hh.pe

Self-hostable Deno Deploy, with the Deno Deploy and Deno Sandbox docs

01	hh.pe	2
02	Deno Deploy	32
03	Reference	42
04	KV	119
05	Deno Sandbox	150

01

hh.pe

About hh.pe 3

State directory 5

Observability 14

Zones 20

Configuration 27

About hh.pe

hh.pe (Hot Handoff Platform Engine, `sandboxo` in the code) is a self-hostable-to-be Deno Deploy ripoff. It speaks enough of Deno Deploy's APIs that the unmodified `deno deploy` CLI, `jsr:@deno/sandbox` and `npm:e2b` work against it, and it runs everything on one host you control: every build, every app instance and every interactive sandbox gets its own gVisor sandbox.

The other sections are Deno's own Deno Deploy and Deno Sandbox documentation (MIT, from denoland/docs). Read them with this page in mind: where they say console.deno.com, think of the hh.pe web UI, and anything listed under What is missing is not there.

How it works

1. Upload: `deno deploy` sends a manifest of file hashes and then uploads only the files the server does not have yet.
2. Build: install and build commands run in a pre-warmed build sandbox, with network for jsr and npm, a timeout and optional cgroup memory and CPU limits. The result is a self-contained directory, packed into a read-only EROFS image, so the app starts with `--cached-only`.
3. Promote: a production revision becomes the app's current revision once its build succeeds.
4. Serve: the first request takes a pre-warmed runtime sandbox, mounts the build image into it and proxies to it. Warm means V8 has already booted, using Deno's prewarm control socket (the mechanism Deno Deploy itself uses), so a cold start takes hundreds of milliseconds.
5. Idle: an instance is stopped after zoned's `idle_ms` without requests, and the pool refills in the background.

Apps are served on `<app>.<domain>`, previews of a single revision on `<app>--<rev>.<domain>`. Records live in Redis and blobs in S3, see State directory. The control plane (sandboxod), the sandboxes (zoned, one per availability zone) and the routers that take public traffic over HTTPS (obice, envoy driven by its own xDS server) are separate programs that can run on separate hosts; zones only dial out, so they can sit behind NAT, see Zones.

What works

- `deno deploy` with `DENO_DEPLOY_ENDPOINT`: deploys (production and preview), `create`, `apps`, `deployments`, `env`, `logs`, `login`. The `deno.json` `deploy` section (`install`, `build`, `predeploy`, `runtime`, `buildTimeout`) overrides the app config per revision.
- Quick deploy links compatible with Deno Deploy's `/new?clone=...`, for public `https://` git repositories. Empty settings are detected from the repository.
- Sandboxes through `jsr:@deno/sandbox`, `deno deploy sandbox` and `npm:e2b`, all on one core: a sandbox created through one API can be used through the other. Volumes, timeouts, `exposeHttp` and an SSH gateway for `exposeSsh`.
- Logs and OpenTelemetry: app, build and sandbox logs are stored in Loki; apps export traces and metrics the way they do on Deno Deploy, without endpoints or credentials in the app; see Observability.
- Web UI: apps, revisions with promote and redeploy, live build and app logs, config and env editors, sandboxes with a web terminal, and these docs at `/docs`.
- Cron: `Deno.cron()` definitions are discovered when a revision is built and run on the production revision, with retries, execution history and traces. Differences: no branch timelines, names are capped at 64

characters by Deno, and the app must call `Deno.serve`.

What is missing

- More than one control plane host, latency-aware routing (a router sends an app's requests to its zones by replica count, not by distance), authentication on tunnels and route fetches beyond a private network, moving volumes between zones.
- Custom domains: routers serve `<domain>` and `*.<domain>` with one certificate.
- WebSockets and other HTTP upgrades through the app router.
- KV, databases, queues, cloud connections, OIDC, `deno run --tunnel`.
- GitHub integration beyond public clones: no GitHub app, no builds on push.
- Env contexts beyond Local: production and preview always share variables; secrets are write-only but stored unencrypted on the host.
- Framework presets other than Next.js: give install, build and runtime settings explicitly.
- Sandboxes: booting from a snapshot, bootable volumes, VS Code, `spawnDeno`, `fetch`, deploying from a sandbox.
- Volume capacity is recorded but not enforced.

Running

```
nix develop
pnpm install
deno task ui:build
deno task docs:build
process-compose up
SANDBOXO_TOKEN=secret deno task dev
```

`dev.ts` runs sandboxod and two zones in one process, with the config the dev shell generates (`SANDBOXO_DEV_CONFIG`); `sandboxod/main.ts` and `zoned/main.ts` run them separately, each with its own config file, see Configuration. The web UI, the API and the Deno Deploy endpoint listen on `http://127.0.0.1:8001`, apps on `https://<app>.sandboxo.localhost:8443` (an app named `root` on `https://sandboxo.localhost:8443` itself). Redis, Garage, the telemetry backends and the router come from `process-compose up`. The router's certificate is signed by a development CA (`packages.dev-tls`): deno in the dev shell trusts it through `DENO_CERT`, curl needs `--cacert $SANDBOXO_DEV_TLS/ca.pem`.

```
export DENO_DEPLOY_ENDPOINT=http://127.0.0.1:8001 DENO_DEPLOY_TOKEN=secret
deno deploy create --org local --app hello --source local \
  --runtime-mode dynamic --entrypoint main.ts --region us
deno deploy --org local --app hello --prod
curl --cacert $SANDBOXO_DEV_TLS/ca.pem https://hello.sandboxo.localhost:8443/
```

Deno sandboxes additionally need `DENO_SANDBOX_ENDPOINT=http://127.0.0.1:8001` and `DENO_DEPLOY_ORG=local`. For e2b, set `E2B_API_URL=http://127.0.0.1:8001`, `E2B_SANDBOX_URL=https://sandboxo.localhost:8443` and `E2B_API_KEY=secret`.

Caveats

- The `deno deploy` CLI keeps one keychain token for all endpoints, so `deno deploy login` against hh.pe replaces your real Deno Deploy token. Prefer `DENO_DEPLOY_TOKEN`, which is never stored.
- The `deno deploy` CLI evaluates server responses as code: only point it at an hh.pe you trust.

State directory

Persistent server state goes through three interfaces: records through a key-value store (`Kv` in `sandboxod/kv.ts`), content-addressed blobs through `BlobStorage` (`common/blobs.ts`) and volume data through `VolumeStorage` (`zoned/btrfs.ts`). By default records live in Redis (`RedisKv`) and blobs in an S3 bucket (`S3BlobStorage`, Garage in the dev shell and the NixOS module); `"kv": {"backend": "fs"}` and `"blobs": {"backend": "fs"}` keep them as plain files instead.

Records belong to sandboxod alone; zones never read Redis (see Zones). Both programs have a state directory, `state` in their config (see Configuration): sandboxod's (default `./state`) holds its git cache and SSH host key, plus records and blobs with the `fs` backends. A zone's (default `./state/zone`, `state/zones/<name>/` with the all-in-one `dev.ts`) holds everything that runs there: volumes as btrfs subvolumes, paused checkpoints, sandbox bundles and a local cache of S3 blobs. With `fs` blobs every zone must see sandboxod's `blobs/` directory (`blobs.dir`), so `fs` only works on one host. Logs are not stored here: app, build and sandbox logs go to Loki only, see Observability.

`kv` is sandboxod's only; `blobs` is required in both programs and must point at the same storage:

Key	Default	Meaning
`kv.backend`	`redis`	`redis` or `fs` (`state/kv/`)
`kv.url`	`redis://127.0.0.1:6379`	secret, `redis://[:password@]host[:port][/db]`, `rediss://` for TLS
`kv.prefix`	`sandboxo:`	prefix of every Redis key, so servers can share a database
`blobs.backend`	required	`s3` or `fs`
`blobs.dir`	`<state>/blobs`	blob directory for `fs`
`blobs.endpoint`	`http://127.0.0.1:3900`	S3 endpoint URL, path-style requests
`blobs.region`	`garage`	signing region
`blobs.bucket`	`sandboxo`	bucket, must exist
`blobs.access_key_id`, `blobs.secret_access_key`	required for `s3`	secrets, credentials with read, write and delete on the bucket
`blobs.prefix`	empty	prefix of every object key
`blobs.concurrency`	32	parallel S3 requests

Two servers must not share a Redis prefix or an S3 prefix: garbage collection deletes every blob under its prefix that its own records do not reference.

Layout

sandboxod:

```
state/
  kv/                          fs kv only: records, see Key-value store
    apps/<app>/
      config.json              app config
      meta.json                app id and timestamps
      current.json             promoted revision id
```

```
    playground.json            playgrounds only: code volume and creation time
    env/<NAME>.json            one environment variable
    revisions/<rev>.json       revision record
    status/<rev>.json          build status
    crons/<name>.json          one cron job's recent runs (name base64url)
  sandboxes/<sandbox id>.json  generic sandbox record
  volumes/<volume id>.json     volume record
  snapshots/<snapshot id>.json snapshot record
  org-slugs/<org>/<slug>.json  volume and snapshot slug index, per organization
  templates/<template id>.json e2b template record
  template-builds/<template id>/<build id>.json   template build record
  template-files/<template id>/<files hash>.json  uploaded COPY archive
  org-template-names/<org>/<name>.json  template name index, per organization
  zones/<name>.json            zone record and token
  routers/<name>.json          router, its tunnel address and when it last fetched routes
  editors/<org>/<user>.json    VS Code user settings, synced from editors
  sessions/<sha256(id)>.json   UI login session (oidc only)
  meta/legacy-org.json         organization that owns pre-tenancy records
  .trash/                     subtrees being deleted
blobs/                         fs blobs only, shared with the zones
  <sha256>                     file contents, tree objects, build and template images, manifests, logs
  .tmp-<uuid>                  blob being written
blob-cache/                    s3 blobs only
  .tmp-<uuid>                  upload being hashed
git/
  <hash(url)>.git/             bare repository cache (gix)
ssh_host_ed25519_key           SSH gateway host key
```

zoned:

```
state/zone/
  blob-cache/                    s3 blobs only
    <sha256>                     local copy of a blob opened as a file
    .tmp-<uuid>                  blob being hashed before upload
    <sha256>.<uuid>.part         blob being downloaded
  volumes/                       btrfs only
    <volume id>/                 volume data, a subvolume
    <snapshot id>/               read-only snapshot of a volume
    .trash/                      subvolumes being deleted
  paused/
    <sandbox id>/                runsc checkpoint of a paused sandbox
    <sandbox id>.json            its spec and creation time
  egress/
    ca/                          iron-proxy CA
    sandboxes/<sandbox id>/      proxy config and secrets of one sandbox
  templates/
    layers/                      registry layer cache
    bases/                       unpacked base images
    work/                        template builds in progress, wiped on startup
  nix/
    <encoded flake ref>          GC root symlink to a sandbox profile
    sandbox-store/nix/store/     private Nix store, mounted at /nix/store
  slots/                         sandbox bundles, wiped on startup
    none/sandbox-<id>/           warm runtime sandboxes, network none
    sandboxed/sandbox-<id>/      warm runtime sandboxes, network sandboxed
    build/<id>/                  build sandboxes
    sandbox-<network>/sandbox-<id>/   generic sandboxes (e2b, Deno Sandbox)
    volume-tar/                  sandboxes packing volume archives
    templates/                   template build sandboxes
  tmp/
  tools/
    inspect.js                   repository inspection, run in build sandboxes
    agent.js                     generic sandbox agent
```

```
  envoy/                        the zone's obice: bootstrap.json (holds the zone token), obice.log,
envoy.log, tunnels.log
```

Path	Lifetime	Created by
kv/, blobs/	persistent (sandboxod)	FsKv.init, FsBlobStorage.init (sandboxod/kv.ts, common/blobs.ts)
volumes/, paused/, egress/	persistent (zone)	BtrfsVolumeStorage.init (zoned/btrfs.ts), SandboxHost (zoned/sandboxes.ts), Egress.create (@in3pre3/sandbox/egress)
blob-cache/	cache, safe to delete while the program is stopped	S3BlobStorage.init (common/blobs.ts)
git/	cache, safe to delete	GitCache.init (sandboxod/git.ts), git_cache moves it
nix/	cache, rebuilt on startup	SandboxTemplate.prepare (@in3pre3/sandbox), zoned's gc_root_dir moves it
templates/	cache, work/ removed on every startup	TemplateBuilder (zoned/templates.ts)
slots/	ephemeral, removed on every zone startup	sandbox pools (zoned/pool.ts, zoned/)
tools/	regenerated on every startup	bundleTools (zoned/tools.ts, Deno.bundle)
ssh_host_ed25519_key	persistent, created once (mode 0600)	SSH gateway (sandboxod/ssh.ts), unless sandbox.ssh is null

Key-value store

Kv (sandboxod/kv.ts) stores JSON values under keys that are lists of segments matching [A-Za-z0-9_-]+. Store (sandboxod/store.ts) is the only user; it validates records with zod on read. Operations: get, set, create (only if absent), update (read-modify-write), delete, deleteTree (a key and everything below it) and list (the immediate children of a prefix, sorted). Keys with other segments read as missing; writing them throws.

FsKv maps key [a, b, c] to the file kv/a/b/c.json and its children to the directory kv/a/b/c/, so a key can hold a value and have children.

- set writes .tmp-<uuid> next to the target (mode 0600) and renames it into place.
- create writes the temp file and hardlinks it to the target, which fails atomically if the target exists.
- update runs one at a time per key within the process.
- deleteTree renames the subtree into kv/.trash/<uuid> (atomic), deletes it there, then deletes the key's own value.
- On startup FsKv.init empties .trash/ and removes stray .tmp-* files.

RedisKv keeps the value of [a, b, c] as a JSON string under <prefix>v:a/b/c and the children of every prefix in a set, <prefix>i:a/b holding c, <prefix>i:a holding b and <prefix>i: holding a.

- set, create and the final write of update are one Lua script each, so the value and the parent sets change atomically. create is SET only if absent.
- update reads the value, runs the function and writes only if the value is still the one it read (compare-and-

set in the script), retrying on conflict, so it is atomic across processes. The function may run more than once.

- `delete` removes the value and, when the key has no children, its name from the parent set, in one script.
- `deleteTree` walks the sets below the prefix and deletes every value and set it finds; it is not atomic.
- `list` reads the prefix's set. A name stays listed while it has children, like a directory in `FsKv`.

Apps

`kv/apps/<app>/` exists for every app; `<app>` is a lowercase DNS label of at most 32 characters, so preview hosts `<app>--<rev>` stay within the 63-character label limit. It starts with a letter, so it never looks like a sandbox port host (`<port>-sbx-<region>-<20>`); ssh and names shaped like `exposeHttp` labels (`x-<region>-<20>`) are reserved.

`config.json`

The app config. Fields: `entrypoint`, `args`, `installCommand`, `buildCommand`, `preDeployCommand`, `buildDirectory`, `network` (`none` or `sandboxed`), `buildTimeout` (minutes or `null`), `buildMemoryLimit` (MiB or `null`), `region` (3-letter region code or `global`) and `maxReplicas` (1 to 16), see Scaling. Missing fields take their defaults (`appConfig` in `common/appconfig.ts`), so older files stay readable. The environment is not stored here: `readConfig` fills `env` from `env/` and `writeConfig` drops it.

`env/<NAME>.json`

One environment variable per key, so every change is a single atomic write: `{"value", "secret", "local", "updatedAt"}`. `<NAME>` matches `[A-Za-z_][A-Za-z0-9_]*`. `local` puts the variable in the Local context (`context_ids: null` in the deno deploy API, otherwise production and preview). A `secret` variable is write-only: its value is never sent to a browser, the admin API or the deno deploy CLI, it cannot become non-secret again, its `local` flag is fixed, and saving it with an empty value keeps the sealed one. Values are stored as plain text; builds and instances get all variables, and revision records keep a snapshot of them.

`meta.json`

`{"id", "createdAt", "updatedAt"}`: a UUID used as the app id by the deno deploy API, and ISO timestamps. `updatedAt` changes on every config write.

`current.json`

The promoted revision id as a JSON string. Promotion is a single `set`, so the switch is atomic. Missing until the first promotion.

`playground.json`

Playgrounds only: `{"volume", "createdAt"}`, the volume holding the code.

Builds and revisions

A revision id is `<milliseconds in base 36>-<8 hex>` (for example `mvfobbyg-21e28c6c`), so ids sort by creation time. Both live under `kv/apps/<app>/`; revisions are listed by their `status/` keys.

`status/<rev>.json`

`{"status", "updatedAt"}`: `status` is `uploading`, `queued`, `building`, `ok` or `failed`, `updatedAt` the ISO time of the last change.

revisions/<rev>.json

The revision record:

```
{
  "prod": true,
  "createdAt": "2026-10-19T19:54:54.664Z",
  "config": { "entrypoint": "...", "...": "..." },
  "source": { "tree": "<sha256 of the root tree>" },
  "git": { "url": "https://...", "ref": "main", "commit": "<sha1>" },
  "manifest": { "blob": "<sha256>", "files": 42 },
  "detect": { "install": true, "build": true, "runtime": true },
  "build": { "image": "<sha256>", "size": 80928768, "nodeModules": true },
  "crons": [{ "name": "cleanup", "schedule": "0 * * * *", "backoffSchedule": [1000] }]
}
```

`config` is a snapshot of the app config (env included) taken when the revision was created; later config changes apply to the next revision. `source.tree` is the root of the source tree in `blobs/`; it is set once the upload completes or the git fetch finishes, and a redeploy copies it. `git` is set for quick-deploy revisions (`ref` only when given, `commit` after the fetch). `manifest` points to the deno deploy upload manifest, stored in `blobs/` because it can list hundreds of thousands of files, and keeps its file count. It stays after the upload. `detect` marks the settings to fill from repository inspection. `build` is set by a successful build: `image` is the build output as an EROFS image in `blobs/` (the source copy plus everything the build produced: `node_modules`, `.deno_dir` with the module cache, framework output such as `.next`), mounted read-only at `/app` in runtime sandboxes; `nodeModules` adds `--node-modules-dir=manual` to the start line. `crons` lists the `Deno.cron()` jobs the revision registered when the build booted it once (the warmup); a revision without the field was built before cron support and runs no crons. Failed builds keep no output.

crons/<name>.json

`{"name", "runs"}` for one cron job, keyed by the base64url of its name (cron names may contain spaces). `runs` holds the last `cron.history` runs, oldest first:

```
{
  "id": "mvhngj9o-1a2b3c4d",
  "schedule": "0 * * * *",
  "rev": "mvhngbru-139140f2",
  "attempt": 1,
  "scheduledAt": "2026-10-21T05:00:00.000Z",
  "startedAt": "2026-10-21T05:00:00.012Z",
  "finishedAt": "2026-10-21T05:00:00.140Z",
  "status": "ok",
  "traceId": "<32 hex>",
  "zone": "ord1"
}
```

`status` is `running`, `ok` or `error` (with `error`); a retry is a new run with `attempt` + 1.

Blob storage

`blobs/` holds content-addressed objects named by the hex sha256 of their bytes (`FsBlobStorage` in `common/blobs.ts`):

- file blobs: raw file contents, the same hashes the deno deploy CLI uses in upload manifests;
- tree blobs: one directory each, canonical JSON `{"tree":1,"entries":[...]}` sorted by name, with entries `{name, type: "file", mode, size, hash}`, `{name, type: "dir", hash}` or `{name, type: "symlink", target}` (`common/tree.ts`);

- build images: uncompressed EROFS images of build output, packed inside the build sandbox (`mkfs.erofs`, `erofsMkfsArgs` in `@in3pre3/sandbox`);
- upload manifests: JSON `{"<path>": "<sha256>"}` from the deno deploy CLI (`Sources.putManifest`).

New blobs are written to `.tmp-<uuid>`, hashed while streaming and renamed into place with mode 0444. Blobs from before the read-only mode are 0644. The directory is flat.

`S3BlobStorage` stores the same objects under `<blobs.prefix><sha256>`. Byte arrays are hashed in memory; streams are written to `blob-cache/.tmp-<uuid>` while hashing, uploaded (multipart above 64 MiB) unless the object already exists, and the file becomes the cache entry. Readers that need a file (blob mounts, the runtime `/app` image, template layers) go through `local`, which downloads into `blob-cache/<sha256>` once; `FsBlobStorage` returns its own file. Deleting a blob deletes the object and the cache entry. A missing object reads as `Deno.errors.NotFound`, as a missing file does.

Garbage collection (`Sources.gc` on sandboxod, every `gc_interval_ms`) keeps blobs reachable from any revision's `source.tree`, every `build.image`, every `manifest.blob`, the files listed in the manifests of revisions created less than a day ago that have no tree yet (uploads in progress), template build images and logs, uploaded template files, and the images of sandboxes that are not stopped, and deletes the rest once they are older than `gc_grace_ms`; stale `.tmp-*` files go too. Zones only add blobs (build and template images, template logs); their `blob-cache/` copies of deleted blobs stay until the cache is cleared. Build sandboxes see blobs through hardlinks in their blob mount directory (below), so deleting a blob does not break a running build.

Git cache

`git/<first 32 hex of sha256(url)>.git/` is a bare repository per source URL, created and fetched with gix (`crates/gitclone`). Fetches are shallow (depth 1), go to `refs/sandboxo/fetched` (`+<ref or HEAD>:refs/sandboxo/fetched`, no tags) and use only the repository's own config. Nothing is checked out; the fetched commit is imported into `blobs/` and the revision records the tree. The cache only speeds up refetches and can be deleted at any time.

Nix

`nix/<encodeURIComponent(flake ref)>` are GC root symlinks to the built sandbox profiles (`SANDBOXO_PROFILE`, `SANDBOXO_SANDBOX_PROFILE`), so the host Nix garbage collector keeps them. `nix/sandbox-store/` is a private Nix store (`nix copy --to local?root=...`) holding only the profiles' closures; its `nix/store` is bind-mounted read-only at `/nix/store` in every sandbox. The directory belongs to the zone and can be moved with `gc_root_dir`.

Slots

`slots/` holds the runsc bundles of all live sandboxes of a zone and is deleted when the zone starts. Every bundle directory (`sandbox-<16 hex>/`) has:

```
sandbox-<id>/
  config.json                   OCI spec passed to runsc
  rootfs/                       root skeleton: /etc files, mountpoints
  state/                        runsc --root: <container>.state, .lock, runsc-<container>.sock
  slots/<name>/                 host side of bind-mounted slots without a hostDir
  blobfs/<n>/                   blob mount control dir (build sandboxes)
    blobfs.sock                 served by sandboxo, used by the blobfs gofer
    files/                      hardlinks of blobs opened in the sandbox
```

```
vsock/                          mounted at /run/vsock (runtime sandboxes)
  2/4318                        host OTLP receiver (guest vsock connect to cid 2)
  2/4320                        host cron socket (DENO_UNSTABLE_CRON_SOCK)
  3/<port>                      guest vsock listeners (host connects here)
```

`slots/none/`, `slots/sandboxed/`

Warm runtime sandboxes per network flavor. `/app` is an empty mountpoint in the rootfs; on assignment the revision's build image is mounted there (`runsc debug --mount`).

`slots/build/<8 hex>/`

One directory per build sandbox: `out/` is the read-write `/out` slot that receives `app.erofs` (kept on shutdown, then streamed into `blobs/`) next to the bundle. The read-write `/app` slot is the bundle's `slots/app/`. The bundle's `blobfs/0/` backs the read-only `/src` mount that serves the revision's source tree.

`slots/sandbox-<network>/`

Generic sandboxes (e2b and Deno Sandbox APIs). Their rootfs has an FHS layout (`/usr/local/bin`, `/home/...`) and a sparse `.gvisor.filestore.<container>` file (1 GiB apparent size) backing the writable root overlay.

Generic sandboxes and volumes

`kv/sandboxes/<id>.json`

One record per generic sandbox, kept after it stops. Ids are `sbx_<region>_<20 base32 chars>`. Fields: `id`, `api` (e2b or deno), `template`, `labels`, `network`, `memoryMb`, `cpuCount`, `volumes` (`[{volume, path}]`), `createdAt`, `stopAt`, `stoppedAt`, `pausedAt`, `autoPause`, `envdAccessToken`, `zone`, `profile` and `envdVersion` (of the zone that runs it, kept so a checkpoint resumes with the same profile), `stopReason`, `image` (`{build, hash, user, workdir}` for template sandboxes), `egress` (policy without secret values) and the front-door state that has to survive a sandboxod restart: `exposures` (`[{label, port}]` of `exposeHttp` hosts, `port` is `null` while a label is reserved but not bound), `sshUsers` and `vscode` (`{label, pid}`).

`kv/volumes/<id>.json`

Volume record: `id`, `slug`, `capacity`, `createdAt`, `zone`, and `baseSnapshot` (`{id, slug}`) for volumes created from a snapshot. Ids are `vol_<region>_<20 base32 chars>`. `capacity` is not enforced.

`kv/snapshots/<id>.json`

Snapshot record: `id`, `slug`, `volume` (`{id, slug}` of the source volume), `createdAt`, `zone`. Ids are `snp_<region>_<20 base32 chars>`.

`kv/zones/<name>.json`

`{"name", "token", "region"?, "cordoned"?, "url"?, "bootId"?, "registeredAt"?}`, see Zones. The token is stored in plain text.

`kv/templates/<id>.json`, `kv/template-builds/<id>/<build>.json`

e2b templates: `{id, name, createdAt, tags: {<tag>: {buildID, createdAt}}}` and one record per build with `status` (`waiting`, `building`, `ready`, `error`), `tags`, `cpuCount`, `memoryMB`, `createdAt`, `finishedAt`, `reason`, `zone`, and once ready `image` (`{hash, size}` of the EROFS image in blob storage), `config` (`{env, workdir, user}`) and `logBlob` (the build log as JSON lines).

`kv/sessions/<sha256(id)>.json`

`{sub, name, email, picture, idToken, createdAt, expiresAt}`: a UI login. The cookie holds the random id, the key only its hash. The ID token is kept as the `id_token_hint` for logout. Expired sessions

are dropped when read and swept on the blob GC interval.

kv/meta/legacy-org.json

The organization that owns records without an `org` field (written before multi-tenancy) and records of the single-org `org`. Startup moves the old `slugs/`, `template-names/` and `editor/user` keys under it and refuses to start when `oidc.legacy_org` names a different organization than the stored one.

Ownership

App metas, sandbox, volume, snapshot and template records carry `org`, the owning organization id. Every API resolves ids and slugs within the caller's organization, so an id of another organization reads as not found.

kv/template-files/<id>/<files hash>.json

`{"blob", "size", "createdAt"}`: a `COPY` archive uploaded by the e2b SDK, stored in blob storage under its own hash.

kv/editors/<org>/<user>.json

`{"settings.json"?, "keybindings.json"?}`: the raw text (comments kept) of the VS Code user files of one user in one organization (`admin` for the admin token), at most 256 KiB each. Written into every VS Code that user opens from the UI and updated when one of them changes the files.

kv/org-slugs/<org>/<slug>.json

`{"id"}` of a volume or snapshot. Volumes and snapshots of an organization share one slug namespace, as in the Deno Sandbox API. The slug is claimed with `create` before anything else is written and released again if creation fails, so two objects never share a slug. Lookups by slug go through it.

volumes/<vol id>/, volumes/<snp id>/

`BtrfsVolumeStorage` keeps every volume as a btrfs subvolume (`btrfs subvolume create`), bind-mounted read-write into the sandboxes that attach it. A snapshot is a read-only snapshot of that subvolume (`btrfs subvolume snapshot -r`). A volume created from a snapshot is a writable snapshot of it, so it shares data copy-on-write with its base and outlives it. Snapshots of a volume in use are crash-consistent: sandbox volume mounts use runsc's default `--file-access-mounts=shared`, so gVisor writes through to the host and btrfs flushes pending writes when it takes the snapshot.

Deleting a volume or snapshot removes its record and slug, clears the read-only flag of a snapshot (a read-only subvolume cannot be moved), moves the subvolume into `volumes/.trash/` and destroys it there: `btrfs subvolume delete` when permitted (root, or `user_subvol_rm_allowed`), otherwise a recursive delete, which works rootless. Startup destroys whatever is left in `.trash/`.

The `btrfs` binary (btrfs-progs, in the dev shell) must be on `PATH`. If `volumes/` is not on btrfs the server logs a warning at startup and creating volumes or snapshots fails; everything else works.

Outside the state directory

`$XDG_RUNTIME_DIR/sandboxo-<8 hex of sha256(zone state path)>/` (or `/tmp/...`) holds the sockets a zone shares with its sandboxes, because unix socket paths are limited to 108 bytes. It is recreated on startup and removed on shutdown:

- `<network>-<8 hex>/`: runtime sandbox `/run/sbx`, with `ctl.sock` (Deno prewarm control) and `http.sock` (the app's HTTP server);

- s-<8 hex>/: generic sandbox /run/sbx, with ctl.sock, agent.sock and p<port>.sock for exposed ports.

Build cgroups (build.cgroup_parent) and sandbox cgroups (sandbox.cgroup_parent) live in the cgroup hierarchy, not in the state directory.

Filesystem requirements

- A zone's blob-cache/ (or the shared blobs/ with fs blobs) should be on the same filesystem as its slots/ : blob mounts hardlink blobs into slots/build/.../blobfs/<n>/files/ and fall back to copying.
- volumes/ must be on btrfs for volumes and snapshots. Smokes that use volumes keep their state in SANDBOXD_SMOKE_DIR (default ./.smoke/) instead of $TMPDIR, which is often tmpfs.
- State directories from before blob storage may still contain uploads/; nothing reads it, and revisions without source.tree cannot be redeployed.
- State directories from before the key-value store keep records in apps/, builds/, sandboxes/ and volumes/<id>.json; nothing reads them. Their volume directories are plain directories, not subvolumes, and cannot be snapshotted.
- State directories from before zones keep volumes/, paused/, slots/ and nix/ at the top level, where no zone looks. Volume and sandbox records without zone cannot be placed; recreate them.

Observability

Logs live in Loki and nowhere else: app instance output, build output and generic sandbox output are pushed there, and the web UI, the admin API and `deno deploy logs` read them back from it. Apps also export OpenTelemetry traces and metrics through Deno's built-in OTLP exporter, using the same channel Deno Deploy uses: vsock to the host. The app never sees an endpoint, a tenant or a credential; hh.pe attributes data to the sandbox it came from and adds those on the host.

Logs

Source	Loki stream labels	Structured metadata
App instance stdout/stderr and lifecycle events (`starting`, `serving`, `idle`, `stopped`, exit codes)	`service_name=<app>`, `source=app`	`rev`, `replica`, `stream` (`out`, `err`, `sandboxo`)
Build output: shell trace (`+ ...`), command output, limits, peak memory, image size, failure reason	`service_name=<app>`, `source=build`	`rev`, `stream=build`
Generic sandbox agent and envd output, lifecycle events	`service_name=sandbox`, `source=sandbox`	`sandbox`, `stream`
OTel log records an app emits itself (OTLP, see below)	`service_name=<app>`, `service_instance_id=<replica>` and Loki's other OTLP labels	`service_version=<rev>`, `severity_text`, `trace_id`, ...

Lines are batched for 250 ms (1000 lines max) and retried every 2 s while Loki is unreachable, keeping at most 20000 pending lines per instance, build or sandbox. Every build step's output is flushed before the build status changes, so a finished build's log is complete. Console output goes through stdout and stderr (`OTEL_DENO_CONSOLE=ignore`): Deno's OTLP log batches are lost when an instance is stopped before they are exported, stdout is not.

Readers: the app page shows the last 1000 lines of the last 7 days, build and sandbox pages everything since the revision or sandbox was created, `GET /apps/<app>/logs?lines=N` and `GET /apps/<app>/revisions/<rev>/log` return plain text, and `deno deploy logs` prints the history and then follows. Queries span at most 30 days (Loki's default `max_query_length`). Every reader goes through one place in sandboxod (`sandboxod/otel.ts`): identical windows asked for within 2 s share one Loki query, and the org page's usage totals share one Mimir query per 15 s.

The web UI renders those lines with the page and then streams new ones as server-sent events from `/otel/logql`; `deno deploy logs` continues the same way. Every follower of the same log, in the UI or the CLI, shares one Loki live tail (`/loki/api/v1/tail`, a WebSocket; `delay_for` holds lines for 1 s so late writers still arrive in order), so a new line shows up about a second after it was written. Loki allows 10 tails per tenant by default (`max_concurrent_tail_requests`) and ends each after an hour (`tail_max_duration`); sandboxod reconnects, and polls `query_range` every second while Loki refuses a tail. Metric charts refresh from `/otel/promql` every

step while the tab is visible; identical chart queries from several viewers share one Mimir request per step. Both endpoints only run the query and tenant the page signed (see `signing_secret` below).

Without `telemetry.loki` logs are dropped (with a warning at startup) and the readers report that log storage is not configured.

OpenTelemetry

```
app (deno, OTEL_DENO=true, OTEL_DENO_VSOCK=2:4318)
  -> AF_VSOCK connect(cid 2, port 4318)        gVisor sentry (bundled runsc)
  -> /run/vsock/2/4318 in the sandbox
  -> slots/<network>/sandbox-<id>/vsock/2/4318 on the host
  -> hh.pe OtlpForwarder (one receiver per running instance)
       /v1/traces  -> telemetry.otlp.traces
       /v1/metrics -> telemetry.mimir.url + /otlp/v1/metrics
       /v1/logs    -> telemetry.loki.url + /otlp/v1/logs
       + X-Scope-OrgID: <tenant>, + configured headers
```

- gVisor has no vsock. The bundled runsc ships a sentry with an `AF_VSOCK` socket provider: vsock sockets are unix stream sockets underneath, and `(cid, port)` maps to `/run/vsock/<cid>/<port>` (guest CID 3, host CID 2), the same convention Firecracker uses for guest-initiated vsock connections. The runtime sandboxes mount a host directory at `/run/vsock` and run with `--host-uds=all`.
- When an instance starts, hh.pe opens an HTTP server on that sandbox's `vsock/2/4318` and sends these variables in the start line, after the app's own env so the app config cannot override them: `OTEL_DENO=true`, `OTEL_DENO_VSOCK=2:4318`, `OTEL_EXPORTER_OTLP_PROTOCOL=http/protobuf`, `OTEL_DENO_CONSOLE=ignore`, `OTEL_SERVICE_NAME=<app>`, `OTEL_RESOURCE_ATTRIBUTES=service.version=<rev>,service.instance.id=<replica>`. Other `OTEL_*` settings from the app env apply, for example `OTEL_METRIC_EXPORT_INTERVAL`.
- The forwarder accepts OTLP/HTTP (protobuf or JSON, compressed or not) up to 16 MiB per request, forwards it with a 10 s timeout and answers 503 when the backend is unreachable, so the exporter retries. Signals without a configured URL are accepted and dropped.
- The tenant is set by the host, so an app can only write into its own tenant, even if it speaks OTLP over `/run/vsock` itself.

Only app instances export OpenTelemetry; builds and generic sandboxes only log. Export is on when Loki, Mimir or a traces URL is configured; otherwise apps get no OTLP variables. Runtime sandboxes always have vsock: the cron socket (`vsock/2/4320`, `DENO_UNSTABLE_CRON_SOCK`) uses it too.

Usage

The organization page (`/<org>`) shows requests, CPU time, outbound and inbound traffic of the last 24 hours, in total and requests per app. hh.pe meters this itself, so it does not depend on what an app exports:

- requests, inbound and outbound bytes are counted by the zone for every request it forwards to an app: request line, headers and body in, status line, headers and body out (as sent to the client, after decompression). Traffic of an app's own outgoing connections is not counted;
- CPU time is the host CPU of the instance's sandbox (sentry and application), read from its cgroup `cpu.stat` every flush and when it stops, minus the value when it was taken from the pool, so the warm V8 boot is not billed. This needs runtime sandboxes in cgroups (`runtime_cgroup_parent`, `auto` by default); zones without them are named on the organization page and their CPU time stays `-`. gVisor's own accounting (`runsc events --stats`) is not used: it reports about half a core for an idle Deno;

- every `telemetry.usage_interval_ms` (15 s) the deltas since the last flush are pushed to Mimir as OTLP gauges `sandboxo.app.requests`, `sandboxo.app.cpu.seconds`, `sandboxo.app.inbound.bytes` and `sandboxo.app.outbound.bytes` with an `app` attribute, in the org tenant. The page sums them with `sum_over_time(...[24h])`, which is exact and survives restarts. Deltas that fail to push are kept for the next flush.

Metrics

`/<org>/<app>/metrics` charts the app over the last 15 minutes, hour, 6 hours, day or week, about 120 points per chart (step 15 s for 15 minutes, 30 s for an hour, 5 minutes for a day, 1.5 hours for a week), times in UTC:

Chart	Source	Query
Requests	usage, org tenant	`sum(sum_over_time(sandboxo_app_requests{app}[step]))`, bars per step
Latency	Deno, app tenant	`histogram_quantile` 0.5, 0.95, 0.99 of `rate(http_server_request_duration_bucket{job}[max(step, 2m)])`
Responses	Deno, app tenant	`rate(http_server_request_duration_count{job})` summed by status class (2xx … 5xx)
CPU	usage, org tenant	`sandboxo_app_cpu_seconds` per step as percent of one core; needs runtime cgroups
Traffic	usage, org tenant	`sandboxo_app_inbound_bytes` and `_outbound_bytes` per step, in bytes per second
Memory	Deno, app tenant	`sum(v8js_memory_heap_size{job})` over all instances

Charts are server-rendered SVG in a `MetricChart` island that adds a crosshair and the values under the cursor. Deno exports metrics every 60 s by default (`OTEL_METRIC_EXPORT_INTERVAL`), so the latency and response charts lag by up to a minute and stay empty for apps without OTel export.

Traces

`/<org>/<app>/traces` lists the app's requests from Tempo (`telemetry.tempo`, queried with the app's tenant): one row per root span, newest first, with the revision identicon, method and path, HTTP status and duration. It shows up to 100 traces of the last 15 minutes to 7 days, optionally only failed requests (span status error or a 5xx response) or only one revision ("Traces of this revision" on a build page). The query is TraceQL:

```
{ resource.service.name = "<app>" && nestedSetParent < 0 [&& ...] }
  | select(span.http.request.method, span.url.path,
           span.http.response.status_code, span.cron.name,
           resource.service.version, status)
  with (most_recent=true)
```

The search box takes the cron filters of Deno Deploy: `kind:cron` adds `span.cron.name =~ ".+"`, `cron.name:<name>` and `cron.schedule:<schedule>` (quote values with spaces, `cron.schedule:"0 * * * *"`) match those span attributes; `span.cron.name` is selected as well. Tempo reports selected attributes a span does not have as the string `nil`, which the list treats as missing.

Cron runs are traced from the platform side: the scheduler mints the trace id and passes it to deno as `traceparent`, and the zone exports a root span `cron <name>` (internal, `cron.name`, `cron.schedule`, `cron.attempt`, `cron.run.id`, error status with the failure reason) as OTLP/JSON to the traces URL with the

app's tenant and resource. deno's own `deno.cron` span, with the handler's exception, and everything the handler does nest under it. The Cron pages link each run to its trace.

A trace page (`/<org>/<app>/traces/<id>`, from Tempo's `/api/v2/traces/<id>`) shows the spans as a waterfall: nested by parent, bars on the trace's time axis, colored by kind (server, client, internal) and red on error. Opening a row shows its attributes, events (such as recorded exceptions) and status message. Below are the app's logs from 20 ms before the trace to 100 ms after it, for the trace's revision: console lines carry no trace id, so this is a time window and can include lines of concurrent requests. Times on these pages are UTC.

