Comprehensive Testing Strategy for Next.js 15 Applications

Vitest + Playwright + Codecov forms the optimal free testing stack for your application, delivering 4x faster unit tests than Jest, unlimited parallel E2E execution, and built-in visual regression—all at zero cost. For AI/ML output validation, combine metrics-based checks with the AI-as-judge pattern using GPT-4o-mini at roughly **$0.05 per 100 evaluations**. This report provides complete configuration files, code examples, and a phased implementation plan.

Unit testing: Vitest outperforms Jest for Next.js 15

The choice between Vitest and Jest for Next.js 15 App Router projects is now clear-cut. Vitest delivers **~3.8 seconds for 100 tests** compared to Jest's ~15.5 seconds— (Wisp CMS) a 4x performance advantage. More importantly, Vitest provides native ESM and TypeScript support without additional configuration, while Jest requires babel transforms and ts-jest setup.

Feature	Vitest	Jest
Native ESM	✅ Built-in	❌ Requires config
TypeScript	✅ Zero-config	❌ Needs ts-jest
Watch mode	✅ Vite HMR (instant)	⚠️ Slower refresh
API compatibility	Jest-compatible	N/A

Complete Vitest configuration for your stack:

typescript

```
// vitest.config.mts
import { defineConfig } from 'vitest/config'
import react from '@vitejs/plugin-react'
import tsconfigPaths from 'vite-tsconfig-paths'

export default defineConfig({
  plugins: [tsconfigPaths(), react()],
  test: {
    environment: 'jsdom',
    globals: true,
    setupFiles: ['./src/tests/setup.ts'],
    include: ['**/*.{test,spec}.{ts,tsx}'],
    coverage: {
      provider: 'v8',
      reporter: ['text', 'json', 'html', 'lcov'],
      thresholds: {
        lines: 80,
        functions: 80,
        branches: 75,
        statements: 80,
      },
    },
  },
})
```

Testing Zustand stores requires the official mock pattern to reset state between tests:

typescript

```typescript
// __mocks__/zustand.ts
import { act } from '@testing-library/react'
import type * as ZustandExportedTypes from 'zustand'

export * from 'zustand'
const { create: actualCreate } = await vi.importActual<typeof ZustandExportedTypes>('zustand')

export const storeResetFns = new Set<() => void>()

export const create = (<T>(stateCreator: ZustandExportedTypes.StateCreator<T>) => {
  const store = actualCreate(stateCreator)
  const initialState = store.getInitialState()
  storeResetFns.add(() => store.setState(initialState, true))
  return store
}) as typeof ZustandExportedTypes.create

afterEach(() => {
  act(() => storeResetFns.forEach((resetFn) => resetFn()))
})
```

Testing TanStack Query hooks requires a wrapper that disables retries and garbage collection:

```typescript
// src/tests/query-test-utils.tsx
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'

export function createWrapper() {
  const queryClient = new QueryClient({
    defaultOptions: {
      queries: { retry: false, gcTime: Infinity },
    },
  })
  return ({ children }) => (
    <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>
  )
}
```

Testing React Hook Form with Zod starts with schema unit tests:

```typescript
```

```
// __tests__/resume-schema.test.ts
import { resumeSchema } from '../schemas/resume-schema'

test('validates complete resume data', () => {
  const validResume = {
    name: 'Jane Doe',
    email: 'jane@example.com',
    experience: [{ company: 'Acme', role: 'Engineer', years: 3 }],
  }
  expect(resumeSchema.safeParse(validResume).success).toBe(true)
})

test('rejects missing required fields', () => {
  const result = resumeSchema.safeParse({ name: 'Jane' })
  expect(result.success).toBe(false)
  expect(result.error.issues[0].path).toContain('email')
})
```

