Handle errors and degradation

- Classify the failure {#1-classify-the-failure}

Distinguish a rejected request from an accepted operation that later fails. This keeps retries from hiding invalid input or duplicating work.

The SDK exposes typed request and wait errors. A command's nonzero exit code is a third outcome: the SDK can successfully observe a command that failed.

Classify the failure {#1-classify-the-failure}

Use the error type and structured code, not the text of its message. The example groups common request failures and retains their details for the caller.

- Invalid arguments need a corrected request.
- Authentication failures need a valid credential; permission failures need appropriate account access.
- A wrong-state error needs a state check before another attempt.
- A capacity or quota refusal may require waiting or reducing usage.
- A transient availability failure may be retryable if the operation can be repeated safely.

Wait errors retain the state or resource observed before the wait stopped. Inspect that information and read the resource again when necessary. A failed create, an interrupted wait, and an expired client deadline do not have the same cleanup outcome.

A process helper can fail while obtaining a sandbox credential or reading output after the process has started. The error retains the accepted process and its cause. That process may still be running: inspect it before deciding whether to retry the command. Inspect the cause chain for a typed credential-service rate-limit error, including any request identifier and retry delay supplied by the service.

Unknown errors remain unknown in the example. They may be connection failures before any response arrived. Do not assume that a write was never accepted.

Keep credentials and secret-bearing request bodies out of diagnostics. Record resource names, error codes, and request identifiers instead. See safe retries before repeating a mutation.

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

```
if (!(error instanceof RequestError)) {
  return { kind: 'other', code: undefined, details: [] };
}
const { code, details } = error.raw;
const known = KINDS.get(code);
return { kind: known ?? 'other', code, details: details ?? [] };
```

Complete TypeScript example: failures/classify.ts

```
import { RequestError, ResourceWaitError } from '@docker/sandboxes';

type Status = RequestError['raw'];
type Detail = NonNullable<Status['details']>[number];

export type FailureKind =
  | 'invalid-request'
  | 'not-served'
  | 'wrong-state'
  | 'exhausted'
  | 'access-or-missing'
  | 'unavailable'
  | 'other';

export interface Failure {
  kind: FailureKind;
  code: string | undefined;
  details: Detail[];
}

const KINDS = new Map<string, FailureKind>([
  ['invalidArgument', 'invalid-request'],
```

```
  ['unimplemented', 'not-served'],
  ['failedPrecondition', 'wrong-state'],
  ['resourceExhausted', 'exhausted'],
  ['unauthenticated', 'access-or-missing'],
  ['permissionDenied', 'access-or-missing'],
  ['notFound', 'access-or-missing'],
  ['unavailable', 'unavailable'],
]);

export function readFailure(error: unknown): Failure {
  if (!(error instanceof RequestError)) {
    return { kind: 'other', code: undefined, details: [] };
  }
  const { code, details } = error.raw;
  const known = KINDS.get(code);
  return { kind: known ?? 'other', code, details: details ?? [] };
}

export function readSandboxWait(
  error: unknown,
): 'degraded' | 'failed' | 'other' {
  if (error instanceof ResourceWaitError) {
    if (error.kind === 'interrupted' && error.state === 'degraded')
      return 'degraded';
    if (error.kind === 'failure' && error.state === 'failed')
      return 'failed';
  }
  return 'other';
}
```

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

□□□ #2
□ □□□ □□□ 2026-10-31 18:56:17 UTC
□ □□□ □□□ 2026-10-31 18:56:18 UTC