Configuration

Zones push: app, build and sandbox logs, the OTLP forwarder and usage metering run in zoned, so a zone needs the Loki, Mimir and traces URLs and headers. Usage gauges carry a `zone` attribute. sandboxod pushes only the build lines it writes itself (git fetch, dispatch failures) and queries Loki, Mimir and Tempo for the web UI and the APIs. Both take the same `telemetry` section in their config (see Configuration); header values are secrets:

```
"telemetry": {
  "loki": { "url": "https://loki.example.com", "headers": { "Authorization": { "_env": "LOKI_AUTH" } } },
  "mimir": { "url": "http://127.0.0.1:9009" },
  "tempo": { "url": "http://127.0.0.1:3200" },
  "otlp": { "traces": "http://127.0.0.1:4318/v1/traces" }
}
```

Key	Meaning
`telemetry.loki.url`	Loki base URL, for example `http://127.0.0.1:3100`: push API, query API and OTLP logs endpoint
`telemetry.loki.headers`	Extra headers for every Loki request, for example credentials of a hosted Loki
`telemetry.otlp.traces`	Full OTLP/HTTP traces URL, for Tempo `http://127.0.0.1:4318/v1/traces`
`telemetry.otlp.headers`	Extra headers for forwarded traces
`telemetry.otlp.max_body_bytes`, `telemetry.otlp.timeout_ms`	Largest forwarded OTLP request (16 MiB) and the forward timeout (10 s)
`telemetry.tempo.url`	Tempo query API base URL, for example `http://127.0.0.1:3200`, for the traces pages (ingest goes to `telemetry.otlp.traces`)
`telemetry.tempo.headers`	Extra headers for Tempo queries
`telemetry.mimir.url`	Mimir base URL, for example `http://127.0.0.1:9009`: OTLP metrics endpoint for app metrics and usage, Prometheus query API for the organization page
`telemetry.mimir.headers`	Extra headers for every Mimir request
`telemetry.tenant`	`X-Scope-OrgID` value: `app` (default, one tenant per app; generic sandboxes use the org), `org`, or a fixed string
`telemetry.usage_interval_ms`	How often zones push usage deltas (15 s)
`runtime_cgroup_parent` (zoned)	Cgroup parent for runtime sandboxes, `auto` (default), a path or `null` (see Configuration). Enables CPU time usage

signing_secret (sandboxod)	Required, a secret of at least 32 characters, keep it stable: it signs the query URLs pages hand to the log and chart islands (valid for 12 hours and only with the same login) and the e2b template upload links. Rotating it stops open pages from refreshing until they are reloaded. The dev config reads a fixed value from the dev shell's SANDBOXO_SIGNING_SECRET; the NixOS module generates one into /var/lib/sandboxo-secrets/ unless services.sandboxod.signingSecretFile is set

Local backends

process-compose.yaml runs Grafana Mimir, Tempo and Loki from nixpkgs (all in the dev shell), single binary each, multi-tenant, listening on 127.0.0.1 only. Their object storage is a single-node Garage with buckets loki, mimir and tempo, next to the sandboxo bucket for hh.pe blobs; garage-init (nix/garage-init.sh) assigns the layout, imports the dev key and creates the buckets before the backends start. Redis (AOF persistence) holds hh.pe records. The dev shell sets the shared dev credentials (GARAGE_RPC_SECRET, SANDBOXO_S3_ACCESS_KEY_ID, SANDBOXO_S3_SECRET_ACCESS_KEY) and SANDBOXO_REDIS_URL, which the dev config reads through _env placeholders, so run process-compose and dev.ts from it. Local state (WAL, Garage metadata and data, the AOF) lives in observability/data/, configs in observability/:

Backend	HTTP	Other ports
Mimir	9009	gRPC 9095
Tempo	3200	OTLP gRPC 4317, OTLP HTTP 4318, gRPC 9195
Loki	3100	gRPC 9096
Garage	S3 3900	RPC 3901, admin 3903
Redis	-	6380

The dev config (SANDBOXO_DEV_CONFIG, see dev.ts) points hh.pe at them through its telemetry section, and the dev shell configures process-compose itself (PC_CONFIG_FILES=process-compose.yaml, API on PC_PORT_NUM=18080):

```
nix develop
process-compose up            # TUI; add -t=false to run headless
SANDBOXO_TOKEN=secret deno task dev
```

Every query needs the tenant header, for an app hello:

```
curl -H 'X-Scope-OrgID: hello' 'http://127.0.0.1:3200/api/search?tags=service.name%3Dhello'
curl -H 'X-Scope-OrgID: hello' 'http://127.0.0.1:3100/loki/api/v1/query_range?
query=%7Bservice_name%3D%22hello%22%7D'
curl -H 'X-Scope-OrgID: hello' 'http://127.0.0.1:9009/prometheus/api/v1/label/__name__/values'
```

In Grafana, add one Prometheus (http://127.0.0.1:9009/prometheus), Tempo and Loki data source per tenant with the X-Scope-OrgID header set.

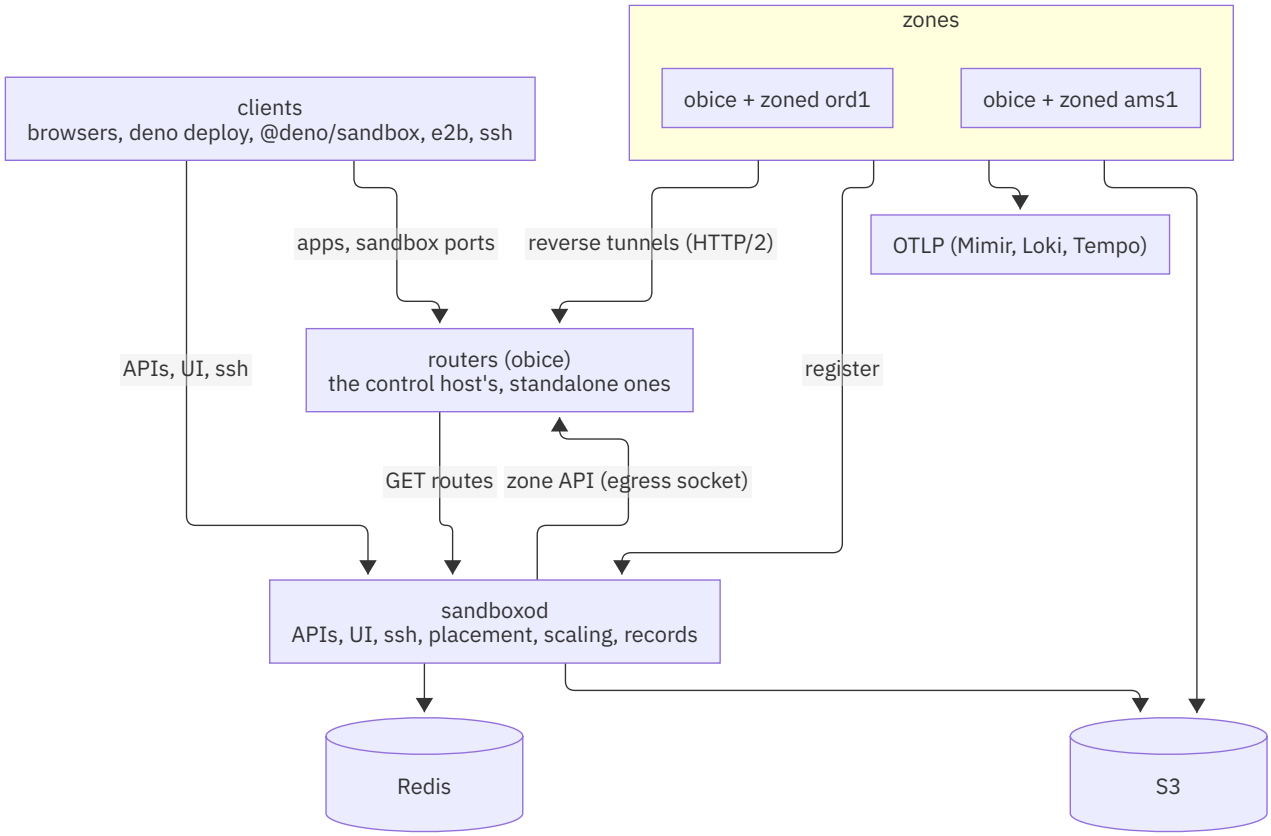
Mimir must advertise 127.0.0.1 for every ring and for the query frontend (frontend.address); with an auto-detected interface address its queriers cannot report finished queries and every query hangs. Loki's ingestion limits are raised (32 MB/s, 64 MB bursts) so large build logs are not rejected.

The smoke tests that read logs (smoke/deploy.ts, smoke/compat.ts, smoke/build_limits.ts, smoke/ui.ts) need the backends running and exit early otherwise. smoke/otel.ts also checks that an app's traces, logs and HTTP metrics arrive under its tenant, that another tenant sees none, that its requests are counted

exactly on the organization page and, with `SANDBOXO_RUNTIME_CGROUP` set (`auto` works in a user session; the smoke passes it as `runtime_cgroup_parent`), that its CPU time is plausible.

Zones

hh.pe runs as two programs plus obice routers. **sandboxod** is the control plane: it owns every record (Redis), takes uploads and git fetches into blob storage (S3), serves the APIs, the web UI and the SSH gateway, and serves the routers their routes. **zoned** runs the gVisor sandboxes of one availability zone: app instances, builds, generic sandboxes, template builds and btrfs volumes. **Routers** are obice instances (envoy driven by obice's own xDS server) that take the public traffic for apps and sandbox ports over HTTPS and send it straight to the zones. A zone talks to S3, to the telemetry backends and to sandboxod; it never touches Redis and takes no inbound connections: its own obice dials a reverse tunnel to every router, so zones can sit behind NAT.



Zones and regions

A zone has a name (a DNS label of at most 32 characters, for example `ord1`) and belongs to one region (three lowercase letters, `ord`). Ids keep the region (`sbx_ord_...`, `vol_ams_...`); records store the zone. A zone's region is fixed once it has registered: registering the same name with another region is rejected. sandboxod's `region` is the default region for new sandboxes and volumes.

Registration

zoned starts with a new random boot id and keeps only `slots/` and its run directory ephemeral: paused checkpoints, volumes, egress state and caches survive restarts. It posts its `ZoneInfo` (name, region, URL, boot id, protocol version, org, tenant mode, profiles, features, limits) to `<control>/zone/v1/register` with its token in `x-sandboxo-zone-token`, every 30 s, and every 2 s while that fails. The answer lists the routers the zone should tunnel to.

sandboxod accepts the registration when the token matches the zone's record (zones/<name> in Redis), the org and tenant mode match its own, the region matches the recorded one and the protocol version is supported. It then dials GET /v1/events on the zone through its router: an NDJSON stream that starts with a hello snapshot of everything the zone holds, then carries events, a ping with current capacity every 5 s and a load with the in-flight requests of every replica. A zone is **ready** while the stream is open and something arrived in the last 15 s; operations on a zone that is not ready answer 503 ZONE_UNAVAILABLE, and nothing new is placed there. sandboxod reconnects every second. Known zones are dialed again when sandboxod starts, before they register.

Tokens:

- zone_tokens in sandboxod's config, for example {"ord1": {"_file": "/run/keys/ord1"}}, writes those tokens into the zone records at startup (the NixOS module, dev.ts and the smoke harness use this);
- PUT /zones/<name> on the API port (with the API token) creates or rotates a zone's token and returns it once: {"name", "token"}.

zoned reads its own from token in its config. The zone's envoy adds the token to every request that arrives over a tunnel, so routers never hold zone tokens. Tunnels, route fetches and registration are not authenticated beyond that yet: keep routers, zones and sandboxod on a private network or put TLS in front. Registration and route fetches are served on sandboxod's API port, so that port must only be reachable from that network.

GET /zones lists zones with status (ready, disconnected, unregistered), region, url, cordoned, bootId, features, limits and the last reported capacity. POST /zones/<name>/cordon stops new placements in a zone; {"cordoned": false} as the body lifts it. Work pinned to a cordoned zone (its volumes, snapshots and paused sandboxes) still runs there.

Routers and tunnels

A router is an obice. Its envoy bootstrap (sandboxod router-bootstrap, or nix/router-bootstrap.nix in the NixOS modules) holds the tunnel listener that zones dial and, on the router next to sandboxod, an egress listener on a unix socket through which sandboxod reaches the zone API. The zone's envoy adds the zone token to everything that comes through its tunnels, so the socket is mode 0660 and only sandboxod's group may use it (/run/sandboxo-egress/egress.sock, group sandboxo-egress, in the NixOS module). Everything else is an obice remote source: the router long-polls GET /router/v1/source?name=<router>&tunnel=<host:port> on sandboxod's API port and gets a document in obice's source format:

- one zone-<name> tunnel cluster per zone, reaching that zone's tunnels;
- routes on obice's https listener, matched on the host (port ignored) in the order sandboxod's front door always had: running sandbox port labels (<port>-sbx-<region>-<id>.<domain>) and exposure labels, the e2b-sandbox-id header of running sandboxes, 404 sandbox not found for other sandbox labels and ids, then apps (<app>.<domain>, the root app on <domain>, previews <app>--<rev>.<domain>), 404 app is not deployed, 503 app is unavailable for an app without a zone in its region, and 404 unknown app for the rest of <domain> and *.<domain>. Other hosts are left to the virtual hosts of the router's own obice config.

Routes add x-sandboxo-route, the original URL and the client address. The router's obice config drops client headers starting with x-sandboxo- (http.strip_headers), and obice sets x-forwarded-for, x-forwarded-proto, x-forwarded-host and x-real-ip and answers with HSTS. sandboxod rebuilds the document on every promotion, placement change and sandbox change. Promotions, sandbox creation, resume and exposing a port only return once every router that polled in the last 5 s reports (Obice-Accepted) that its envoy applied

the new routes, so the next request finds them.

A poll registers the router: sandboxod keeps it in Redis (`routers/<name>`) and pushes the router list to the zones (`PUT /v1/routers`), so every zone dials every router. A router that stops polling stays in the list for 10 minutes, so a sandboxod restart does not make zones drop their tunnels.

Routers serve `<domain>` and `*.<domain>` with the certificate in their obice config (`certificates.sandboxo`), from obice's ACME or from files. In development, `process-compose up` runs the router `local` on port 8443 (`obice run --http-port`, without a plain http listener) with the `packages.dev-tls` certificate for `sandboxo.localhost`, and the dev config sets `domain` and `port` to match. A router elsewhere:

```
sandboxod router-bootstrap --name edge1 --tunnel '[::]:8005' \
  --log-dir /var/log/router > bootstrap.json
obice run --bootstrap bootstrap.json --config obice.json \
  --remote-source 'sandboxo=http://control.example.com:8001/router/v1/source?
name=edge1&tunnel=edge1.example.com:8005' \
  -- --concurrency 2
```

where `obice.json` sets `http.strip_headers: ["x-sandboxo-*"]` and the certificate. On NixOS, `services.sandboxo-router` sets this up in the host's `services.obice`, and `services.sandboxod` makes the host's obice the router `local` (`router.enable`).

Each zone runs its own obice as a child of zoned and serves it the router list as a remote source on loopback. It keeps 4 tunnels per router: envoy keeps tunnels per worker thread, so routers must not run more than 4 workers. The modules pass `--concurrency 2`, which also applies to everything else the host's obice serves. zoned itself only listens on loopback.

Placement

There is no replicated filesystem, so volumes, snapshots and paused checkpoints stay in the zone that created them, and so does everything that uses them:

- a sandbox mounting volumes runs in their zone; volumes from two zones in one sandbox are rejected with 400 `VOLUMES_IN_DIFFERENT_ZONES` (`volume 'a' (zone ord1) and volume 'b' (zone ams1) are in different zones`);
- a snapshot is taken in its volume's zone, a volume created from a snapshot lives in the snapshot's zone;
- a paused sandbox resumes in the zone that holds its checkpoint;
- if that zone is not ready the request fails with 503 `ZONE_UNAVAILABLE`; if an explicit region does not match it, with 400 `REGION_MISMATCH`.

Everything else is free to move:

- unpinned sandboxes and volumes go to a ready, uncordoned zone with the feature (`sandboxes`, `volumes`, egress policies need `egress`), in the requested region if one is given (400 `INVALID_REGION` for an unknown one), otherwise preferably in the default region and in any region when it has none. Sandboxes prefer the zone with the most free sandbox slots, then the most warm sandboxes; ties are broken at random. When a zone answers `LIMIT_REACHED`, the next one is tried;
- apps run as replicas, see Scaling;
- a build goes to the zone with the most free build slots; a template build to the zone running the fewest template builds. Images and logs are stored in S3, so a template built in one zone boots in every other.

Scaling

An app runs as one or more replicas, each an isolated runtime sandbox with its own id (6 characters, `[0-9a-z]`), the way Deno Deploy runs several instances of an application. Two settings in the app config (UI Configuration page, admin API) control them; like the rest of the config they apply to the next deploy:

`region`
: A region code (`ord`) or `global` (default). Replicas only run in zones of that region; with no ready zone there the app answers 503. `deno deploy create --region` only accepts `us`, `eu` and `global`, so from the CLI only `global` is taken over.

`maxReplicas`
: 1 (default) to 16.

sandboxod decides how many replicas each zone runs; the zones pick the replica for every request:

- Every deployed app is assigned a home zone, preferably in the default region (sandboxod's `region`), then the zone that built the revision, then where most warm runtime sandboxes wait. sandboxod sends the zone the instance spec and its replica count (`PUT /v1/instances/<key>`); the zone starts a replica on the first request.
- Routers spread an app's requests over its zones by their replica counts. Inside a zone, a request goes to the replica with the fewest in-flight requests; when the zone runs fewer replicas than it was given, it starts the missing ones.
- Zones report the in-flight requests of every replica, including requests waiting for a replica to start and cron runs, every second. When they exceed `scaling.target_inflight` (20) per replica, sandboxod adds a replica (at most one every 2 s, up to `maxReplicas`) in the region with the fewest replicas of the app, then the zone with the fewest, so a `global` app spreads over regions and zones.
- When the smoothed load would fit one replica fewer at half the target for `scaling.scale_down_ms` (60 s), the zone with the most replicas gets one less; it drains its least loaded replica (up to zoned's `drain_ms`) and stops it. A lower `maxReplicas` or a changed region shrinks the same way.
- A zone that gets a request for an app it holds no replicas for answers 409 `unknown-instance` before reading the body, and the router retries in another zone. A zone that does not know an app at all, after a restart or for a preview, asks sandboxod for the spec (`GET /zone/v1/instances/<key>`).
- Idle replicas stop after `idle_ms` on their zone; an app without traffic runs nothing, and the next request starts a replica in its zone.
- A zone that stops responding loses its replicas, and they move to another allowed zone. After a sandboxod restart the zones' `hello` snapshots bring the replica counts back.
- A preview runs in the zone that first asked for it.

Every app log line carries the replica as structured metadata (`replica`), and the app's own OpenTelemetry data carries `service.instance.id`, so logs show a second, round identicon for the replica next to the revision's. `GET /apps/<app>` and the app overview list the replicas the zones report with their zone, in-flight requests and state.

Failures

Zone down

Its sandboxes answer 503, listing them still works. Apps move to other zones; routers retry requests that cannot reach a zone in another one. After 60 s without the zone, its builds go back to the queue (at most two attempts per revision) and its template builds fail; if the zone comes back still running them, they are cancelled there.

Zone restart (new boot id)

Sandboxes that were running there are recorded as stopped (`zone restarted`). Paused checkpoints and volumes are kept and resume there. Builds lost by the restart are queued again, template builds fail.

Router down

Clients of that router lose their connections; zones keep their tunnels to the others. A router that comes back fetches its routes again and the zones dial it again.

sandboxod restart

Zones keep their sandboxes running and keep enforcing deadlines (sandboxod sends absolute times, so clocks must be synchronized). Deno session sandboxes, those without a timeout, end with their owning connection and are stopped. Routers keep serving the routes they have and fetch new ones once sandboxod is back.

Reconciliation

On every `hello` sandboxod compares the snapshot with its records: tombstones of sandboxes that ended while it was away are applied, running records the zone no longer has become stopped (or paused, if it holds their checkpoint), sandboxes the zone runs without a record are killed (after a 60 s grace for creations in flight), paused records whose checkpoint is gone become stopped (`checkpoint lost`), and orphan checkpoints are discarded. Finished builds, cron runs and tombstones are kept by the zone until sandboxod acknowledges them, and the first result applied wins. Cron runs the zone no longer knows fail; runs still `running` after a sandboxod restart fail if no zone claims them within 30 s.

Zone API

All requests carry `x-sandboxo-zone-token`; ids are minted by sandboxod and every operation is idempotent by id. Errors are `{"error": {"code", "message"}}`.

Endpoint	Meaning
`GET /v1/info`	zone info and capacity
`GET /v1/events`	NDJSON: `hello`, `ping`, `load`, `sandbox.ended`, `build.finished`, `templateBuild.log`, `templateBuild.finished`, `cron.finished`
`POST /v1/ack`	`{"sandboxes"?, "builds"?, "templateBuilds"?, "cronRuns"?}`: forget delivered results
`PUT /v1/routers`	`[{"name", "address"}]`: routers to tunnel to
`PUT, DELETE /v1/instances/<key>`	`{"spec", "replicas"}`: app instance spec (image, config, env, cron names) and how many replicas the zone runs, 0 drains them; stop all replicas with `?drain=1`
`PUT /v1/cron-runs/<id>`	run a cron on one of the app's replicas (started if needed); 202, the result arrives as `cron.finished`; 409 `UNKNOWN_INSTANCE` when the zone runs no replicas of the app
`PUT, DELETE /v1/builds/<id>`	start or cancel a build

`PUT /v1/template-builds/<id>`, `GET .../log?offset=`, `DELETE`	template builds
`PUT /v1/sandboxes/<id>`, `DELETE`, `POST .../pause`, `POST .../resume`, `PUT .../deadline`	generic sandbox lifecycle
`GET /v1/sandboxes/<id>/agent`	WebSocket, NDJSON agent protocol
`GET /v1/sandboxes/<id>/tcp/<port>`	WebSocket byte tunnel to a port, text frame `eof` half-closes
`PUT`, `DELETE /v1/volumes/<id>`, `GET .../tar`, `PUT`, `DELETE /v1/snapshots/<id>`	volumes and snapshots

Proxied traffic (app requests, sandbox ports, envd) is any request with `x-sandboxo-route`: `kind=instance&key=<app>&rev=<rev>`, `kind=preview&app=<app>` (the revision comes from the host), or `kind=sandbox&id=<id>&port=<port>` where the port is a number, `host` (from the port label) or `header` (from `e2b-sandbox-port`, 49983 without it). The original URL is in `x-sandboxo-url` and the client address in `x-sandboxo-client`; zoned strips every `x-sandboxo-*` header before forwarding.

Running

Both programs and `dev.ts` take a JSON config file, see Configuration. `SANDBOXO_TOKEN=secret deno task dev` starts sandboxod and zones in one process for development (dev.ts), and `process-compose up` runs the router. As separate processes, with `ord1.token` holding the zone token:

```
{
  "token": { "_env": "SANDBOXO_TOKEN" },
  "signing_secret": { "_env": "SANDBOXO_SIGNING_SECRET" },
  "zone_tokens": { "ord1": { "_file": "ord1.token" } },
  "blobs": { "backend": "fs" }
}
```

```
{
  "name": "ord1",
  "token": { "_file": "ord1.token" },
  "state": "state/zones/ord1",
  "blobs": { "backend": "fs", "dir": "state/blobs" }
}
```

```
echo dev-zone-token > ord1.token
SANDBOXO_TOKEN=secret deno run -A sandboxod/main.ts --config sandboxod.json &
deno run -A zoned/main.ts --config zoned.json
```

The flake builds `packages.sandboxod` (no gVisor, profiles or passt in its closure, checked by `checks.closures`), `packages.zoned`, `packages.sandboxo` (the all-in-one launcher) and `packages.dev-tls` (the development CA and certificate for `sandboxo.localhost`). The NixOS module imports obice's and offers `services.sandboxo` for one host (sandboxod, Redis, Garage and a zone `local1` with a generated token), or the parts:

```
# control host
services.sandboxod = {
  enable = true;
  domain = "deploy.example.com";
  tokenFile = "/run/keys/sandboxo-token";
  apiHost = "10.0.0.1";
  tunnel.address = "10.0.0.1:8005";
  router.enableACME = true;
  garage.s3Address = "10.0.0.1:3900";
  zones.ord1.tokenFile = "/run/keys/zone-ord1";
};
```

```
services.obice.acme.nameserver = "10.0.0.53:53";

# zone host, behind NAT is fine
services.zoned = {
  enable = true;
  name = "ord1";
  region = "ord";
  tokenFile = "/run/keys/zone-ord1";
  controlUrl = "http://10.0.0.1:8001";
  settings.blobs = {
    backend = "s3";
    endpoint = "http://10.0.0.1:3900";
    access_key_id._env = "SANDBOXD_S3_ACCESS_KEY_ID";
    secret_access_key._env = "SANDBOXD_S3_SECRET_ACCESS_KEY";
  };
  environmentFiles = [ "/run/keys/zone-s3.env" ];
};

# another router
services.sandboxd-router = {
  enable = true;
  name = "edge1";
  domain = "deploy.example.com";
  tunnel.address = "edge1.example.com:8005";
  controlAddress = "10.0.0.1:8001";
  enableACME = true;
  openFirewall = true;
};
services.obice.acme.nameserver = "10.0.0.53:53";
```

The zone's S3 key needs read and write access to the bucket. `/var/lib/zoned` should be on btrfs for volumes.

Tests

`sandboxd/placement_test.ts` covers the placement rules, `scaler_test.ts` the replica counts, `routes_test.ts` the route order and that `obice check` takes the source, and `router_test.ts` runs obice with the router bootstrap and checks that envoy accepts the generated routes and serves them over HTTPS; `common/obice_test.ts` covers the route source server. `smoke/multizone.ts` runs sandboxd and zones `ord1` and `ams1` in one process and checks app failover, volume placement, the mixed-zone rejection, snapshots and forks, pausing and resuming across a zone restart, a sandboxd restart and a template built in one zone used in the other. `smoke/routers.ts` adds a standalone router and checks that both zones reach it and that the routers serve without each other. `smoke/tunnel.ts` measures the path through the tunnel and a tunnel loss. `smoke/cron.ts` covers cron discovery, scheduled and manual runs, retries, timeouts, a sandboxd restart and a promotion during a run. `checks.module` boots the single-host module; `checks.multizone` a control VM, two zone VMs that take no inbound connections and a router VM.

Configuration

sandboxod, zoned and the all-in-one `dev.ts` each read one JSON file given with `--config <file>`. Keys are snake_case and unknown keys are errors. Every key not marked required has a default, so a minimal sandboxod config is:

```
{
  "token": { "_env": "SANDBOXO_TOKEN" },
  "signing_secret": { "_file": "/run/keys/sandboxo-signing" },
  "blobs": { "backend": "fs" }
}
```

Errors name keys in camelCase (`build.timeoutMs` for `build.timeout_ms`). Relative paths resolve against the working directory.

Secrets

Keys marked *secret* take a plain string or a placeholder, the same ones obice uses:

`{ "_env": "NAME" }`
: the value of the environment variable `NAME`.

`{ "_file": "/path" }`
: the content of the file, without trailing newlines.

A missing variable or an unreadable file fails startup. Secret keys: sandboxod's `token`, `signing_secret`, the values of `zone_tokens` and `kv.url`, zoned's `token`, and in both `blobs.access_key_id`, `blobs.secret_access_key` and the values of every `headers` in `telemetry`.

Environment

Only the Nix profiles come from the environment, and binaries from `PATH`. The `zoned` and `sandboxo` packages set both in their wrappers; the dev shell has them for `deno run`. zoned and `dev.ts` fail at startup when one is missing.

Variable	Meaning
`SANDBOXO_PROFILE`	required, Nix profile of build and runtime sandboxes
`SANDBOXO_SANDBOX_PROFILE`	required, Nix profile of generic sandboxes (e2b, Deno Sandbox)

zoned needs on `PATH`: `sbx-runsc` (sbx-gvisor), `pasta` (passt), `mkfs.erofs` (erofs-utils), `iron-proxy` for sandbox egress policies, `obice` for the zone's tunnels (its envoy comes from obice's own `PATH`), `btrfs`, `unshare` and `nsenter`.

sandboxod

Key	Default	Meaning
`token`	required, secret	admin token for the API, the UI and `/zones`, characters `[A-Za-z0-9._~-]` only; with `oidc` set it stays a break-glass superadmin

signing_secret	required, secret	signs UI query links and upload links, see Observability
blobs	required	blob storage, see State directory
kv	Redis on 127.0.0.1:6379	record storage, see State directory
telemetry	none	Loki, Mimir, Tempo and OTLP, see Observability
state	state	state directory, see State directory
org	local	deployment id ([A-Za-z0-9_-]): zones must use the same, it names the shared telemetry tenant and cannot be used as an app name. Without oidc it is also the one organization
domain	localhost	apps are served on <app>.<domain>
port	443	https port of the routers, used in app and sandbox URLs
api_hostname, api_port	127.0.0.1, 8081	API and UI, also where zones register and routers fetch their routes
egress	/run/sandboxo-egress/egress.sock	unix socket of the egress listener of the router on this host, sandboxod reaches the zones through it
ui	production	production (needs ui/dist), development (bundles and hot-reloads the UI) or off
region	ord	default region for new sandboxes, volumes and apps
zone_tokens	{}	zone name to secret, written into the zone records at startup, see Zones
sandbox.max_lifetime_ms	1800000	longest lifetime of a generic sandbox
sandbox.template_build_timeout_ms	3600000	e2b template build timeout
sandbox.ssh	{}	SSH gateway, null turns it off
sandbox.ssh.hostname, sandbox.ssh.port	::, 2222	SSH gateway listen address
sandbox.ssh.public_host	ssh.<domain>	host the SSH commands name
scaling.target_inflight	20	in-flight requests per replica above which an app scales up, see Scaling
scaling.scale_down_ms	60000	how long the load has to stay low before a replica is removed
build.timeout_ms	600000	build timeout of apps that set none
build.max_timeout_ms	1800000	largest build timeout an app may set, at least build.timeout_ms
build.memory_mb	1024	build memory limit of apps that set none, null for no limit
build.max_memory_mb	4096	largest build memory limit an app may set
cron.max	10	cron jobs a revision may register, more fail the build
cron.timeout_ms	900000	a cron run that takes longer fails and its instance is replaced

cron.history	100	runs kept per cron job
git_cache	<state>/git	bare repository cache for git deploys
git_fetch_timeout_ms	300000	git fetch timeout
source_max_files, source_max_bytes	200000, 2147483648	limits of one upload or git tree
gc_interval_ms, gc_grace_ms	3600000, 3600000	blob garbage collection interval (0 disables it) and the age a blob needs before it is deleted; expired login sessions are swept on the same interval
oidc	none	multi-tenant mode with users and organizations from an OpenID provider (pykrete), see OpenID Connect

OpenID Connect

With oidc set, users log in through the provider and every organization they belong to there is a tenant: its members share its apps, sandboxes, volumes, snapshots and templates. API keys are minted, listed and revoked through the provider's backend API (a Clerk-compatible subset) and checked with RFC 7662 introspection. telemetry.tenant must be app, and redirect_base must not be under domain (apps would receive the session cookie).

Key	Default	Meaning
oidc.issuer	required	issuer URL; discovery must report the same issuer
oidc.client_id, oidc.client_secret	required, secret	confidential client; the secret is also the backend API key. The client needs "Backend API" (set by the client owner) and "All organizations" (first-party, set by a pykrete admin), otherwise every organization has to enable the client and only users who authorized it are visible
oidc.token_auth_method	client_secret_post	or client_secret_basic
oidc.redirect_base	required	public origin of the UI; the redirect URI is <redirect_base>/login/callback and the post-logout URI <redirect_base>/login
oidc.api	<issuer>api/v1/	backend API base URL
oidc.legacy_org	org	provider organization id that owns records created before multi-tenancy (and in single-org mode); it cannot change once set
oidc.scopes	openid profile email organizations	requested scopes
oidc.session_ttl_ms	604800000	lifetime of a UI login
oidc.introspect_ttl_ms, oidc.membership_ttl_ms	60000, 30000	how long API key checks and organization memberships are cached, so how late a revoked key or a removed member is noticed
oidc.admin_login	false	also offer the admin token form on /login

`oidc.org_url`, `oidc.account_url`	none	links to the provider's organization page (`{slug}`, `{id}` are replaced) and account page, for members, invitations and API keys

zoned

Key	Default	Meaning
`name`	required	zone name, a DNS label of at most 32 characters
`token`	required, secret	zone token, see Zones
`blobs`	required	blob storage, the same as sandboxod's, see State directory
`telemetry`	none	where logs, traces, metrics and usage are pushed, see Observability
`region`	`ord`	region of the zone
`org`	`local`	must match sandboxod's
`hostname`, `port`	`127.0.0.1`, `8002`	zone API listen address, only the zone's envoy connects
`url`	the listen address	zone URL reported to sandboxod
`control`	`http://127.0.0.1:8001`	sandboxod API listener
`state`	`state/zone`	zone state directory, see State directory
`gc_root_dir`	`<state>/nix`	GC roots and private store of the profiles
`pool.none`, `pool.sandboxed`	`20`, `4`	warm runtime sandboxes per network flavor
`pool.build`	`2`	warm build sandboxes
`build.concurrency`	`2`	builds the zone runs at once
`build.memory_mb`	`1024`	build memory limit of apps that set none, `null` for no limit
`build.memory_overhead_mb`	`1024`	added to the app's limit for the build sandbox's cgroup
`build.cpus`	none	`cpu.max` of build cgroups, in cores
`build.cgroup_parent`, `runtime_cgroup_parent`, `sandbox.cgroup_parent`	`auto`	cgroup parents of build, runtime and generic sandboxes: a path, `auto` or `null` (no cgroups). `auto` uses the cgroup of zoned's unit when it is delegated (`Delegate=yes`, as the NixOS module sets), otherwise one `systemd-run --user --scope` unit per sandbox under `app.slice`. Without a systemd user manager `auto` fails: the zone logs a warning and runs without cgroups. Limits and CPU usage need them
`start_timeout_ms`	`10000`	how long a replica may take to start
`idle_ms`	`300000`	idle replicas stop after this long without requests
`drain_ms`	`30000`	how long a replica that is scaled down or replaced finishes its requests
`sandbox.pool_size`	`2`	warm generic sandboxes

sandbox.max	10	generic sandboxes the zone runs at once

dev.ts

dev.ts starts sandboxod and its zones in one process. Its config has three keys:

sandboxod
: a sandboxod config.

zoned
: zoned keys shared by every zone, except the ones dev.ts sets itself: name, region, token, org, hostname, port, url, control, state, blobs and telemetry.

zones
: [{"name", "region"?}], default <region>1 and <region>2 of sandboxod's region.

Every zone gets a random token, a free loopback port, sandboxod's org, telemetry and blob storage (fs blobs in <state>/blobs), its state in <state>/zones/<name>/ and gc_root_dir <state>/nix unless set. The dev shell exports SANDBOXO_DEV_CONFIG (and SANDBOXO_DEV_UI_CONFIG with ui: development), generated from flake.nix, which points at the process-compose backends and reads the token from SANDBOXO_TOKEN:

```
SANDBOXO_TOKEN=secret deno task dev
```

NixOS

services.sandboxod.settings and services.zoned.settings are written to the config file. The modules set the keys their options cover (domain, name, ports, state directory, ...), the tokens as _file placeholders on systemd credentials, and for the local Redis and Garage of services.sandboxod kv and blobs with _env placeholders. Secrets for other placeholders go in environmentFiles:

```
services.zoned = {
  enable = true;
  name = "ord1";
  tokenFile = "/run/keys/zone-ord1";
  controlUrl = "http://10.0.0.1:8001";
  settings = {
    pool.none = 4;
    blobs = {
      backend = "s3";
      endpoint = "http://10.0.0.1:3900";
      access_key_id._env = "SANDBOXO_S3_ACCESS_KEY_ID";
      secret_access_key._env = "SANDBOXO_S3_SECRET_ACCESS_KEY";
    };
  };
  environmentFiles = [ "/run/keys/zone-s3.env" ];
};
```

02

Deno Deploy

About Deno Deploy 33

Getting started 35

Migrating from Deploy Classic to Deno Deploy 39

About Deno Deploy

What is Deno Deploy?

Deno Deploy is a serverless platform for running JavaScript and TypeScript applications in the cloud (or self-hosted on your own infrastructure). It provides a management plane for deploying and running applications with the built-in CI or through integrations such as GitHub actions.

Deno Deploy comes with an easy to use dashboard at console.deno.com. Here, you can create and host new apps, create and manage Deno Deploy organizations and manage and view your databases and app telemetry.

Comparison to Deploy Classic

Deno Deploy is a complete rework of Deploy Classic. It has a new dashboard, and a new execution environment that uses Deno 2.0 and is much more powerful than Deploy Classic. The below table compares the two versions of Deno Deploy.

Feature	Deno Deploy	Deploy Classic
Web interface	console.deno.com	dash.deno.com
Dark mode	✓ Supported	✗ Not supported
Builds	✓ Fully integrated	✓ Runs in GitHub Actions, no live streamed logs in the dashboard, caching requires manual setup, changing config requires editing YAML
Can run Deno apps	✓ Full support	✓ Limited (no FFI, subprocesses, write permission)
Can run Node apps	✓ Full support	✓ Limited (no FFI, native addons, subprocesses, write permission, and degraded NPM compatibility)
Can run Next.js/Astro/SvelteKit	✓ First-class support	✓ Framework dependent, requires manual setup
First class static sites	✓ Supported	✗ Not supported
Environment Variables	✓ Different dev/prod env vars	✓ One set of env vars for all deployments
CDN caching	✓ Supported	✗ Not supported
Web Cache API	✓ Supported	✓ Supported
Databases	✓ Supported	✓ Deno KV
Queues	✗ Not supported	✓ Supported
Cron	✓ Supported	✓ Supported
Deploy from GitHub	✓ Supported	✓ Supported
Deploy from CLI	✓ Supported	✓ Supported

Instant Rollback	Supported	Supported
Logs	Supported	Supported
Tracing	Supported	✕ Not supported
Metrics	Supported	✕ Not supported
OpenTelemetry export	Work in progress	✕ Not supported
Regions	2	6
Self hostable regions	Supported	✕ Not supported

> ⚠ **WARNING**
> Deno Deploy Classic (dash.deno.com) and the subhosting v1 API will be shut down on July 20, 2026. See the migration guide for details.

How to access Deno Deploy

To begin using Deno Deploy:

1. Visit console.deno.com to access the new dashboard
2. Create a new Deno Deploy organization
3. Create your first application within this organization
4. Deploy from your GitHub repository or directly from the dashboard

For detailed configuration instructions and framework-specific guides, please refer to our reference documentation.

Getting started

Create an organization

Deno Deploy Classic will be shut down on July 20, 2026. If you are migrating from Deploy Classic, see the migration guide for details on what has changed.

To get started with Deno Deploy:

1. Visit console.deno.com
2. Create an organization:

[The Deno Deploy organization creation screen.]

Note that you cannot create an organization with the same slug as any existing project in Deploy Classic. Organization names and slugs cannot be changed after creation.

Create an app

After creating an organization, you'll be directed to the organization apps page, which shows all your applications and provides access to organization settings and custom domains.

To create an app, press the `+ New App` button:

[Screenshot of deploy app creation screen]

An application is a single deployed web service with one build configuration, build history, environment variables, attached custom domains, a linked GitHub repository, etc.

> **NOTE**
> You can also create and configure apps from the command line with `deno deploy create`. See the Apps reference for details.

Select a repo

1. Choose the GitHub repository for your application:

[Screenshot of deploy org selection screen]

If your repository doesn't appear, use the `Add another GitHub account` or `Configure GitHub App permissions` buttons to grant the Deno Deploy GitHub app access to your repositories.

> Mono-repos (repositories where the application lives in a subdirectory) are not yet supported.