Mocking Vercel AI SDK uses the official `MockLanguageModelV2` from `ai/test`: `ai-sdk`

typescript

```
import { generateText, simulateReadableStream } from 'ai'
import { MockLanguageModelV2 } from 'ai/test'

test('generates resume bullet point', async () => {
  const result = await generateText({
    model: new MockLanguageModelV2({
      doGenerate: async () => ({
        finishReason: 'stop',
        usage: { inputTokens: 50, outputTokens: 100, totalTokens: 150 },
        content: [{ type: 'text', text: 'Led team of 5 engineers to deliver $2M project' }],
        warnings: [],
      }),
    }),
    prompt: 'Improve this bullet point: managed team',
  })
  expect(result.text).toContain('Led')
  expect(result.text).toMatch(/\$\d+/)
})
```

Integration testing with Drizzle ORM and API routes

Testing Next.js 15 route handlers requires `next-test-api-route-handler`, which supports the App Router's async params pattern:

typescript

```typescript
// app/api/resumes/[id]/route.test.ts
import { testApiHandler } from 'next-test-api-route-handler'
import * as appHandler from './route'

test('GET returns resume by ID', async () => {
  await testApiHandler({
    appHandler,
    params: { id: 'resume-123' },
    test: async ({ fetch }) => {
      const response = await fetch({ method: 'GET' })
      expect(response.status).toBe(200)
      const data = await response.json()
      expect(data).toHaveProperty('content')
    },
  })
})
```

Testing Drizzle ORM with in-memory SQLite provides fast, isolated database tests:

typescript

```
// tests/setup/db.ts
import { drizzle } from 'drizzle-orm/better-sqlite3'
import Database from 'better-sqlite3'
import { migrate } from 'drizzle-orm/better-sqlite3/migrator'
import * as schema from '@/db/schema'

export function createTestDb() {
  const sqlite = new Database(':memory:')
  const db = drizzle(sqlite, { schema })
  migrate(db, { migrationsFolder: './drizzle' })
  return { db, sqlite }
}

// tests/db/resumes.test.ts
describe('Resume Database Operations', () => {
  let db, sqlite

  beforeEach(() => {
    const testDb = createTestDb()
    db = testDb.db
    sqlite = testDb.sqlite
  })

  afterEach(() => sqlite.close())

  test('inserts and retrieves resume', async () => {
    const [resume] = await db.insert(resumes)
      .values({ userId: 'user-1', content: 'Test resume' })
      .returning()

    const found = await db.select().from(resumes).where(eq(resumes.id, resume.id)).get()
    expect(found.content).toBe('Test resume')
  })
})
```

Mocking Apify Actor integrations prevents API calls during tests while validating your scraper logic:

typescript

```
// tests/mocks/apify.ts
export const mockLinkedInScraperResponse = {
  items: [{
    url: 'https://linkedin.com/company/acme',
    name: 'Acme Corp',
    industry: 'Technology',
    employeeCount: '500-1000',
    description: 'Leading innovator in...',
  }],
}

// tests/services/company-research.test.ts
vi.mock('apify-client', () => ({
  ApifyClient: vi.fn(() => ({
    actor: vi.fn().mockReturnValue({
      call: vi.fn().mockResolvedValue({ defaultDatasetId: 'ds-123' }),
    }),
    dataset: vi.fn().mockReturnValue({
      listItems: vi.fn().mockResolvedValue(mockLinkedInScraperResponse),
    }),
  })),
}))

test('aggregates company data from multiple sources', async () => {
  const service = new CompanyResearchService('token')
  const result = await service.research('Acme Corp')
  expect(result.sources).toContain('linkedin')
  expect(result.employeeCount).toBe('500-1000')
})
```

Testing Puppeteer PDF generation validates both content and visual output:

```
typescript
```

```
// tests/pdf/resume-generation.test.ts
import puppeteer from 'puppeteer'
import pdfParse from 'pdf-parse'

test('generated PDF contains all resume sections', async () => {
  const browser = await puppeteer.launch({ headless: true })
  const page = await browser.newPage()

  await page.setContent(resumeHTML)
  const pdfBuffer = await page.pdf({ format: 'A4', printBackground: true })

  const { text } = await pdfParse(pdfBuffer)
  expect(text).toContain('Work Experience')
  expect(text).toContain('Education')
  expect(text).toContain('Skills')

  await browser.close()
})
```

