Cloudflare Workflows

Version Wrangler 4 · verified 2026-11-02

Durable multi-step programs on Workers — step.do checkpoints, retries and rollbacks, sleeps, waiting for events, instance control, limits, and tests.

Contents

Mental model 2

Defining a Workflow 2

Steps and replay 4

Retries, timeouts, and rollbacks 5

Sleeping and waiting for events 6

Triggering and managing instances 7

Limits and pricing 9

Observability and testing 10

Further reading 11

Mental model

A Workflow is ordinary async code whose `step.do()` calls are checkpoints. When an instance sleeps, waits, retries, or the engine restarts, `run()` executes again from the top, and every step that already succeeded returns its stored result instead of running. Code between steps can therefore run many times, so side effects belong inside steps, step names must be stable, and anything a step returns must be serializable.

Defining a Workflow

Piece	Purpose
`extends WorkflowEntrypoint`	The Workflow class
`run(event, step)`	Entry point; may return a result
`event.payload`	Params passed at creation
`event.instanceId`	This run's ID
`"workflows"` in config	Binds it as `env.NAME`

A Workflow is a class in a Worker module. The `workflows` binding gives the Worker `create()`, `get()`, and friends; a Worker can instead declare the class under `exports` and call it through `ctx.exports`. `run()` may return a value, which appears in the instance status.

```
import {
  WorkflowEntrypoint,
} from "cloudflare:workers";
```

```
export class Signup extends WorkflowEntrypoint {
  async run(event, step) {
    const { email } = event.payload;
    const user = await step.do("create user",
      () => this.env.API.createUser(email));
    await step.do("send welcome",
      () => this.env.MAIL.welcome(user.id));
    return { userId: user.id };
  }
}
```

```
{
  "workflows": [
    {
      "name": "signup",
      "binding": "SIGNUP",
      "class_name": "Signup"
    }
  ]
}
```

Steps and replay

Rule	Why
Side effects inside `step.do`	Outside code reruns
Deterministic step names	The name is the cache key
Build state from step results	Memory is lost on hibernation
Make steps idempotent	A step can retry
Return ≤ 1 MiB per step	Larger results fail

The engine hibernates instances with nothing to do and replays `run()` when they wake, so variables set outside steps are gone. Treat each step as one unit of work that can be retried, and key external calls on something stable so a retry does not charge a card twice. Branch only on `event.payload` or earlier step results — never on `Date.now()` or `Math.random()` outside a step.

```
// Wrong: runs again on every replay
const startedAt = Date.now();
console.log("charging");

// Right: recorded once, replayed from storage
const charged = await step.do("charge", () =>
  pay(event.payload.orderId, {
    idempotencyKey: event.instanceId,
  }));
```

> **Gotcha:** A step named with a timestamp or random value — `` `fetch ${Date.now()}` `` — never matches its cached result, so it re-executes on every replay.

Retries, timeouts, and rollbacks

Option	Default
`retries.limit`	5
`retries.delay`	10,000 ms
`retries.backoff`	`"exponential"`
`timeout` (per attempt)	10 minutes
`NonRetryableError`	Fails the step at once

Pass a config object as the second argument to `step.do()`. `backoff` is `constant`, `linear`, or `exponential`, and `delay` can be a function of the attempt and error. Throw `NonRetryableError` for failures that retrying cannot fix. A rollback handler, passed last, runs if the instance later fails; rollbacks run in reverse step-start order.

```
const order = await step.do("reserve stock",
  { retries: { limit: 3, delay: "5 seconds",
    backoff: "exponential" },
    timeout: "2 minutes" },
  async () => {
    const r = await reserve(event.payload);
    if (!r.ok) {
```

```
      throw new NonRetryableError("sold out");
    }
    return { reservationId: r.id };
  },
  { rollback: async ({ output }) =>
    release(output.reservationId) },
);
```

> **Note:** `NonRetryableError` is imported from `cloudflare:workflows`. Uncaught at the top of `run()`, it fails the instance.