Configure your app

Deno Deploy automatically attempts to detect your application type and configure an appropriate build setup. You can see the detected configuration in the `App Config` box:

[Screenshot of Deploy application configuration screen]

To modify this configuration, click `Edit build config`.

[Screenshot of Deploy build configuration screen]

Configure your build

In the build config drawer, you can customize:

Framework preset

Select your framework or choose `No Preset` if using a custom setup.

Install command

Command for installing dependencies (e.g., `npm install`, `deno install`). This can be empty for Deno applications without a `package.json`.

Build command

Command to compile/bundle your application (e.g., `next build`, `deno task build`). Leave empty if your application doesn't require building.

Runtime configuration

For most frameworks there are no options to configure here, as Deno Deploy will figure out the ideal runtime configuration for the app based on the framework preset. When a framework is not configured, you can choose here whether the app is a `Dynamic` app that needs to execute code server side for every request, such as an API server, server-side rendered application, etc., or a `Static` app that consists only of a set of static files that need to be hosted.

Dynamic Entrypoint

The JavaScript or TypeScript file that should be executed to start the application. This is the file path that you would pass locally to `deno run` or `node` to start the app. The path has to be relative to the working directory.

Dynamic arguments

Additional command line arguments to pass to the app on startup, after the entrypoint. These are arguments that are passed to the application not to Deno itself.

Static Directory

The directory in the working directory that contains the static files to be served. For example,`dist`,`_site`, or `.output`.

Single Page App mode

Whether the application is a single page app that should have the root `index.html` served for any paths that do not exist as files in the static directory, instead of a 404 page.

Closing the drawer saves the settings.

Environment variables

To add environment variables:

1. Click `Add/Edit environment variables`
2. Click `+ Add variable` in the drawer
3. Enter the name and value
4. Choose whether it's a plain text variable or secret
5. Select the contexts where it should be available:
   - **Production**: For requests to production domains
   - **Development**: For requests to preview/branch domains
6. Click `Save` to apply your changes

[Screenshot of the Deploy env variables config screen]

You can re-open the drawer to edit / remove environment variables you have added. You can also edit the app name on this page, and select which region(s) the application should be served from.

Build and deploy your app

1. Click `Create App` to create the application and start the first build
2. Watch the build progress through the live logs:

[Screenshot of app build logs]

On the build page you can see live streaming build logs split into multiple sections:

- **Prepare:** cloning the GitHub repository and restoring build cache
- **Install:** executing the install command, and any framework specific pre-install setup
- **Build:** executing the build command, any framework specific pre- and post-build setup, and preparing the build artifact for deployment
- **Warm up:** sending a request to the preview URL of the deployment to ensure it starts up correctly. The logs shown in the Warm up section are Runtime logs, not build logs.
- **Route:** Deno Deploy is rolling out the new version of this build into all global regions.

In the top left of this build is a button to cancel the build. For failed builds, there is also a button to restart the build.

For completed builds, the top right shows the preview URL. Further down all timelines that this build is deployed to are shown, such as `Production`, or `Git Branch` timelines.

You can also see how the build was triggered on this page. This can either be `manual action`, for builds triggered through the UI, or `GitHub repo` for builds triggered through the GitHub integration.

You can view the application through either the preview URL, or any of the other URLs shown in the timelines list.

Monitor your application

After deploying, you can view telemetry about your application in the form of the logs and traces available in our observability panels. You can visit these pages by clicking the respective buttons in the left sidebar.

Logs

View application logs with filtering options for context, revision, and text content:

[Screenshot of the Logs page]

The logs page shows all recent logs in the project. By default logs from all contexts (production and development) are shown, but using the filter button and search bar at the top, the shown logs can be restricted. For example, to filter to only production logs, add `context:production` to the search bar. To only show logs from a certain revision, use `revision:<id>` etc.

You can also use full text search in the search bar. The full text search will filter down the log entries to only those containing the text written, case-insensitively.

By default logs from the last hour are shown. The time picker in the top right can be used to adjust the time frame that logs are shown for. The time zone of the timestamps shown is the time zone set in the time picker.

The "view trace" button on the right of a log line shows up if a log line is correlated with a trace. This happens when a log line occurs within an active trace. Clicking this button will open the respective trace as an overlay.

Traces

View request traces with detailed timing information:

[Screenshot of the Traces page]

The traces page shows all recent traces in the project. By default traces from all contexts (production and development) are shown, but using the filter button and search bar at the top, the shown traces can be restricted. For example, to filter to only production traces, add `context:production` to the search bar. To only show traces from a certain revision, use `revision:<id>` etc.

All traces that contain an incoming HTTP request are shown in the list. The text shown for each trace is the path of the request, and the duration of the trace in milliseconds.

Clicking on a trace will open the trace view, which shows the full trace including all spans and logs that are part of the trace.

[Screenshot of the Trace view]

For each span in the trace you can see the duration of the span, the name of the span, the start and end time, and the recorded attributes. By clicking on a span in the timeline, the details of that span will be shown in the summary panel at the bottom.

The logs that are emitted as part of a given span are shown in the logs tab at the bottom. Changing the selected span will update which logs are shown in this panel.

Migrating from Deploy Classic to Deno Deploy

Deno Deploy Classic (dash.deno.com) and the subhosting v1 API (apidocs.deno.com) will be shut down on **July 20, 2026**. This guide covers migrating your applications and API integrations to the new Deno Deploy platform. If you are migrating from the subhosting v1 API, see the subhosting API migration guide.

Create a new organization

The new Deno Deploy uses a separate account system. To get started:

1. Go to console.deno.com and sign in
2. Create a new organization (if you don't have one already) — this is required before you can deploy any applications
3. Invite team members to your organization as needed

Your Deploy Classic projects at dash.deno.com are not automatically transferred. You will need to create new apps and redeploy.

Create and deploy your app

In the new Deno Deploy dashboard, create a new app within your organization. You can deploy via GitHub integration or the CLI.

GitHub integration

Connect your GitHub repository from the app settings in the dashboard. The new Deploy has fully integrated builds — no more GitHub Actions YAML configuration. Build logs stream directly in the dashboard.

CLI deployment

The `deployctl` CLI is also being sunset. Use the `deno deploy` subcommand instead:

```
deno deploy
```

See the getting started guide for detailed setup instructions.

Environment variables

Deploy Classic used a single set of environment variables for all deployments. The new Deploy supports separate **production**, **development**, and **build** timelines.

Review your environment variables and set them up in the new dashboard under your app's settings. See environment variables and contexts for details.

Custom domains

To migrate a custom domain:

1. Add the domain to your new Deploy app in the dashboard
2. Configure the `_acme-challenge` CNAME record for TLS certificate provisioning

3. Update your DNS records (CNAME or ANAME) to point to the new Deploy
4. Allow up to 48 hours for DNS propagation before removing the domain from Deploy Classic

See the custom domain migration tutorial for a step-by-step walkthrough.

HTTP server API

Deploy Classic supported the `serve()` function from the `deno.land/std` HTTP module (e.g. `https://deno.land/std@0.170.0/http/server.ts`). The new Deploy does **not** support this legacy `serve()` function.

Update your code to use the built-in `Deno.serve()` API instead:

```
- import { serve } from "https://deno.land/std@0.170.0/http/server.ts";
-
- serve(() => new Response("hello, world"));
+ Deno.serve(() => new Response("hello, world"));
```

`Deno.serve()` has been the recommended HTTP server API since Deno 1.35 and is more performant. If your application uses the old standard library `serve()`, update it before migrating to the new Deploy.

If you deploy code that uses the old `serve()` function to the new Deploy, it will fail with a **timeout error during the warmup phase** of deployment. If you encounter this error, check whether your entrypoint or any of your dependencies is using the standard library `serve()`. Some older versions of third-party libraries may internally depend on the legacy `serve()` function — in that case, upgrade the library to a version that uses `Deno.serve()` instead.

Cron jobs

The `Deno.cron()` API works the same way on the new Deploy. Your existing cron job code should work without modification. See the cron reference for details.

Queues

Deno queues (`Deno.Kv.enqueue()` / `Deno.Kv.listenQueue()`) are **not supported** on the new Deno Deploy. If your application relies on queues, you will need to adopt an alternative approach — for example, an external message queue service or a database-backed job queue.

KV database

Deno KV is available on the new Deploy, but your existing KV data is not automatically migrated. Contact support@deno.com for assistance migrating your KV database.

See Deno KV on Deploy for information about how KV works on the new platform, including per-timeline database isolation.

Subhosting API migration

The subhosting v1 API at `apidocs.deno.com` will be shut down alongside Deploy Classic on July 20, 2026. Migrate your integrations to the v2 API.

The v2 API has significant architectural changes — projects become apps, deployments become revisions, and each app represents a single deployable service. See the subhosting API migration guide for detailed

endpoint mappings, request/response changes, and new features like labels and layers.

Official SDKs for the v2 API:

- **TypeScript/JavaScript**: @deno/sandbox
- **Python**: deno-sandbox

Regions

Deploy Classic serves from 6 regions. The new Deploy currently has 2 regions, US (`us`) and Europe (`eu`), with the ability to self-host additional regions on your own infrastructure. Use `global` to serve from both regions. If your application is latency-sensitive and depends on specific regions, please plan accordingly. See the `deno deploy` CLI reference for the available region values.

What's new

The new Deploy includes several features not available in Deploy Classic:

- **Full Deno 2.0 runtime** — FFI, subprocesses, file system write access, and improved npm compatibility
- **Integrated builds** — build steps run on Deno Deploy with live-streamed logs, no GitHub Actions YAML required
- **First-class framework support** — Next.js, Astro, SvelteKit, Fresh, and more work out of the box
- **CDN caching** — built-in edge caching with `Cache-Control` headers and programmatic cache invalidation
- **Observability** — logs, traces, and metrics in the dashboard
- **Static site support** — deploy static sites directly
- **Separate dev/prod environments** — different environment variables and databases per context

If you have questions or need help, contact support@deno.com.

03

Reference

Accounts 43
Organizations 44
Applications 46
Builds 50
Environment Variables and Contexts 55
Timelines 58
Tunnel 60
Observability 62
Domains 65
Cron 69
Deno KV 73
Databases 75
Cloud Connections 84
OIDC 100
Runtime 103
Frameworks 106
CDN and caching 109
Playgrounds 116
Deploy Button 118

Accounts

Deno Deploy supports logging in with GitHub.

Your primary contact email address and name are synced from GitHub. Both your username and email address update on every sign in. After changing your email, login, or name on GitHub, sign in again to see these changes reflected in the Deno Deploy dashboard.

Deno also supports logging in with Google accounts.

Users authenticating with Deno Deploy using a Google account will also need to provide GitHub credentials when creating new applications in order to access GitHub repositories for deployments.

Organizations

Organizations are groups of users that collectively own apps and domains. When signing up for Deno Deploy, each user can either create an organization or join an existing organization through invitation.

All users must belong to an organization to use Deno Deploy, as all resources are owned at the organization level.

Organizations have both a name and a slug. The name is visible only to organization members and appears in the organization dropdown in both Deno Deploy and Deploy Classic. The slug forms part of the default domain for all applications in the organization.

Every organization has a default domain used for production, git branch, and preview URLs for projects in that organization. For example, an organization with the slug `acme-inc` would have a default domain of `acme-inc.deno.net`.

Organizations can have multiple members. Currently, all members have owner permissions for the organization, which means they can invite other members, create and delete apps, and manage domains.

Create an organization

Organizations in Deno Deploy are created when you sign up for a Deno Deploy account.

If you do not yet have a Deno Deploy account, you can create one by visiting the Deno Deploy dashboard and signing in with your GitHub account. You will be prompted to create an organization as part of the sign-up process.

> **NOTE**
> Organization slugs must be unique across all Deno Deploy organizations and cannot match any existing project name in Deno Deploy Classic.

Update the organization name

The organization name can be updated from the organization settings page in the Deno Deploy dashboard. The organization name is only cosmetic, and changing it does not affect any apps or domains. It is the display name of your organization, on the dashboard and in invitation emails.

Update the organization slug

The organization slug can be updated from the organization settings page in the Deno Deploy dashboard. Changing the organization slug will update the default domain for all apps in the organization. For example, changing the slug from `acme-inc` to `acme-corp` will change the default domain from `acme-inc.deno.net` to `acme-corp.deno.net`.

> ⚠ **WARNING**
>
> Changing the organization slug will immediately affect all apps in the organization that use the default domain. Any existing URLs using the old domain will stop working. You will need to update any bookmarks or links that use the old domain.
>
> After changing the organization slug, it can take multiple minutes for the new default domain to start working, due to TLS certificate provisioning. In this time, apps will be unreachable via the new default domain.
>
> Custom domains are not affected by changing the organization slug.

This will also affect the URLs of resources inside of the organization on the Deno Deploy dashboard. For example, the URL for an app will change from `https://console.deno.com/acme-inc/my-app` to `https://console.deno.com/acme-corp/my-app`.

Deleting an organization

Organizations cannot currently be deleted from the dashboard. Please contact Deno support if you need to delete an organization.

Inviting users to an organization

To invite a user:

1. Go to the organization settings page and click "+ Invite User"
2. Enter the user's GitHub account username (e.g., `ry`)
3. Optionally enter an email address to send the invitation to
4. Click "Invite"

If you don't specify an email address, we'll attempt to send the invitation to the email in the user's public GitHub profile or another email we may have on record.

After inviting a user, they will receive an email with an invite link (if we have their email address). They must click this link and accept the invitation to join the organization. You can also directly share the personalized invite link displayed in the members table after inviting a user.

You can cancel an invitation before it's accepted by clicking the delete button next to the invited user in the members table and confirming by clicking "Save". This invalidates the previously sent invitation link.

Removing users from an organization

To remove a member from the organization, find the user in the members table in the organization settings, click the remove button, and confirm by clicking "Delete". "Delete".

Applications

Applications are web services that serve traffic within an organization. Each application contains a history of revisions (previous versions), typically corresponding to Git commits when using the GitHub integration.

Applications are identified by a slug, which must be unique within the organization and is used in default domain names.

Creating an application

Applications can be created from the Deno Deploy dashboard or from the command line using the `deno deploy` CLI.

Using the dashboard

To create an application from the web dashboard:

1. Click the "+ New App" button on the Application overview page
2. Select the GitHub repository to deploy from
3. Configure the app slug (name)
4. Set up build configuration
5. Add any required environment variables

The build configuration determines how the application is built during the deployment process. Builds are automatically triggered on each push to the linked repository or when manually clicking "Deploy Default Branch". For detailed build configuration information, see the Builds documentation.

You can add environment variables during app creation by clicking "Edit Environment Variables". For more details on environment variables, see the Environment Variables and Contexts documentation.

Using the CLI

You can also create applications from the command line with `deno deploy create`. Running without flags starts an interactive wizard that walks you through each step:

```
deno deploy create
```

For CI/CD pipelines or scripting, provide flags to skip the wizard entirely:

```
deno deploy create \
  --org my-org \
  --app my-api \
  --source local \
  --framework-preset fresh \
  --build-timeout 5 \
  --build-memory-limit 1024 \
  --region us
```

You can also create apps that deploy from a GitHub repository:

```
deno deploy create \
  --org my-org \
  --app my-app \
  --source github \
```

```
  --owner my-github-org \
  --repo my-repo \
  --framework-preset astro \
  --build-timeout 10 \
  --build-memory-limit 2048 \
  --region global
```

For the full list of flags and options, see the deno deploy create CLI reference.

Renaming an application

Applications can be renamed by editing the app slug on the app settings page. This will update the default domain names associated with the app since they are based on the app slug. The new slug must be unique within the organization (i.e. must not be in use by another app or playground in the same organization).

> **⚠ WARNING**
>
> Any previous `deno.net` URLs pointing to the app will no longer work after renaming.
>
> Custom domains will continue to work, as they are not tied to the app slug.

Deleting an application

Applications can be deleted from the app settings page. This will remove the app and all its revisions from the organization. All existing deployments will immediately stop serving traffic, and all custom domain associations will be removed.

The app and its revisions will no longer be accessible after deletion, and no traffic will be served from it. Deleted apps cannot be restored through the Deno Deploy UI.

> **ⓘ NOTE**
>
> Deleted an app by mistake? Contact Deno support within 30 days to restore it.

Limitations

> Apps cannot currently be transferred to another organization.

GitHub integration

The GitHub integration enables automatic deployments of the app from a GitHub repository. Every push to the repository will trigger a new build of the app. Depending on the branch of the commit, the build will be deployed to different timelines.

Apps are linked to a GitHub repository during creation. However, it is possible to unlink the repository after creation, and optionally link it to a new GitHub repository. This can be done from the app settings page.

Only accounts that have been authorized with the Deno Deploy GitHub app will be visible in the GitHub repository dropdown. You can authorize new organizations or repositories by clicking the "+ Add another GitHub account" button in the user or organization dropdown, or the "Configure GitHub app permissions" button in the repository dropdown. This will redirect you to GitHub to authorize the Deno Deploy GitHub app with the selected GitHub account or organization. After authorization, you will be redirected back to the app settings page, where you can select the newly authorized GitHub repository.

GitHub events integration

Whenever Deno Deploy builds an app from a GitHub repository, it will send a repository_dispatch event to the repository at the start and end of the build. This allows you to trigger GitHub Actions workflows based on the build status.

Deno Deploy will send the following events:

Event Name	Description
deno_deploy.build.enqueued	Sent when a build is enqueued, i.e. when a push is made to the repository.
deno_deploy.build.cancelled	Sent when a build is cancelled, either manually or due to a timeout.
deno_deploy.build.failed	Sent when a build fails.
deno_deploy.build.routed	Sent when a build completes successfully, and traffic is routed to it.

The payload of the event follows the following TypeScript type definition:

```
interface DenoDeployBuildEventPayload {
  app: {
    /** The UUID of the Deno Deploy app. */
    id: string;
    /** The slug (name) of the Deno Deploy app. */
    slug: string;
  };
  organization: {
    /** The UUID of the Deno Deploy organization containing the app. */
    id: string;
    /** The slug (name) of the Deno Deploy organization containing the app. */
    slug: string;
  };
  revision: {
    /** The ID of the revision being built. */
    id: string;
    /** A URL to view the revision and build status in the Deno Deploy dashboard. */
    html_url: string;
    /** The Git commit SHA being built. */
    git: { sha: string };
    /** The preview URL the revision is available at, if the build succeeded. */
    preview_url: string | null;
  };
}
```

You can receive these events in a GitHub Actions workflow by adding a repository_dispatch trigger. For example:

```
on:
  repository_dispatch:
    types: [deno_deploy.build.routed] # Listen for successful builds

jobs:
  notify:
    runs-on: ubuntu-latest
    steps:
      - name: Test the preview_url
        run: |
```

```
          echo "The Deno Deploy app is available at ${{ github.event.client_payload.revision.preview_url }}"
          curl -I ${{ github.event.client_payload.revision.preview_url }}
```

Builds

In Deno Deploy, each version of your application code is represented as a revision (or build). When deploying from GitHub, revisions generally map one-to-one to git commits in your repository.

Build triggers

Builds can be triggered in three ways:

- **Automatically from GitHub**: When a new commit is pushed to a GitHub repository linked to your app.
- **Manually from the CLI**: Using the `deno deploy` command.
- **Manually from the UI**: Using the "Deploy Default Branch" button on the builds page, which deploys the default git branch (usually `main`). The dropdown menu lets you select a different branch.

Build stages

A revision goes through these stages before becoming available:

1. **Queuing**: The revision waits to be assigned to a builder.
2. **Preparing**: A builder downloads the source code and restores any available build caches.
3. **Install**: The install command executes (if specified), typically downloading dependencies.
4. **Build**: The build command executes (if specified), creating a build artifact that is uploaded to the runtime infrastructure.
5. **Deploy**: The revision is prepared for deployment into each timeline. For each timeline, the following occurs:
    1. **Create database**: If the application has an attached database, ensure one exists for this timeline (creating it if necessary).
    2. **Pre-deploy command**: Any pre-deploy command configured for the application executes, typically for tasks like database migrations. The command receives the timeline it is preparing in the `DENO_TIMELINE` environment variable, so migrations can target the correct environment.
    3. **Warmup**: Only in the "Preview" timeline, the application is started to ensure it boots correctly.
    4. **Routing**: Roll out the new revision to the URLs associated with this timeline.

If any step fails, the build enters a "Failed" state and does not receive traffic.

Build logs are streamed live to the dashboard during the build process and remain available on the build page after completion.

Build caching speeds up builds by reusing files that haven't changed between builds. This happens automatically for framework presets and the `DENO_DIR` dependency cache.

You can cancel a running build using the "Cancel" button in the top-right corner of the build page. Builds automatically time out based on application configuration. By default, builds time out after 5 minutes, but this can be increased for users on the Pro plan.

App configuration

App configuration defines how to convert source code into a deployable artifact.

There are two places you can set app configuration:

- **In source code**: Using a `deno.json` or `deno.jsonc` file in the application directory.
- **In the Deno Deploy dashboard**: Using the app configuration settings.

If you specify both options, settings in the source code take precedence over those in the dashboard. You will be unable to edit any of the app configuration values in the dashboard if the most recent successful build used configuration from source code.

The application directory must be configured through the dashboard. This setting is not configurable from source code, as it determines where to find the source code itself.

Editing app configuration in the dashboard

You can modify app configuration in three places:

- During app creation by clicking "Edit app config"
- In app settings by clicking "Edit" in the app configuration section
- In the retry drawer on a failed build's page

When creating an app, app configuration may be automatically detected from your repository if you're using a recognized framework or common build setup.

Configuration options

- **App directory**: The directory within the repository to use as the application root. Useful for monorepos. Defaults to the repository root.
- **Framework preset**: Optimized configuration for supported frameworks like Next.js or Fresh. Learn more about framework integrations.
- **Install command**: Shell command for installing dependencies, such as `npm install` or `deno install`.
- **Build command**: Shell command for building the project, often a task from `package.json` or `deno.json`, such as `deno task build` or `npm run build`.
- **Pre-deploy command**: Shell command that runs after the build is complete but before deployment, typically for tasks like database migrations.
- **Runtime configuration**: Determines how the application serves traffic:
  - **Dynamic**: For applications that respond to requests using a server (API servers, server-rendered websites, etc.)
    - **Entrypoint**: The JavaScript or TypeScript file to execute
    - **Arguments** (optional): Command-line arguments passed to the application, after the entrypoint. These are not flags for the Deno runtime: runtime flags such as `--unstable-*` cannot be passed, since the runtime always runs with a fixed set of flags. See the runtime reference for details.
    - **Runtime working directory** (optional): The working directory for the application at runtime
    - **Runtime memory limit** (optional): The maximum amount of memory the application can use at runtime. Defaults to 768 MB, can be increased to 4 GB on the Pro plan.
  - **Static**: For static websites serving pre-rendered content
    - **Directory**: Folder containing static assets (e.g., `dist`, `.output`)

        - **Single page app mode** (optional): Serves `index.html` for paths that don't match static files instead of returning 404 errors
    - **Automatic**: When using a framework preset, the runtime configuration is set automatically.
        - **Runtime memory limit** (optional): The maximum amount of memory the application can use at runtime. Defaults to 768 MB, can be increased to 4 GB on the Pro plan.

- **Build timeout**: Maximum time allowed for the build process. Defaults to 5 minutes, can be increased to 30 minutes on the Pro plan. For apps configured from `deno.json`, set it with `deploy.buildTimeout` instead.
- **Build memory**: Amount of memory allocated to the build process. Defaults to 3 GB, can be increased to 4 GB on the Pro plan.

Editing app configuration from source code

To configure your application from source code, add a `deno.json` or `deno.jsonc` file to the root of your application directory with a `deploy` key. If any of the following app configuration options other than `deploy.buildTimeout` are specified under this key, the entire configuration will be sourced from the file instead of the dashboard (any configuration specified in the dashboard will be ignored).

`deno.json` options

- `deploy.framework` (required unless `deploy.runtime` is set): The framework preset to use, such as `nextjs` or `fresh`. Setting this option automatically configures defaults for the framework. Available presets are listed in the framework integrations docs.
- `deploy.install` (optional): Shell command to install dependencies.
- `deploy.build` (optional): Shell command to build the project.
- `deploy.predeploy` (optional): Shell command to run after the build is complete but before deployment, typically for tasks like database migrations.
- `deploy.buildTimeout` (optional): Maximum time allowed for the build process. Either a whole number of seconds (`600`), or a whole number with an `s`, `m` or `h` suffix (`"90s"`, `"10m"`, `"1h"`). Defaults to 5 minutes. Any other value (such as `0`, `"10"` or `"1.5m"`) causes the whole `deploy` section to be ignored. A value above your plan's maximum (5 minutes on the Free plan, 30 minutes on the Pro plan) is capped to that maximum, and a value below 1 minute is raised to 1 minute. Because the dashboard setting is ignored for apps configured from `deno.json`, this is the only way to change their build timeout. On its own, this option does not move the app configuration into `deno.json`: it only takes effect together with at least one of the other options in this list.
- `deploy.runtime` (required unless `deploy.framework` is set): Configuration for how the app serves traffic. The app can either be static or dynamic, as defined below:
    - For dynamic apps:
        - `deploy.runtime.type`: Must be set to `"dynamic"`, or omitted (dynamic is the default).
        - `deploy.runtime.entrypoint`: The JavaScript or TypeScript file to execute.
        - `deploy.runtime.args` (optional): Command-line arguments passed to the application, after the entrypoint. These are not flags for the Deno runtime; runtime flags such as `--unstable-*` cannot be passed. See the runtime reference.
        - `deploy.runtime.cwd` (optional): The working directory for the application at runtime.
        - `deploy.runtime.memoryLimit` (optional): The maximum amount of memory the application can use at runtime, in MB. Defaults to 768 MB, can be increased to 4 GB on the Pro plan. The earlier

spelling `memory_limit` is still accepted but deprecated.

- For static apps:
  - `deploy.runtime.type`: Must be set to `"static"`.
  - `deploy.runtime.cwd`: Folder containing static assets (e.g., `dist`, `.output`).
  - `deploy.runtime.spa` (optional): If `true`, serves `index.html` for paths that don't match static files instead of returning 404 errors.
- For apps using a framework preset:
  - `deploy.runtime.memoryLimit` (optional): The maximum amount of memory the application can use at runtime, in MB. Defaults to 768 MB, can be increased to 4 GB on the Pro plan. The earlier spelling `memory_limit` is still accepted but deprecated.

Examples

Example dynamic app configuration from `deno.json`:

```
{
  "deploy": {
    "install": "npm install",
    "build": "npm run build",
    "predeploy": "deno run --allow-net --allow-env migrate.ts",
    "runtime": {
      "type": "dynamic",
      "entrypoint": "./app/server.js",
      "args": ["--port", "8080"],
      "cwd": "./app"
    }
  }
}
```

Example static app configuration from `deno.jsonc`:

```
{
  "deploy": {
    "install": "npm install",
    "build": "npm run build",
    "runtime": {
      "type": "static",
      "cwd": "./public",
      "spa": true
    }
  }
}
```

Example framework preset configuration with Next.js from `deno.json`:

```
{
  "deploy": {
    "framework": "nextjs",
    "install": "npm install",
    "build": "npm run build",
    "buildTimeout": "15m"
  }
}
```

Build environment

The build environment runs on Linux using either x64 or ARM64 architecture. Available tools include:

- `deno` (same version as at runtime)
- `node`
- `npm`
- `npx`
- `yarn` (v1)
- `pnpm`
- `git`
- `tar`
- `gzip`

> **NOTE**
>
> All JavaScript inside of the builder is executed using Deno.
>
> The `node` command is actually a shim that translates Node.js invocations to `deno run`. Similarly, `npm`, `npx`, `yarn`, and `pnpm` run through Deno rather than Node.js.

Environment variables configured for the "Build" context are available during builds, but variables from "Production" or "Development" contexts are not. Learn more about environment variables.

The following environment variables are additionally always available during builds:

- `CI`: `true`
- `DENO_DEPLOY`: `true` - Indicates that the code is running in Deno Deploy.
- `DENO_DEPLOY_ORG_ID`: The ID of the organization that owns the application. This is a UUID.
- `DENO_DEPLOY_ORG_SLUG`: The slug of the organization that owns the application. This is the human-readable identifier used in URLs that was set when creating the organization.
- `DENO_DEPLOY_APP_ID`: The ID of the application. This is a UUID.
- `DENO_DEPLOY_APP_SLUG`: The slug of the application. This is the human-readable identifier used in URLs that was set when creating the application, or changed later in the application settings.
- `DENO_DEPLOY_BUILD_ID`: The ID of the currently running build.

Builders have the following resources available during the build process:

- 2 vCPUs
- 3 GB of RAM (can be increased to 4 GB on Pro plan)
- 8 GB of storage

Environment Variables and Contexts

Environment variables in Deno Deploy allow you to configure your application with static values such as API keys or database connection strings.

Types of environment variables

Environment variables can be stored as:

- **Plain text**: Visible in the UI and suitable for non-sensitive values like feature flags
- **Secrets**: Never visible in the UI after creation, only readable from application code, suitable for sensitive values like API keys

Variables can be set at:

- **Application level**: Specific to a single application
- **Organization level**: Applied to all applications in the organization, but can be overridden by application-level variables

Contexts

Each environment variable applies to one or more contexts. Contexts represent the logical "environments" in which your code runs, each with its own set of variables and secrets.

By default, there are two contexts:

- **Production**: Used for the production timeline serving production traffic
- **Development**: Used for development timelines serving non-production traffic (preview URLs and branch URLs)

> **NOTE**
>
> Need additional contexts? Please contact support.

Additionally, there is a **Build** context used during the build process. Environment variables in the Build context are only available during builds and aren't accessible in Production or Development contexts (and vice versa). This separation enables different configuration for build-time vs. runtime.

Within a single application or organization, you cannot have multiple environment variables with the same name in the same context. You can, however, have variables with the same name in different non-overlapping contexts.

Adding, editing and removing environment variables

You can manage environment variables from several locations:

- On the "New App" page while creating an application
- In the application settings under the "Environment Variables" section
- In the organization settings under the "Environment Variables" section

In each location, click the relevant edit button to open the environment variables drawer. Changes only apply when you click "Save." Clicking "Cancel" discards your changes.

To add a variable:

1. Click "Add Environment Variable"
2. Enter the name and value
3. Specify whether it's a secret
4. Select the contexts where it should apply

You can also bulk import variables from a `.env` file:

1. Click "+ Add from .env file"
2. Paste the contents of your `.env` file
3. Click "Import variables"

Note that lines starting with # are treated as comments.

To remove a variable, click the "Remove" button next to it.

To edit a variable, click the "Edit" button next to it to modify its name, value, secret status, or applicable contexts.

Using environment variables in your code

Access environment variables using the `Deno.env.get` API:

```
const myEnvVar = Deno.env.get("MY_ENV_VAR");
```

Limits

Environment variables have the following limits:

- Environment variable keys can be at most 128 bytes long.
- Environment variable keys can not contain =, null bytes, or leading or trailing whitespace.
- Environment variable keys can not start with:
  - `DENO_`, except for `DENO_AUTH_TOKENS`, `DENO_COMPAT`, `DENO_CONDITIONS`, `DENO_DEPLOY_ENDPOINT`, or `DENO_DEPLOY_TOKEN`
  - `LD_`
  - `OTEL_`
- Environment variable values can be at most 128 kB (128,000 bytes) long.
- Environment variables can total at most 256 KiB (262,144 bytes) of keys and values within one scope. Organization, application and layer scopes are budgeted separately.

No individual key name is reserved. If you use Cloud Connections, the credentials they provide are set by Deno Deploy in the contexts the connection applies to, and take precedence over an environment variable you define with the same name.

Predefined environment variables

Deno Deploy provides these predefined environment variables in all contexts:

- `DENO_DEPLOY`: `true` - Indicates that the code is running in Deno Deploy.

- `DENO_DEPLOY_ORG_ID`: The ID of the organization that owns the application. This is a UUID.
- `DENO_DEPLOY_ORG_SLUG`: The slug of the organization that owns the application. This is the human-readable identifier used in URLs that was set when creating the organization.
- `DENO_DEPLOY_APP_ID`: The ID of the application. This is a UUID.
- `DENO_DEPLOY_APP_SLUG`: The slug of the application. This is the human-readable identifier used in URLs that was set when creating the application, or changed later in the application settings.
- `DENO_DEPLOY_BUILD_ID`: The ID of the revision that is currently running. During builds, this is the ID of the revision being built.
- `DENO_DEPLOYMENT_ID`: A unique identifier representing the entire configuration set (application ID, build ID, context, environment variables, cloud connections, database). Changes if any of these components change.
- `DENO_TIMELINE`: The timeline the application is currently running in. Possible values are `production`, `git-branch/<branch-name>`, and `preview/<revision-id>`. This is not set during builds, as builds are not specific to any timeline. It is set for the pre-deploy command, which runs once per timeline and receives the timeline it is preparing.

During builds, the environment variable `CI=true` is additionally set.

Timelines

A timeline is a representation of the history of one branch of the application. Each timeline has a set of revisions, which are the individual items in the timeline. One of the revisions (usually the most recent one) is the "active" revision, which is the one that is currently serving traffic. The active revision receives traffic on all URLs that are assigned to the timeline.

Each timeline is associated with a context, which determines which environment variables are available to the code running in that timeline.

By default, there are multiple timelines set up for each application:

- **Production**: The production timeline contains all of the revisions from the default git branch. This is the timeline that serves production traffic. This timeline is associated with `https://<app-name>.<org-name>.deno.net`, and any custom domains that are mapped to the application. It uses the production context.
- **Git Branch / `<branch-name>`**: Each git branch has its own timeline. This timeline contains all of the revisions from that git branch. This timeline is associated with `https://<app-name>--<branch-name>.<org-name>.deno.net`. It uses the development context.

> There is also one timeline for each revision, that contains only that revision. This is the timeline that backs the preview URL for that revision. This timeline is associated with `https://<app-name>-<revision-id>.<org-name>.deno.net`. It uses the development context.
>
> Preview timelines are not visible in timeline pages in the UI. You can view the preview URL for a revision on that revision's build page.

You can view the timelines that each revision is associated with on the revision's build page. You can also view the revisions that are associated with a given timeline from the timeline pages.

Active revision

Each timeline has an active revision. The active revision is the revision that is currently serving traffic for that timeline. You can view the active revision for a timeline on the timeline page.

Usually, the active revision is the most recently built revision on the timeline. However, a different revision can be manually locked to be the active revision. This enables rollback, and timeline locking:

Rollback

Rollback is the process of reverting the active revision to a previous revision, usually because the newer revision has some sort of bug or issue. By rolling back to a known good revision, you can restore the application to a working state without having to deploy new code via Git, and waiting for a build to complete.

Refer to "changing the active revision" below for more information on how to rollback a timeline.

Timeline locking

Timeline locking is the process of locking a timeline to a specific revision, to ensure that new builds do not automatically become the active revision. This is useful if you are in a feature freeze situation, for example during a big event, and want to de-risk by not allowing new builds to be deployed. When a timeline is locked to a specific revision you can still create new builds by pushing to Git, but they will not automatically become

the active revision on the locked timeline.

Refer to "changing the active revision" below for more information on how to lock a timeline to a specific revision.

Changing the active revision

On the timelines page, you can lock any revision on that timeline to be the active revision. This will lock the timeline to that revision, and new builds will not automatically become the active revision on this timeline anymore. You can then either unlock the revision from the timeline, reverting back to the default behavior of the latest revision being the active revision, or you can lock a different revision to be the active revision.

Tunnel

Deno Deploy's tunnel feature allows you to securely expose your local development server to the internet. This is particularly useful for testing webhooks, sharing your work with collaborators, or accessing your local server from remote locations.

In addition to providing secure access to your local server, Deno Deploy's tunnel can also:

- Pull environment variables in the "Local" context from your Deno Deploy project to your local Deno process.
- Push Open Telemetry traces, metrics, and logs from your local Deno process to your Deno Deploy app, where you can view them in the Deno Deploy dashboard.
- Automatically connect to local development databases assigned to your Deno Deploy app.

Getting started

To start using the tunnel feature, you'll need to have Deno installed on your local machine. You can then pass the `--tunnel` flag when running your Deno application locally, either with `deno task` or `deno run`. For example:

```
deno run --tunnel -A main.ts
```

The first time you run this command, you'll be prompted to authenticate with Deno Deploy and choose which Deno Deploy app you want to connect the tunnel to. Once authenticated, a secure tunnel will be established, and you'll receive a public URL that forwards traffic to your local server.

You can also specify `--tunnel` for `deno task` commands defined in your `deno.json` or `package.json` file:

```
{
  "tasks": {
    "dev": "astro dev"
  }
}
```

Then run the task with:

```
deno task --tunnel dev
```

Using the tunnel

Once the tunnel is established, any requests made to the public URL will be forwarded to your local development server. You can use this URL to test webhooks, share your work with others, or access your local server from remote locations.

Stopping the tunnel

To stop the tunnel, simply terminate the Deno process running your application. This will close the secure connection and stop forwarding traffic to your local server.

View open tunnels

The "Tunnels" tab of your application dashboard on Deno Deploy shows all active tunnels connected to your application. From this tab, you can view details about each tunnel, including the public URL, the local address it's forwarding to, and the time it was established.

Environment variables

When using the tunnel feature, the "Local" context environment variables from your Deno Deploy application are made available to your local Deno process. This allows you to use the same configuration locally as you do in your Deno Deploy application.

You can view and manage the environment variables for your Deno Deploy application in the "Environment Variables" tab of your application settings. See the docs on adding, editing, and removing environment variables for more information.

Viewing traces and logs

When using the tunnel feature, Open Telemetry traces, metrics, and logs from your local Deno process are pushed to your Deno Deploy application. You can view these traces and logs in the "Observability" tab of your application dashboard on Deno Deploy.

You can filter to see only traces and logs from your local process by searching for `context:local` in the search bar.

Observability

Deno Deploy provides comprehensive observability features to help you understand application performance, debug errors, and monitor usage. These features leverage OpenTelemetry and the built-in OpenTelemetry integration in Deno.

The three main observability features in Deno Deploy are:

- **Logs**: Unstructured debug information emitted by your application code
- **Traces**: Structured information about request handling, including execution time for each step and automatic capture of outbound I/O operations
- **Metrics**: Structured, high-level data about application performance and usage, such as request count, error count, and latency

Logs

Logs in Deno Deploy are captured using the standard `console` API and can be queried from the logs page in the dashboard.

Logs are organized by application. You can use the search bar to filter logs based on various attributes and message content.

When logs are emitted inside the context of a trace, they become associated with that specific trace and span. For such logs, a "View trace" button appears in the logs interface, allowing you to open the relevant trace in an overlay drawer for detailed inspection.

Traces

Traces in Deno Deploy are captured in three ways:

- **Automatically for built-in operations**: Incoming HTTP requests, outbound fetch calls, and other system operations are traced automatically. This cannot be disabled.
- **Automatically for supported frameworks**: Frameworks like Next.js, Fresh, and Astro include built-in instrumentation. The specific frameworks and operations covered may change over time.
- **Manually through custom instrumentation**: Your application code can create new traces or spans using the OpenTelemetry API.

Traces are organized by application. The search bar lets you filter based on various attributes and span names.

The Traces list shows top-level request traces. Custom spans you create through manual instrumentation appear as child spans nested within the request trace they run under, not as separate entries in the list. To find them, open the relevant request trace rather than looking for a standalone trace.

Clicking a trace opens the trace overlay drawer, showing all spans within that trace in a waterfall view. This visualization displays the start time, end time, and duration of each span, grouped by parent span with the root span at the top.

Clicking any span shows its details at the bottom of the drawer, including all captured attributes. For example, outbound HTTP requests include the method, URL, and status code.

The span details section also includes a "Logs" tab showing all logs emitted within the selected span's context.

You can click "View logs" on any trace to open the logs page with the trace ID pre-filled in the search bar, showing all logs related to that trace.