MSW 2.x setup mocks external APIs consistently across unit and integration tests:

typescript

```
// src/mocks/handlers.ts
import { http, HttpResponse } from 'msw'

export const handlers = [
  http.post('/api/ai/generate', async ({ request }) => {
    const { prompt } = await request.json()
    return HttpResponse.json({
      text: 'Spearheaded initiative resulting in 40% efficiency improvement',
    })
  }),

  http.get('https://api.apify.com/v2/datasets/:id/items', () => {
    return HttpResponse.json(mockLinkedInScraperResponse.items)
  }),
]

// vitest.setup.ts
import { server } from './src/mocks/node'
beforeAll(() => server.listen({ onUnhandledRequest: 'error' }))
afterEach(() => server.resetHandlers())
afterAll(() => server.close())
```

E2E testing: Playwright delivers free unlimited parallelization

Playwright wins decisively over Cypress for Next.js 15 projects. Beyond free unlimited parallel execution (Cypress Cloud starts at **$67-75/month**), SaaSworthy Playwright provides WebKit/Safari testing critical for iOS users, native multi-tab support, and experimental server-side fetch mocking for App Router.

Complete Playwright configuration:

typescript

```typescript
// playwright.config.ts
import { defineConfig, devices } from '@playwright/test'

export default defineConfig({
  testDir: './tests/e2e',
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 2 : 0,
  workers: process.env.CI ? '50%' : undefined,

  reporter: process.env.CI
    ? [['html'], ['junit', { outputFile: 'test-results/junit.xml' }], ['github']]
    : [['html', { open: 'never' }], ['list']],

  use: {
    baseURL: 'http://localhost:3000',
    trace: 'on-first-retry',
    screenshot: 'only-on-failure',
    video: 'on-first-retry',
  },

  expect: {
    toHaveScreenshot: { maxDiffPixels: 100, threshold: 0.2, animations: 'disabled' },
  },

  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } },
    { name: 'mobile-chrome', use: { ...devices['Pixel 5'] } },
    { name: 'mobile-safari', use: { ...devices['iPhone 13'] } },
  ],

  webServer: {
    command: process.env.CI ? 'npm run start' : 'npm run dev',
    url: 'http://localhost:3000',
    reuseExistingServer: !process.env.CI,
  },
})
```

Testing PDF file uploads:

```typescript
```

```
// tests/e2e/resume-upload.spec.ts
import { test, expect } from '@playwright/test'
import path from 'path'

test('uploads PDF resume and extracts content', async ({ page }) => {
  await page.goto('/upload')

  await page.getByLabel('Upload resume').setInputFiles(
    path.join(__dirname, 'fixtures', 'sample-resume.pdf')
  )

  await expect(page.getByText('sample-resume.pdf')).toBeVisible()
  await expect(page.getByTestId('upload-progress')).toBeVisible()
  await expect(page.getByTestId('parsed-content')).toBeVisible({ timeout: 30000 })
  await expect(page.getByText('Work Experience')).toBeVisible()
})

test('drag and drop PDF upload', async ({ page }) => {
  await page.goto('/upload')

  const buffer = fs.readFileSync(path.join(__dirname, 'fixtures', 'resume.pdf'))
  const dataTransfer = await page.evaluateHandle((b64) => {
    const dt = new DataTransfer()
    const file = new File([Uint8Array.from(atob(b64), c => c.charCodeAt(0))], 'resume.pdf', { type: 'application/pdf' })
    dt.items.add(file)
    return dt
  }, buffer.toString('base64'))

  await page.dispatchEvent('[data-testid="dropzone"]', 'drop', { dataTransfer })
  await expect(page.getByText('resume.pdf')).toBeVisible()
})
```