Sleeping and waiting for events

Call	Does
`step.sleep(name, "3 days")`	Pauses for a duration
`step.sleepUntil(name, date)`	Pauses until a time
`step.waitForEvent(name, opts)`	Waits for `sendEvent`
`instance.sendEvent({ type, payload })`	Delivers the event

A sleeping or waiting instance moves to the `waiting` state: it consumes no CPU and does not count toward the concurrency limit, so millions can wait at once. `waitForEvent` matches on `type` and throws when its `timeout` expires; the default is 24 hours. Sleeps can last up to 365 days and do not count toward the step limit.

```
// in run():
const approval = await step.waitForEvent(
  "manager approval",
  { type: "approved", timeout: "7 days" },
);

// in a Worker handling the webhook:
const inst = await env.SIGNUP.get(id);
await inst.sendEvent({
  type: "approved",
  payload: { by: "ada" },
});
```

> **Tip:** Prefer `waitForEvent` to a long step `timeout`. Cloudflare recommends step timeouts of 30 minutes or less; anything longer is a wait, not a step.

Triggering and managing instances

Call	Effect
`env.WF.create({ id, params })`	Starts an instance
`env.WF.createBatch(...)`	Up to 100 at once
`env.WF.get(id)`	Handle to an instance
`inst.status()`	`{ status, output, error }`
`pause` / `resume` / `restart`	Control a run
`terminate({ rollback })`	Stops it

Statuses are `queued`, `running`, `paused`, `waiting`, `waitingForPause`, `complete`, `errored`, `terminated`, and `unknown`. A custom ID must be unique among retained instances: `create()` throws on a duplicate, while `createBatch()` skips it and reports it in `errors`. For periodic runs, put cron expressions in `schedules` on the binding instead of writing a separate `scheduled` handler.

```
export default {
  async fetch(req, env) {
    const order = await req.json();
    const inst = await env.SIGNUP.create({
      id: `order-${order.id}`,
      params: order,
    });
    return Response.json({
      id: inst.id,
      ...(await inst.status()),
    });
  },
};
```

> **Gotcha:** Instance IDs are never reused while the instance is retained. Re-running "the same" order with `create()` fails; call `restart()` on the existing instance or choose a new ID.

Limits and pricing

Limit	Free	Paid
CPU per step	10 ms	30 s, up to 5 min
Steps per Workflow	1,024	10,000, up to 25,000
Concurrent running instances	100	50,000
Instance creation rate	100/s	300/s per account
State per instance	100 MB	1 GB
Completed-state retention	3 days	30 days max

Wall-clock time per step is unlimited: waiting on network I/O is free of CPU limits. Raise CPU with `limits.cpu_ms` and steps with `limits.steps` in the Wrangler config. Paid billing covers requests, CPU time, steps (500,000 included per month, then $0.80 per 100,000), and stored state (1 GB-month included, then $0.20 per GB-month).

```
{
  "limits": { "cpu_ms": 300000 },
  "workflows": [
    {
      "name": "signup",
      "binding": "SIGNUP",
      "class_name": "Signup",
      "limits": { "steps": 25000 },
      "schedules": ["0 9 * * MON-FRI"]
    }
```

```
  ]
}
```

> **Note:** On Workers Paid, Workflows created on or after 10 September 2026 keep finished instances for 7 days by default; set `retention` on `create()` to change it.

Observability and testing

Tool	Use
`wrangler workflows instances describe`	Step-by-step status
`wrangler workflows trigger`	Start an instance
`inst.subscribe()`	Stream instance events
`introspectWorkflowInstance()`	Mock steps in Vitest
`disableSleeps()`	Skip sleeps in tests

The dashboard and `instances describe` show each step's attempts, output, and errors. In tests with `@cloudflare/vitest-plugin`, an introspector from `cloudflare:test` can disable sleeps, mock step results and errors, and inject events, then wait for a final status.

```
npx wrangler workflows trigger signup \
  '{"email":"ada@example.com"}'
npx wrangler workflows instances \
  describe signup latest
npx wrangler workflows trigger signup --local
```

Further reading

- Rules of Workflows
- Workflows Workers API
- Sleeping and retrying
- Events and parameters
- Workflows limits
- Workflows test APIs