Metrics

Metrics in Deno Deploy are automatically captured for various operations such as incoming HTTP requests and outbound fetch calls. This automatic capture cannot be disabled.

Metrics are organized by application and displayed in time-series graphs showing values over time. You can use the search bar to filter metrics based on various attributes.

Filtering

Logs, traces, and metrics can be filtered using these general attributes:

- **Revision**: The ID of the application revision that emitted the data
- **Context**: The context in which the data was emitted ("Production" or "Development")

For logs and traces, this additional filter is available:

- **Trace**: The ID of the trace containing the log or spans

For traces only, these additional filters are available:

- **HTTP Method**: The HTTP method of the request that triggered the trace
- **HTTP Path**: The path of the request that triggered the trace
- **HTTP Status**: The HTTP status code of the response

Time range filter

By default, the observability pages show data for the last hour. You can change this using the time range filter in the top right corner of each page.

You can select predefined time ranges like "Last 1 hour," "Last 24 hours," or "Last 7 days," or set a custom time range by clicking the "Custom" button.

Custom time ranges can be either absolute (specific start and end times) or relative (e.g., 3 days ago, 1 hour from now). Relative time ranges use the same syntax as Grafana:

- `now` - the current time
- `now-1h` - 1 hour ago
- `now/h` - the start of the current hour
- `now-1h/h` - the start of the previous hour
- `now/d+3h` - 3 hours from the start of the current day
- `now-1d/d` - the start of the previous day page. The time range filter can be set to a predefined time range, like "Last 1 hour", "Last 24 hours", or "Last 7 days", or a custom time range.

The custom time range can be set by clicking on the "Custom" button. A custom time range can either be absolute (a specific start and end time) or relative (3 days ago, 1 hour from now, etc.). The time range filter is

shown in the top right corner of the page.

Relative time ranges use the same syntax as Grafana, where `now` is the current time, and `now-1h` is 1 hour ago. Furthermore syntax such as `now-1h/h` can be used to round the time to the nearest hour. Some examples:

- `now-1h` - 1 hour ago
- `now/h` - the start of the current hour
- `now-1h/h` - the start of the previous hour
- `now/d+3h` - 3 hours from the start of the current day
- `now-1d/d` - the start of the previous day

Domains

Every organization has a default domain used for all applications deployed within that organization. For example, an organization with the slug `acme-inc` would have a default domain of `acme-inc.deno.net`. An application named `my-app` would automatically receive the production domain `my-app.acme-inc.deno.net`.

In addition to these default domains, you can add custom domains to your applications. Custom domains are domains that you own and control. To use a custom domain, you must:

1. Own the domain (purchased from a domain registrar)
2. Have access to edit its DNS records

Custom domains belong to an organization and can be attached to any application within that organization.

A custom domain can be added as:

- A base domain (e.g., `example.com` or a specific subdomain)
- A wildcard domain (e.g., `*.example.com`)

A base domain works with a single application, while a wildcard domain offers more flexibility. You can either:

- Assign the entire wildcard to one application (all subdomains point to the same app)
- Partially assign it to multiple applications (different subdomains point to different apps)

All custom domains require valid TLS certificates. Deno Deploy can automatically provision these certificates using Let's Encrypt. Alternatively, you can bring your own TLS certificates, which you will then need to renew manually.

Adding a custom domain

1. Go to the organization domains page (click your organization name in the top left corner, then the "Domains" tab)
2. Click "Add Domain"
3. Enter your domain (e.g., `example.com`)
4. Select whether to add just this domain or also include the wildcard subdomain
5. Click "Add Domain"

This will open the domain configuration drawer.

DNS configuration

The domain configuration drawer shows the DNS records needed to:

- Verify domain ownership
- Optionally provision TLS certificates
- Route traffic to Deno Deploy

There are three possible configuration methods, depending on your domain registrar's capabilities:

ANAME/ALIAS method (preferred)

If your registrar supports `ANAME` or `ALIAS` records, this is the best option:

- Add one `ANAME/ALIAS` record
- Add one `CNAME` record for verification

CNAME method

Works well for subdomains but not for apex domains:

- Add two `CNAME` records
- Note: This method doesn't allow other DNS records (like `MX` records) on the same domain

A record method

Most compatible but requires more configuration:

- Add one `A` record
- Add one `CNAME` record for verification

> Note: Deno Deploy does not currently support IPv6. When using the `ANAME/ALIAS` or `CNAME` methods, your domain will automatically use IPv6 when supported. With the `A` method, you'll receive an email when it's time to add an `AAAA` record.

> **CAUTION**
> When using Cloudflare as your DNS provider, you **MUST** disable the proxying feature (orange cloud) for the `_acme-challenge` CNAME record, or verification and certificate provisioning will fail.

Verification

After adding the DNS records, Deno Deploy will verify your domain ownership. This process may take a few minutes depending on your DNS provider. You can leave the domain configuration drawer open during verification — it will refresh automatically when complete.

You can manually trigger verification by clicking the "Provision Certificate" button. Successful verification also initiates TLS certificate provisioning.

TLS certificates

After domain verification, you need a valid TLS certificate to use the domain with Deno Deploy. You can either have Deno Deploy provision a certificate for you using Let's Encrypt, or you can bring your own certificate.

Automatic provisioning (Let's Encrypt)

After domain verification, click "Provision Certificate" to generate a TLS certificate through Let's Encrypt. This process can take up to 90 seconds.

Once provisioned, you'll see certificate details including expiration date and issue time.

Certificates are automatically renewed near expiration. You can check the current certificate status in the domain configuration drawer.

If automatic renewal fails (for example, because DNS records changed), you will receive an email notification 14 days before the certificate expires. You then have a chance to fix the issue and contact support to retry the renewal. If the certificate is not renewed before expiration, the domain will stop working.

Bring your own certificate

If you prefer to use your own TLS certificate, you can upload it in the domain configuration drawer. You'll need to provide the following:

- The certificate file (PEM format)
- The private key file (PEM format)

Once uploaded, the certificate will be used for the domain. You are responsible for renewing and updating the certificate before it expires.

You will receive email notifications 14 days before the certificate expires reminding you to update it. If the certificate expires, the domain will stop working.

The TLS certificate must be valid at the time of upload. It must cover the base domain (and, if you have a wildcard domain, the wildcard subdomain as well) through either the common name or the subject alternative names in the certificate. The private key and certificate must match, and must be either RSA (2048, 3072, or 4096 bits) or ECDSA (P-256, P-384, or P-521).

Assigning a custom domain to an application

After adding a custom domain to your organization:

1. Go to the organization domains page
2. Click "Assign" next to the custom domain
3. Select the target application
4. If using a wildcard domain, choose whether to attach the base domain, the wildcard, or a specific subdomain
5. Click "Assign Domain"

Redirecting one domain to another

A custom domain serves whatever application you assign it to, so to make one domain redirect to another — for example, sending `example.com` to `www.example.com` so the `www` host is canonical — deploy a small application that issues the redirect and assign the redirecting domain to it.

The redirect application is a single `Deno.serve()` handler that rewrites the host while preserving the path and query string:

redirect.ts

```ts
Deno.serve((req) => {
  const url = new URL(req.url);
  const target = new URL(url.pathname + url.search, "https://www.example.com");
  return Response.redirect(target, 301);
});
```

`301` is a permanent redirect, which is what you want for a canonical host so browsers and search engines remember it. Use `302` if the redirect is only temporary.

Then wire up the two domains:

1. Assign the canonical domain (`www.example.com`) to your main application, following Assigning a custom domain.
2. Deploy the redirect application above as a separate application and assign the other domain (`example.com`) to it.

Requests to the non-canonical domain now return a `301` to the same path on the canonical domain. To redirect the other way (`www` to the apex domain), swap the two domain assignments and change the target URL in the handler.

Unassigning a custom domain from an application

1. Go to the application settings page
2. Find the "Custom Domains" section
3. Click "Remove" next to the domain you want to unassign

This removes the domain from the application but keeps it available in your organization for use with other applications.

Removing a custom domain

1. Go to the organization domains page
2. Open the domain configuration drawer
3. Click "Delete" and confirm

This removes the custom domain from your organization and deletes all domain assignments across all applications.

Migrating a custom domain from Deploy Classic to Deno Deploy

If you have previously set up a custom domain on Deploy Classic and want to migrate it to Deno Deploy, we've created a step-by-step tutorial to guide you through the process.

Cron

Cron jobs are scheduled tasks that run automatically on a defined schedule. You define cron jobs in your code using the `Deno.cron()` API, deploy your application, and the platform discovers and runs them on schedule.

`Deno.cron()` is a Deno runtime API — it ships with Deno itself. Deno Deploy builds on top of that runtime API: it discovers your `Deno.cron()` definitions at deployment time, schedules and invokes them, handles retries, and surfaces runs in the dashboard and logs, so you don't need to keep a long-running process up yourself.

Defining cron jobs in code

`Deno.cron()` takes a human-readable name, a schedule, and a handler function. The name identifies the cron job in the dashboard and logs, the schedule determines when it fires, and the handler contains the code to run on each invocation. The schedule can be either a standard 5-field cron expression or a structured object. All times are UTC — this avoids ambiguity around daylight saving transitions. See the full API reference for details.

```
Deno.cron("cleanup-old-data", "0 * * * *", () => {
  console.log("Cleaning up old data...");
});

Deno.cron(
  "sync-data",
  "*/15 * * * *",
  {
    backoffSchedule: [1000, 5000, 10000],
  },
  async () => {
    await syncExternalData();
  },
);
```

Common schedule expressions

Schedule	Expression
Every minute	`* * * * *`
Every 15 minutes	`*/15 * * * *`
Every hour	`0 * * * *`
Every 3 hours	`0 */3 * * *`
Daily at 1 AM	`0 1 * * *`
Every Wednesday at midnight	`0 0 * * WED`
First of the month at midnight	`0 0 1 * *`

Organizing cron declarations

You must register cron jobs at the module top level, before `Deno.serve()` starts. The platform extracts `Deno.cron()` definitions at deployment time by evaluating your module's top-level code — this is how it

discovers which cron jobs exist and what their schedules are. Cron jobs registered inside request handlers, conditionals, or after the server starts will not be picked up.

As the number of cron jobs grows, keeping them all in your main entrypoint can get noisy. A common convention is to define cron handlers in a dedicated file and import it at the top of your entrypoint.

crons.ts

```
Deno.cron("cleanup-old-data", "0 * * * *", async () => {
  await deleteExpiredRecords();
});

Deno.cron("sync-data", "*/15 * * * *", async () => {
  await syncExternalData();
});
```

main.ts

```
import "./crons.ts";

Deno.serve(handler);
```

Because `Deno.cron()` calls execute at the module top level, the import alone is enough to register them — no need to call or re-export anything.

For projects with many cron jobs, you can use a `crons/` directory with one file per cron job or group of related cron jobs, and a barrel file that re-exports them:

crons/mod.ts

```
import "./cleanup.ts";
import "./sync.ts";
```

main.ts

```
import "./crons/mod.ts";

Deno.serve(handler);
```

Restrictions

There are a couple of restrictions to keep in mind when defining cron jobs. The name passed to `Deno.cron()` can be at most **256 characters** long — this is the identifier that appears in the dashboard and logs, so keeping it concise is good practice regardless. On free organizations, each revision can register at most **10 cron jobs**. If your project needs more, upgrading to a paid plan removes this limit.

Execution lifecycle & status

When a cron job is due, it fires independently on each timeline where it's registered. Each execution runs the handler from that timeline's active revision. For example, if a `cleanup-old-data` cron job is registered in both the production timeline and a `staging` branch timeline, the production execution runs the handler from the production revision, and the staging execution runs the handler from the staging revision. Each execution will be billed as one inbound HTTP request.

Cron job executions progress through these statuses:

Status	Color	Description
Running	Yellow	Cron job handler is currently executing

OK	Green	Completed successfully
Error	Red	Failed — hover to see the error message

The platform prevents overlapping executions: the same cron job cannot run concurrently. If a cron job is still running when the next scheduled invocation is due, that invocation is skipped. This avoids resource contention and ensures each execution can complete without interference.

Retries & backoff

By default, failed cron job executions are not retried. You can optionally provide a `backoffSchedule` array to enable retries and control exactly when each one happens.

The `backoffSchedule` is an array of delays in milliseconds, where each element specifies how long to wait before the next retry attempt:

- Maximum **5 retries** per execution
- Maximum delay per retry: **1 hour** (3,600,000 ms)
- Retries do not affect the cron job schedule — the next scheduled run will be executed on time, even if the previous run has retries pending. If a retry and the next scheduled run coincide, the later one will be skipped as per the overlapping policy described above.

Example: `backoffSchedule: [1000, 5000, 10000]` retries up to 3 times with delays of 1s, 5s, and 10s.

Dashboard

The **Cron** tab in the app sidebar gives you an overview of all registered cron jobs across your project. Each entry shows the cron job's schedule, its most recent executions, and the active timelines it belongs to. When a cron job with the same name is registered with different schedules on different timelines, each distinct schedule appears as its own entry so you can track them independently.

Click "View Details" on any cron job to open its detail page, which shows the full execution history. You can filter executions using the search bar:

- **`status:<value>`** — filter by status (`ok`, `error`, `running`)
- **`timeline:<name>`** — filter by timeline name (substring match, case-insensitive)

Observability integration

Cron job executions produce OpenTelemetry traces. Click "View Trace" in the execution history to navigate to the Observability traces page for that specific trace.

On the traces page, you can use these cron-related filters:

- **`kind:cron`** — show only cron spans
- **`cron.name:<name>`** — filter by cron name
- **`cron.schedule:<schedule>`** — filter by schedule expression

Timelines

Cron jobs run on production and git branch timelines. The platform extracts `Deno.cron()` definitions at deployment time and schedules them for execution, so each timeline runs the set of cron jobs defined in its active revision's code. To add, remove, or modify cron jobs, update your code and deploy a new revision. Rolling back to a previous deployment re-registers the cron jobs from that deployment. You can see which cron jobs are currently registered in a given timeline from its page in the dashboard.

Deno KV

Deno KV is a Key Value database supported in Deno Deploy as a database engine option in the databases feature. Thanks to the new timelines capability in Deno Deploy Early Access (EA), your apps have full control over the Deno KV databases they use (for example, one for production and one for each Git branch), ensuring data isolation and security across environments.

As with other database engines, your code automatically connects to the correct database for each environment—no timeline detection or manual database naming required.

Getting Started

Add a KV database

Navigate to your organization dashboard and click "Databases" in the navigation bar. Click "Provision Database", choose Deno KV as the database engine, provide a memorable name, and save.

Connect an app to a KV database

Once you have a database instance you can assign it to an app. From the database instances list, click "Assign" next to the database you wish to use and select the app from the dropdown.

Deno Deploy automatically creates a separate database for each timeline. This keeps your production data safe while you develop and test. You can monitor provisioning and watch the status change to "Connected." If any errors occur, click "Fix" to retry.

Connect from Deno Deploy

Once you've assigned a database to your app, connecting from code is simple. Deno Deploy sets up the connection to the correct database based on the current environment — no database ID or access token required.

Example

Here's how to connect to Deno KV in your Deno Deploy app:

```
const kv = await Deno.openKv();

Deno.serve(async () => {
  const res = await kv.get<number>(["requests"]);
  const requests = res.value + 1;
  await kv.set(["requests"], requests);
  return new Response(JSON.stringify(requests));
});
```

For detailed information about Deno KV and its features, see the Deno KV documentation.

Connect from outside Deno Deploy

You can also open a managed Deno KV database from local Deno CLI apps using the URL connector:

```
const kv = await Deno.openKv(
  "https://api.deno.com/v2/databases/<Database ID>/connect",
);
```

Replace <Database ID> with the ID of the database you want to connect to. Each database's ID is shown in the Databases table on the database instance page in the Deno Deploy console.

Authenticate the connection by setting either a personal or an organization access token as the DENO_KV_ACCESS_TOKEN environment variable:

```
export DENO_KV_ACCESS_TOKEN=ddo_...
```

Organization access tokens can be created from your organization's settings page in the Deno Deploy console.

Un-assigning a KV database

If you remove a database assignment from an app, the app will no longer be able to access that database. However, the database itself and its data will remain intact and can be reassigned to another app or the same app at a later time. Hover over the name of the assigned app in the databases list and click the 'remove app assignment' icon to un-assign it.

Data Distribution and GDPR

Deno KV databases are replicated across at least three data centers in the primary region, Northern Virginia (us-east4). Once a write operation is committed, its mutations are durably stored in a quorum of data centers within the primary region with *read* replicas in Europe and Asia.

Data written to KV is therefore stored in, and transits through the US. For workloads that require EU-only data residency or strict GDPR guarantees, Deno KV is not suitable in its current form. Data is encrypted in transit and at rest, and KV is isolated per project, but we do not currently provide formal GDPR or compliance documentation specific to KV.

For projects with strict GDPR compliance requirements, we recommend one of our Postgres database solutions such as our Prisma integration

Data storage

By default, in local development data is kept in memory. You do not need to create or allocate a database before using the KV APIs locally, and your KV code remains consistent across environments. To develop against a managed Deno KV database from a local Deno program instead, see Connect from outside Deno Deploy.

Deleting a database instance

Click "Delete" on the Deno KV entry in the database instances list. Unlike other database engines, this action deletes all existing Deno KV databases and their data. Be sure to back up your data before proceeding.

Databases

Deno Deploy's databases feature enables your applications to easily connect to a variety of databases, enabling seamless state management in your apps. Currently, PostgreSQL and Deno KV are supported.

After creating or linking a database instance, Deno Deploy automatically creates isolated (logical) databases inside of that instance for each deployment environment, including production, Git branches, and preview timelines. Your application code connects to the appropriate database based on the current environment, using automatically injected environment variables. This ensures that your data remains consistent and isolated across different stages of development and deployment.

Deno Deploy currently supports two database engines:

- **PostgreSQL** — Connect an existing externally hosted PostgreSQL instance, or provision a managed PostgreSQL database through Deno Deploy, hosted by Prisma.
- **Deno KV** — Provision a fast, globally distributed key-value store built for the edge.

Creating a database instance

There are two ways to add a database instance to your Deno Deploy organization:

- **Link Database**: Connect an existing external database instance (for example, a PostgreSQL server you run or a managed instance from a cloud provider).
- **Provision Database**: Create and attach a managed data store from Deno Deploy (Deno KV or Prisma Postgres).

Linking an external database

To link an existing external database instance you can either:

- go to the "Databases" page in your organization dashboard and click the "Link Database" button,
- go to the "Databases" tab in your app settings, click on "Attach Database", then select "Link Database" in database instance selection dropdown.

From here, enter the connection details for your external database instance. You will need to provide:

- **Engine**: Select the database engine (currently only PostgreSQL is supported).
- **Connection Details**: Enter the hostname, port, username, password, and optionally a CA certificate if required by your provider. You can also paste a connection string to automatically populate these fields.
- **Slug**: Give your database instance a descriptive name to identify it in the dashboard. This name is only used within Deno Deploy and does not affect your actual database server.

Once you've filled out the form, click "Test Connection" to verify your settings. If the connection is successful, click "Save" to add the database instance to your organization.

If the connection fails, double-check your connection details and ensure that your database server is accessible from Deno Deploy's network. We are unable to provide a list of IP addresses for Deno Deploy at this time, so please ensure your database server allows connections from all IPs. If you continue to have trouble, you can contact support for assistance.

> **NOTE**
>
> Because Deno Deploy creates isolated databases for each environment (production, Git branches, and previews), ensure that the database user you provide has sufficient privileges to create new databases on the server.

TLS/SSL configuration

When linking an external database, Deno Deploy supports secure SSL/TLS connections. Depending on your database provider, you may need to upload a CA certificate to verify the server's identity.

If your database provider uses a trusted root Certificate Authority (CA), such as Let's Encrypt, no certificate upload is needed and SSL connections work automatically.

For users of AWS RDS, we will automatically detect RDS instances and provide an option to "Use AWS Certificate Bundle" to configure the necessary certificates without manual downloads.

For Google Cloud SQL users, you will need to download the Google Cloud SQL CA certificate from your Google Cloud Console and upload it when linking your database.

For other providers using self-signed certificates or private CAs, you will need to upload the specific CA certificate that was used to sign your database's certificate. You can usually obtain this from your database provider's documentation or console.

Provisioning a managed database

To create and attach a managed database instance from Deno Deploy, you can either:

- go to the "Databases" page in your organization dashboard and click the "Provision Database" button,
- go to the "Databases" tab in your app settings, click on "Attach Database", then select "Provision Database" in database instance selection dropdown.

From here, select the database engine you want to provision. Available today:

- **Deno KV** — a fast, globally distributed key-value store built for the edge, hosted by Deno on your behalf.
- **Prisma Postgres** — the world's most advanced open source relational database, hosted by Prisma.

You will have to provide a **Slug** to identify the database instance in the dashboard. This name is only used within Deno Deploy and does not affect your actual database server.

Depending on the engine you select, there may be additional configuration options, such as choosing a region. Choosing a region close to your application's users can help reduce latency and improve performance of your database queries.

Once you're ready, click "Provision" to create the database instance. Deno Deploy will handle the provisioning process and set up the necessary infrastructure on your behalf.

Linking databases to your apps

After creating or linking a database instance, you can assign it to your apps. Each database instance can be assigned to multiple apps. Each app gets its own isolated databases within the instance for each deployment environment (production, Git branches, and preview timelines).

> **NOTE**
>
> It is not currently possible to link multiple database instances to a single app. It is thus not possible to link both a Deno KV and a PostgreSQL database to the same app at this time.

Once assigned, Deno Deploy automatically creates an isolated database for each timeline within that app. The naming scheme for these databases is as follows:

- The single production database uses the format `{app-id}-production`
- Each Git branch database uses the format `{app-id}--{branch-name}`
- A singular preview database exists, with the format `{app-id}-preview`

> **NOTE**
>
> Currently only one preview database is created per app, shared across all preview deployments. In future releases, each preview deployment will get its own isolated database.

To assign a database to an app you can either:

- Go to the "Databases" page in your organization dashboard, find the database instance you want to assign, click "Assign", then select the app from the dropdown.
- Go to the "Databases" tab in your app settings, click on "Attach Database", then select the database instance from the dropdown.

Once assigned, Deno Deploy will automatically create the necessary isolated databases for each environment within that app.

Connecting to databases from your code

Once you've assigned a database to your app, connecting to it from your code is simple. Deno Deploy automatically handles connection details, credentials, and environment variables for you.

Deno KV

For Deno KV, you can use the built-in `Deno.openKv()` API to connect to your assigned Deno KV instance. No additional configuration is needed - Deno Deploy automatically connects your app to the correct Deno KV instance based on the current environment.

```
// No arguments needed - Deno Deploy handles this automatically
const kv = await Deno.openKv();

Deno.serve(async () => {
  // Use the Deno KV instance
  await kv.set(["user", "123"], { name: "Alice", age: 30 });
  const user = await kv.get(["user", "123"]);

  return new Response(JSON.stringify(user.value), {
    headers: { "content-type": "application/json" },
  });
});
```

PostgreSQL

For PostgreSQL databases (both external and provisioned), Deno Deploy automatically injects standard database environment variables into your app's runtime environment:

- `DATABASE_URL`: A full connection string for the current environment, in the format `postgresql://username:password@hostname:port/database`.
- `PGHOST`: The database server hostname.
- `PGPORT`: The database server port.
- `PGDATABASE`: The database name for the current environment.
- `PGUSER`: The database username.
- `PGPASSWORD`: The database password.

If your database requires a custom SSL/TLS certificate, Deno Deploy also injects that certificate into the default certificate store, so that all SSL connections work automatically.

You can use your favorite PostgreSQL client library (such as `pg` from npm) to connect to your database using these environment variables. Most libraries automatically detect and use these standard environment variables without any configuration.

As an example, here's how to connect to PostgreSQL in your Deno Deploy app:

```
import { Pool } from "npm:pg";

// No arguments needed - the library reads connection details from environment variables automatically
const pool = new Pool();

Deno.serve(async () => {
  // Use the database
  const result = await pool.query("SELECT * FROM users WHERE id = $1", [123]);

  return new Response(JSON.stringify(result.rows), {
    headers: { "content-type": "application/json" },
  });
});
```

Running migrations and seeding data

Since each environment has its own isolated database, you will often have many separate databases to manage in every app. It is not practical to manually run migrations or insert seed data into each database individually every time you deploy a new revision.

To streamline this process, Deno Deploy allows you to configure an automated pre-deploy command that runs every time a revision is rolled out to a timeline, before the deployment starts serving traffic.

This command runs with the same environment variables available to your application, including `PGHOST`, `PGPORT`, `PGDATABASE`, etc., so you can use it to run migrations or seed data using your existing migration tools.

To set up an automated migration command, head to the Settings page of your app, and go to the App Config section. There you can edit the app configuration to set a pre-deploy command in the "Pre-Deploy Command" field (for example, `deno task migrate` or `npm run migrate`).

You can see the detailed logs of the pre-deploy command execution in the revision build logs, in the "Deployment" section.

As an example, you could set up a migration script using node-pg-migrate:

1. Add a task to your `deno.json`:

```
{
  "tasks": {
    "migrate": "deno run --allow-net --allow-env --allow-read --allow-write npm:node-pg-migrate up"
```

```
  }
}
```

2. Create a migrations directory and add migration files. For example, `migrations/1234567890_create-users-table.js`:

```
exports.up = (pgm) => {
  pgm.createTable("users", {
    id: "id",
    name: { type: "varchar(100)", notNull: true },
    email: { type: "varchar(100)", notNull: true },
    created_at: {
      type: "timestamp",
      notNull: true,
      default: pgm.func("current_timestamp"),
    },
  });
};
exports.down = (pgm) => {
  pgm.dropTable("users");
};
```

3. Set your pre-deploy command to `deno task migrate` in the app settings.

Deno Deploy will automatically run this command before each deployment, ensuring all your environment-specific databases stay up to date.

Other migration tools such as Prisma Migrate, Drizzle, or Kysely can also be used in a similar way.

Local Development

When developing locally, you have two options for your database setup:

- Use a local database instance running on your machine, for example a local PostgreSQL server or the built-in Deno KV backend in Deno.
- Connect to a hosted isolated local development instance provisioned on Deno Deploy, through `--tunnel`.

Using a local database instance

Deno KV

For Deno KV, you can use the built-in `Deno.openKv()` API to connect to a local Deno KV instance. By default, this uses a local file-based backend stored in your home directory.

```
const kv = await Deno.openKv(); // connects to local Deno KV instance

Deno.serve(async () => {
  // Use the Deno KV instance
  await kv.set(["user", "123"], { name: "Alice", age: 30 });
  const user = await kv.get(["user", "123"]);

  return new Response(JSON.stringify(user.value), {
    headers: { "content-type": "application/json" },
  });
});
```

PostgreSQL

To install PostgreSQL locally, follow the instructions for your operating system on postgresql.org. On macOS, you can use `brew install postgresql`, while many Linux distributions provide PostgreSQL packages through their package managers.

After installing PostgreSQL, create a new database and user for your local development:

```
createdb myapp_dev
createuser myuser --pwprompt
```

Set up a `.env` file in your project root with the connection details for your local PostgreSQL instance:

```
DATABASE_URL=postgresql://myuser:mypassword@localhost:5432/myapp_dev
```

You can then use your favorite PostgreSQL client library (such as pg from npm) to connect to your local database using the `DATABASE_URL` environment variable.

If you are using Deno for local development, you can pass the `--env-file` flag to automatically load environment variables from your `.env` file on startup:

```
deno run -A --env-file main.ts
```

Using a hosted isolated local development instance

Deno Deploy allows you to connect to a hosted isolated local development database when using the `--tunnel` flag in Deno.

To set this up, first provision a database instance in your Deno Deploy organization, and assign it to your app as usual.

Next, start your local development server with the `--tunnel` flag:

```
deno task --tunnel dev
# or
deno run -A --tunnel main.ts
```

Deno Deploy will automatically inject the appropriate database environment variables into your local environment, allowing your code to connect to the hosted isolated database instance. Please note that this database instance is shared across all developers using the same app, so be cautious when modifying data.

> **NOTE**
>
> When using the `--tunnel` flag, Deno Deploy also enables other features such as:
>
> - exporting your logs, traces, and metrics from your local environment to the Deno Deploy dashboard
> - pulling environment variables from the "Local" context to your local environment
> - exposing your local server to a public URL for easy sharing and testing
>
> If you do not want to use these features, consider using a local database instance instead.

Database management

Organization level

You can view and manage your database instances from the "Databases" page in your organization dashboard. Here you will find all your linked and provisioned database instances, along with their status, assigned apps, and connection details (if applicable).

Clicking on a database instance takes you to the database detail page, where you can view more information about the instance, including assigned apps, individual databases created within the instance. For each database provisioned in the instance, you can see its name, status, and associated app and timeline.

If a database fails to create, you'll see an error status with a "Fix" button that you can use to retry the operation and view more detailed error information.

From this page, you can also detach database instances from an app, or delete the instance entirely.

Next to each database in the instance, you can open the database explorer (for PostgreSQL databases), or copy the connection string (`DATABASE_URL`), allowing you to easily connect to your databases from your local machine for debugging.

App level

You can also manage database instances from the "Databases" tab in your app settings. Here you can see the database instance assigned to your app, along with its status and connection details.

You can see all databases created for your app within the assigned instance, including those for production, Git branches, and preview timelines. For each database, you can see its name and status.

You can open the database explorer (for PostgreSQL databases), and copy the connection string (`DATABASE_URL`) for each database directly from this page, allowing you to easily connect to your databases from your local machine for debugging.

Database explorer

Deno Deploy provides a built-in database explorer for PostgreSQL databases, allowing you to browse and manage your data directly from the Deno Deploy dashboard.

To access the database explorer, go to the "Databases" tab in your app settings, find the assigned PostgreSQL database instance, and click "Explore Database". This opens the database explorer interface, where you can view tables, run queries, and manage your data.

> **NOTE**
> The database explorer is not available for Deno KV databases at this time.

Prisma Postgres instances and claiming

When you provision a managed Prisma Postgres database instance through Deno Deploy, the instance is initially "unclaimed". This means that the instance is managed by Deno Deploy on your behalf, but you do not have direct access to the Prisma account that owns the instance. Unclaimed instances have default limits on the number of databases and size based on Deno Deploy's plans.

If you want to have full control over the Prisma Postgres instance, including to upgrade to a higher Prisma plan, you can "claim" the instance into your own Prisma account. To do this, go to the database instance detail page in the Deno Deploy dashboard, and click the "Claim Instance" button. You will be guided through the process of linking the instance to your Prisma account.

> ⚠ **WARNING**
>
> Claiming a Prisma Postgres instance is a permanent action and cannot be undone. Once claimed, the instance is managed through your Prisma account and will make use of your Prisma account limits (based on plan). The Deno Deploy integration to automatically create isolated databases for each app will continue to work as before.

Limits

There is no limit on the number of linked database instances, or Deno KV instances you can create in your organization.

For managed Prisma Postgres instances, only a couple of instances can be created per organization. In each Prisma Postgres instance, there is a limit to the number of databases that can be created, which directly affects the number of apps that can be assigned to that instance. Once a Prisma Postgres instance reaches its database limit, no more databases can be created in that instance.

If a Prisma Postgres instance is "claimed" through your own Prisma account, the database limit is determined by your Prisma plan. For unclaimed Prisma Postgres instances, a default limit applies.

For managed Prisma Postgres instances, each database has a size limit and operation count limit based on the plan you are on. The usage is shown at the bottom of the database instance detail page. If you are on a claimed Prisma Postgres instance, the limits are determined by your Prisma plan. For unclaimed Prisma Postgres instances, a default limit applies.

Frequently Asked Questions

Q: Can multiple apps share the same database instance?

Yes! Multiple apps can be assigned to the same database instance. Each app gets its own isolated databases within that instance.

Q: What happens to my data when I remove an app assignment?

The databases remain on your database server. Only the connection between your app and the database instance is removed.

Q: Can I use the same database for multiple environments?

By default, each environment (production, branch, preview) gets its own database to ensure isolation and prevent data conflicts. However, you can customize the database your code connects to using options in your database library.

Q: How do I access my databases directly?

You can connect directly to your database server using the connection details you provided. Use the database names shown in the Deno Deploy dashboard.

Q: Can I change database connection details?

Yes, click "Edit" on any database instance to update connection details. Test the connection before saving to ensure it works.

Q: How do I delete a database instance?

First remove all app assignments, then click "Delete" on the database instance. This only removes the connection from Deno Deploy - your actual database server is not affected.

Cloud Connections

Deno Deploy allows you to connect to cloud providers like AWS and Google Cloud Platform (GCP) without needing to manually manage static credentials. This is done through the use of OpenID Connect (OIDC) and identity federation.

How it works

Deno Deploy is an OIDC provider. Every running application of Deno Deploy can be issued short-lived JWT tokens that are signed by Deno Deploy. These tokens contain information about the application, such as the organization and application ids and slugs, the context in which an application is executing, and the running revision ID. Learn more about OIDC in Deno Deploy.

By sending these tokens to AWS or GCP, one can exchange them for short-lived AWS or GCP credentials that can be used to access cloud resources such as AWS S3 buckets or Google Cloud Spanner instances. When sending the token to AWS or GCP, the token is verified by the cloud provider, which checks that it was issued by Deno Deploy and that it is valid for the application and context that should be allowed to access the cloud resources.

To enable AWS or GCP to exchange OIDC tokens for credentials, the cloud provider needs to be configured to trust Deno Deploy as an OIDC identity provider, and an AWS IAM role or GCP service account needs to be created that allows the exchange of tokens for credentials, for a specific Deno Deploy application.

Environment variables set by a connection

When a cloud connection applies to the context your application is running in, Deno Deploy sets the environment variables the cloud SDKs look for, so your code does not have to configure credentials itself:

- AWS: `AWS_ROLE_ARN` and `AWS_WEB_IDENTITY_TOKEN_FILE`
- GCP: `GOOGLE_APPLICATION_CREDENTIALS`, `GCP_WORKLOAD_PROVIDER_ID`, `GCP_SERVICE_ACCOUNT_EMAIL`, and `GCP_PROJECT_ID`

Deno Deploy sets these, and they take precedence over an environment variable you have defined with the same name, in every scope. In contexts where no connection applies, these names are ordinary environment variables that you are free to use for your own purposes.

Setting up AWS

This guide contains three guides for setting up these AWS resources. You can use any of these to set up the AWS resources.

- Using the `deno deploy setup-aws` command from your local machine (recommended)
- Using the `aws` CLI
- Using the AWS Console
- Using Terraform

To set up AWS with Deno Deploy, the following resources need to be created inside of your AWS account:

- An AWS IAM OIDC Identity Provider that trusts Deno Deploy as an OIDC provider.
  - The OIDC provider URL is `https://oidc.deno.com`.

    - The audience (client ID) is `sts.amazonaws.com`.
- An AWS IAM Role that can be "assumed" (signed into) using a Deno Deploy OIDC token.
    - The trust policy of the role should allow the OIDC provider to assume the role, such as:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::<account-id>:oidc-provider/oidc.deno.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "oidc.deno.com:aud": "sts.amazonaws.com",
          "oidc.deno.com:sub": "deployment:<organization-slug>/<application-slug>/<context-name>"
        }
      }
    }
  ]
}
```

    - The role should have permissions to access the AWS resources you want to use, such as S3 buckets or DynamoDB tables.

After setting up the AWS resources, navigate to the AWS cloud integration setup page from the app settings. There you must select the context(s) in which the cloud connection should be available.

Then you must enter the ARN (Amazon Resource Name) for the AWS IAM Role created earlier. After entering the ARN you can start a connection test by pressing the "Test connection" button. The connection test will check that the AWS IAM Role and OIDC provider are configured correctly, and does not allow access from apps, orgs, or contexts that should not have access.

After testing the connection, you can save the cloud connection.

Usage

After setting up a cloud connection between AWS and Deno Deploy you can access AWS resources such as S3 directly from your application code, without having to configure any credentials.

The AWS SDK v3 automatically picks up on the cloud connection configuration. Here is an example of accessing an S3 bucket from a Deno Deploy application with a configured AWS account.

```
import { ListBucketsCommand, S3Client } from "@aws-sdk/client-s3";

const s3 = new S3Client({ region: "us-west-2" });

Deno.serve(() => {
  const { Buckets } = await s3.send(new ListBucketsCommand({}));
  return Response.json(Buckets);
});
```

Setting up GCP

To set up GCP with Deno Deploy, the following resources need to be created inside of your GCP account:

- A Workload Identity Pool and Workload Identity Provider that trusts Deno Deploy as an OIDC provider.
  - The OIDC provider URL is `https://oidc.deno.com`.
  - The audience should be the default (starts with `https://iam.googleapis.com`).
  - At least the following attribute mappings must be set:
    - `google.subject = assertion.sub`
    - `attribute.full_slug = assertion.org_slug + "/" + assertion.app_slug`
- A Service account that can be "impersonated" (signed into) using the OIDC token.
  - A principal or principal set from the workload identity pool should have access to the service account using the Workload Identity User role (`roles/iam.workloadIdentityUser`). Examples:
    - A specific context in an app: `principal://iam.googleapis.com/projects/<PROJECT_NUMBER>/locations/global/workloadIdentityPools/oidc-deno-com/subject/deployment:<ORG_SLUG>/<APP_SLUG>/<CONTEXT_NAME>`
    - All contexts in an app: `principalSet://iam.googleapis.com/projects/<PROJECT_NUMBER>/locations/global/workloadIdentityPools/oidc-deno-com/attribute.full_slug/<ORG_SLUG>/<APP_SLUG>`
  - The service account should have access to the GCP resources you want to use, such as a Google Cloud Storage bucket.

This guide contains three guides for setting up these GCP resources. You can use any of these to set up the GCP resources.

- Using the `deno deploy setup-gcp` command from your local machine (recommended)
- Using the `gcloud` CLI
- Using the GCP Console
- Using Terraform

After setting up the GCP resources, navigate to the GCP cloud integration setup page from the app settings. There you must select the context(s) in which the cloud connection should be available.

Then you must enter the workload identity provider ID, in the form `projects/<PROJECT_NUMBER>/locations/global/workloadIdentityPools/oidc-deno-com/providers/oidc-deno-com`, and the email address of the GCP Service Account created earlier. After entering the email address you can start a connection test by pressing the "Test connection" button. The connection test will check that the GCP Service Account and OIDC provider are configured correctly, and does not allow access from apps, orgs, or contexts that should not have access.

After testing the connection, you can save the cloud connection.

Usage

After setting up a cloud connection between GCP and Deno Deploy you can access GCP resources such as Cloud Storage directly from your application code, without having to configure any credentials.

The Google Cloud SDK automatically picks up on the cloud connection configuration. Here is an example of accessing a Cloud Storage bucket from a Deno Deploy application with a configured GCP account.