Testing streaming AI responses:

typescript

```
test('displays AI-generated improvements incrementally', async ({ page }) => {
  await page.goto('/editor')

  await page.getByTestId('resume-content').fill('managed team')
  await page.getByRole('button', { name: 'Improve' }).click()

  // Verify streaming indicator appears
  await expect(page.getByTestId('typing-indicator')).toBeVisible()

  // Wait for streaming completion
  await expect(page.getByTestId('typing-indicator')).not.toBeVisible({ timeout: 30000 })

  // Verify improved content appears
  const improvedText = await page.getByTestId('ai-suggestion').textContent()
  expect(improvedText).toMatch(/led|spearheaded|directed/i)
})
```

Testing split-pane editor interactions:

typescript

```
test('resize split pane by dragging divider', async ({ page }) => {
  await page.goto('/editor')

  const divider = page.getByTestId('split-pane-divider')
  const leftPane = page.getByTestId('resume-editor')
  const initialBox = await leftPane.boundingBox()

  const dividerBox = await divider.boundingBox()
  await page.mouse.move(dividerBox.x + dividerBox.width / 2, dividerBox.y + 100)
  await page.mouse.down()
  await page.mouse.move(dividerBox.x + 150, dividerBox.y + 100, { steps: 10 })
  await page.mouse.up()

  const newBox = await leftPane.boundingBox()
  expect(newBox.width).toBeGreaterThan(initialBox.width)
})
```

Visual regression testing uses Playwright's built-in screenshot comparison (free) or Chromatic for team collaboration:

typescript

```
test('resume template renders correctly', async ({ page }) => {
  await page.goto('/templates/modern')

  // Mask dynamic content
  await expect(page).toHaveScreenshot('modern-template.png', {
    mask: [page.locator('.timestamp'), page.locator('.user-avatar')],
  })
})

// Disable Framer Motion animations for consistent snapshots
// Add to your test setup or component providers:
import { MotionGlobalConfig } from 'framer-motion'
MotionGlobalConfig.skipAnimations = process.env.NODE_ENV === 'test'
```

AI/ML output validation demands multi-layered testing

Testing AI-generated resume content requires a tiered approach: fast code-based validation first, embedding similarity second, and AI-as-judge for complex quality assessments.

Metrics-based content validation (free, fast):

typescript

```
// lib/resume-validator.ts
const ACTION_VERBS = ['led', 'managed', 'developed', 'achieved', 'increased',
  'reduced', 'created', 'launched', 'optimized', 'spearheaded', 'orchestrated']

export function validateResumeQuality(text: string): QualityReport {
  const words = text.toLowerCase().split(/\s+/)

  // Count action verbs
  const actionVerbCount = ACTION_VERBS.filter(verb =>
    words.some(w => w.startsWith(verb))
  ).length

  // Detect quantifiable achievements
  const quantifiablePattern = /(\d+%|\$[\d,]+|\d+x|\d+\s*(users|customers|revenue|sales))/gi
  const achievements = (text.match(quantifiablePattern) || []).length

  // Flesch-Kincaid readability
  const sentences = text.split(/[.!?]+/).filter(s => s.trim())
  const avgWordsPerSentence = words.length / sentences.length
  const readability = 206.835 - 1.015 * avgWordsPerSentence - 84.6 * (countSyllables(text) / words.length)

  return {
    actionVerbCount,
    quantifiableAchievements: achievements,
    readabilityScore: Math.max(0, Math.min(100, readability)),
    passed: actionVerbCount >= 3 && achievements >= 2 && readability >= 30,
  }
}

// tests/ai/resume-quality.test.ts
test('AI-generated bullet points meet quality standards', async () => {
  const generated = await generateResumeBullet('managed software team')
  const quality = validateResumeQuality(generated)

  expect(quality.actionVerbCount).toBeGreaterThanOrEqual(1)
  expect(quality.quantifiableAchievements).toBeGreaterThanOrEqual(1)
  expect(quality.readabilityScore).toBeGreaterThan(30)
})
```