```
import { Storage } from "@google-cloud/storage";

const storage = new Storage();
```

```
Deno.serve(() => {
  const [buckets] = await storage.getBuckets();
  return Response.json(buckets);
});
```

Removing a cloud integration

You can remove a cloud connection by pressing the "Delete" button in the cloud integration section, next to a specific cloud connection.

Setup Guides

AWS: Easy setup with `deno deploy setup-aws`

For instructions on how to set up AWS with Deno Deploy using the `deno deploy setup-aws` command, please see the instructions on the AWS cloud integration setup page in your app settings.

AWS: Using the `aws` CLI

You can manually set up AWS resources using the AWS CLI. This requires having the AWS CLI installed and configured with appropriate permissions to create IAM roles, OIDC providers, and attach policies.

Prerequisites

- AWS CLI installed and configured
- Permissions to create IAM roles, OIDC providers, and attach policies

Step 1: Create OIDC Provider

First, create the OIDC provider if it doesn't already exist:

```
aws iam create-open-id-connect-provider \
  --url https://oidc.deno.com \
  --client-id-list sts.amazonaws.com
```

Step 2: Create IAM Role with Trust Policy

Create a trust policy file that allows your Deno Deploy application to assume the role. You can choose between allowing access to all contexts or specific contexts only.

For all contexts in your app:

```
# Create trust policy file for entire app
cat > trust-policy-all-contexts.json << EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::YOUR_ACCOUNT_ID:oidc-provider/oidc.deno.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
```

```
        "StringLike": {
          "oidc.deno.com:sub": "deployment:YOUR_ORG/YOUR_APP/*"
        }
      }
    }
  ]
}
EOF
```

For specific contexts only:

```
# Create trust policy file for specific contexts
cat > trust-policy-specific-contexts.json << EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "arn:aws:iam::YOUR_ACCOUNT_ID:oidc-provider/oidc.deno.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "oidc.deno.com:sub": [
            "deployment:YOUR_ORG/YOUR_APP/production",
            "deployment:YOUR_ORG/YOUR_APP/staging"
          ]
        }
      }
    }
  ]
}
EOF
```

Step 3: Create the IAM Role

Create the role using the appropriate trust policy:

```
# For entire app
aws iam create-role \
  --role-name DenoDeploy-YourOrg-YourApp \
  --assume-role-policy-document file://trust-policy-all-contexts.json

# OR for specific contexts
aws iam create-role \
  --role-name DenoDeploy-YourOrg-YourApp \
  --assume-role-policy-document file://trust-policy-specific-contexts.json
```

Step 4: Attach Policies

Attach the necessary policies to grant permissions for the AWS resources your application needs:

```
aws iam attach-role-policy \
  --role-name DenoDeploy-YourOrg-YourApp \
  --policy-arn arn:aws:iam::aws:policy/POLICY_NAME
```

Replace `POLICY_NAME` with the appropriate AWS policies (e.g., `AmazonS3ReadOnlyAccess`, `AmazonDynamoDBReadOnlyAccess`, etc.) based on your requirements.

After completing these steps, use the Role ARN in your Deno Deploy cloud connection configuration.

AWS: Using the AWS Console

You can set up AWS resources using the AWS Management Console web interface. This method provides a visual way to configure the necessary IAM resources.

Step 1: Create OIDC Identity Provider

1. **Navigate to IAM Console** → Identity providers
2. **Create OIDC Provider**:
   - Click "Add provider"
   - Select "OpenID Connect"
   - Provider URL: `https://oidc.deno.com`
   - Audience: `sts.amazonaws.com`
   - Click "Add provider"

Step 2: Create IAM Role

1. **Navigate to IAM Console** → Roles
2. **Create role**:
   - Click "Create role"
   - Trusted entity type: **Web identity**
   - Identity provider: Select the created OIDC provider (`oidc.deno.com`)
   - Audience: `sts.amazonaws.com`

Step 3: Configure Trust Policy Conditions

Add a condition to restrict which Deno Deploy applications can assume this role. Choose one approach:

For all contexts in your app:

- Condition key: `oidc.deno.com:sub`
- Operator: `StringLike`
- Value: `deployment:YOUR_ORG/YOUR_APP/*`

For specific contexts only:

- Condition key: `oidc.deno.com:sub`
- Operator: `StringEquals`
- Value: `deployment:YOUR_ORG/YOUR_APP/production`
- Add additional conditions for each context (e.g., staging, development)

Click "Next" to continue.

Step 4: Attach Permissions Policies

1. Search and select appropriate policies based on your needs:
   - For S3 access: `AmazonS3ReadOnlyAccess` or `AmazonS3FullAccess`

- For DynamoDB access: `AmazonDynamoDBReadOnlyAccess` or `AmazonDynamoDBFullAccess`
- For other services: Select relevant policies

2. Click "Next"

Step 5: Name and Create Role

1. **Role name**: `DenoDeploy-YourOrg-YourApp` (replace with your actual organization and app names)
2. **Description**: Optional description of the role's purpose
3. Review the trust policy and permissions
4. Click "Create role"

Step 6: Copy Role ARN

After creating the role:

1. Go to the role details page
2. Copy the Role ARN (it looks like `arn:aws:iam::123456789012:role/DenoDeploy-YourOrg-YourApp`)
3. Use this ARN in your Deno Deploy cloud connection configuration

AWS: Using Terraform

You can use Terraform to programmatically create the AWS resources needed for cloud connections. This approach is ideal for infrastructure-as-code workflows.

Terraform Configuration

Create a Terraform configuration file with the following content:

```
# Variables
variable "org" {
  description = "Deno Deploy organization name"
  type        = string
}

variable "app" {
  description = "Deno Deploy app name"
  type        = string
}

variable "contexts" {
  description = "List of specific contexts to allow (leave empty for all contexts)"
  type        = list(string)
  default     = []
}

# OIDC Provider
resource "aws_iam_openid_connect_provider" "deno_deploy" {
  url = "https://oidc.deno.com"
  client_id_list = ["sts.amazonaws.com"]
}

# IAM Role with dynamic trust policy based on contexts
resource "aws_iam_role" "deno_deploy_role" {
  name = "DenoDeploy-${var.org}-${var.app}"
```

```
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Principal = {
          Federated = aws_iam_openid_connect_provider.deno_deploy.arn
        }
        Action = "sts:AssumeRoleWithWebIdentity"
        Condition = length(var.contexts) > 0 ? {
          # Specific contexts only
          StringEquals = {
            "oidc.deno.com:sub" = [
              for context in var.contexts : "deployment:${var.org}/${var.app}/${context}"
            ]
          }
        } : {
          # All contexts (wildcard)
          StringLike = {
            "oidc.deno.com:sub" = "deployment:${var.org}/${var.app}/*"
          }
        }
      }
    ]
  })
}

# Attach policies
resource "aws_iam_role_policy_attachment" "example" {
  role       = aws_iam_role.deno_deploy_role.name
  policy_arn = "arn:aws:iam::aws:policy/POLICY_NAME"
}

# Output the role ARN
output "role_arn" {
  value = aws_iam_role.deno_deploy_role.arn
}
```

Usage Examples

For entire app access (all contexts):

```
module "deno_deploy_aws" {
  source = "./path-to-terraform-module"

  org      = "your-org"
  app      = "your-app"
  contexts = []  # Empty list allows all contexts
}
```

For specific contexts only:

```
module "deno_deploy_aws" {
  source = "./path-to-terraform-module"

  org      = "your-org"
  app      = "your-app"
  contexts = ["production", "staging"]
}
```

Applying the Configuration

1. Initialize Terraform:

```
terraform init
```

2. Plan the deployment:

```
terraform plan
```

3. Apply the configuration:

```
terraform apply
```

After applying, Terraform will output the Role ARN that you can use in your Deno Deploy cloud connection configuration.

Customizing Policies

Replace `POLICY_NAME` in the `aws_iam_role_policy_attachment` resource with the appropriate AWS managed policies or create custom policies based on your requirements. You can add multiple policy attachments by creating additional `aws_iam_role_policy_attachment` resources.

GCP: Easy setup with `deno deploy setup-gcp`

For instructions on how to set up GCP with Deno Deploy using the `deno deploy setup-gcp` command, please see the instructions on the Google cloud integration setup page in your app settings.

GCP: Using the `gcloud` CLI

You can manually set up GCP resources using the gcloud CLI. This requires having the gcloud CLI installed and authenticated with appropriate permissions to create workload identity pools, service accounts, and grant IAM roles.

Prerequisites

- gcloud CLI installed and authenticated
- Access to create workload identity pools, service accounts, and grant IAM roles
- Required APIs enabled:
  - `iam.googleapis.com`
  - `iamcredentials.googleapis.com`
  - `sts.googleapis.com`

Step 1: Enable Required APIs

First, enable the required APIs for your project:

```
gcloud services enable iam.googleapis.com
gcloud services enable iamcredentials.googleapis.com
gcloud services enable sts.googleapis.com
```

Step 2: Create Workload Identity Pool

Create a workload identity pool to manage external identities:

```
gcloud iam workload-identity-pools create oidc-deno-com \
  --location=global \
  --display-name="Deno Deploy Workload Identity Pool"
```

Step 3: Create Workload Identity Provider

Configure the OIDC provider within the workload identity pool:

```
gcloud iam workload-identity-pools providers create-oidc oidc-deno-com \
  --workload-identity-pool=oidc-deno-com \
  --location=global \
  --issuer-uri=https://oidc.deno.com \
  --attribute-mapping="google.subject=assertion.sub,attribute.org_slug=assertion.org_slug,attribute.app_slug=as
sertion.app_slug,attribute.full_slug=assertion.org_slug+\"/\"+assertion.app_slug"
```

Step 4: Create Service Account

Create a service account that will be used by your Deno Deploy application:

```
gcloud iam service-accounts create deno-your-org-your-app \
  --display-name="Deno Deploy YourOrg/YourApp"
```

Step 5: Configure Workload Identity Binding

Get your project number and configure the workload identity binding. Choose between allowing access to all contexts or specific contexts only.

```
# Get project number
PROJECT_NUMBER=$(gcloud projects describe PROJECT_ID --format="value(projectNumber)")
```

For all contexts in your app:

```
gcloud iam service-accounts add-iam-policy-binding \
  deno-your-org-your-app@PROJECT_ID.iam.gserviceaccount.com \
  --role=roles/iam.workloadIdentityUser \
  --member="principalSet://iam.googleapis.com/projects/$PROJECT_NUMBER/locations/global/workloadIdentityPools/
oidc-deno-com/attribute.full_slug/YOUR_ORG/YOUR_APP"
```

For specific contexts only:

```
# Bind for production context
gcloud iam service-accounts add-iam-policy-binding \
  deno-your-org-your-app@PROJECT_ID.iam.gserviceaccount.com \
  --role=roles/iam.workloadIdentityUser \
  --member="principal://iam.googleapis.com/projects/$PROJECT_NUMBER/locations/global/workloadIdentityPools/
oidc-deno-com/subject/deployment:YOUR_ORG/YOUR_APP/production"

# Bind for staging context
gcloud iam service-accounts add-iam-policy-binding \
  deno-your-org-your-app@PROJECT_ID.iam.gserviceaccount.com \
  --role=roles/iam.workloadIdentityUser \
  --member="principal://iam.googleapis.com/projects/$PROJECT_NUMBER/locations/global/workloadIdentityPools/
oidc-deno-com/subject/deployment:YOUR_ORG/YOUR_APP/staging"

# Add more bindings for each specific context as needed
```

Step 6: Grant Roles to Service Account

Grant the necessary roles to the service account for accessing GCP resources:

```
gcloud projects add-iam-policy-binding PROJECT_ID \
  --member="serviceAccount:deno-your-org-your-app@PROJECT_ID.iam.gserviceaccount.com" \
  --role="roles/ROLE_NAME"
```

Replace `ROLE_NAME` with appropriate roles such as:

- `roles/storage.objectViewer` for Cloud Storage read access
- `roles/storage.objectAdmin` for Cloud Storage full access
- `roles/cloudsql.client` for Cloud SQL access
- Other roles based on your requirements

Step 7: Get Required Values

After completing the setup, you'll need two values for your Deno Deploy configuration:

1. **Workload Provider ID**: `projects/PROJECT_NUMBER/locations/global/workloadIdentityPools/oidc-deno-com/providers/oidc-deno-com`
2. **Service Account Email**: `deno-your-org-your-app@PROJECT_ID.iam.gserviceaccount.com`

Use these values in your Deno Deploy cloud connection configuration.

GCP: Using the GCP Console

You can set up GCP resources using the Google Cloud Console web interface. This method provides a visual way to configure workload identity federation and service accounts.

Step 1: Enable Required APIs

1. **Navigate to APIs & Services** → Library
2. Search for and enable the following APIs:
   - "Identity and Access Management (IAM) API"
   - "IAM Service Account Credentials API"
   - "Security Token Service API"

Step 2: Create Workload Identity Pool

1. **Navigate to IAM & Admin** → Workload Identity Federation
2. **Create Pool**:
   - Click "Create Pool"
   - Pool name: `Deno Deploy Workload Id Pool`
   - Pool ID: `oidc-deno-com`
   - Click "Continue"

Step 3: Add Provider to Pool

1. **Add a provider**:
   - Click "Add a provider"
   - Provider type: **OpenID Connect (OIDC)**

- Provider name: `Deno Deploy OIDC Provider`
- Provider ID: `oidc-deno-com`
- Issuer URL: `https://oidc.deno.com`

2. **Configure attribute mappings**:
   - `google.subject` → `assertion.sub`
   - `attribute.org_slug` → `assertion.org_slug`
   - `attribute.app_slug` → `assertion.app_slug`
   - `attribute.full_slug` → `assertion.org_slug + "/" + assertion.app_slug`
3. Click "Save"

Step 4: Create Service Account

1. **Navigate to IAM & Admin** → Service Accounts
2. **Create Service Account**:
   - Click "Create Service Account"
   - Service account name: `deno-your-org-your-app`
   - Service account ID: `deno-your-org-your-app`
   - Description: `Service account for Deno Deploy project your-org/your-app`
   - Click "Create and Continue"

Step 5: Grant Roles to Service Account

1. Select appropriate roles based on your needs:
   - For Cloud Storage: `Storage Object Viewer` or `Storage Admin`
   - For Cloud SQL: `Cloud SQL Client`
   - For other services: Select relevant roles
2. Click "Continue" then "Done"

Step 6: Configure Workload Identity Binding

1. **Go back to the created service account**
2. Click on the "Principals with access" tab
3. Click "Grant Access"
4. Configure principals - choose one approach:

   For all contexts in your app: * New principals: `principalSet://iam.googleapis.com/projects/YOUR_PROJECT_NUMBER/locations/global/workloadIdentityPools/oidc-deno-com/attribute.full_slug/YOUR_ORG/YOUR_APP` + **For specific contexts only:** * New principals: `principal://iam.googleapis.com/projects/YOUR_PROJECT_NUMBER/locations/global/workloadIdentityPools/oidc-deno-com/subject/deployment:YOUR_ORG/YOUR_APP/production` * Repeat for each context (staging, etc.)
5. Role: **Workload Identity User**
6. Click "Save"

Step 7: Get Required Values

You'll need two values for your Deno Deploy configuration:

1. **Workload Provider ID**:
   - Navigate back to Workload Identity Federation
   - Click on your pool, then your provider
   - Copy the provider resource name (full path starting with `projects/`)
2. **Service Account Email**: Copy from the service account details page

Step 8: Verify Configuration

The final workload identity pool overview should show:

- Your pool with the OIDC provider
- The connected service account
- Proper bindings configured

Use the Service Account Email and Workload Provider ID in your Deno Deploy cloud connection configuration.

GCP: Using Terraform

You can use Terraform to programmatically create the GCP resources needed for cloud connections. This approach is ideal for infrastructure-as-code workflows.

Terraform Configuration

Create a Terraform configuration file with the following content:

```
# Variables
variable "org" {
  description = "Deno Deploy organization name"
  type        = string
}

variable "app" {
  description = "Deno Deploy app name"
  type        = string
}

variable "contexts" {
  description = "List of specific contexts to allow (leave empty for all contexts)"
  type        = list(string)
  default     = []
}

variable "project_id" {
  description = "GCP Project ID"
  type        = string
}

variable "roles" {
  description = "List of IAM roles to grant to the service account"
  type        = list(string)
```

```
  default     = []
}

# Data source for project information
data "google_project" "project" {
  project_id = var.project_id
}

# Workload Identity Pool
resource "google_iam_workload_identity_pool" "deno_deploy" {
  workload_identity_pool_id = "oidc-deno-com"
  display_name              = "Deno Deploy Workload Id Pool"
}

# Workload Identity Provider
resource "google_iam_workload_identity_pool_provider" "deno_deploy" {
  workload_identity_pool_id          = google_iam_workload_identity_pool.deno_deploy.workload_identity_pool_id
  workload_identity_pool_provider_id = "oidc-deno-com"
  display_name                       = "Deno Deploy OIDC Provider"

  attribute_mapping = {
    "google.subject"        = "assertion.sub"
    "attribute.org_slug"    = "assertion.org_slug"
    "attribute.app_slug"    = "assertion.app_slug"
    "attribute.full_slug"   = "assertion.org_slug + \"/\" + assertion.app_slug"
  }

  oidc {
    issuer_uri = "https://oidc.deno.com"
  }
}

# Service Account
resource "google_service_account" "deno_deploy" {
  account_id   = "deno-${var.org}-${var.app}"
  display_name = "Deno Deploy ${var.org}/${var.app}"
}

# Workload Identity Binding - dynamic based on contexts
resource "google_service_account_iam_binding" "workload_identity" {
  service_account_id = google_service_account.deno_deploy.name
  role               = "roles/iam.workloadIdentityUser"

  members = length(var.contexts) > 0 ? [
    # Specific contexts only
    for context in var.contexts :
    "principal://iam.googleapis.com/projects/${data.google_project.project.number}/locations/global/
workloadIdentityPools/${google_iam_workload_identity_pool.deno_deploy.workload_identity_pool_id}/subject/
deployment:${var.org}/${var.app}/${context}"
  ] : [
    # All contexts (using attribute mapping)
    "principalSet://iam.googleapis.com/projects/${data.google_project.project.number}/locations/global/
workloadIdentityPools/${google_iam_workload_identity_pool.deno_deploy.workload_identity_pool_id}/
attribute.full_slug/${var.org}/${var.app}"
  ]
}

# Grant roles to service account
resource "google_project_iam_member" "service_account_roles" {
  for_each = toset(var.roles)
  project  = var.project_id
  role     = each.value
  member   = "serviceAccount:${google_service_account.deno_deploy.email}"
}
```

```
# Outputs
output "workload_provider_id" {
  value = "projects/${data.google_project.project.number}/locations/global/workloadIdentityPools/
${google_iam_workload_identity_pool.deno_deploy.workload_identity_pool_id}/providers/
${google_iam_workload_identity_pool_provider.deno_deploy.workload_identity_pool_provider_id}"
}

output "service_account_email" {
  value = google_service_account.deno_deploy.email
}
```

Usage Examples

For entire app access (all contexts):

```
module "deno_deploy_gcp" {
  source = "./path-to-terraform-module"

  org        = "your-org"
  app        = "your-app"
  project_id = "your-gcp-project-id"
  contexts   = []  # Empty list allows all contexts
  roles      = [
    "roles/storage.objectViewer",
    "roles/cloudsql.client"
  ]
}
```

For specific contexts only:

```
module "deno_deploy_gcp" {
  source = "./path-to-terraform-module"

  org        = "your-org"
  app        = "your-app"
  project_id = "your-gcp-project-id"
  contexts   = ["production", "staging"]
  roles      = [
    "roles/storage.objectAdmin",
    "roles/cloudsql.client"
  ]
}
```

Applying the Configuration

1. Initialize Terraform:

   ```
   terraform init
   ```

2. Plan the deployment:

   ```
   terraform plan
   ```

3. Apply the configuration:

   ```
   terraform apply
   ```

After applying, Terraform will output the Workload Provider ID and Service Account Email that you can use in your Deno Deploy cloud connection configuration.

Customizing Roles

The `roles` variable accepts a list of GCP IAM roles. Common roles include:

- `roles/storage.objectViewer` - Read access to Cloud Storage
- `roles/storage.objectAdmin` - Full access to Cloud Storage objects
- `roles/cloudsql.client` - Access to Cloud SQL instances
- `roles/secretmanager.secretAccessor` - Access to Secret Manager secrets
- Custom roles can also be specified

OIDC

Deno Deploy is an OIDC provider. Every running application of Deno Deploy can be issued short-lived JWT tokens that are signed by Deno Deploy. These tokens contain information about the application, such as the organization and application ids and slugs, the context in which an application is executing, and the running revision ID.

The tokens can be used to authenticate with third-party services that support OIDC authentication, such as major cloud providers, but also HashiCorp Vault, NPM, and others.

> **TIP**
> Do you want to use OIDC tokens to authenticate with AWS or Google Cloud? Use the Cloud Connections feature instead of manually configuring OIDC authentication. Cloud Connections handle the entire configuration for you, including setting up trust relationships and permissions. OIDC is used under the hood.

Issuing Tokens

To issue a token for the currently running application, use the `getIdToken()` function from the `@deno/oidc` module on JSR.

First, install `@deno/oidc` as a dependency of your application:

```
deno add jsr:@deno/oidc
```

Then, import the `getIdToken()` function and call it with the desired audience:

```
import { getIdToken } from "jsr:@deno/oidc";

const token = await getIdToken("https://example.com/");
console.log(token);
```

The `audience` parameter is a string that identifies the intended recipient of the token. It is typically a URL or an identifier that represents the service or application that will consume the token. The audience value must match the value configured in the third-party service that you want to authenticate with. It will be placed into the `aud` claim of the issued JWT token.

The `getIdToken()` function returns a promise that resolves to a JWT token as a string.

To check whether your current environment supports OIDC (i.e. whether your application is running on Deno Deploy), you can use the `supportsIssuingIdTokens` namespaced property:

```
import { supportsIssuingIdTokens } from "jsr:@deno/oidc";

if (supportsIssuingIdTokens) {
  // OIDC is supported
} else {
  // OIDC is not supported
}
```

Token Structure

The issued tokens are JWT tokens that are signed using the RS256 algorithm. The tokens contain the following claims:

Claim Name	Example Value	Description
iss	https://oidc.deno.com	The issuer of the token, which is always https://oidc.deno.com.
aud	https://example.com/	The audience of the token, which is the value passed to the getIdToken() function.
iat	1757924011	The issued-at time of the token, which is a Unix timestamp indicating when the token was issued.
exp	1757924311	The expiration time of the token, which is a Unix timestamp indicating when the token will expire.
nbf	1757923951	The not-before time of the token, which is a Unix timestamp indicating when the token becomes valid.
sub	deployment:deno/astro-app/production	The subject of the token, which is a string concatenation of deployment:<org>/<app>/<context>
org_id	729adb8f-20d6-4b09-bb14-fac14cb260d1	The unique identifier of the organization that owns the application.
org_slug	deno	The slug of the organization that owns the application.
app_id	16ad21d8-7aeb-4155-8aa3-9f58df87cd3e	The unique identifier of the application.
app_slug	astro-app	The slug of the application.
context_id	1d685676-92d7-418d-b103-75b46f1a58b4	The unique identifier of the context in which the application is running.
context_name	production	The context in which the application is running.
revision_id	rh2r15rgy802	The unique identifier of the revision of the application that is currently running.
deployment_id	<random string>	A unique hash containing the entire deployment metadata, including the application, revision, and context IDs.

Tokens expire 5 minutes after they are issued. To account for clock skew, the tokens nbf claim is set to 1 minute before the iat claim.

Verifying Tokens

To verify the tokens issued by Deno Deploy, you need to fetch the public keys from the OIDC provider's JWKS endpoint. The JWKS endpoint for Deno Deploy is:

```
https://oidc.deno.com/.well-known/jwks.json
```

Use the `kid` (key ID) from the JWT token header to select the correct key from the JWKS response.

Deno Deploy also provides a standard OIDC discovery document at:

```
https://oidc.deno.com/.well-known/openid-configuration
```

Deno Deploy rotates its signing keys periodically. Therefore, it is important to fetch the JWKS keys dynamically from the JWKS endpoint rather than hardcoding them.

Currently, Deno Deploy signing keys use the `ES256` algorithm. This may change in the future, depending on security requirements, best practices, and support in third-party services.

To verify the tokens, you can use a JWT library that supports OIDC and JWKS. In TypeScript, you can use the jose library.

Runtime

In Deno Deploy, all applications execute using a standard Deno runtime in a secure, isolated Linux environment.

The Deno runtime used in Deno Deploy is the standard Deno runtime, with full support for all features of the Deno CLI, including JSR and NPM dependencies, reading and writing to the file system, making network requests, spawning subprocesses, and loading FFI and node native addons.

The Deno runtime runs using `--allow-all` permissions.

Custom flags, including `--unstable-*` flags, cannot be passed to the Deno runtime.

Runtime environment

The runtime environment is a Linux-based environment running either x64 or ARM64 architecture. The exact set of tools available in the runtime environment is subject to change and thus cannot be relied upon.

Currently Deno Deploy runs on Deno 2.5.0

Lifecycle

Deno Deploy runs applications in a serverless environment. This means that an application is not always running and is only started when a request is received. When no incoming traffic is received for a period of time, the application is stopped.

Applications can be started and stopped at any time. They should start quickly to respond to incoming requests without delay.

Multiple instances of the same application can run simultaneously. For example, one instance could be running in the US and another in Europe. Each instance is completely isolated from the others and they do not share CPU, memory, or disk resources. Multiple instances can also start in the same region when needed, such as to handle high traffic or during infrastructure updates.

Startup

When the system decides to start an application, it provisions a new sandbox environment for the application. This environment is isolated from all other applications.

It then starts the application using the configured entrypoint and waits for the HTTP server to start. If the application crashes before the HTTP server starts, the request that triggered the start will fail with a 502 Bad Gateway error.

Once the application is started, incoming requests are routed to it and responses are sent back to the client.

Shutdown

The application remains alive until no new incoming requests are received or responses (including response body bytes) are sent for a period of time. The exact timeout is between 5 seconds and 10 minutes. WebSocket connections that actively transmit data (including ping/pong frames) also keep the application alive.

Once the system decides to stop the application, it sends a `SIGINT` signal to the application as a trigger to shut down. From this point on, the application has 5 seconds to shut down gracefully before it will be forcibly killed with a `SIGKILL` signal.

Eviction

Sometimes an isolate may shut down even if the application is actively receiving traffic. Some examples of when this can happen are:

- An application was scaled up to handle load, but the load has decreased enough to be handled by a single instance again.
- The underlying server executing the instance is too resource constrained to continue running this application instance.
- The underlying infrastructure is being updated or has experienced a failure.

When the system decides to evict an application, it attempts to divert traffic away from the instance being evicted as early as possible. Sometimes this means that a request will wait for a new instance to boot up even though an existing instance is already running.

When an application only serves requests that finish quickly, evictions are usually unnoticeable. For applications that serve long-running requests or WebSockets, evictions can be more noticeable because the application may need to be evicted while still processing a request. The system will try to avoid these scenarios, but it is not always possible.

After traffic has been diverted away from the old instance, the system sends a `SIGINT` signal to trigger a graceful shutdown. The application should finish processing any remaining requests quickly and shut down websockets and other long-running connections. Clients making long-running requests should be prepared to handle these disruptions and reconnect when disconnected.

5 seconds after the `SIGINT` signal is sent, the old instance will be forcibly killed with a `SIGKILL` signal if it has not already shut down gracefully.

Cold starts

Because applications are not always running, they may need to start when a request is received. This is called a cold start. Cold starts in Deno Deploy are highly optimized and complete within 100 milliseconds for hello world applications, and within a few hundred milliseconds for larger applications.

Deno Deploy uses multiple optimizations to enable fast cold starts:

- Linux microVMs and the Deno runtime are pre-provisioned to ensure they don't need to be created from scratch when starting an application.
- Applications start immediately when the client sends the first TCP packet to establish a TLS connection. For fast-starting applications, depending on the network round trip latency, the application may already be running before the client sends the HTTP request.
- File system access is optimized for frequently used startup files. Deno Deploy analyzes file access patterns during the build step's warmup phase and optimizes the file system for faster access.

When cold starts are slow, they can negatively impact user experience. To optimize your application for quick startup:

1. Minimize dependencies used by your application.

2. Load infrequently accessed code and dependencies lazily using dynamic `import()`.
3. Minimize I/O operations during startup, especially top-level `await` operations and network requests.

If your application starts slowly, please contact Deno support for help investigating the issue.

Frameworks

Deno Deploy supports a number of JavaScript and TypeScript frameworks out of the box. This means that you can use these frameworks without any additional configuration or setup.

Natively supported frameworks are tested to work with Deno Deploy and are automatically detected when you create a new app. Deno Deploy automatically optimizes the build and runtime configuration for these frameworks to be as optimal as possible.

Frameworks not listed here are still likely to work, but may require manually configuring the install and/or build command and the runtime configuration in the build settings.

Feel like a framework is missing? Let us know in the Deno Deploy Discord channel or contact Deno support.

Supported frameworks

Next.js (`nextjs`)

Next.js is a React framework for building full-stack web applications. You use React Components to build user interfaces, and Next.js for additional features and optimizations.

Both pages and app router are supported out of the box. ISR, SSG, SSR, and PPR are supported. Caching is supported out of the box, including using the new `"use cache"`.

`next/image` works out of the box.

Next.js on Deno Deploy always builds in standalone mode.

Tracing is supported out of the box, and Next.js automatically emits some spans for incoming requests, routing, rendering, and other operations.

Deno compatibility notes

Next.js projects run on Deno Deploy through Deno's Node and npm compatibility layer. Most Next.js apps work without extra changes, but the following issues can show up in projects generated by scaffolding tools or coding agents:

- Prefer `next.config.mjs` with `export default` for the broadest Next.js version compatibility. If you are using Next.js 15 or newer, `next.config.ts` with `export default` is also supported. Deno's `detect-cjs` option helps CommonJS `next.config.js` files, but some Next.js versions and loaders can still report a "module is not defined" error when a config is loaded as ESM. Using an ESM config avoids that ambiguity.
- Some Next.js versions can crash during the lint/type-check phase under Deno with `Cannot read properties of undefined (reading 'bold')`. Until this is fixed upstream, you can unblock the build by setting `typescript.ignoreBuildErrors` and `eslint.ignoreDuringBuilds` in your Next.js config. Because this skips Next.js's build-time checks, keep running type checking and linting separately in CI.
- App Router pages are still prerendered at build time when Next.js decides they are static. If a client component calls browser-only hooks or APIs, defer that code until after the component mounts, such as in `useEffect`. If you need to disable SSR with `next/dynamic` and `{ ssr: false }`, put the dynamic import in a Client Component wrapper; Server Components cannot use `{ ssr: false }` directly. Marking the route dynamic is not enough: dynamic routes still render on the server, where browser globals are undefined.

Example `next.config.mjs`:

next.config.mjs

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  typescript: {
    // Workaround for known Next.js-on-Deno build-time type-check crashes.
    ignoreBuildErrors: true,
  },
  eslint: {
    // Keep linting in CI while allowing `next build` to complete on Deno.
    ignoreDuringBuilds: true,
  },
};

export default nextConfig;
```

Astro (`astro`)

Astro is a web framework for building content-driven websites like blogs, marketing, and e-commerce. Astro leverages server rendering over client-side rendering in the browser as much as possible.

For static Astro sites, no additional configuration is needed to use Deno Deploy .

When using SSR in Astro with Deno Deploy , you need to install the [`@deno/astro-adapter`] package and configure your `astro.config.mjs` file to use the adapter:

```
$ deno add npm:@deno/astro-adapter
# or npm install @deno/astro-adapter
# or yarn add @deno/astro-adapter
# or pnpm add @deno/astro-adapter
```

astro.config.mjs

```
  import { defineConfig } from 'astro/config';
+ import deno from '@deno/astro-adapter';

  export default defineConfig({
+   output: 'server',
+   adapter: deno(),
  });
```

Sharp image optimization is supported.

The `astro:env` API is supported.

Nuxt (`nuxt`)

Create high-quality web applications with Nuxt, the open source framework that makes full-stack development with Vue.js intuitive.

Nuxt requires no additional setup. Deno Deploy automatically configures Nitro, so no additional preset configuration is needed.

SolidStart (`solidstart`)

SolidStart is an open source meta-framework designed to unify components that make up a web application. It is built on top of Solid.

SolidStart requires no additional setup. Deno Deploy automatically configures Nitro, so no additional preset configuration is needed.

TanStack Start (`tanstackstart`)

TanStack Start is a full-stack React or Solid framework powered by TanStack Router. It uses Nitro as its server layer, which enables deployment to Deno Deploy.

To deploy TanStack Start on Deno Deploy, you need to install `nitro` and configure the Nitro Vite plugin:

1. Add `nitro` to your dependencies:

```
deno add npm:nitro-nightly@latest
# or npm install nitro-nightly@latest
```

1. Configure your `vite.config.ts` to use the Nitro plugin:

vite.config.ts

```
import { tanstackStart } from "@tanstack/react-start/plugin/vite";
import { defineConfig } from "vite";
import { nitro } from "nitro/vite";
import viteReact from "@vitejs/plugin-react";

export default defineConfig({
  plugins: [tanstackStart(), nitro(), viteReact()],
});
```

Deno Deploy automatically configures Nitro, so no additional preset configuration is needed.

SvelteKit (`sveltekit`)

SvelteKit is a framework for rapidly developing robust, performant web applications using Svelte.

SvelteKit requires no additional setup.

Fresh (`fresh`)

Fresh is a full stack modern web framework for JavaScript and TypeScript developers. Fresh uses Preact as the JSX rendering engine.

Fresh requires no additional setup.

Lume (`lume`)

Lume is a static site generator for building fast and modern websites using Deno.

Lume requires no additional setup.

Remix (`remix`)

> **Experimental**: Remix is not yet fully supported. It is in the process of being integrated into Deno Deploy. Some features may not work as expected. Please report any issues you encounter to the Deno team.

CDN and caching

Deno Deploy includes a built-in HTTP caching layer that automatically caches eligible responses at the edge, reducing latency and origin load. This document covers how caching works, how to control cache behavior, and how to invalidate cached content.

How Caching Works

When a request arrives at Deno Deploy:

1. The cache checks if a valid cached response exists for the request
2. If found and fresh, the cached response is served immediately (cache hit)
3. If not found or stale, the request is forwarded to your application (cache miss)
4. Cacheable responses are stored for future requests

The cache follows RFC 9110 and RFC 9111 semantics, implementing standard HTTP caching behavior.

Cache-Control Headers

Deno Deploy respects the standard `Cache-Control` header with the following directives:

Response Directives

Directive	Description
`public`	Response can be cached by shared caches
`private`	Response is user-specific and cannot be cached (bypasses CDN)
`no-store`	Response must not be cached
`no-cache`	Response must be revalidated before use
`max-age=N`	Response is fresh for N seconds
`s-maxage=N`	Like `max-age`, but only for shared caches (takes precedence)
`stale-while-revalidate=N`	Serve stale content while revalidating in background
`stale-if-error=N`	Serve stale content if origin returns an error
`must-revalidate`	Stale responses must not be used without revalidation

Example: Cache for 1 hour at the edge

```
Deno.serve(() => {
  return new Response("Hello, World!", {
    headers: {
      "Cache-Control": "public, s-maxage=3600",
    },
  });
});
```

Example: Cache with stale-while-revalidate

```
Deno.serve(() => {
  return new Response(JSON.stringify({ data: "..." }), {
    headers: {
      "Content-Type": "application/json",
      // Cache for 60s, serve stale for up to 5 minutes while revalidating
      "Cache-Control": "public, s-maxage=60, stale-while-revalidate=300",
    },
  });
});
```

Deno Deploy-Specific Headers

Deno Deploy supports additional headers for fine-grained cache control:

`Deno-CDN-Cache-Control`

A CDN-specific cache control header that takes precedence over both `CDN-Cache-Control` and `Cache-Control`. Use this when you want different caching behavior for Deno Deploy's CDN versus browsers or other caches.

Header priority (highest to lowest):

1. `Deno-CDN-Cache-Control`
2. `CDN-Cache-Control`
3. `Cache-Control`

```
Deno.serve(() => {
  return new Response("Hello!", {
    headers: {
      // Browser caches for 60s
      "Cache-Control": "public, max-age=60",
      // Deno Deploy CDN caches for 1 hour
      "Deno-CDN-Cache-Control": "public, s-maxage=3600",
    },
  });
});
```

`Deno-Cache-Tag` / `Cache-Tag`

Associate responses with cache tags for targeted invalidation. Tags allow you to invalidate groups of cached responses without knowing their exact URLs.

```
Deno.serve((req) => {
  const url = new URL(req.url);
  const productId = url.pathname.split("/")[2];

  return new Response(JSON.stringify({ id: productId, name: "Widget" }), {
    headers: {
      "Content-Type": "application/json",
      "Cache-Control": "public, s-maxage=3600",
      // Tag this response for later invalidation
      "Deno-Cache-Tag": `product-${productId},products,catalog`,
    },
  });
});
```

Tag format:

- Multiple tags can be specified as a comma-separated list
- Tags are case-insensitive
- Maximum 1024 characters per tag
- Maximum 500 tags per response
- Tags must be UTF-8 encoded

`Deno-Cache-Id`

A special header that serves two purposes:

1. **Opt out of automatic invalidation**: Responses with `Deno-Cache-Id` use a shared cache namespace that persists across deployments. Without this header, cached responses are automatically invalidated when you deploy a new version.
2. **Acts as a cache tag**: The value can be used to invalidate the cached response.

```
Deno.serve(() => {
  return new Response("Static content that rarely changes", {
    headers: {
      "Cache-Control": "public, s-maxage=86400",
      // This response survives redeployments
      "Deno-Cache-Id": "static-content-v1",
    },
  });
});
```

Use cases for `Deno-Cache-Id`:

- Content that should remain cached across deployments (e.g., static assets with content-based hashes)
- Long-lived cached responses where you want explicit control over invalidation
- Sharing cached responses between deployment revisions

`Deno-Vary`

Extend cache variation beyond standard HTTP `Vary` semantics. *(Coming soon)*

Cache Invalidation

Deno Deploy supports on-demand cache invalidation via cache tags. This allows you to purge specific cached content without redeploying.

Invalidation API

Send a POST request to `http://cache.localhost/invalidate/http` from within your Deno Deploy application:

```
Deno.serve(async (req) => {
  const url = new URL(req.url);

  if (req.method === "POST" && url.pathname === "/admin/purge") {
    // Invalidate all responses tagged with "products"
    const res = await fetch("http://cache.localhost/invalidate/http", {
      method: "POST",
```

```
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        tags: ["products"],
      }),
    });

    if (res.ok) {
      return new Response("Cache purged successfully");
    }
    return new Response("Purge failed", { status: 500 });
  }

  // ... handle other requests
});
```

Invalidation Request Format

```
{
  "tags": ["tag1", "tag2", "tag3"]
}
```

- `tags`: Array of cache tags to invalidate (required)
- Maximum 500 tags per request

Wildcard Invalidation

Use "*" to invalidate all cached responses for your deployment:

```
await fetch("http://cache.localhost/invalidate/http", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    tags: ["*"],
  }),
});
```

Cross-Region Invalidation

Cache invalidation is automatically synchronized across all Deno Deploy regions. When you invalidate a tag, the purge propagates globally within seconds.