AI-as-judge pattern uses a cheaper model to evaluate outputs:

```
typescript
```

```typescript
// lib/ai-evaluator.ts
import { generateObject } from 'ai'
import { openai } from '@ai-sdk/openai'
import { z } from 'zod'

const EvalSchema = z.object({
  relevance: z.number().min(0).max(5),
  accuracy: z.number().min(0).max(5),
  professionalism: z.number().min(0).max(5),
  overall: z.number().min(0).max(1),
  reasoning: z.string(),
})

export async function evaluateResumeContent(
  original: string,
  improved: string,
  jobDescription: string
): Promise<z.infer<typeof EvalSchema>> {
  const { object } = await generateObject({
    model: openai('gpt-4o-mini'), // ~$0.05 per 100 evaluations
    schema: EvalSchema,
    temperature: 0,
    prompt: `Evaluate this resume improvement:

Original: ${original}
Improved: ${improved}
Target job: ${jobDescription}

Score 0-5 on relevance, accuracy, professionalism. Provide overall score 0-1.`,
  })

  return object
}

// tests/ai/judge-evaluation.test.ts
test('improved bullet point scores higher than original', async () => {
  const original = 'did software development'
  const improved = 'Led development of microservices architecture serving 50K daily users'

  const originalScore = await evaluateResumeContent(original, original, 'Senior Engineer')
  const improvedScore = await evaluateResumeContent(original, improved, 'Senior Engineer')

  expect(improvedScore.overall).toBeGreaterThan(originalScore.overall)
```

```
  expect(improvedScore.professionalism).toBeGreaterThanOrEqual(4)
})
```

Semantic similarity with BGE-M3 embeddings:

typescript

```
// lib/embedding-validator.ts
function cosineSimilarity(a: number[], b: number[]): number {
  const dot = a.reduce((sum, val, i) => sum + val * b[i], 0)
  const normA = Math.sqrt(a.reduce((sum, val) => sum + val * val, 0))
  const normB = Math.sqrt(b.reduce((sum, val) => sum + val * val, 0))
  return dot / (normA * normB)
}

export async function validateSemanticPreservation(
  original: string,
  transformed: string,
  threshold = 0.75
): Promise<{ similarity: number; passed: boolean }> {
  const [origEmbed, transEmbed] = await Promise.all([
    getEmbedding(original),  // BGE-M3 embedding
    getEmbedding(transformed),
  ])

  const similarity = cosineSimilarity(origEmbed, transEmbed)
  return { similarity, passed: similarity >= threshold }
}

// tests/ai/semantic-preservation.test.ts
test('improved resume preserves original meaning', async () => {
  const original = 'Built React applications for clients'
  const improved = await improveWithAI(original)

  const { similarity, passed } = await validateSemanticPreservation(original, improved)
  expect(passed).toBe(true)
  expect(similarity).toBeGreaterThan(0.75)
})
```

Testing GLiNER NER extraction accuracy:

typescript

```
// tests/ml/ner-extraction.test.ts
const GOLDEN_DATASET = [
  {
    text: 'Led team at Google from 2019-2023, managing $5M budget',
    expected: [
      { text: 'Google', label: 'ORG' },
      { text: '2019-2023', label: 'DATE' },
      { text: '$5M', label: 'MONEY' },
    ],
  },
]

function calculateF1(predicted, expected) {
  const predSet = new Set(predicted.map(e => `${e.text}:${e.label}`))
  const expSet = new Set(expected.map(e => `${e.text}:${e.label}`))

  let tp = 0, fp = 0, fn = 0
  predSet.forEach(p => expSet.has(p) ? tp++ : fp++)
  expSet.forEach(e => predSet.has(e) ? null : fn++)

  const precision = tp / (tp + fp) || 0
  const recall = tp / (tp + fn) || 0
  return 2 * (precision * recall) / (precision + recall) || 0
}

test('GLiNER extracts entities with F1 > 0.8', async () => {
  for (const testCase of GOLDEN_DATASET) {
    const predicted = await gliner.predict(testCase.text, ['ORG', 'DATE', 'MONEY'], { threshold: 0.5 })
    const f1 = calculateF1(predicted, testCase.expected)
    expect(f1).toBeGreaterThan(0.8)
  }
})
```

Handling non-deterministic outputs through property-based testing:

typescript

```
// tests/ai/property-tests.test.ts
const RESUME_PROPERTIES = [
  { name: 'hasActionVerb', test: (_, out) => /led|managed|built|created/i.test(out) },
  { name: 'hasReasonableLength', test: (_, out) => out.length >= 50 && out.length <= 500 },
  { name: 'preservesContext', test: (inp, out) => {
    const keywords = inp.match(/\b\w{4,}\b/g) || []
    return keywords.some(k => out.toLowerCase().includes(k.toLowerCase()))
  }},
]

test('AI outputs satisfy invariant properties across runs', async () => {
  const input = 'managed development team'
  const runs = 5

  for (const prop of RESUME_PROPERTIES) {
    let passCount = 0
    for (let i = 0; i < runs; i++) {
      const output = await improveWithAI(input)
      if (prop.test(input, output)) passCount++
    }
    expect(passCount / runs).toBeGreaterThanOrEqual(0.8) // 80% must pass
  }
})
```

Running local AI models in CI works on standard runners using CPU inference (10-50x slower than GPU but free). For faster GPU execution, GitHub's larger runners cost ~$0.07/minute—use label-triggered workflows to run ML tests only when needed.

Tool pricing for budget-conscious teams

Visual regression testing

Tool	Free Tier	Starter	Notes
Playwright built-in	Unlimited	N/A	Self-managed baselines
Lost Pixel	OSS unlimited	Free for OSS	Open source, self-hostable lost-pixel
Argos CI	5K screenshots/mo	Contact	100% open source argos-ci
Chromatic	5K snapshots/mo	$149/mo	Best Storybook integration
Percy	5K screenshots/mo	$149/mo	BrowserStack ecosystem

Coverage and test management

Tool	Free Tier	Paid	Notes
Codecov	5 users, OSS free Vercel	$4-10/user/mo GitHub	Industry standard
Coveralls	OSS free	$5/mo (1 repo) GitHub	Per-repo pricing
Qase	3 users	$20/user/mo SaaSworthy	Modern test management
TestRail	None	$36-74/user/mo SaaSworthy	Enterprise-focused

Recommended free stack ($0/month)

For teams under 5 developers, this stack provides comprehensive testing at zero cost:

- **CI/CD**: GitHub Actions (2,000 free minutes/month)
- **Unit testing**: Vitest (free, fastest option)
- **E2E testing**: Playwright (free unlimited parallelization)
- **Visual regression**: Playwright screenshots or Lost Pixel
- **Coverage**: Codecov free tier (5 users) Vercel
- **Test management**: Qase free tier (3 users) Testingtools
- **LLM testing**: DeepEval (open source) GitHub

Upgrade path: When growing beyond 5 developers, add Chromatic ($149/month) for visual testing collaboration and Codecov Pro ($10/user/month) for advanced analytics—total **~$200/month** for a 10-person team.

Testing specific application features

Testing Docling PDF parsing accuracy uses golden file comparisons:

typescript

```
// tests/pdf-parsing/docling.test.ts
const FIXTURES_DIR = path.join(__dirname, 'fixtures')
const GOLDEN_DIR = path.join(__dirname, 'golden')

test('parses multi-column resume correctly', async () => {
  const pdfBuffer = await fs.readFile(path.join(FIXTURES_DIR, 'two-column-resume.pdf'))
  const result = await parsePdfWithDocling(pdfBuffer)

  const golden = JSON.parse(await fs.readFile(path.join(GOLDEN_DIR, 'two-column-resume.json'), 'utf-8'))

  // Compare extracted sections
  expect(result.sections.map(s => s.title)).toEqual(golden.sections.map(s => s.title))

  // Fuzzy text comparison (OCR may have minor differences)
  const similarity = await computeTextSimilarity(result.fullText, golden.fullText)
  expect(similarity).toBeGreaterThan(0.95)
})
```