Example: Content Management Webhook

```
Deno.serve(async (req) => {
  const url = new URL(req.url);

  // Webhook endpoint for CMS updates
  if (req.method === "POST" && url.pathname === "/webhook/cms") {
    const payload = await req.json();

    // Invalidate based on content type
    const tags: string[] = [];

    if (payload.type === "product") {
      tags.push(`product-${payload.id}`, "products");
    } else if (payload.type === "category") {
      tags.push(`category-${payload.id}`, "categories", "navigation");
    }
```

```
    if (tags.length > 0) {
      await fetch("http://cache.localhost/invalidate/http", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ tags }),
      });
    }

    return new Response("OK");
  }

  // ... serve content
});
```

Cache Status Header

Deno Deploy adds a `Cache-Status` response header to indicate the cache result:

Value	Description
`deno; hit`	Response served from cache
`deno; fwd=uri-miss; stored`	Cache miss, response stored for future requests
`deno; fwd=miss; stored`	Vary miss, response stored with new vary key
`deno; fwd=stale`	Stale response, forwarding to origin
`deno; fwd=method`	Non-cacheable method (POST, PUT, etc.)
`deno; fwd=bypass`	Response explicitly bypassed cache
`deno; fwd=request`	Request directives prevented caching

Bypass Details

When a response bypasses the cache, the `detail` field indicates why:

- `detail=not-cacheable` - Response doesn't meet caching criteria
- `detail=no-cache-or-private` - `no-cache` or `private` directive present
- `detail=set-cookie` - Response contains `Set-Cookie` header
- `detail=pragma-no-cache` - Legacy `Pragma: no-cache` header present
- `detail=too-large` - Response body exceeds maximum cacheable size
- `detail=zero-ttl` - Calculated TTL is zero
- `detail=vary-star` - `Vary: *` header prevents caching
- `detail=header-overflow` - Too many response headers

Cacheable Responses

A response is cacheable when:

1. The request method is `GET` or `HEAD`
2. The response status is cacheable (200, 203, 204, 206, 300, 301, 308, 404, 405, 410, 414, 501)
3. The response includes caching headers (`Cache-Control`, `Expires`, etc.)
4. The response doesn't include `no-store`, `private`, or `no-cache` directives

5. The response doesn't include a `Set-Cookie` header
6. The response body is within the maximum cacheable size

Cache Variation (Vary)

The cache respects the standard `Vary` header to store different versions of a response based on request headers:

```
Deno.serve((req) => {
  const acceptLanguage = req.headers.get("Accept-Language") || "en";
  const language = acceptLanguage.startsWith("es") ? "es" : "en";

  return new Response(`Hello in ${language}!`, {
    headers: {
      "Cache-Control": "public, s-maxage=3600",
      "Vary": "Accept-Language",
      "Content-Language": language,
    },
  });
});
```

Note: `Vary: *` prevents caching entirely.

HEAD Requests

HEAD requests can be served from cached GET responses. The cache automatically strips the response body while preserving headers.

Range Requests

The cache supports HTTP range requests (`Range` header) and can serve partial content from cached full responses.

Best Practices

[[1-use-s-maxage-for-cdn-caching]] === 1. Use `s-maxage` for CDN caching

```
// Let browsers cache for 1 minute, CDN for 1 hour
headers: {
  "Cache-Control": "public, max-age=60, s-maxage=3600"
}
```

[[2-tag-content-for-targeted-invalidation]] === 2. Tag content for targeted invalidation

```
// Tag by content type, ID, and category for flexible purging
headers: {
  "Deno-Cache-Tag": `article-${id},articles,category-${category}`
}
```

[[3-use-stale-while-revalidate-for-better-ux]] === 3. Use `stale-while-revalidate` for better UX

```
// Serve stale content while refreshing in background
headers: {
  "Cache-Control": "public, s-maxage=60, stale-while-revalidate=600"
}
```

[[4-use-deno-cache-id-for-stable-assets]] === 4. Use `Deno-Cache-Id` for stable assets

```
// Content-addressed assets that shouldn't be invalidated on deploy
const hash = computeHash(content);
headers: {
  "Cache-Control": "public, s-maxage=31536000, immutable",
  "Deno-Cache-Id": `asset-${hash}`
}
```

[[5-set-appropriate-ttls]] === 5. Set appropriate TTLs

- Static assets with hash in filename: `max-age=31536000` (1 year)
- API responses: `s-maxage=60` to `s-maxage=300` with `stale-while-revalidate`
- Personalized content: Don't cache or use `Vary` appropriately

Automatic Cache Invalidation

By default, all cached responses (without `Deno-Cache-Id`) are automatically invalidated when you deploy a new version of your application. This ensures users always see fresh content after a deployment.

To opt out of automatic invalidation, use the `Deno-Cache-Id` header.

Playgrounds

Playground applications enable you to create, edit, and deploy applications entirely from the Deno Deploy web dashboard, without needing to create a GitHub repository.

Playgrounds contain one or more files (JavaScript, TypeScript, TSX, JSON, etc.) that you can edit directly in the playground editor.

Creating a playground

You can create playgrounds from the "Applications" page in your organization. Click the "New Playground" button to create a basic "Hello World" playground. Using the dropdown on the "New Playground" button lets you create playgrounds from other templates, such as Next.js or Hono.

Editing a playground

To edit a playground, open it from the "Applications" page in your organization.

The playground editor consists of five main sections:

- **Code editor**: The central area where you edit code for the currently selected file. Above the editor is a navbar showing the current file name, which you can click to edit.
- **File browser**: Located on the left of the code editor, this panel shows all files in the playground. Click any file to open it in the editor. Create new files by clicking the "New" icon at the top of the file browser. Delete files using the delete button next to each file name.
- **Top bar**: Located above the code editor, this contains action buttons for the playground. The "Deploy" button saves current changes and triggers a build. "Build Config" and "Env Variables" buttons open their respective configuration drawers. The left side of the top bar displays the playground URL (unless the playground hasn't been deployed yet).
- **Bottom drawer**: Located beneath the code editor, this contains debugging tools including "Build Logs" that show build progress during deployment, and tabs for viewing logs and traces.
- **Right drawer**: Located to the right of the code editor, this contains tools for inspecting application output. The "Preview" tab displays an iframe showing the deployed application, while "HTTP Explorer" lets you send individual HTTP requests to your deployment.

The playground content automatically saves when you click the "Deploy" button or when the editor loses focus.

Uploading files

You can upload a zip file containing files and directories to the playground by dragging it into the file browser area. The contents of the zip file will be extracted into the playground, preserving the directory structure.

> The playground editor does not support uploading individual files or directories.

Using the HTTP explorer

The HTTP Explorer tab in the playground allows you to make arbitrary HTTP requests to any URL served by the playground. This is useful for testing APIs or other services that do not serve a web page.

To use the HTTP Explorer, enter the path and query parameters for the request you want to make, select the HTTP method (GET, POST, etc.), and click on the button labeled with the selected method.

Additional request headers can be added by clicking the "Set Headers" button.

After the response has been made, the HTTP Explorer will display the response status, headers, and body.

To view the trace for the request, click on the "Trace" button in the response section. This will open the request trace for the request in a drawer on top of the playground editor. From there you can also view any `console.log` output that was captured during the request.

Renaming a playground

You can rename a playground by editing the playground slug on the playground settings page. This will update the default domain names associated with the playground since they are based on the playground slug. The new slug must be unique within the organization (i.e. must not be in use by another app or playground in the same organization).

> **NOTE**
>
> Any previous `deno.net` URLs pointing to the playground will no longer work after renaming.
>
> Custom domains will continue to work, as they are not tied to the playground slug.

Deleting a playground

Playgrounds can be deleted from the playground settings page. This will remove the playground and all its revisions from the organization. All existing deployments will immediately stop serving traffic, and all custom domain associations will be removed.

The playground and its revisions will no longer be accessible after deletion. Deleted playgrounds cannot be restored through the Deno Deploy UI.

> **NOTE**
>
> Deleted a playground by mistake? Contact Deno support within 30 days to restore it.

Limitations

> Playgrounds cannot currently be transferred to another organization.

Deploy Button

The Deploy Button offers a shortcut for users to create and deploy a new application on Deno Deploy based on existing code hosted in a Git repository.

It provides a link directly into the Deno Deploy application creation flow, and populates settings in the creation flow based on provided query parameters or framework detection.

The specified repository will be cloned to the user's GitHub account and set as the source for a new project. By default, the new repository will be public, but can be set to be private if required.

Example

The deploy button below demonstrates the creation of a new application based on a simple starter project

[Deploy on Deno]

Create and deploy a new application

Use the code below to give a button which creates and deploys a new application:

Markdown

```
[![Deploy on Deno](https://deno.com/button)](https://console.deno.com/new?clone=REPOSITORY_URL)
```

HTML

```
[![Deploy on Deno](https://deno.com/button)](https://console.deno.com/new?clone=REPOSITORY_URL)
```

URL

```
https://console.deno.com/new?clone=REPOSITORY_URL
```

Parameters

The following query parameters can be used to configure a Deploy Button:

- `clone` — (required) The URL of the source repo to clone as a new repo which will then be deployed
- `path` — (optional) The path within the source repo to clone from. Providing this will create a new repo whose root is this directory from within the source repository.
- `app_directory` — (optional) The directory within the new repository to use as the application root. This is useful when the repository is structured as a monorepo.
- `install` — (optional) the command to execute prior to a build in order to install dependencies
- `build` — (optional) the command to execute to build the application
- `predeploy` — (optional) the command to execute after a build but before deployment

04

KV

Deno KV Quick Start 120

Key Space 125

Operations 129

Key Expiration (TTL for keys) 134

Secondary Indexes 135

Transactions 139

Using KV in Node.js 141

Data Modeling in TypeScript 143

Backups 146

Deno KV Quick Start

Deno KV is a key-value database built directly into the Deno runtime, available in the `Deno.Kv` namespace. It can be used for many kinds of data storage use cases, but excels at storing simple data structures that benefit from very fast reads and writes. Deno KV is available in the Deno CLI and on Deno Deploy.

> **CAUTION**
> Deno KV is still in development and may change. To use it, you must pass the `--unstable-kv` flag to Deno.

Let's walk through the key features of Deno KV.

Opening and closing a database

In your Deno program, you can get a reference to a KV database using `Deno.openKv()`. You may pass in an optional file system path to where you'd like to store your database, otherwise one will be created for you based on the current working directory of your script.

To close the database connection, use the `.close()` method.

```
const kv = await Deno.openKv();
// Execute some queries ...
kv.close();
```

Creating, updating, and reading a key-value pair

Data in Deno KV is stored as key-value pairs, much like properties of a JavaScript object literal or a Map. Keys are represented as an array of JavaScript types, like `string`, `number`, `bigint`, or `boolean`. Values can be arbitrary JavaScript objects. In this example, we create a key-value pair representing a user's UI preferences, and save it with `kv.set()`.

```
const kv = await Deno.openKv();

const prefs = {
  username: "ada",
  theme: "dark",
  language: "en-US",
};

const result = await kv.set(["preferences", "ada"], prefs);
```

Once a key-value pair is set, you can read it from the database with `kv.get()`:

```
const entry = await kv.get(["preferences", "ada"]);
console.log(entry.key);
console.log(entry.value);
console.log(entry.versionstamp);
```

Both `get` and `list` operations return a KvEntry object with the following properties:

- `key` - the array key you used to set the value
- `value` - the JavaScript object you set for this key
- `versionstamp` - a generated value used to determine if a key has been updated.

The `set` operation is also used to update objects that already exist for a given key. When a key's value is updated, its `versionstamp` will change to a new generated value.

Listing several key-value pairs

To get values for a finite number of keys, you may use `kv.getMany()`. Pass in several keys as arguments, and you'll receive an array of values for each key. Note that **values and versionstamps can be `null`** if no value exists for the given key(s).

```
const kv = await Deno.openKv();
const result = await kv.getMany([
  ["preferences", "ada"],
  ["preferences", "grace"],
]);
result[0].key; // ["preferences", "ada"]
result[0].value; // { ... }
result[0].versionstamp; // "00000000000000010000"
result[1].key; // ["preferences", "grace"]
result[1].value; // null
result[1].versionstamp; // null
```

Often, it is useful to retrieve a list of key-value pairs from all keys that share a given prefix. This type of operation is possible using `kv.list()`. In this example, we get a list of key-value pairs that share the "preferences" prefix.

```
const kv = await Deno.openKv();
const entries = kv.list({ prefix: ["preferences"] });
for await (const entry of entries) {
  console.log(entry.key); // ["preferences", "ada"]
  console.log(entry.value); // { ... }
  console.log(entry.versionstamp); // "00000000000000010000"
}
```

Returned keys are ordered lexicographically based on the next component of the key after the prefix. So KV pairs with these keys:

- `["preferences", "ada"]`
- `["preferences", "bob"]`
- `["preferences", "cassie"]`

Will be returned in that order by `kv.list()`.

Read operations can either be performed in **strong or eventual consistency mode**. Strong consistency mode guarantees that the read operation will return the most recently written value. Eventual consistency mode may return a stale value, but is faster. By contrast, writes are always performed in strong consistency mode.

Deleting key-value pairs

You can delete a key from the database using `kv.delete()`. No action is taken if no value is found for the given key.

```
const kv = await Deno.openKv();
await kv.delete(["preferences", "alan"]);
```

Atomic transactions

Deno KV is capable of executing atomic transactions, which enables you to conditionally execute one or

many data manipulation operations at once. In the following example, we create a new preferences object only if it hasn't been created already.

```
const kv = await Deno.openKv();

const key = ["preferences", "alan"];
const value = {
  username: "alan",
  theme: "light",
  language: "en-GB",
};

const res = await kv.atomic()
  .check({ key, versionstamp: null }) // `null` versionstamps mean 'no value'
  .set(key, value)
  .commit();
if (res.ok) {
  console.log("Preferences did not yet exist. Inserted!");
} else {
  console.error("Preferences already exist.");
}
```

Learn more about transactions in Deno KV here.

Improve querying with secondary indexes

Secondary indexes store the same data by multiple keys, allowing for simpler queries of the data you need. Let's say that we need to be able to access user preferences by both username AND email. To enable this, you could provide a function that wraps the logic to save the preferences to create two indexes.

```
const kv = await Deno.openKv();

async function savePreferences(prefs) {
  const key = ["preferences", prefs.username];

  // Set the primary key
  const r = await kv.set(key, prefs);

  // Set the secondary key's value to be the primary key
  await kv.set(["preferencesByEmail", prefs.email], key);

  return r;
}

async function getByUsername(username) {
  // Use as before...
  const r = await kv.get(["preferences", username]);
  return r;
}

async function getByEmail(email) {
  // Look up the key by email, then second lookup for actual data
  const r1 = await kv.get(["preferencesByEmail", email]);
  const r2 = await kv.get(r1.value);
  return r2;
}
```

Learn more about secondary indexes in the manual here.

Watching for updates in Deno KV

You can also listen for updates from Deno KV with `kv.watch()`, which will emit a new value or values of the key or keys you provide. In the below chat example, we watch for updates on the key `["last_message_id", roomId]`. We retrieve `messageId`, which we then use with `kv.list()` to grab all the new messages from seen and `messageId`.

```
let seen = "";
for await (const [messageId] of kv.watch([["last_message_id", roomId]])) {
  const newMessages = await Array.fromAsync(kv.list({
    start: ["messages", roomId, seen, ""],
    end: ["messages", roomId, messageId, ""],
  }));
  await websocket.write(JSON.stringify(newMessages));
  seen = messageId;
}
```

Learn more about using Deno KV watch here.

Production usage

Deno KV is available for use in live applications on Deno Deploy. In production, Deno KV is backed by FoundationDB, the open source key-value store created by Apple.

Testing

By default, `Deno.openKv()` creates or opens a persistent store based on the path from which the script that invoked it was run. This isn't usually desirable for tests, which need to produce the same behavior when run many times in a row.

To test code that uses Deno KV, you can use the special argument `":memory:"` to create an ephemeral Deno KV datastore.

```
async function setDisplayName(
  kv: Deno.Kv,
  username: string,
  displayname: string,
) {
  await kv.set(["preferences", username, "displayname"], displayname);
}

async function getDisplayName(
  kv: Deno.Kv,
  username: string,
): Promise<string | null> {
  return (await kv.get(["preferences", username, "displayname"]))
    .value as string;
}

Deno.test("Preferences", async (t) => {
  const kv = await Deno.openKv(":memory:");

  await t.step("can set displayname", async () => {
    const displayName = await getDisplayName(kv, "example");
    assertEquals(displayName, null);

    await setDisplayName(kv, "example", "Exemplary User");
```

```
    const displayName = await getDisplayName(kv, "example");
    assertEquals(displayName, "Exemplary User");
  });
});
```

This works because Deno KV is backed by SQLite when run for local development. Just like in-memory SQLite databases, multiple ephemeral Deno KV stores can exist at once without interfering with one another. For more information about special database addressing modes, see the SQLite docs on the topic.

Next steps

At this point, you're just beginning to scratch the surface with Deno KV. Be sure to check out our guide on the Deno KV key space, and a collection of tutorials and example applications here.

Key Space

Deno KV is a key value store. The key space is a flat namespace of key+value+versionstamp pairs. Keys are sequences of key parts, which allow modeling of hierarchical data. Values are arbitrary JavaScript objects. Versionstamps represent when a value was inserted / modified.

Keys

Keys in Deno KV are sequences of key parts, which can be `string`s, `number`s, `boolean`s, `Uint8Array`s, or `bigint`s.

Using a sequence of parts, rather than a single string eliminates the possibility of delimiter injection attacks, because there is no visible delimiter.

> A key injection attack occurs when an attacker manipulates the structure of a key-value store by injecting delimiters used in the key encoding scheme into a user controlled variable, leading to unintended behavior or unauthorized access. For example, consider a key-value store using a slash (/) as a delimiter, with keys like "users/alice/settings" and "users/bob/settings". An attacker could create a new user with the name "alice/settings/hacked" to form the key "users/alice/settings/hacked/settings", injecting the delimiter and manipulating the key structure. In Deno KV, the injection would result in the key `["users", "alice/settings/hacked", "settings"]`, which is not harmful.

Between key parts, invisible delimiters are used to separate the parts. These delimiters are never visible, but ensure that one part can not be confused with another part. For example, the key parts `["abc", "def"]`, `["ab", "cdef"]`, `["abc", "", "def"]` are all different keys.

Keys are case sensitive and are ordered lexicographically by their parts. The first part is the most significant, and the last part is the least significant. The order of the parts is determined by both the type and the value of the part.

Key Part Ordering

Key parts are ordered lexicographically by their type, and within a given type, they are ordered by their value. The ordering of types is as follows:

1. `Uint8Array`
2. `string`
3. `number`
4. `bigint`
5. `boolean`

Within a given type, the ordering is:

- `Uint8Array`: byte ordering of the array
- `string`: byte ordering of the UTF-8 encoding of the string
- `number`: -Infinity < -1.0 < -0.5 < -0.0 < 0.0 < 0.5 < 1.0 < Infinity < NaN
- `bigint`: mathematical ordering, largest negative number first, largest positive number last
- `boolean`: false < true

This means that the part `1.0` (a number) is ordered before the part `2.0` (also a number), but is greater than the part `0n` (a bigint), because `1.0` is a number and `0n` is a bigint, and type ordering has precedence over the ordering of values within a type.

Key Examples

```
["users", 42, "profile"]; // User with ID 42's profile
["posts", "2023-04-23", "comments"]; // Comments for all posts on 2023-04-23
["products", "electronics", "smartphones", "apple"]; // Apple smartphones in the electronics category
["orders", 1001, "shipping", "tracking"]; // Tracking information for order ID 1001
["files", new Uint8Array([1, 2, 3]), "metadata"]; // Metadata for a file with Uint8Array identifier
["projects", "openai", "tasks", 5]; // Task with ID 5 in the OpenAI project
["events", "2023-03-31", "location", "san_francisco"]; // Events in San Francisco on 2023-03-31
["invoices", 2023, "Q1", "summary"]; // Summary of Q1 invoices for 2023
["teams", "engineering", "members", 1n]; // Member with ID 1n in the engineering team
```

Universally Unique Lexicographically Sortable Identifiers (ULIDs)

Key part ordering allows keys consisting of timestamps and ID parts to be listed chronologically. Typically, you can generate a key using the following: `Date.now()` and `crypto.randomUUID()`:

```
async function setUser(user) {
  await kv.set(["users", Date.now(), crypto.randomUUID()], user);
}
```

Run multiple times sequentially, this produces the following keys:

```
["users", 1691377037923, "8c72fa25-40ad-42ce-80b0-44f79bc7a09e"]; // First user
["users", 1691377037924, "8063f20c-8c2e-425e-a5ab-d61e7a717765"]; // Second user
["users", 1691377037925, "35310cea-58ba-4101-b09a-86232bf230b2"]; // Third user
```

However, having the timestamp and ID represented within a single key part may be more straightforward in some cases. You can use a Universally Unique Lexicographically Sortable Identifier (ULID) to do this. This type of identifier encodes a UTC timestamp, is lexicographically sortable and is cryptographically random by default:

```
import { ulid } from "jsr:@std/ulid";

const kv = await Deno.openKv();

async function setUser(user) {
  await kv.set(["users", ulid()], user);
}
```

```
["users", "01H76YTWK3YBV020S6MP69TBEQ"]; // First user
["users", "01H76YTWK4V82VFET9YTYDQ0NY"]; // Second user
["users", "01H76YTWK5DM1G9TFR0Y5SCZQV"]; // Third user
```

Furthermore, you can generate ULIDs monotonically increasingly using `monotonicUlid` function:

```
import { monotonicUlid } from "jsr:@std/ulid";

async function setUser(user) {
  await kv.set(["users", monotonicUlid()], user);
}
```

```
// Strict ordering for the same timestamp by incrementing the least-significant random bit by 1
["users", "01H76YTWK3YBV020S6MP69TBEQ"]; // First user
["users", "01H76YTWK3YBV020S6MP69TBER"]; // Second user
["users", "01H76YTWK3YBV020S6MP69TBES"]; // Third user
```

Values

Values in Deno KV can be arbitrary JavaScript values that are compatible with the structured clone algorithm. This includes:

- `undefined`
- `null`
- `boolean`
- `number`
- `string`
- `bigint`
- `Uint8Array`
- `Array`
- `Object`
- `Map`
- `Set`
- `Date`
- `RegExp`

Objects and arrays can contain any of the above types, including other objects and arrays. `Map`s and `Set`s can contain any of the above types, including other `Map`s and `Set`s.

Circular references within values are supported.

Objects with a non-primitive prototype are not supported (such as class instances or Web API objects). Functions and symbols can also not be serialized.

`Deno.KvU64` type

In addition to structured serializable values, the special value `Deno.KvU64` is also supported as a value. This object represents a 64-bit unsigned integer, represented as a bigint. It can be used with the `sum`, `min`, and `max` KV operations. It can not be stored within an object or array. It must be stored as a top-level value.

It can be created with the `Deno.KvU64` constructor:

```
const u64 = new Deno.KvU64(42n);
```

Value Examples

```
undefined;
null;
true;
false;
42;
-42.5;
42n;
"hello";
new Uint8Array([1, 2, 3]);
[1, 2, 3];
{ a: 1, b: 2, c: 3 };
new Map([["a", 1], ["b", 2], ["c", 3]]);
new Set([1, 2, 3]);
```

```
new Date("2023-04-23");
/abc/;

// Circular references are supported
const a = {};
const b = { a };
a.b = b;

// Deno.KvU64 is supported
new Deno.KvU64(42n);
```

Versionstamp

All data in the Deno KV key-space is versioned. Every time a value is inserted or modified, a versionstamp is assigned to it. Versionstamps are monotonically increasing, non-sequential, 12 byte values that represent the time that the value was modified. Versionstamps do not represent real time, but rather the order in which the values were modified.

Because versionstamps are monotonically increasing, they can be used to determine whether a given value is newer or older than another value. This can be done by comparing the versionstamps of the two values. If versionstamp A is greater than versionstamp B, then value A was modified more recently than value B.

```
versionstampA > versionstampB;
"000002fa526aaccb0000" > "000002fa526aacc90000"; // true
```

All data modified by a single transaction are assigned the same versionstamp. This means that if two `set` operations are performed in the same atomic operation, then the versionstamp of the new values will be the same.

Versionstamps are used to implement optimistic concurrency control. Atomic operations can contain checks that ensure that the versionstamp of the data they are operating on matches a versionstamp passed to the operation. If the versionstamp of the data is not the same as the versionstamp passed to the operation, then the transaction will fail and the operation will not be applied.

Operations

The Deno KV API provides a set of operations that can be performed on the key space.

There are two operations that read data from the store, and five operations that write data to the store.

Read operations can either be performed in strong or eventual consistency mode. Strong consistency mode guarantees that the read operation will return the most recently written value. Eventual consistency mode may return a stale value, but is faster.

Write operations are always performed in strong consistency mode.

`get`

The `get` operation returns the value and versionstamp associated with a given key. If a value does not exist, get returns a `null` value and versionstamp.

There are two APIs that can be used to perform a `get` operation. The `Deno.Kv.prototype.get(key, options?)` API, which can be used to read a single key, and the `Deno.Kv.prototype.getMany(keys, options?)` API, which can be used to read multiple keys at once.

Get operations are performed as a "snapshot read" in all consistency modes. This means that when retrieving multiple keys at once, the values returned will be consistent with each other.

```
const res = await kv.get<string>(["config"]);
console.log(res); // { key: ["config"], value: "value", versionstamp: "000002fa526aaccb0000" }

const res = await kv.get<string>(["config"], { consistency: "eventual" });
console.log(res); // { key: ["config"], value: "value", versionstamp: "000002fa526aaccb0000" }

const [res1, res2, res3] = await kv.getMany<[string, string, string]>([
  ["users", "sam"],
  ["users", "taylor"],
  ["users", "alex"],
]);
console.log(res1); // { key: ["users", "sam"], value: "sam", versionstamp: "00e0a2a0f0178b270000" }
console.log(res2); // { key: ["users", "taylor"], value: "taylor", versionstamp: "0059e9035e5e7c5e0000" }
console.log(res3); // { key: ["users", "alex"], value: "alex", versionstamp: "00a44a3c3e53b9750000" }
```

`list`

The `list` operation returns a list of keys that match a given selector. The associated values and versionstamps for these keys are also returned. There are 2 different selectors that can be used to filter the keys matched.

The `prefix` selector matches all keys that start with the given prefix key parts, but not inclusive of an exact match of the key. The prefix selector may optionally be given a `start` OR `end` key to limit the range of keys returned. The `start` key is inclusive, and the `end` key is exclusive.

The `range` selector matches all keys that are lexicographically between the given `start` and `end` keys. The `start` key is inclusive, and the `end` key is exclusive.

> Note: In the case of the prefix selector, the `prefix` key must consist only of full (not partial) key parts. For example, if the key `["foo", "bar"]` exists in the store, then the prefix selector `["foo"]` will match it, but the prefix selector `["f"]` will not.

The list operation may optionally be given a `limit` to limit the number of keys returned.

List operations can be performed using the Deno.Kv.prototype.list<string>(selector, options?) method. This method returns a `Deno.KvListIterator` that can be used to iterate over the keys returned. This is an async iterator, and can be used with `for await` loops.

```
// Return all users
const iter = kv.list<string>({ prefix: ["users"] });
const users = [];
for await (const res of iter) users.push(res);
console.log(users[0]); // { key: ["users", "alex"], value: "alex", versionstamp: "00a44a3c3e53b9750000" }
console.log(users[1]); // { key: ["users", "sam"], value: "sam", versionstamp: "00e0a2a0f0178b270000" }
console.log(users[2]); // { key: ["users", "taylor"], value: "taylor", versionstamp: "0059e9035e5e7c5e0000" }

// Return the first 2 users
const iter = kv.list<string>({ prefix: ["users"] }, { limit: 2 });
const users = [];
for await (const res of iter) users.push(res);
console.log(users[0]); // { key: ["users", "alex"], value: "alex", versionstamp: "00a44a3c3e53b9750000" }
console.log(users[1]); // { key: ["users", "sam"], value: "sam", versionstamp: "00e0a2a0f0178b270000" }

// Return all users lexicographically after "taylor"
const iter = kv.list<string>({ prefix: ["users"], start: ["users", "taylor"] });
const users = [];
for await (const res of iter) users.push(res);
console.log(users[0]); // { key: ["users", "taylor"], value: "taylor", versionstamp: "0059e9035e5e7c5e0000" }

// Return all users lexicographically before "taylor"
const iter = kv.list<string>({ prefix: ["users"], end: ["users", "taylor"] });
const users = [];
for await (const res of iter) users.push(res);
console.log(users[0]); // { key: ["users", "alex"], value: "alex", versionstamp: "00a44a3c3e53b9750000" }
console.log(users[1]); // { key: ["users", "sam"], value: "sam", versionstamp: "00e0a2a0f0178b270000" }

// Return all users starting with characters between "a" and "n"
const iter = kv.list<string>({ start: ["users", "a"], end: ["users", "n"] });
const users = [];
for await (const res of iter) users.push(res);
console.log(users[0]); // { key: ["users", "alex"], value: "alex", versionstamp: "00a44a3c3e53b9750000" }
```

The list operation reads data from the store in batches. The size of each batch can be controlled using the `batchSize` option. The default batch size is 500 keys. Data within a batch is read in a single snapshot read, so the values are consistent with each other. Consistency modes apply to each batch of data read. Across batches, data is not consistent. The borders between batches is not visible from the API as the iterator returns individual keys.

The list operation can be performed in reverse order by setting the `reverse` option to `true`. This will return the keys in lexicographically descending order. The `start` and `end` keys are still inclusive and exclusive respectively, and are still interpreted as lexicographically ascending.

```
// Return all users in reverse order, ending with "sam"
const iter = kv.list<string>({ prefix: ["users"], start: ["users", "sam"] }, {
  reverse: true,
});
const users = [];
for await (const res of iter) users.push(res);
console.log(users[0]); // { key: ["users", "taylor"], value: "taylor", versionstamp: "0059e9035e5e7c5e0000" }
console.log(users[1]); // { key: ["users", "sam"], value: "sam", versionstamp: "00e0a2a0f0178b270000" }
```

> Note: in the above example we set the `start` key to `["users", "sam"]`, even though the first key returned is `["users", "taylor"]`. This is because the `start` and end keys are always evaluated in lexicographically ascending order, even when the list operation is performed in reverse order (which returns the keys in lexicographically descending order).

set

The `set` operation sets the value of a key in the store. If the key does not exist, it is created. If the key already exists, its value is overwritten.

The `set` operation can be performed using the `Deno.Kv.prototype.set(key, value)` method. This method returns a `Promise` that resolves to a `Deno.KvCommitResult` object, which contains the `versionstamp` of the commit.

Set operations are always performed in strong consistency mode.

```
const res = await kv.set(["users", "alex"], "alex");
console.log(res.versionstamp); // "00a44a3c3e53b9750000"
```

delete

The `delete` operation deletes a key from the store. If the key does not exist, the operation is a no-op.

The `delete` operation can be performed using the `Deno.Kv.prototype.delete(key)` method.

Delete operations are always performed in strong consistency mode.

```
await kv.delete(["users", "alex"]);
```

sum

The `sum` operation atomically adds a value to a key in the store. If the key does not exist, it is created with the value of the sum. If the key already exists, its value is added to the sum.

The `sum` operation can only be performed as part of an atomic operation. The `Deno.AtomicOperation.prototype.mutate({ type: "sum", value })` method can be used to add a sum mutation to an atomic operation.

The sum operation can only be performed on values of type `Deno.KvU64`. Both the operand and the value in the store must be of type `Deno.KvU64`.

If the new value of the key is greater than `2^64 - 1` or less than `0`, the sum operation wraps around. For example, if the value in the store is `2^64 - 1` and the operand is `1`, the new value will be `0`.

Sum operations are always performed in strong consistency mode.

```
await kv.atomic()
  .mutate({
    type: "sum",
    key: ["accounts", "alex"],
    value: new Deno.KvU64(100n),
  })
  .commit();
```

min

The `min` operation atomically sets a key to the minimum of its current value and a given value. If the key does not exist, it is created with the given value. If the key already exists, its value is set to the minimum of its current value and the given value.

The `min` operation can only be performed as part of an atomic operation. The `Deno.AtomicOperation.prototype.mutate({ type: "min", value })` method can be used to add a min mutation to an atomic operation.

The min operation can only be performed on values of type `Deno.KvU64`. Both the operand and the value in the store must be of type `Deno.KvU64`.

Min operations are always performed in strong consistency mode.

```
await kv.atomic()
  .mutate({
    type: "min",
    key: ["accounts", "alex"],
    value: new Deno.KvU64(100n),
  })
  .commit();
```

max

The `max` operation atomically sets a key to the maximum of its current value and a given value. If the key does not exist, it is created with the given value. If the key already exists, its value is set to the maximum of its current value and the given value.

The `max` operation can only be performed as part of an atomic operation. The `Deno.AtomicOperation.prototype.mutate({ type: "max", value })` method can be used to add a max mutation to an atomic operation.

The max operation can only be performed on values of type `Deno.KvU64`. Both the operand and the value in the store must be of type `Deno.KvU64`.

Max operations are always performed in strong consistency mode.

```
await kv.atomic()
  .mutate({
    type: "max",
    key: ["accounts", "alex"],
    value: new Deno.KvU64(100n),
  })
  .commit();
```

watch

The `watch` operation accepts an array of keys, and returns a `ReadableStream`, which emits a new value whenever any of the watched keys change their `versionstamp`. The emitted value is an array of `Deno.KvEntryMaybe` objects.

Note that the returned stream does not return every single intermediate state of the watched keys, but keeps you up to date with the latest state of keys. This means if a key is modified multiple times quickly, you may not receive a notification for every change, but the latest state of the key.

```
const db = await Deno.openKv();

const stream = db.watch([["foo"], ["bar"]]);
for await (const entries of stream) {
  entries[0].key; // ["foo"]
  entries[0].value; // "bar"
  entries[0].versionstamp; // "00000000000000010000"
  entries[1].key; // ["bar"]
  entries[1].value; // null
  entries[1].versionstamp; // null
}
```

Key Expiration (TTL for keys)

Since version 1.36.2, Deno KV supports key expiration, allowing developers to control time to live (TTL) for keys in a KV database. This allows an expiration timestamp to be associated with a key, after which the key will be automatically deleted from the database:

```
const kv = await Deno.openKv();

// `expireIn` is the number of milliseconds after which the key will expire.
function addSession(session: Session, expireIn: number) {
  await kv.set(["sessions", session.id], session, { expireIn });
}
```

Key expiration is supported on both Deno CLI and Deno Deploy.

Atomic expiration of multiple keys

If multiple keys are set in the same atomic operation and have the same `expireIn` value, the expiration of those keys will be atomic. For example:

```
const kv = await Deno.openKv();

function addUnverifiedUser(
  user: User,
  verificationToken: string,
  expireIn: number,
) {
  await kv.atomic()
    .set(["users", user.id], user, { expireIn })
    .set(["verificationTokens", verificationToken], user.id, { expireIn })
    .commit();
}
```

Caveats

The expire timestamp specifies the *earliest* time after which the key can be deleted from the database. An implementation is allowed to expire a key at any time after the specified timestamp, but not before. If you need to strictly enforce an expiration time (e.g. for security purposes), please also add it as a field of your value and do a check after retrieving the value from the database.

Secondary Indexes

Key-value stores like Deno KV organize data as collections of key-value pairs, where each unique key is associated with a single value. This structure enables easy retrieval of values based on their keys but does not allow for querying based on the values themselves. To overcome this constraint, you can create secondary indexes, which store an additional key that lets you look up related data by an alternate attribute (for example, email → user). A best practice is to store a pointer to the primary key in the secondary index, rather than duplicating the full value.

> **TIP**
>
> Prefer storing the primary key (or a compact reference to it) as the value in a secondary index. This reduces storage usage and avoids keeping multiple copies of the same data in sync. The trade-off is a double read when querying through the index (index → primary).
>
> Pros
>
> - Lower storage and write amplification
> - Fewer updates when non-indexed fields change
> - Clearer transactional updates: update primary + index together
>
> Cons
>
> - Requires a second read to resolve the primary value
> - You must maintain referential integrity atomically (create/update/delete in a single transaction)

Maintaining consistency between primary and secondary keys is crucial when using secondary indexes. If a value is updated at the primary key without updating the secondary key, the data returned from a query targeting the secondary key will be incorrect. To ensure that primary and secondary keys always represent the same data, use atomic operations when inserting, updating, or deleting data. This approach ensures that the group of mutation actions are executed as a single unit, and either all succeed or all fail, preventing inconsistencies.

Unique indexes (one-to-one)

Unique indexes have each key in the index associated with exactly one primary key. For example, when storing user data and looking up users by both their unique IDs and email addresses, store user data under two separate keys: one for the primary key (user ID) and another for the secondary index (email → user ID). This setup allows querying users based on either their ID or their email. The secondary index can also enforce uniqueness constraints on values in the store. In the case of user data, use the index to ensure that each email address is associated with only one user.

To implement a unique secondary index for this example, follow these steps:

1. Create a `User` interface representing the data:

```
interface User {
  id: string;
  name: string;
  email: string;
}
```

2. Define an `insertUser` function that stores user data at the primary key and stores a pointer (the primary key) at the secondary key:

```
async function insertUser(user: User) {
  const primaryKey = ["users", user.id] as const;
  const byEmailKey = ["users_by_email", user.email.toLowerCase()] as const;
  const res = await kv.atomic()
    .check({ key: primaryKey, versionstamp: null })
    .check({ key: byEmailKey, versionstamp: null })
    .set(primaryKey, user)
    // store pointer, not full user
    .set(byEmailKey, user.id)
    .commit();
  if (!res.ok) {
    throw new TypeError("User with ID or email already exists");
  }
}
```

> This function performs the insert using an atomic operation that checks that no user with the same ID or email already exists. If either of these constraints is violated, the insert fails and no data is modified.

3. Define a `getUser` function to retrieve a user by their ID:

```
async function getUser(id: string): Promise<User | null> {
  const res = await kv.get<User>(["users", id]);
  return res.value;
}
```

4. Define a `getUserByEmail` function to retrieve a user by their email address using a double lookup (email → user ID → user):

```
async function getUserByEmail(email: string): Promise<User | null> {
  const idRes = await kv.get<string>([
    "users_by_email",
    email.toLowerCase(),
  ]);
  if (!idRes.value) return null;
  const res = await kv.get<User>(["users", idRes.value]);
  return res.value;
}
```

   This function queries the store using the secondary key (`["users_by_email", email]`).

5. Define a `deleteUser` function to delete users by their ID, removing the index entry too:

```
async function deleteUser(id: string) {
  let res = { ok: false } as { ok: boolean };
  while (!res.ok) {
    const cur = await kv.get<User>(["users", id]);
    if (cur.value === null) return;
    res = await kv.atomic()
      .check(cur)
      .delete(["users", id])
      .delete(["users_by_email", cur.value.email.toLowerCase()])
      .commit();
  }
}
```

> This function first retrieves the user by their ID to get the users email address. This is needed to retrieve the email that is needed to construct the key for the secondary index for this user address. It then performs an atomic operation that checks that the user in the database has not changed, and then deletes both the primary and secondary key pointing to the user value. If this fails (the user has been modified between query and delete), the atomic operation aborts. The entire procedure is retried until the delete succeeds. The check is required to prevent race conditions where value may have been modified between the retrieve and delete. This race can occur if an update changes the user's email, because the secondary index moves in this case. The delete of the secondary index then fails, because the delete is targeting the old secondary index key.

Non-Unique Indexes (One-to-Many)

Non-unique indexes are secondary indexes where a single key can be associated with multiple primary keys, allowing you to query for multiple items based on a shared attribute. For example, when querying users by their favorite color, implement this using a non-unique secondary index. The favorite color is a non-unique attribute since multiple users can have the same favorite color.

To implement a non-unique secondary index for this example, follow these steps:

1. Define the `User` interface:

```
interface User {
  id: string;
  name: string;
  favoriteColor: string;
}
```

2. Define the `insertUser` function (store the primary key as the value in the non-unique index; note the composite key includes the user ID to avoid collisions):

```
async function insertUser(user: User) {
  const primaryKey = ["users", user.id] as const;
  const byColorKey = [
    "users_by_favorite_color",
    user.favoriteColor,
    user.id,
  ] as const;
  await kv.atomic()
    .check({ key: primaryKey, versionstamp: null })
    .set(primaryKey, user)
    // store pointer, not full user
    .set(byColorKey, user.id)
    .commit();
}
```

3. Define a function to retrieve users by their favorite color. This performs a double lookup per result (index → primary):

```
async function getUsersByFavoriteColor(color: string): Promise<User[]> {
  const iter = kv.list<string>({
    prefix: ["users_by_favorite_color", color],
  });
  const ids: string[] = [];
  for await (const { value: id } of iter) {
    ids.push(id);
  }
  if (ids.length === 0) return [];
  const results = await kv.getMany<User>(
```

```
    ids.map((id) => ["users", id] as const),
  );
  return results.map((r) => r.value!).filter(Boolean);
}
```

This example demonstrates the use of a non-unique secondary index, `users_by_favorite_color`, which allows querying users based on their favorite color. The index stores pointers (user IDs) and requires resolving to the primary key to read full values.

The primary difference between unique and non-unique indexes lies in the structure and organization of secondary keys. In unique indexes, each secondary key is associated with exactly one primary key, ensuring that the indexed attribute is unique across all records. In non-unique indexes, a single secondary key can be associated with multiple primary keys, as the indexed attribute may be shared among multiple records. To achieve this, non-unique secondary keys are typically structured with an additional unique identifier (e.g., primary key) as part of the key, allowing multiple records with the same attribute to coexist without conflicts.

When duplicating values may be acceptable

While pointer indexes are recommended, duplicating the full value in a secondary index can be acceptable when:

- The value is small and reads occur almost exclusively via the secondary index
- You want to avoid a second read and can tolerate the extra storage
- You can reliably keep the primary and secondary in sync via atomic transactions

If duplicating, ensure inserts/updates/deletes modify both keys in the same atomic transaction.

Migration from duplicated-value indexes

To migrate existing duplicated-value indexes to pointer indexes:

1. Backfill: scan primary keys and set secondary index values to the primary key (e.g., user ID).
2. Cutover: update write paths to maintain pointer indexes; keep the old index temporarily for reads.
3. Cleanup: switch readers to the pointer index, then remove the duplicated index entries.

Transactions

The Deno KV store utilizes *optimistic concurrency control transactions* rather than *interactive transactions* like many SQL systems like PostgreSQL or MySQL. This approach employs versionstamps, which represent the current version of a value for a given key, to manage concurrent access to shared resources without using locks. When a read operation occurs, the system returns a versionstamp for the associated key in addition to the value.

To execute a transaction, one performs an atomic operations that can consist of multiple mutation actions (like set or delete). Along with these actions, key+versionstamp pairs are provided as a condition for the transaction's success. The optimistic concurrency control transaction will only commit if the specified versionstamps match the current version for the values in the database for the corresponding keys. This transaction model ensures data consistency and integrity while allowing concurrent interactions within the Deno KV store.

Because OCC transactions are optimistic, they can fail on commit because the version constraints specified in the atomic operation were violated. This occurs when an agent updates a key used within the transaction between read and commit. When this happens, the agent performing the transaction must retry the transaction.

To illustrate how to use OCC transactions with Deno KV, this example shows how to implement a `transferFunds(from: string, to: string, amount: number)` function for an account ledger. The account ledger stores the balance for each account in the key-value store. The keys are prefixed by `"account"`, followed by the account identifier: `["account", "alice"]`. The value stored for each key is a number that represents the account balance.

Here's a step-by-step example of implementing this `transferFunds` function:

```
async function transferFunds(sender: string, receiver: string, amount: number) {
  if (amount <= 0) throw new Error("Amount must be positive");

  // Construct the KV keys for the sender and receiver accounts.
  const senderKey = ["account", sender];
  const receiverKey = ["account", receiver];

  // Retry the transaction until it succeeds.
  let res = { ok: false };
  while (!res.ok) {
    // Read the current balance of both accounts.
    const [senderRes, receiverRes] = await kv.getMany([senderKey, receiverKey]);
    if (senderRes.value === null) {
      throw new Error(`Account ${sender} not found`);
    }
    if (receiverRes.value === null) {
      throw new Error(`Account ${receiver} not found`);
    }

    const senderBalance = senderRes.value;
    const receiverBalance = receiverRes.value;

    // Ensure the sender has a sufficient balance to complete the transfer.
    if (senderBalance < amount) {
      throw new Error(
        `Insufficient funds to transfer ${amount} from ${sender}`,
      );
    }
```

```
    // Perform the transfer.
    const newSenderBalance = senderBalance - amount;
    const newReceiverBalance = receiverBalance + amount;

    // Attempt to commit the transaction. `res` returns an object with
    // `ok: false` if the transaction fails to commit due to a check failure
    // (i.e. the versionstamp for a key has changed)
    res = await kv.atomic()
      .check(senderRes) // Ensure the sender's balance hasn't changed.
      .check(receiverRes) // Ensure the receiver's balance hasn't changed.
      .set(senderKey, newSenderBalance) // Update the sender's balance.
      .set(receiverKey, newReceiverBalance) // Update the receiver's balance.
      .commit();
  }
}
```

In this example, the `transferFunds` function reads the balances and versionstamps of both accounts, calculates the new balances after the transfer, and checks if there are sufficient funds in account A. It then performs an atomic operation, setting the new balances with the versionstamp constraints. If the transaction is successful, the loop exits. If the version constraints are violated, the transaction fails, and the loop retries the transaction until it succeeds.

Limits

In addition to a max key size of 2 KiB and max value size of 64 KiB, there are certain limits with the Deno KV transaction API:

- **Max keys per `kv.getMany()`**: 10
- **Max batch size per `kv.list()`**: 1000
- **Max checks in an atomic operation**: 100
- **Max mutations in an atomic operation**: 1000
- **Max total size of an atomic operation**: 800 KiB. This includes all keys and values in checks and mutations, and encoding overhead counts toward this limit as well.
- **Max total size of keys**: 90 KiB. This includes all keys in checks and mutations, and encoding overhead counts toward this limit as well.
- **Max watched keys per `kv.watch()`** 10

Using KV in Node.js

> ⚠ **WARNING**
>
> This page covers connecting to a Deno KV database hosted on **Deno Deploy Classic** (dash.deno.com). Deploy Classic will be shut down on July 20, 2026 — see the migration guide for details.
>
> On the new Deno Deploy platform, KV databases are provisioned per app through the Databases feature and accessed from your app code via `Deno.openKv()` without a connection URL — see Deno KV on Deno Deploy. External KV Connect access to new-Deploy KV databases is not currently exposed.

Connecting from Node.js to a Deno KV database hosted on Deno Deploy Classic is supported via our official client library on npm. You can find usage instructions for this option below. The same library can also connect to any other endpoint implementing the open KV Connect protocol, such as a self-hosted denokv instance.

Installation and usage

Use your preferred npm client to install the client library for Node.js using one of the commands below.

npm

```
npm install @deno/kv
```

pnpm

```
pnpm add @deno/kv
```

yarn

```
yarn add @deno/kv
```

Once you've added the package to your Node project, you can import the `openKv` function (supports both ESM `import` and CJS `require`-based usage):

```
import { openKv } from "@deno/kv";

// Connect to a KV instance
const kv = await openKv("<KV Connect URL>");

// Write some data
await kv.set(["users", "alice"], { name: "Alice" });

// Read it back
const result = await kv.get(["users", "alice"]);
console.log(result.value); // { name: "Alice" }
```

By default, the access token used for authentication comes from the `DENO_KV_ACCESS_TOKEN` environment variable. You can also pass it explicitly:

```
import { openKv } from "@deno/kv";

const kv = await openKv("<KV Connect URL>", { accessToken: myToken });
```

Once your Deno KV client is initialized, the same API available in Deno may be used in Node as well.

KV Connect URLs

Connecting to a KV database outside of Deno requires a KV Connect URL. A KV Connect URL for a database hosted on Deno Deploy Classic will be in this format: `https://api.deno.com/databases/<database-id>/connect`.

The `database-id` for your project can be found in the Deno Deploy Classic dashboard, under your project's "KV" tab.

[Connection string locations in Deploy]

More information

More information about how to use the Deno KV module for Node can be found on the project's README page.

Data Modeling in TypeScript

In TypeScript applications, it is usually desirable to create strongly-typed, well-documented objects to contain the data that your application operates on. Using interfaces or classes, you can describe both the shape and behavior of objects in your programs.

If you are using Deno KV, however, there is a bit of extra work required to persist and retrieve objects that are strongly typed. In this guide, we'll cover strategies for working with strongly typed objects going into and back out from Deno KV.

Using interfaces and type assertions

When storing and retrieving application data in Deno KV, you might want to begin by describing the shape of your data using TypeScript interfaces. Below is an object model which describes some key components of a blogging system:

model.ts

```
export interface Author {
  username: string;
  fullName: string;
}

export interface Post {
  slug: string;
  title: string;
  body: string;
  author: Author;
  createdAt: Date;
  updatedAt: Date;
}
```

This object model describes a blog post and an associated author.

With Deno KV, you can use these TypeScript interfaces like data transfer objects (DTOs) - a strongly typed wrapper around the otherwise untyped objects you might send to or receive from Deno KV.

Without any additional work, you can happily store the contents of one of these DTOs in Deno KV.

```
import { Author } from "./model.ts";

const kv = await Deno.openKv();

const a: Author = {
  username: "acdoyle",
  fullName: "Arthur Conan Doyle",
};

await kv.set(["authors", a.username], a);
```

When retrieving this same object from Deno KV, however, it won't by default have type information associated with it. If you know the shape of the object that was stored for the key, however, you can use type assertion to inform the TypeScript compiler about the shape of an object.

```
import { Author } from "./model.ts";

const kv = await Deno.openKv();

const r = await kv.get(["authors", "acdoyle"]);
```

```
const ac = r.value as Author;

console.log(ac.fullName);
```

You can also specify an optional type parameter for `get`:

```
import { Author } from "./model.ts";

const kv = await Deno.openKv();

const r = await kv.get<Author>(["authors", "acdoyle"]);

console.log(r.value.fullName);
```

For simpler data structures, this technique may be sufficient. But often, you will want or need to apply some business logic when creating or accessing your domain objects. When this need arises, you can develop a set of pure functions that can operate on your DTOs.

Encapsulating business logic with a service layer

When your application's persistence needs become more complex - such as when you need to create secondary indexes to query your data by different keys, or maintain relationships between objects - you will want to create a set of functions to sit on top of your DTOs to ensure that the data being passed around is valid (and not merely typed correctly).

From our business objects above, the `Post` object is complex enough where it is likely to need a small layer of code to save and retrieve an instance of the object. Below is an example of two functions that wrap the underlying Deno KV APIs, and return strongly typed object instances for the `Post` interface.

Notably, we need to store an identifier for an `Author` object, so we can retrieve author information from KV later.

```
import { Author, Post } from "./model.ts";

const kv = await Deno.openKv();

interface RawPost extends Post {
  authorUsername: string;
}

export async function savePost(p: Post): Promise<Post> {
  const postData: RawPost = Object.assign({}, p, {
    authorUsername: p.author.username,
  });

  await kv.set(["posts", p.slug], postData);
  return p;
}

export async function getPost(slug: string): Promise<Post> {
  const postResponse = await kv.get(["posts", slug]);
  const rawPost = postResponse.value as RawPost;
  const authorResponse = await kv.get(["authors", rawPost.authorUsername]);

  const author = authorResponse.value as Author;
  const post = Object.assign({}, postResponse.value, {
    author,
  }) as Post;

  return post;
}
```

This thin layer uses a `RawPost` interface, which extends the actual `Post` interface, to include some additional data that is used to reference data at another index (the associated `Author` object).

The `savePost` and `getPost` functions take the place of a direct Deno KV `get` or `set` operation, so that they can properly serialize and "hydrate" model objects for us with appropriate types and associations.

Backups

KV databases hosted on Deno Deploy can be continuously backed up to your own S3-compatible storage buckets. This is in addition to the replication and backups that we internally perform for all data stored in hosted Deno KV databases to ensure high availability and data durability.

This backup happens continuously with very little lag, enabling *https://en.wikipedia.org/wiki/Point-in-time_recovery[point-in-time-recovery]* and live replication. Enabling backup for KV databases unlocks various interesting use-cases:

- Retrieving a consistent snapshot of your data at any point in time in the past
- Running a read-only data replica independent of Deno Deploy
- Pushing data into your favorite data pipeline by piping mutations into streaming platforms and analytical databases like Kafka, BigQuery and ClickHouse

Configuring backup to Amazon S3

First you must create a bucket on AWS:

AWS Console

1. Go to the AWS S3 console
2. Click "Create bucket"
3. Enter a bucket name and choose a AWS region, then scroll down and click "Next"

AWS CLI

1. Install the AWS CLI
2. Run `aws s3api create-bucket --bucket <bucket-name> --region <region> --create-bucket-configuration LocationConstraint=<region>` (replace `<bucket-name>` and `<region>` with your own values)

Then, create an IAM policy with `PutObject` access to the bucket, attach it to an IAM user, and create access keys for that user:

AWS Console

1. Go to the AWS IAM console
2. Click "Policies" in the left sidebar
3. Click on "Create policy"
4. Select the "JSON" the policy editor and paste the following policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "KVBackup",
      "Effect": "Allow",
      "Action": "s3:PutObject",
      "Resource": "arn:aws:s3:::<bucket-name>/*"
    }
  ]
}
```

Replace <bucket-name> with the name of the bucket you created earlier.

5. Click "Review policy"
6. Enter a name for the policy and click "Create policy"
7. Click "Users" in the left sidebar
8. Click "Add user"
9. Enter a name for the user and click "Next"
10. Click "Attach policies directly"
11. Search for the policy you created earlier and click the checkbox next to it
12. Click "Next"
13. Click "Create user"
14. Click on the user you just created
15. Click "Security credentials" and then "Create access key"
16. Select "Other", then click "Next"
17. Enter a description for the access key and click "Create access key"
18. Copy the access key ID and secret access key and save them somewhere safe. You will need them later, and you will not be able to retrieve them again.

AWS CLI

1. Copy the following command to your terminal, and replace <bucket-name> with the name of the bucket you created earlier, then run it:

```
aws iam create-policy --policy-name <policy-name> --policy-document '{"Version":"2012-10-17","Statement":
[{"Sid":"KVBackup","Effect":"Allow","Action":"s3:PutObject","Resource":"arn:aws:s3:::<bucket-name>/*"}]}'
```

2. Copy the following command to your terminal, and replace <user-name> with a name for the user you are creating, then run it:

```
aws iam create-user --user-name <user-name>
```

3. Copy the following command to your terminal, and replace <policy-arn> with the ARN of the policy you created in step 1, and <user-name> with the name of the user you created in the previous step, then run it:

```
aws iam attach-user-policy --policy-arn <policy-arn> --user-name <user-name>
```

4. Copy the following command to your terminal, and replace <user-name> with the name of the user you created in step 2, then run it:

```
aws iam create-access-key --user-name <user-name>
```

5. Copy the access key ID and secret access key and save them somewhere safe. You will need them later, and you will not be able to retrieve them again.

Now visit the Deno Deploy dashboard, and click on the "KV" tab in your project. Scroll to the "Backup" section, and click on "AWS S3". Enter the bucket name, access key ID, and secret access key you created earlier, and the region the bucket is in. Then click "Save".

[add backup to dashboard]

The backup will start immediately. Once the data has been backed up, and continuous backup is active, you will see the status change to "Active".

Configuring backup to Google Cloud Storage

Google Cloud Storage (GCS) is compatible with the S3 protocol, and can also be used as a backup target.

First you must create a bucket on GCP:

GCP Console

1. Go to the GCP Cloud Storage console
2. Click on "Create" in the top bar
3. Enter a bucket name, choose a location, and click "Create"

gcloud CLI

1. Install the gcloud CLI
2. Run `gcloud storage buckets create <bucket-name> --location <location>` (replace `<bucket-name>` and `<location>` with your own values)

Then, create a service account with `Storage Object Admin` access to the bucket, and create an HMAC access key for the service account:

GCP Console

1. Go to the GCP IAM console
2. Click on "Service accounts" in the left sidebar
3. Click on "Create service account"
4. Enter a name for the service account and click "Done"
5. Copy the email for the service account you just created. You will need it later.
6. Go to the GCP Cloud Storage console
7. Click on the bucket you created earlier
8. Click on "Permissions" in the toolbar
9. Click "Grant access"
10. Paste the email for the service account you copied earlier into the "New principals" field
11. Select "Storage Object Admin" from the "Select a role" dropdown
12. Click "Save"
13. Click on "Settings" in the left sidebar (still in the Cloud Storage console)
14. Click on the "Interoperability" tab
15. Click on "Create a key for a service account"
16. Select the service account you created earlier
17. Click "Create key"
18. Copy the access key and secret access key and save them somewhere safe. You will need them later, and you will not be able to retrieve them again.

gcloud CLI

1. Run the following command, replacing `<service-account-name>` with a name for the service account you are creating:

```
gcloud iam service-accounts create <service-account-name>
```

2. Run the following command, replacing `<bucket-name>` with the name of the bucket you created earlier, and `<service-account-email>` with the email of the service account you created in the previous step:

```
gsutil iam ch serviceAccount:<service-account-email>:objectAdmin gs://<bucket-name>
```

3. Run the following command, replacing `<service-account-email>` with the email of the service account you created in the previous step:

```
gcloud storage hmac create <service-account-email>
```

4. Copy the `accessId` and `secret` and save them somewhere safe. You will need them later, and you will not be able to retrieve them again.

Now visit the Deno Deploy dashboard, and click on the "KV" tab in your project. Scroll to the "Backup" section, and click on "Google Cloud Storage". Enter the bucket name, access key ID, and secret access key you created earlier, and the region the bucket is in. Then click "Save".

The backup will start immediately. Once the data has been backed up, and continuous backup is active, you will see the status change to "Active".

Using backups

S3 backups can be used with the `denokv` tool. Please refer to the documentation for more details.

05

Deno Sandbox

Deno Sandbox 151

Getting started 155

Create a Deno Sandbox 161

Management via CLI 165

Expose HTTP 171

SSH 176

Programmatic management of Deno Deploy Apps 179

Volumes & Snapshots 183

Sandbox Timeouts 190

Security 192

Deno Sandbox

Deno Sandbox brings instant Linux microVMs to Deno Deploy. Each sandbox boots in under a second, is API driven from the `@deno/sandbox` SDK, and is torn down as soon as you are done. The result is on-demand compute that feels like opening a terminal, yet ships with production-grade isolation and observability.

What is a Deno Sandbox?

- Individual Linux microVMs orchestrated by Deno Deploy
- Designed for running untrusted code
- Instantly available; boot times measured in milliseconds
- Ephemeral by default but able to persist beyond the current connection lifetime
- Able to access durable storage via volumes
- Fully API driven: create, run commands, and tear down from code

Ideal use cases

Deno Sandbox specializes in workloads where code needs to be generated, evaluated, or safely executed on behalf of an untrusted user. They are ideal for:

- AI agents and copilots that need to run code as they reason
- Secure plugin or extension systems
- Vibe-coding and collaborative IDE experiences
- Ephemeral CI runners and smoke tests
- Customer supplied or user generated code paths
- Instant dev servers and preview environments

This is compute built not just for developers, but for software that builds software.

Run real workloads

Once the Deno Sandbox exists, you get a full Linux environment with files, processes, package managers, and background services:

JavaScript

```
import { Sandbox } from "@deno/sandbox";
await using sandbox = await Sandbox.create();
await sandbox.sh`ls -lh /`;
```

Python

```
from deno_sandbox import DenoDeploy

def main():
  sdk = DenoDeploy()

  with sdk.sandbox.create() as sandbox:
    process = sandbox.spawn("ls", args=["-lh"])
    process.wait()
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

async def main():
  sdk = AsyncDenoDeploy()

  async with sdk.sandbox.create() as sandbox:
    process = await sandbox.spawn("ls", args=["-lh"])
    await process.wait()
```

Security policies

Provision a sandbox so that it can only talk to approved hosts:

JavaScript

```
await Sandbox.create({
  allowNet: ["google.com"],
});
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandboxes.create(allowNet=["google.com"]) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python

```
sdk = DenoDeploy()

with sdk.sandboxes.create(allowNet=["google.com"]) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Secrets never enter the sandbox environment. The real value is substituted only when the sandbox makes outbound requests to approved hosts.

JavaScript

```
await Sandbox.create({
  allowNet: ["api.openai.com"],
  secrets: {
    OPENAI_API_KEY: {
      hosts: ["api.openai.com"],
      value: process.env.OPENAI_API_KEY,
    },
  },
});
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandboxes.create(
  allowNet=["api.openai.com"],
  secrets={
    "OPENAI_API_KEY": {
      "hosts": ["api.openai.com"],
      "value": os.environ.get("OPENAI_API_KEY"),
    }
  },
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python

```
sdk = DenoDeploy()

with sdk.sandboxes.create(
  allowNet=["api.openai.com"],
  secrets={
    "OPENAI_API_KEY": {
      "hosts": ["api.openai.com"],
      "value": os.environ.get("OPENAI_API_KEY"),
    }
  },
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Built for instant, safe compute

Developers and AI systems now expect compute that is instant, safe, and globally accessible. Deno Sandbox delivers:

- Instant spin-up with no warm pool to manage
- Dedicated isolation with strict network egress policies
- Full observability alongside Deno Deploy logs and traces
- Region selection, memory sizing, and lifetime controls per sandbox
- Seamless hand-off to Deno Deploy apps when code is production ready

Together, Deno Deploy and Deno Sandbox form a single workflow: code is created, proved safe in a sandbox, and deployed globally without new infrastructure or orchestration layers.

Runtime support

The Deno Sandbox SDK is tested and supported on:

- **Deno:** Latest stable version
- **Node.js:** Version 24+
- **Python:** >=3.10

You can use Deno Sandbox from any environment that can make outbound HTTPS requests to the Deno Deploy API. The JavaScript SDK is available as `@deno/sandbox` on both jsr and npm (the JSR package is optimized for Deno usage). The Python SDK is available as `deno-sandbox` on PyPI. For direct API access, see the REST API documentation.

> **NOTE**
>
> The `await using` syntax requires Node.js 24+. If your project uses earlier Node.js versions, use try/finally blocks instead:
>
> ```
> import { Sandbox } from "@deno/sandbox";
>
> const sandbox = await Sandbox.create();
> try {
>   // ... use sandbox ...
> } finally {
>   await sandbox.close();
> }
> ```

Limits

Deno Sandbox has the following limits:

- **Memory:** 768 MB to 4096 MB (1.2GB default) configurable per sandbox
- **CPU:** 2 vCPU
- **Lifetime:** Configurable per sandbox and bound to a session, up to 30 minutes
- **Disk**: 10 GB of ephemeral storage
- **Concurrency**: 5 concurrent sandboxes per organization (This is the default concurrency limit during the pre-release phase of Deno Sandbox. Contact deploy@deno.com to request a higher limit.)

Exceeding these limits may result in throttling or termination of your sandbox.

Regions

Regions currently supported are:

- `ams` - Amsterdam, Netherlands
- `ord` - Chicago, USA

You can specify the region where the sandbox will be created when creating a new sandbox:

JavaScript

```
import { Sandbox } from "@deno/sandbox";

await using sandbox = await Sandbox.create({ region: "ams" });
```

Python

```
from deno_sandbox import DenoDeploy

def main():
  sdk = DenoDeploy()

  with sdk.sandboxes.create(region="ams") as sandbox:
    print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

async def main():
  sdk = AsyncDenoDeploy()

  async with sdk.sandboxes.create(region="ams") as sandbox:
    print(f"Sandbox {sandbox.id} is ready.")
```

If not specified, the sandbox will be created in the default region.

Learn more

Ready to try it? Follow the Getting started guide to create your first sandbox, obtain an access token, and run code on the Deploy edge.

Getting started

To use Deno Sandbox, you need a Deno Deploy account. If you do not have one yet you can sign up for a free account at console.deno.com.

Access the Deno Sandbox dashboard

1. Visit console.deno.com and sign in with your Deploy account.
2. Choose or create the organization where you want to run Deno Sandbox.
3. Open the **Sandboxes** tab to view existing sandboxes, lifetime usage, and access tokens.

Deno Sandbox and Deno Deploy apps share the same organization boundary, so you can reuse members, tokens, and observability settings across both products.

Create an organization token

The `@deno/sandbox` SDK authenticates using the `DENO_DEPLOY_TOKEN` environment variable. Generate it from **Settings → Organization tokens**, copy the value, and store it securely. Then export it in your local shell or CI job:

```
export DENO_DEPLOY_TOKEN=<your-token>
```

[The Deno Deploy organization tokens screen.]

> **TIP**
> Treat this token like any other production secret. Rotate it from the dashboard if it is ever exposed.

Install the SDK

JavaScript

The SDK works in both Deno and Node.js environments.

```
# Using Deno
deno add jsr:@deno/sandbox

# Using npm
npm install @deno/sandbox

# Using pnpm
pnpm install jsr:@deno/sandbox

# Using yarn
yarn add jsr:@deno/sandbox
```

Python

The SDK works in Python versions `>=3.10`.

```
# Install with uv
uv add deno-sandbox

# or with pip
pip install deno-sandbox
```

Create your first sandbox

JavaScript

```
// main.ts
import { Sandbox } from "@deno/sandbox";
await using sandbox = await Sandbox.create();
await sandbox.sh`ls -lh /`;
```

Python

```
# main.py
from deno_sandbox import DenoDeploy

def main():
  sdk = DenoDeploy()

  with sdk.sandbox.create() as sandbox:
    process = sandbox.spawn("ls", args=["-lh"])
    process.wait()

if __name__ == '__main__':
  main()
```

Python (Async)

```
# main.py
import asyncio
from deno_sandbox import DenoDeploy

async def main():
  sdk = DenoDeploy()

  async with sdk.sandbox.create() as sandbox:
    process = await sandbox.spawn("ls", args=["-lh"])
    await process.wait()

if __name__ == '__main__':
  asyncio.run(main())
```

Run your sandbox code

JavaScript

This code will require access to the network to reach the Deploy edge where the sandbox will be created, and also access to the environment variables to authenticate with the Deploy API, so we'll pass in the `--allow-net` and `--allow-env` flags to the `deno run` command (or use the shorthand `-EN`).

```
deno -EN main.ts
```

Python

To run the script we just created, execute:

```
uv run main.py
```

Any sandbox you create will be listed in the **Sandboxes** tab of your Deno Deploy organization.

[The list of sandboxes created in the Deno Deploy console.]

Details about the sandbox will be shown in its **Event log**.

[The sandbox event log details in the Deno Deploy console.]

Configuring your sandbox

When creating a sandbox with `Sandbox.create()`, you can configure it with the following options:

- `allowNet` / `allow_net`: Optional list of allowed outbound hosts. See Outbound network control.
- `secrets`: Secret substitution rules for outbound requests. See Secret redaction and substitution.
- `region`: Deploy region where the sandbox will be created.
- `memoryMb` / `memory_mb`: Amount of memory allocated to the sandbox.
- `timeout`: Timeout of the sandbox.
- `labels`: Arbitrary key/value tags to help identify and manage sandboxes

JavaScript

```
await using sandbox = await Sandbox.create({
  allowNet: ["api.stripe.com", "api.openai.com"], // optional: list of hosts that this sandbox can communicate
with
  region: "ams", // optional: choose the Deploy region
  memoryMb: 1024, // optional: pick the RAM size (768-4096)
});
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create(
  allow_net=["api.stripe.com", "api.openai.com"],  # optional: list of hosts that this sandbox can communicate
with
  region="ams",  # optional: choose the Deploy region
  memory_mb=1024,  # optional: pick the RAM size (768-4096)
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(
  allow_net=["api.stripe.com", "api.openai.com"],  # optional: list of hosts that this sandbox can communicate
with
  region="ams",  # optional: choose the Deploy region
  memory_mb=1024,  # optional: pick the RAM size (768-4096)
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Running commands and scripts

Deno Sandbox exposes familiar filesystem and process APIs to run commands, upload files, and spawn long-running services.

You can for example list files in the root directory:

JavaScript

```
await sandbox.sh`ls -lh /`;
```

Python

```
process = sandbox.spawn("ls", args=["-lh", "/"])
process.wait()
```

Python (Async)

```
process = await sandbox.spawn("ls", args=["-lh", "/"])
await process.wait()
```

Or upload a script from the local filesystem and run it:

JavaScript

```
// Upload a file to a specific path in the sandbox
await sandbox.fs.upload("./local-hello.ts", "./hello.ts");
const proc = await sandbox.spawn("deno", {
  args: ["run", "hello.ts"],
  stdout: "piped",
});
for await (const chunk of proc.stdout) {
  console.log(new TextDecoder().decode(chunk));
}
await proc.status;
```

Python

```
# Upload a file to a specific path in the sandbox
sandbox.fs.upload("./local-hello.py", "./hello.py")
proc = sandbox.spawn("python", args=["hello.py"], stdout="piped")
for chunk in proc.stdout:
  print(chunk.decode())
proc.wait()
```

Python (Async)

```
# Upload a file to a specific path in the sandbox
await sandbox.fs.upload("./local-hello.py", "./hello.py")
proc = await sandbox.spawn("python", args=["hello.py"], stdout="piped")
async for chunk in proc.stdout:
  print(chunk.decode())
await proc.wait()
```

You can keep state between commands, stream stdout and stderr, or open an interactive REPL for agent-style workflows.

Deploying from a Deno Sandbox

The snippet below walks through an end-to-end workflow: it creates a Deploy app, boots a high-memory sandbox for heavier builds, scaffolds and builds a Next.js project inside that VM, then calls `sandbox.deno.deploy()` to push the compiled artifacts while streaming build logs back to your terminal.

JavaScript

```
import { Client, Sandbox } from "@deno/sandbox";

const client = new Client();
const app = await client.apps.create();

await using sandbox = await Sandbox.create({ memoryMb: 4096 });
console.log("Created sandbox", sandbox);

await sandbox
```

```
  .sh`deno -A npm:create-next-app@latest --yes --skip-install my-app`;
await sandbox.sh`cd my-app && deno install`;
await sandbox.sh`cd my-app && deno task build`;
await sandbox.sh`cd my-app && du -sh .`;
const build = await sandbox.deno.deploy(app.slug, {
  path: "my-app",
  production: true,
  build: {
    entrypoint: "node_modules/.bin/next",
    args: ["start"],
  },
});

for await (const log of build.logs()) {
  console.log(log.message);
}
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

app = sdk.apps.create(slug="my-next-app")

with sdk.sandbox.create(memory_mb=4096) as sandbox:
  print(f"Created sandbox {sandbox.id}")

  sandbox.spawn("deno", args=["-A", "npm:create-next-app@latest", "--yes", "--skip-install", "my-app"]).wait()
  sandbox.spawn("sh", args=["-c", "cd my-app && deno install"]).wait()
  sandbox.spawn("sh", args=["-c", "cd my-app && deno task build"]).wait()

  build = sandbox.deno.deploy(
    app["slug"],
    path="my-app",
    production=True,
    entrypoint="node_modules/.bin/next",
    args=["start"],
  )

  for log in build.logs():
    print(log["message"])
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

app = await sdk.apps.create(slug="my-next-app")

async with sdk.sandbox.create(memory_mb=4096) as sandbox:
  print(f"Created sandbox {sandbox.id}")

  proc = await sandbox.spawn("deno", args=["-A", "npm:create-next-app@latest", "--yes", "--skip-install", "my-
app"])
  await proc.wait()
  proc = await sandbox.spawn("sh", args=["-c", "cd my-app && deno install"])
  await proc.wait()
  proc = await sandbox.spawn("sh", args=["-c", "cd my-app && deno task build"])
  await proc.wait()

  build = await sandbox.deno.deploy(
    app["slug"],
    path="my-app",
    production=True,
```

```
    entrypoint="node_modules/.bin/next",
    args=["start"],
  )

  async for log in build.logs():
    print(log["message"])
```

Tuning timeout, cleanup, and reconnect

- `timeout: "session"` (default) destroys the VM once your script finishes.
- Provide durations such as `"5m"` to keep the sandbox alive even after the client disconnects. You can later `Sandbox.connect({ id })` to resume work.
- Cleanup happens automatically when your code drops the last reference (or the `await using` block ends). Call `sandbox.kill()` only if you need to tear the VM down ahead of that schedule.

Observability is shared with Deno Deploy: every sandbox logs, trace, and metric is visible in the Deno Deploy dashboard so you can debug agent runs the same way you debug production apps.

Create a Deno Sandbox

The sandbox creation method is the primary entry point for provisioning an isolated Linux microVM on the Deploy edge. It returns a connected sandbox instance that you can use to run commands, upload files, expose HTTP endpoints, or request SSH access.

JavaScript

```
import { Sandbox } from "@deno/sandbox";

await using sandbox = await Sandbox.create();
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create() as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create() as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

By default, this creates an ephemeral sandbox in the closest Deploy region with 1280 MB of RAM, no outbound network access, and a timeout bound to the current process. You can tailor the sandbox by passing an options object.

Available options

Option	Description
`region`	Eg `ams` or `ord`
`allowNet` / `allow_net`	Optional list of allowed outbound hosts. See Outbound network control.
`secrets`	Secrets to substitute on outbound requests to approved hosts. See Secret redaction and substitution.
`memoryMb` / `memory_mb`	Allocate between 768 and 4096 MB of RAM for memory-heavy tasks or tighter budgets.
`timeout`	How long the sandbox stays alive in (m) or (s) such as `5m`
`labels`	Attach arbitrary key/value labels to help identify and manage sandboxes
`env`	Environment variables to start the sandbox with.

Example configurations

Allow outbound traffic to specific APIs

JavaScript

```
const sandbox = await Sandbox.create({
  allowNet: ["api.openai.com", "api.stripe.com"],
});
```

Python

```
sdk = DenoDeploy()

with sdk.sandbox.create(
  allow_net=["api.openai.com", "api.stripe.com"]
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(
  allow_net=["api.openai.com", "api.stripe.com"]
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Configure secret substitution for approved hosts

JavaScript

```
const sandbox = await Sandbox.create({
  allowNet: ["api.openai.com"],
  secrets: {
    OPENAI_API_KEY: {
      hosts: ["api.openai.com"],
      value: process.env.OPENAI_API_KEY,
    },
  },
});
```

Python

```
import os
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create(
  allow_net=["api.openai.com"],
  secrets={
    "OPENAI_API_KEY": {
      "hosts": ["api.openai.com"],
      "value": os.environ.get("OPENAI_API_KEY"),
    }
  }
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
import os
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()
```

```
async with sdk.sandbox.create(
  allow_net=["api.openai.com"],
  secrets={
    "OPENAI_API_KEY": {
      "hosts": ["api.openai.com"],
      "value": os.environ.get("OPENAI_API_KEY"),
    }
  }
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Run in a specific region with more memory

JavaScript

```
const sandbox = await Sandbox.create({
  region: "ams",
  memoryMb: 2048,
});
```

Python

```
sdk = DenoDeploy()

with sdk.sandbox.create(
  region="ams",
  memory_mb=2048
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(
  region="ams",
  memory_mb=2048
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Keep the sandbox alive for later inspection

JavaScript

```
const sandbox = await Sandbox.create({ timeout: "10m" });
const id = sandbox.id;
await sandbox.close(); // disconnect but leave VM running

// ...later...
const reconnected = await Sandbox.connect({ id });
```

Python

```
sdk = DenoDeploy()

with sdk.sandbox.create(timeout="10m") as sandbox:
  sandbox_id = sandbox.id
  sandbox.close()  # disconnect but leave VM running

# ...later...
with sdk.sandbox.connect(sandbox_id) as reconnected:
  print(f"Reconnected to {reconnected.id}")
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(timeout="10m") as sandbox:
  sandbox_id = sandbox.id
  await sandbox.close()  # disconnect but leave VM running

# ...later...
async with sdk.sandbox.connect(sandbox_id) as reconnected:
  print(f"Reconnected to {reconnected.id}")
```

Provide default environment variables

JavaScript

```
const sandbox = await Sandbox.create({
  env: {
    NODE_ENV: "development",
    FEATURE_FLAG: "agents",
  },
});
```

Python

```
sdk = DenoDeploy()

with sdk.sandbox.create(
  env={
    "NODE_ENV": "development",
    "FEATURE_FLAG": "agents",
  }
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(
  env={
    "NODE_ENV": "development",
    "FEATURE_FLAG": "agents",
  }
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Tips

- Keep network allowlists as narrow as possible to block exfiltration attempts.
- Use metadata keys such as `agentId` or `customerId` to trace sandboxes in the Deploy dashboard.
- Let context managers (Python) or automatic disposal (JavaScript) handle cleanup. Call `sandbox.kill()` only when you need to terminate it prior to that automatic cleanup.
- For long-lived services, migrate from a Deno Sandbox to a Deploy app once the code stabilizes.

Management via CLI

The Deno CLI includes built-in commands for managing your Deno Sandbox instances, allowing you to create, control, and interact with them from your terminal.

This integration makes Deno Sandbox management feel natural within your existing Deno workflow.

Creating your first sandbox

The simplest way to get started is with `deno sandbox create`. By default, this creates an interactive session-based sandbox that automatically opens an SSH connection when ready:

```
deno sandbox create
```

If SSH isn't available on your system, it will display connection information instead. The sandbox cleans itself up when you exit the session.

For development work, you'll often want to copy your project files into the sandbox. The `--copy` option uploads files to the `/app` directory inside the sandbox:

```
deno sandbox create --copy ./my-project
```

You can copy multiple directories during creation:

```
deno sandbox create --copy ./src --copy ./config
```

If you need the sandbox to run longer than a single session, specify a timeout with `--timeout`:

```
deno sandbox create --timeout 2m
```

You can also create a sandbox with a custom memory limit:

```
deno sandbox create --memory 2gib
```

To expose HTTP ports for web applications:

```
deno sandbox create --expose-http 3000
```

You can mount persistent volumes to your sandbox using the `--volume` flag:

```
deno sandbox create --volume my-volume:/data
```

To create a sandbox and run a command immediately:

```
deno sandbox create ls /
```

This is especially useful for building and testing projects. You can copy files and run your build process in one command:

```
deno sandbox create --copy ./app --cwd /app "npm i && npm start"
```

For web applications, you can expose ports to access running services:

```
deno sandbox create --expose-http 3000 --copy ./web-app --cwd /app "npm i && npm run dev"
```

Complex workflows can be expressed as quoted command chains:

```
deno sandbox create --copy ./app --cwd /app "npm install && npm test && npm run build"
```

Viewing your Deno Sandbox

Use `deno sandbox list` (or `deno sandbox ls`) to see all sandboxes in your organization:

```
$ deno sandbox list
ID                          CREATED                 REGION   STATUS    UPTIME
sbx_ord_1at5nn58e77rtd11e3k3   2026-01-30 18:33:40.79   ord      running   26.9s
sbx_ord_fwnygdsnszfe5ghafyx8   2026-01-30 18:31:40.90   ord      stopped   5.1s
sbx_ord_4xqcyahb8ye2r5a643de   2026-01-30 18:29:59.10   ord      stopped   9.4s
```

This shows each sandbox's ID, creation time, region, status, and uptime.

Running commands remotely

The `deno sandbox exec` command lets you run individual commands in any running sandbox without opening an interactive session. This is perfect for automation, CI/CD pipelines, or quick one-off tasks:

```
deno sandbox exec sbx_ord_abc123def456 ls -la
```

Most of the time, you'll want to work in the `/app` directory where your copied files live. Use `--cwd` to set the working directory:

```
deno sandbox exec sbx_ord_abc123def456 --cwd /app npm install
```

For scripting or automation, use `--quiet` to suppress command output:

```
deno sandbox exec sbx_ord_abc123def456 --quiet --cwd /app npm test
```

You can also run complex command chains by quoting the entire command:

```
deno sandbox exec sbx_ord_abc123def456 --cwd /app "npm install && npm test"
```

The exec command works naturally with Unix pipes and standard input/output. You can pipe the output of sandbox commands to local tools:

```
deno sandbox exec sbx_ord_abc123def456 'ls -lh /' | wc -l
```

Or pipe local data into sandbox processes for processing:

```
cat large-dataset.csv | deno sandbox exec sbx_ord_abc123def456 --cwd /app "deno run -A main.ts"
```

This makes it easy to integrate sandbox processing into larger Unix workflows and data pipelines.

Transferring files

While you can copy files during Deno Sandbox creation, you might need to update or retrieve files later. The `deno sandbox copy` command (also available as `deno sandbox cp`) transfers files in any direction: from your local machine to a Deno Sandbox, from a Deno Sandbox back to your machine, or even between different sandboxes.

Copy files from your local machine to a sandbox:

```
deno sandbox copy ./app.js sbx_ord_abc123def456:/app/
```

Retrieve files from a sandbox to your local machine:

```
deno sandbox copy sbx_ord_abc123def456:/app/results.json ./output/
```

Copy files between different sandboxes:

```
deno sandbox copy sbx_ord_abc123def456:/app/data.csv sbx_ord_xyz789uvw012:/app/input/
```

You can use glob patterns to copy multiple files from Deno Sandbox:

```
deno sandbox copy sbx_ord_abc123def456:/app/*.json ./config/
deno sandbox copy sbx_ord_abc123def456:/app/logs/*.log ./logs/
```

You can copy multiple files and directories at once:

```
deno sandbox copy ./src/ ./package.json sbx_ord_abc123def456:/app/
```

The target path can be customized to organize files within the sandbox:

```
deno sandbox copy ./frontend sbx_ord_abc123def456:/app/web/
```

Deploying sandboxes

You can deploy a running sandbox to a Deno Deploy app using the `deno sandbox deploy` command:

```
deno sandbox deploy sbx_ord_abc123def456 my-app
```

By default, this deploys to a preview deployment. To deploy directly to production:

```
deno sandbox deploy --prod sbx_ord_abc123def456 my-app
```

You can specify a custom working directory and entrypoint:

```
deno sandbox deploy --cwd /app --entrypoint main.ts sbx_ord_abc123def456 my-app
```

To pass arguments to the entrypoint script:

```
deno sandbox deploy --args --port 8080 sbx_ord_abc123def456 my-app
```

Managing volumes

The sandbox system supports persistent volumes for data that needs to survive across sandbox instances. Use the `deno sandbox volumes` command to manage them.

Creating volumes

Create a new volume with a specific name, capacity, and region:

```
deno sandbox volumes create my-volume --capacity 10gb --region ord
```

Listing volumes

List all volumes in your organization:

```
deno sandbox volumes list
```

You can also search for specific volumes:

```
deno sandbox volumes list my-volume
```

Deleting volumes

Remove a volume when you no longer need it:

```
deno sandbox volumes delete my-volume
```

Managing snapshots

Snapshots are read-only images created from volumes. Use them to pre-install software once, then boot new sandboxes instantly with everything ready. See Volumes & Snapshots for the full workflow.

Creating snapshots

Create a snapshot from an existing volume:

```
deno sandbox snapshots create my-volume my-snapshot
```

Listing snapshots

List all snapshots in your organization:

```
$ deno sandbox snapshots list
ID                          SLUG          REGION   ALLOCATED    BOOTABLE
snp_ord_spmbe47dysccpy277ma6   my-snapshot   ord      217.05 MiB   TRUE
```

Deleting snapshots

Remove a snapshot when you no longer need it:

```
deno sandbox snapshots delete my-snapshot
```

Switching organizations

The `deno sandbox switch` command allows you to switch between different organizations in your configuration:

```
deno sandbox switch
```

This is useful when working with multiple organizations.

Interactive access

When you need to work interactively within a sandbox; be it editing files, debugging issues, or exploring the environment, you can use `deno sandbox ssh`:

```
deno sandbox ssh sbx_ord_abc123def456
```

This gives you a full Linux shell inside the sandbox where you can use any command-line tools, edit files with vim or nano, monitor processes, or install additional software as needed. The sandbox continues running after you disconnect, so you can reconnect later or use other commands to interact with it remotely.

Managing sandbox timeout

Extending sandbox duration

Sometimes you'll need more time to complete your work in a running sandbox. The `deno sandbox extend` command allows you to extend the timeout of any running sandbox without interrupting ongoing processes:

```
deno sandbox extend sbx_ord_abc123def456 30m
```

The extend command works seamlessly with any sandbox state; whether you're SSH'd into it, running remote commands, or have background processes running. All active connections and processes continue uninterrupted while the sandbox's expiration time is updated.

Cleanup and termination

When you're finished with a sandbox, use `deno sandbox kill` (or `deno sandbox rm`) to terminate it and free up resources:

```
deno sandbox kill sbx_ord_abc123def456
```

This immediately stops all processes in the sandbox and releases its resources. Be sure to save any important work before terminating a sandbox, as all data inside will be lost.

Common workflows

Development and testing

A typical development workflow involves creating a sandbox with your code, setting up dependencies, and running tests:

```
deno sandbox create --copy ./my-app
```

Once created, use the returned sandbox ID to set up and test your project:

```
deno sandbox exec sbx_ord_abc123def456 --cwd /app npm install
deno sandbox exec sbx_ord_abc123def456 --cwd /app npm test
```

As you make changes locally, you can update the sandbox, and retrieve any generated files when done:

```
deno sandbox copy ./src/ sbx_ord_abc123def456:/app/src/
deno sandbox copy sbx_ord_abc123def456:/app/build/ ./dist/
deno sandbox kill sbx_ord_abc123def456
```

Data processing

For data processing workflows where you need to retrieve results, use a combination of remote execution and SSH access:

```
SANDBOX_ID=$(deno sandbox create --timeout 20m --copy ./data)
deno sandbox exec $SANDBOX_ID --cwd /app "deno run -A main.ts"
```

You can also stream data directly into sandbox processes using pipes, which is particularly useful for large datasets or real-time processing:

```
SANDBOX_ID=$(deno sandbox create --timeout 20m --copy ./processing-scripts)
curl -s https://api.example.com/data.json | deno sandbox exec $SANDBOX_ID --cwd /app jq '.items[] |
select(.active)'
```

Or combine local and remote processing in a pipeline:

```
grep "ERROR" /var/log/app.log | deno sandbox exec $SANDBOX_ID --cwd /app "deno run -A main.ts" | sort | uniq -c
```

To retrieve results, copy the generated files back to your local machine, then clean up:

```
deno sandbox copy $SANDBOX_ID:/app/results/*.csv ./output/
deno sandbox kill $SANDBOX_ID
```

Expose HTTP

You can run dev servers, preview apps, webhook receivers, or framework CLIs on any port and publish them instantly to a secure, random HTTPS URL.

Pass a port when creating the sandbox:

JavaScript

```
import { Sandbox } from "@deno/sandbox";

await using sandbox = await Sandbox.create({ port: 8000 });
console.log(sandbox.id);

await sandbox.fs.writeTextFile(
  "main.ts",
  "export default { fetch: () => new Response('hello from a sandbox!') }",
);

const p = await sandbox.sh`deno serve --watch main.ts`.spawn();

console.log("deno now listening on", sandbox.url);

await p.output();
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create(port=8000) as sandbox:
  print(sandbox.id)

  sandbox.fs.write_text_file(
    "main.ts",
    "export default { fetch: () => new Response('hello from a sandbox!') }"
  )

  p = sandbox.spawn("deno", args=["serve", "--watch", "main.ts"])

  print(f"deno now listening on {sandbox.url}")

  p.wait()
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(port=8000) as sandbox:
  print(sandbox.id)

  await sandbox.fs.write_text_file(
    "main.ts",
    "export default { fetch: () => new Response('hello from a sandbox!') }"
  )

  p = await sandbox.spawn("deno", args=["serve", "--watch", "main.ts"])

  print(f"deno now listening on {sandbox.url}")

  await p.wait()
```

This can then be run by setting your Deploy token and executing:

```
deno run -A --watch main.ts
```

Setting the `--watch` flag allows the sandbox to restart automatically when code changes are detected, for a low-fi hot-reload experience.

The URL stays live for the sandbox lifetime, making it perfect for short-lived QA links or agent generated previews.

When to expose HTTP

Expose HTTP whenever you need to share the sandbox with teammates, bots, or external services:

- Preview links for AI-generated web apps or instant demos
- Webhook receivers that must be reachable from Stripe, GitHub, etc.
- Framework dev servers (`next dev`, `astro dev`, `deno task dev`) that should be inspected from a browser
- Temporary APIs, health checks, or observability probes

Because sandboxes are ephemeral, you do not need to manage DNS or certificates. Each call to `exposeHttp()` returns a unique hostname under `*.sandbox.deno.net` with TLS automatically configured.

All requests to a sandbox's URL will send HTTP traffic to the sandbox.

exposeHttp() usage

Sandboxes also support exposing HTTP on-demand:

JavaScript

```
const previewUrl = await sandbox.exposeHttp({ port: 8000 });
console.log(`Preview ready at ${previewUrl}`);
```

Python

```
preview_url = sandbox.expose_http(port=8000)
print(f"Preview ready at {preview_url}")
```

Python (Async)

```
preview_url = await sandbox.expose_http(port=8000)
print(f"Preview ready at {preview_url}")
```

This is useful when you want to start a sandbox without HTTP exposure, then expose it later (for example, after some initialization or build steps).

> **NOTE**
> When you call this API, the target HTTP service will be PUBLICLY EXPOSED WITHOUT AUTHENTICATION. Anyone with knowledge of the public domain will be able to send requests to the exposed service.

Observing traffic

Requests routed through the exposed URL show up alongside your Deploy logs and traces. Use the dashboard to:

- Filter logs by sandbox ID or time range

- Inspect request traces to track latency between the edge and your VM
- Cancel or restart the sandbox if a preview misbehaves

Security and networking

- Exposed URLs are long, random subdomains that are hard to guess.
- TLS termination happens at the Deploy edge; traffic is encrypted end-to-end.

Cleanup and limits

- An exposed URL stops accepting traffic when the sandbox lifetime ends. You can call `sandbox.kill()` to terminate the sandbox (and URL) ahead of schedule if needed.
- For persistent services, graduate the code into a Deploy app rather than relying on a long-running sandbox.

Full example with a framework

JavaScript

```
import { Sandbox } from "@deno/sandbox";

await using sandbox = await Sandbox.create();

// Install dependencies
await sandbox.fs.writeTextFile(
  "package.json",
  JSON.stringify(
    {
      private: true,
      scripts: { dev: "next dev" },
      dependencies: {
        next: "^15.0.0",
        react: "^19.0.0",
        "react-dom": "^19.0.0",
      },
    },
    null,
    2,
  ),
);
await sandbox.fs.mkdir("pages", { recursive: true });
await sandbox.fs.writeTextFile(
  "pages/index.js",
  `export default function Home() {
  return (
    <main style={{ fontFamily: "system-ui", padding: "2rem" }}>
      <h1>Next.js sandbox</h1>
      <p>Edit pages/index.js to get started.</p>
    </main>
  );
}
`,
);
await sandbox.sh`npm install`;

// Start the dev server
const server = await sandbox.spawn("npm", {
```

```
  args: ["run", "dev"],
  stdout: "inherit",
  stderr: "inherit",
});

// Publish it
const previewUrl = await sandbox.exposeHttp({ port: 3000 });
console.log(`Preview ready at ${previewUrl}`);

await server.status; // keep running until the process exits
```

Python

```
import json
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create() as sandbox:
  # Install dependencies
  sandbox.fs.write_text_file(
    "package.json",
    json.dumps({
      "private": True,
      "scripts": {"dev": "next dev"},
      "dependencies": {
        "next": "^15.0.0",
        "react": "^19.0.0",
        "react-dom": "^19.0.0",
      },
    }, indent=2)
  )
  sandbox.fs.mkdir("pages", recursive=True)
  sandbox.fs.write_text_file(
    "pages/index.js",
    """export default function Home() {
  return (
    <main style={{ fontFamily: "system-ui", padding: "2rem" }}>
      <h1>Next.js sandbox</h1>
      <p>Edit pages/index.js to get started.</p>
    </main>
  );
}
"""
  )
  sandbox.spawn("npm", args=["install"]).wait()

  # Start the dev server
  server = sandbox.spawn("npm", args=["run", "dev"], stdout="inherit", stderr="inherit")

  # Publish it
  preview_url = sandbox.expose_http(port=3000)
  print(f"Preview ready at {preview_url}")

  server.wait()  # keep running until the process exits
```

Python (Async)

```
import json
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create() as sandbox:
  # Install dependencies
```

```
  await sandbox.fs.write_text_file(
    "package.json",
    json.dumps({
      "private": True,
      "scripts": {"dev": "next dev"},
      "dependencies": {
        "next": "^15.0.0",
        "react": "^19.0.0",
        "react-dom": "^19.0.0",
      },
    }, indent=2)
  )
  await sandbox.fs.mkdir("pages", recursive=True)
  await sandbox.fs.write_text_file(
    "pages/index.js",
    """export default function Home() {
  return (
    <main style={{ fontFamily: "system-ui", padding: "2rem" }}>
      <h1>Next.js sandbox</h1>
      <p>Edit pages/index.js to get started.</p>
    </main>
  );
}
"""
  )
  proc = await sandbox.spawn("npm", args=["install"])
  await proc.wait()

  # Start the dev server
  server = await sandbox.spawn("npm", args=["run", "dev"], stdout="inherit", stderr="inherit")

  # Publish it
  preview_url = await sandbox.expose_http(port=3000)
  print(f"Preview ready at {preview_url}")

  await server.wait()  # keep running until the process exits
```

Using Deno Sandbox in this way allows you to spin up full-featured framework development servers with minimal code, useful for agents or developers who need to spin up high-fidelity previews, share them for feedback, and tear everything down with a single `Ctrl+C`.

SSH

Deno Sandbox can hand out SSH credentials so you can inspect the filesystem, tail logs, run editors, or forward ports. SSH access is available both in your terminal as a command and in the Deno Deploy Sandbox UI.

JavaScript

```
import { Sandbox } from "@deno/sandbox";

await using sandbox = await Sandbox.create();

const { hostname, username } = await sandbox.exposeSsh();
console.log(`ssh ${username}@${hostname}`);

// keep process alive or interact via SSH until done...
await new Promise((resolve) => setTimeout(resolve, 10 * 60 * 1000));
```

Python

```
import time
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create() as sandbox:
  ssh = sandbox.expose_ssh()
  print(f"ssh {ssh['username']}@{ssh['hostname']}")

  # keep process alive or interact via SSH until done...
  time.sleep(10 * 60)
```

Python (Async)

```
import asyncio
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create() as sandbox:
  ssh = await sandbox.expose_ssh()
  print(f"ssh {ssh['username']}@{ssh['hostname']}")

  # keep process alive or interact via SSH until done...
  await asyncio.sleep(10 * 60)
```

The sandbox remains reachable until the configured timeout expires. Once your script releases its references (for example, the `await using` block ends) the sandbox shuts down and the SSH endpoint disappears; you can also call `sandbox.kill()` if you need to tear it down immediately.

Connecting from your machine

exposeSsh method

1. Request credentials via `sandbox.exposeSsh()`.
2. Connect using the provided username and hostname:

```
ssh ${username}@${hostname}
```

1. Use regular terminal workflows: copy files, run top, tail logs, or attach to running processes.

In the terminal

You can SSH into your sandbox from the terminal using the `--ssh` flag when running your script:

```
deno sandbox create -ssh
```

In the Deno Deploy console

After creating a sandbox, you can SSH into it in the Deno Deploy web app.

1. Log in to console.deno.com and navigate to the **Sandboxes** section.
2. Either create a new sandbox or select an existing one from the list.
3. Click **Start SSH terminal** to open an interactive terminal session in your browser.

When to use SSH access

- Debugging agent-generated code that only fails in the sandbox
- Editing files with a full-screen terminal editor or remote VS Code
- Streaming logs in real time without instrumenting application code
- Running profiling or inspection tools that are easier to use manually

Because each sandbox is already isolated, opening SSH does not compromise other projects or organizations.

Security considerations

- Credentials are single-use and bound to the sandbox lifetime.
- You control how long the sandbox runs; destroy it to revoke access instantly.

Keeping the sandbox alive

The SSH tunnel closes if the sandbox shuts down. Keep it running by:

- Setting `timeout: "session"` (default) and keeping your managing script active
- Passing `timeout: "5m"` (or another duration) when creating the sandbox so it persists after the script exits, then reconnecting later with `Sandbox.connect({ id })`

Cleanup is automatic when your code stops referencing the sandbox, but you can run `sandbox.kill()` (or simply `exit` inside the SSH session) if you want to end it on demand.

Example workflow

JavaScript

```
import { Sandbox } from "@deno/sandbox";

await using sandbox = await Sandbox.create({ timeout: "10m" });

// Prepare the app
await sandbox.fs.upload("./app", ".");
await sandbox.sh`deno task dev`
```

```
  .noThrow(); // start server; leave running for inspection

// Get SSH details
const ssh = await sandbox.exposeSsh();
console.log(`Connect with: ssh ${ssh.username}@${ssh.hostname}`);

// Block until you're done debugging manually
await new Promise((resolve) => setTimeout(resolve, 10 * 60 * 1000));
```

Python

```
import time
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create(timeout="10m") as sandbox:
  # Prepare the app
  sandbox.fs.upload("./app", ".")
  proc = sandbox.spawn("deno", args=["task", "dev"])
  # start server; leave running for inspection

  # Get SSH details
  ssh = sandbox.expose_ssh()
  print(f"Connect with: ssh {ssh['username']}@{ssh['hostname']}")

  # Block until you're done debugging manually
  time.sleep(10 * 60)
```

Python (Async)

```
import asyncio
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(timeout="10m") as sandbox:
  # Prepare the app
  await sandbox.fs.upload("./app", ".")
  proc = await sandbox.spawn("deno", args=["task", "dev"])
  # start server; leave running for inspection

  # Get SSH details
  ssh = await sandbox.expose_ssh()
  print(f"Connect with: ssh {ssh['username']}@{ssh['hostname']}")

  # Block until you're done debugging manually
  await asyncio.sleep(10 * 60)
```

Use this pattern to investigate flaky builds, run interactive REPLs, or pair with teammates without promoting the code to a full Deploy app.

Programmatic management of Deno Deploy Apps

Beyond provisioning microVMs, the SDK provides APIs for creating and managing Deploy apps inside your organization. Automating these workflows can help when you need to:

- spin up isolated apps for previews or QA
- keep multiple environments in sync
- integrate Deploy provisioning directly into your CI/CD
- clean up stale or unused apps on a schedule

The SDK wraps the Deno Deploy REST API.

Getting started

Authentication

You will need a Deno Deploy API token with appropriate permissions to manage apps. You can find your token in the Deno Deploy console under **Sandboxes** > **Integrate into your app**.

Click the **+ Create Token** button to generate a new token if you don't have one yet.

Pass the `DENO_DEPLOY_TOKEN` environment variable, scoped to your organization, when instantiating the client or creating a sandbox to manage apps. The `Client` class exposes create, list, retrieve, update, and delete methods for every app that belongs to the organization the token is scoped to.

Initialize the client

JavaScript

```
import { Client } from "@deno/sandbox";

const client = new Client();
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()
```

The SDK uses the same `DENO_DEPLOY_TOKEN` environment variable for authentication. Provide tokens scoped to the organization you want to manage.

Create an app

JavaScript

```
const app = await client.apps.create({
  slug: "my-app-from-sdk",
});
```

```
console.log(app);
// {
//   id: "4416a358-4a5f-45b2-99b5-3ebcb4b63b5f",
//   slug: "my-app-from-sdk",
//   updated_at: "2025-11-25T05:29:08.777Z",
//   created_at: "2025-11-25T05:29:08.777Z"
// }
```

Python

```
app = sdk.apps.create(slug="my-app-from-sdk")

print(app)
# {
#   "id": "4416a358-4a5f-45b2-99b5-3ebcb4b63b5f",
#   "slug": "my-app-from-sdk",
#   "updated_at": "2025-11-25T05:29:08.777Z",
#   "created_at": "2025-11-25T05:29:08.777Z"
# }
```

Python (Async)

```
app = await sdk.apps.create(slug="my-app-from-sdk")

print(app)
# {
#   "id": "4416a358-4a5f-45b2-99b5-3ebcb4b63b5f",
#   "slug": "my-app-from-sdk",
#   "updated_at": "2025-11-25T05:29:08.777Z",
#   "created_at": "2025-11-25T05:29:08.777Z"
# }
```

The `slug` is required and must be unique inside the organization. You will also be able to provide optional metadata such as `name` and `description` as the API evolves.

List apps

JavaScript

```
const list = await client.apps.list();
console.log(list.items); // first page (30 newest apps)

for await (const app of list) {
  console.log(app.slug); // paginated iterator
}
```

Python

```
page = sdk.apps.list()
print(page.items)  # first page (30 newest apps)

for item in page.items:
  print(item["slug"])
```

Python (Async)

```
page = await sdk.apps.list()
print(page.items)  # first page (30 newest apps)

async for item in page:
  print(item["slug"])  # paginated iterator
```

Use iteration to walk every app in the organization without managing cursors yourself.

Retrieve an app

JavaScript

```
const appBySlug = await client.apps.get("my-app-from-sdk");
const appById = await client.apps.get("bec265c1-ed8e-4a7e-ad24-e2465b93be88");
```

Python

```
app_by_slug = sdk.apps.get("my-app-from-sdk")
app_by_id = sdk.apps.get("bec265c1-ed8e-4a7e-ad24-e2465b93be88")
```

Python (Async)

```
app_by_slug = await sdk.apps.get("my-app-from-sdk")
app_by_id = await sdk.apps.get("bec265c1-ed8e-4a7e-ad24-e2465b93be88")
```

Fetching supports either the slug or the UUID, making it easy to use whichever identifier you have on hand.

Update app metadata

JavaScript

```
const updated = await client.apps.update(
  "bec265c1-ed8e-4a7e-ad24-e2465b93be88",
  { slug: "my-cool-app" },
);
console.log(updated.slug); // "my-cool-app"
```

Python

```
updated = sdk.apps.update(
  "bec265c1-ed8e-4a7e-ad24-e2465b93be88",
  slug="my-cool-app"
)
print(updated["slug"])  # "my-cool-app"
```

Python (Async)

```
updated = await sdk.apps.update(
  "bec265c1-ed8e-4a7e-ad24-e2465b93be88",
  slug="my-cool-app"
)
print(updated["slug"])  # "my-cool-app"
```

This is handy when a team renames a service or you want to enforce consistent slug patterns across organizations.

Delete an app

JavaScript

```
await client.apps.delete("legacy-chaotic-app");
await client.apps.delete("bec265c1-ed8e-4a7e-ad24-e2465b93be88");
```

Python

```
sdk.apps.delete("legacy-chaotic-app")
sdk.apps.delete("bec265c1-ed8e-4a7e-ad24-e2465b93be88")
```

Python (Async)

```
await sdk.apps.delete("legacy-chaotic-app")
await sdk.apps.delete("bec265c1-ed8e-4a7e-ad24-e2465b93be88")
```

Deleting accepts either identifier. Once removed, associated builds and routes are cleaned up automatically.

Publish to a Deploy app from a sandbox

The `sandbox.deno.deploy()` method can be used to publish resources from a sandbox to an existing Deno Deploy app. This allows you to use a sandbox as a deployment pipeline for an application hosted on Deno Deploy.

JavaScript

```
await using sandbox = await Sandbox.create();

// ... build your application ...

const app = await sandbox.deno.deploy("my-app", {
  options: {
    path: "build-output", // optional: path to the directory containing the application to deploy
    production: true, // optional: deploy to production
    build: {
      entrypoint: "server.ts", // optional: entrypoint to deploy
    },
  },
});

console.log(`Deployed to ${app.slug}`);
```

Python

```
sdk = DenoDeploy()

with sdk.sandbox.create() as sandbox:
  # ... build your application ...

  build = sandbox.deno.deploy(
    "my-app",
    path="build-output",  # optional: path to the directory containing the application to deploy
    production=True,  # optional: deploy to production
    entrypoint="server.ts",  # optional: entrypoint to deploy
  )

  print(f"Deployed revision ID: {build.id}")
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandbox.create() as sandbox:
  # ... build your application ...

  build = await sandbox.deno.deploy(
    "my-app",
    path="build-output",  # optional: path to the directory containing the application to deploy
    production=True,  # optional: deploy to production
    entrypoint="server.ts",  # optional: entrypoint to deploy
  )

  print(f"Deployed revision ID: {build.id}")
```

Volumes & Snapshots

Deno Sandbox provides two storage primitives:

- **Volumes** – read-write block storage. Use for caches, databases, and artifacts that persist across sessions.
- **Snapshots** – read-only images created from volumes. Use for pre-installing software so sandboxes boot instantly with everything ready. You can also create new volumes from snapshots.

Volumes

Persistent volumes let you attach regional block storage to a sandbox so data survives process restarts and new connections. They are ideal for package caches, build artifacts, SQLite databases, or any workflow that needs a small amount of durable storage without promoting code to a full Deno Deploy app.

Provision storage

JavaScript

```
import { Client } from "@deno/sandbox";

const client = new Client();

const volume = await client.volumes.create({
  slug: "training-cache",
  region: "ord", // only "ord" (Chicago) is supported today
  capacity: "2GB", // accepts bytes or "1GB"/"512MB" style strings
});

console.log(volume);
// {
//   id: "8a0f...",
//   slug: "training-cache",
//   region: "ord",
//   capacity: 2147483648,
//   used: 0
// }
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

volume = sdk.volumes.create(
  slug="training-cache",
  region="ord",  # only "ord" (Chicago) is supported today
  capacity="2GB"  # accepts bytes or "1GB"/"512MB" style strings
)

print(volume)
# {
#   "id": "8a0f...",
#   "slug": "training-cache",
#   "region": "ord",
#   "capacity": 2147483648,
#   "used": 0
# }
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

volume = await sdk.volumes.create(
  slug="training-cache",
  region="ord",  # only "ord" (Chicago) is supported today
  capacity="2GB"  # accepts bytes or "1GB"/"512MB" style strings
)

print(volume)
# {
#   "id": "8a0f...",
#   "slug": "training-cache",
#   "region": "ord",
#   "capacity": 2147483648,
#   "used": 0
# }
```

Field	Required	Details
slug		Unique per organization. Slugs become part of the mount metadata, so pick something descriptive.
region		Currently only "ord" (Chicago) is supported. Only sandboxes in the same region can mount the volume.
capacity		Between 300 MB and 20 GB. Pass a number of bytes or a string with GB/MB/KB (decimal) or GiB/MiB/KiB (binary) units.

Inspect and search volumes

The volumes API returns paginated results and can fetch a single record by slug or UUID.

JavaScript

```
const page = await client.volumes.list({ search: "training" });
for (const vol of page.items) {
  console.log(vol.slug, vol.estimatedFlattenedSize, vol.capacity);
}

const vol = await client.volumes.get("training-cache");
```

Python

```
page = sdk.volumes.list(search="training")
for vol in page.items:
  print(f"{vol['slug']} {vol['estimatedFlattenedSize']} {vol['capacity']}")

vol = sdk.volumes.get("training-cache")
```

Python (Async)

```
page = await sdk.volumes.list(search="training")
async for vol in page:
  print(f"{vol['slug']} {vol['estimatedFlattenedSize']} {vol['capacity']}")

vol = await sdk.volumes.get("training-cache")
```

The used field reports the most recent estimate the control plane received from the underlying cluster. It can lag a few minutes behind reality, so always size volumes with headroom.

Mount volumes inside a sandbox

Pass a `volumes` mapping when creating a sandbox. Keys are mount paths and values are either the volume slug or ID. The sandbox and volume **must be in the same region**.

> **NOTE**
> `Sandbox.create()` and `client.sandboxes.create()` are equivalent—use whichever fits your code style.

JavaScript

```
import { Client, Sandbox } from "@deno/sandbox";

const client = new Client();
const volume = await client.volumes.create({
  slug: "dataset",
  region: "ord",
  capacity: "1GB",
});

// First run writes a file to the volume
{
  await using sandbox = await Sandbox.create({
    region: "ord",
    volumes: {
      "/data/dataset": volume.slug,
    },
    labels: { job: "prepare" },
  });

  await sandbox.fs.writeTextFile("/data/dataset/hello.txt", "Persist me!\n");
}

// A new sandbox—possibly started hours later—can read the same file
{
  await using sandbox = await Sandbox.create({
    region: "ord",
    volumes: {
      "/data/dataset": volume.id, // IDs work too
    },
  });

  const contents = await sandbox.fs.readTextFile("/data/dataset/hello.txt");
  console.log(contents); // "Persist me!"
}
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

volume = sdk.volumes.create(
  slug="dataset",
  region="ord",
  capacity="1GB"
)

# First run writes a file to the volume
with sdk.sandbox.create(
```

```
  region="ord",
  volumes={
    "/data/dataset": volume["slug"],
  },
  labels={"job": "prepare"}
) as sandbox:
  sandbox.fs.write_text_file("/data/dataset/hello.txt", "Persist me!\n")

# A new sandbox—possibly started hours later—can read the same file
with sdk.sandbox.create(
  region="ord",
  volumes={
    "/data/dataset": volume["id"],  # IDs work too
  }
) as sandbox:
  contents = sandbox.fs.read_text_file("/data/dataset/hello.txt")
  print(contents)  # "Persist me!"
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

volume = await sdk.volumes.create(
  slug="dataset",
  region="ord",
  capacity="1GB"
)

# First run writes a file to the volume
async with sdk.sandbox.create(
  region="ord",
  volumes={
    "/data/dataset": volume["slug"],
  },
  labels={"job": "prepare"}
) as sandbox:
  await sandbox.fs.write_text_file("/data/dataset/hello.txt", "Persist me!\n")

# A new sandbox—possibly started hours later—can read the same file
async with sdk.sandbox.create(
  region="ord",
  volumes={
    "/data/dataset": volume["id"],  # IDs work too
  }
) as sandbox:
  contents = await sandbox.fs.read_text_file("/data/dataset/hello.txt")
  print(contents)  # "Persist me!"
```

Mounts behave like regular directories. You can create subfolders, write binary files, or execute programs directly from the volume.

Delete volumes safely

JavaScript

```
await client.volumes.delete("training-cache");
```

Python

```
sdk.volumes.delete("training-cache")
```

Python (Async)

```
await sdk.volumes.delete("training-cache")
```

Deletion is a two-step process:

1. The API marks the volume as deleted immediately, which detaches it from new sandbox requests and frees the slug for future reuse.
2. A background job waits 24 hours before removing the underlying block storage from the cluster. This grace period allows you to contact support if a volume was removed accidentally.

During the grace period you cannot mount or read the volume.

Snapshots

Snapshots are read-only images created from volumes. When a sandbox boots with a snapshot as its root, the entire filesystem is replaced with the snapshot contents. Run `apt-get install` or `npm install` once, snapshot the result, and every future sandbox starts instantly with everything already installed.

Creating a snapshot

The workflow is:

1. Create a **bootable volume** from a base image
2. Boot a sandbox with that volume as `root` (writable)
3. Install software
4. Snapshot the volume

Sandboxes booted from the snapshot start instantly with everything installed.

A volume is **bootable** when created with the `from` option. Currently `builtin:debian-13` is the only available base image.

JavaScript

```
import { Client } from "@deno/sandbox";

const client = new Client();

// 1. Create a bootable volume
const volume = await client.volumes.create({
  region: "ord",
  slug: "my-toolchain",
  capacity: "10GiB",
  from: "builtin:debian-13",
});

// 2. Boot a sandbox with the volume as root (writable)
await using sandbox = await client.sandboxes.create({
  region: "ord",
  root: volume.slug,
});

// 3. Install software
await sandbox.sh`apt-get update && apt-get install -y nodejs npm`;
await sandbox.sh`npm install -g typescript`;

// 4. Snapshot the volume
const snapshot = await client.volumes.snapshot(volume.id, {
  slug: "my-toolchain-snapshot",
```

```
});
```

CLI

```
# Create snapshot from a volume
deno sandbox snapshots create my-toolchain my-toolchain-snapshot
```

Booting from a snapshot

Once you have a snapshot, use it as the `root` when creating new sandboxes. The sandbox must be created in the same region as the snapshot:

JavaScript

```
await using sandbox = await client.sandboxes.create({
  region: "ord",
  root: "my-toolchain-snapshot", // snapshot slug or ID
});

// TypeScript and Node.js are already installed
await sandbox.sh`tsc --version`;
await sandbox.sh`node --version`;
```

The sandbox boots with the snapshot's filesystem as its root. Any writes during the session are ephemeral—they don't modify the snapshot.

Listing snapshots

JavaScript

```
const page = await client.snapshots.list();
for (const snap of page.items) {
  console.log(snap.slug, snap.region, snap.bootable);
}
```

CLI

```
$ deno sandbox snapshots list
ID                            SLUG                   REGION   ALLOCATED    BOOTABLE
snp_ord_spmbe47dysccpy277ma6  my-toolchain-snapshot   ord      217.85 MiB   TRUE
```

Creating a volume from a snapshot

Create a new writable volume from a snapshot:

JavaScript

```
const volume = await client.volumes.create({
  region: "ord",
  slug: "my-toolchain-fork",
  capacity: "10GiB",
  from: "my-toolchain-snapshot",
});
```

The new volume contains the snapshot's contents and is fully writable. Use this to modify a snapshot's contents, then snapshot again.

Deleting snapshots

JavaScript

```
await client.snapshots.delete("my-toolchain-snapshot");
```

CLI

```
deno sandbox snapshots delete my-toolchain-snapshot
```

Volumes vs Snapshots

Feature	Volumes	Snapshots
Access	Read-write	Read-only
Mount point	Any path, or root if bootable	Root filesystem only
Use case	Caches, databases, install software	Pre-installed software, toolchains
Concurrent use	One sandbox at a time	Many sandboxes simultaneously
Region	Must match sandbox region	Must match sandbox region

Sandbox Timeouts

Sandboxes created by Deno Sandbox are intentionally ephemeral. They boot in milliseconds, serve their purpose, and disappear—reducing the blast radius of untrusted code and removing infrastructure chores. Still, you can control exactly how long a sandbox stays alive and even reconnect later when debugging is required.

Default timeout: "session"

JavaScript

```
await using sandbox = await Sandbox.create();
```

Python

```
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create() as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create() as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

With no options set, the sandbox lives for the duration of your script. Once the sandbox instance is closed, the microVM shuts down and frees all resources. This keeps costs predictable and prevents orphaned infrastructure.

Duration-based timeouts

Provide a duration string to keep a sandbox alive after the client disconnects:

JavaScript

```
const sandbox = await Sandbox.create({ timeout: "5m" });
const id = sandbox.id;
await sandbox.close(); // process can exit now

// later
const reconnected = await Sandbox.connect({ id });
```

Python

```
sdk = DenoDeploy()

with sdk.sandbox.create(timeout="5m") as sandbox:
  sandbox_id = sandbox.id
  sandbox.close()  # process can exit now

# later
with sdk.sandbox.connect(sandbox_id) as reconnected:
  print(f"Reconnected to {reconnected.id}")
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(timeout="5m") as sandbox:
  sandbox_id = sandbox.id
  await sandbox.close()  # process can exit now

# later
async with sdk.sandbox.connect(sandbox_id) as reconnected:
  print(f"Reconnected to {reconnected.id}")
```

Supported suffixes: `s` (seconds) and `m` (minutes). Examples: `"30s"`, `"5m"`, `"90s"`. Use this mode for manual inspection, SSH debugging, or when a bot needs to resume work mid-way.

If you require a longer timeout, it is possible to promote a duration-based sandbox to a Deno Deploy app using `sandbox.deno.deploy()`.

Forcefully ending a sandbox

- `await sandbox.kill()` immediately stops the VM and releases the lifetime if you need to tear it down before it would naturally expire.
- Killing a sandbox invalidates exposed HTTP URLs, SSH sessions, and any attached volumes, but this also happens automatically when your code drops the last reference to the sandbox or the configured duration elapses.

Extending the timeout of a sandbox

Add more time to a running sandbox without interrupting ongoing processes.

JavaScript

```
const newTimeout = await sandbox.extendTimeout("30m");
console.log(`Sandbox now expires at ${newTimeout}`);
```

CLI

```
deno sandbox extend sbx_ord_abc123def456 30m
```

All active connections and processes continue uninterrupted while the sandbox's expiration time is updated.

Related APIs

- `Sandbox.create()` – pass the `timeout` option when provisioning.
- `Sandbox.connect({ id })` – resume control of a duration-based sandbox.
- `Sandbox.kill()` – terminate early.
- Expose HTTP and SSH – note that their URLs/credentials die with the sandbox lifetime.

Security

Deno Sandbox is designed for untrusted or AI-generated workloads. Every VM is ephemeral, isolated at the hypervisor level, and governed by strict outbound policies. This lets you run arbitrary code while keeping organization data and infrastructure safe.

Secret redaction and substitution

Secrets never enter the sandbox environment variables. Instead, Deno Deploy substitutes them only when the sandbox makes outbound requests to an approved host. Configure secrets when creating a sandbox:

JavaScript

```
await using sandbox = await Sandbox.create({
  secrets: {
    OPENAI_API_KEY: {
      hosts: ["api.openai.com"],
      value: process.env.OPENAI_API_KEY,
    },
    ANTHROPIC_API_KEY: {
      hosts: ["api.anthropic.com"],
      value: process.env.ANTHROPIC_API_KEY,
    },
  },
});
```

Python

```
import os
from deno_sandbox import DenoDeploy

sdk = DenoDeploy()

with sdk.sandbox.create(
  secrets={
    "OPENAI_API_KEY": {
      "hosts": ["api.openai.com"],
      "value": os.environ.get("OPENAI_API_KEY"),
    },
    "ANTHROPIC_API_KEY": {
      "hosts": ["api.anthropic.com"],
      "value": os.environ.get("ANTHROPIC_API_KEY"),
    },
  }
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
import os
from deno_sandbox import AsyncDenoDeploy

sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(
  secrets={
    "OPENAI_API_KEY": {
      "hosts": ["api.openai.com"],
      "value": os.environ.get("OPENAI_API_KEY"),
    },
    "ANTHROPIC_API_KEY": {
```

```
      "hosts": ["api.anthropic.com"],
      "value": os.environ.get("ANTHROPIC_API_KEY"),
    },
  }
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Inside the sandbox, the environment variable holds a placeholder:

```
echo $ANTHROPIC_API_KEY
# <placeholder>
```

This confirms that user code cannot read the real secret. This blocks the most common AI attack path of prompt injection followed by secret exfiltration while allowing your automation to call third-party APIs securely.

Outbound network control

By default, Deno Sandbox has unrestricted outbound network access. Use the network allowlist option to restrict traffic to specific hosts:

JavaScript

```
await using sandbox = await Sandbox.create({
  allowNet: ["api.openai.com", "*.anthropic.com"],
});
```

Python

```
sdk = DenoDeploy()

with sdk.sandbox.create(
  allow_net=["api.openai.com", "*.anthropic.com"]
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Python (Async)

```
sdk = AsyncDenoDeploy()

async with sdk.sandbox.create(
  allow_net=["api.openai.com", "*.anthropic.com"]
) as sandbox:
  print(f"Sandbox {sandbox.id} is ready.")
```

Supported patterns include:

Pattern	Matches
`example.com`	Exact hostname, any port
`example.com:443`	Exact hostname on port 443 only
`*.example.com`	Any subdomain of example.com
`192.0.2.1`	Exact IPv4 address
`[2001:db8::1]`	Exact IPv6 address

Any outbound request to a host not in the allow list will be blocked when `allowNet` is provided. When `allowNet` is omitted, all outbound requests are allowed. Combine this with the secrets option to ensure that even if code is tricked into calling an unexpected endpoint, credentials are never sent.

Filesystem isolation and cleanup

- MicroVMs boot from a clean disk image. Any files you upload exist only for the sandbox lifetime unless you explicitly mount a volume.
- Once the last reference to a sandbox is dropped (or `sandbox.kill()` is called), the VM is destroyed and the disk wiped, preventing lingering state.
- Volumes provide shared storage, but access is explicit per sandbox and can be mounted read-only when needed.

Auditing and observability

- Every command, HTTP request, and SSH session can be traced in the Deno Deploy dashboard, giving you a paper trail for agent behavior.
- Attach metadata when creating sandboxes (e.g., `metadata: { owner: "agent" }`) so logs and traces clearly show who initiated activity.