Testing ATS compliance of generated PDFs:

```typescript
```

```
test('generated PDF passes ATS parsing', async () => {
  const resumeHTML = await renderResumeTemplate(sampleData)
  const pdfBuffer = await generatePdfWithPuppeteer(resumeHTML)

  // Parse with pdf-parse (simulates ATS)
  const { text } = await pdfParse(pdfBuffer)

  // Verify all required fields are extractable
  expect(text).toContain(sampleData.name)
  expect(text).toContain(sampleData.email)
  expect(text).toContain(sampleData.phone)

  // Verify sections appear in correct order
  const experienceIdx = text.indexOf('Experience')
  const educationIdx = text.indexOf('Education')
  expect(experienceIdx).toBeLessThan(educationIdx)

  // No invisible text or encoding issues
  expect(text).not.toMatch(/[\uFFFD\u0000-\u001F]/)
})
```

Testing rate limiting for Apify scrapers:

typescript

```typescript
// tests/services/rate-limiter.test.ts
test('respects rate limits for Apify calls', async () => {
  vi.useFakeTimers()

  const client = new RateLimitedApifyClient({
    maxRequestsPerMinute: 2,
  })

  const mockCall = vi.fn().mockResolvedValue({ id: 'run-1' })
  client.apifyClient.actor = vi.fn().mockReturnValue({ call: mockCall })

  // First two calls succeed immediately
  await client.callActor('test-actor', {})
  await client.callActor('test-actor', {})
  expect(mockCall).toHaveBeenCalledTimes(2)

  // Third call is queued
  const thirdCall = client.callActor('test-actor', {})
  expect(mockCall).toHaveBeenCalledTimes(2)

  // After 60 seconds, third call executes
  await vi.advanceTimersByTimeAsync(60000)
  await thirdCall
  expect(mockCall).toHaveBeenCalledTimes(3)
})
```

Test file organization and best practices

Recommended directory structure for Next.js 15 App Router:

```
project/
├── src/
│   ├── app/
│   │   ├── api/
│   │   │   └── resumes/
│   │   │       ├── route.ts
│   │   │       └── route.test.ts      # Co-located API tests
│   │   └── (routes)/
│   │       └── editor/
│   │           └── page.tsx
│   ├── components/
│   │   ├── ResumeEditor/
│   │   │   ├── ResumeEditor.tsx
│   │   │   ├── ResumeEditor.test.tsx  # Co-located component tests
│   │   │   └── ResumeEditor.stories.tsx
│   └── lib/
│       └── validators/
│           ├── resume-validator.ts
│           └── resume-validator.test.ts
├── tests/
│   ├── e2e/                       # Playwright E2E tests
│   │   ├── resume-upload.spec.ts
```

```
│   │   ├── editor-workflow.spec.ts
│   │   └── fixtures/
│   │       └── sample-resume.pdf
│   ├── integration/              # Cross-module integration tests
│   │   └── resume-pipeline.test.ts
│   ├── ml/                       # AI/ML model tests
│   │   ├── embedding-similarity.test.ts
│   │   └── ner-extraction.test.ts
│   ├── golden/                   # Golden reference files
│   │   └── parsed-resumes/
│   ├── mocks/
│   │   ├── handlers.ts           # MSW handlers
│   │   └── apify.ts
│   └── setup/
│       ├── db.ts                 # Test database factory
│       └── query-client.ts
├── vitest.config.mts
├── playwright.config.ts
└── .github/workflows/test.yml
```

Fixture management for complex test data:

```typescript
```

```ts
// tests/fixtures/resume-factory.ts
import { faker } from '@faker-js/faker'

export function createMockResume(overrides = {}) {
  return {
    id: faker.string.uuid(),
    userId: faker.string.uuid(),
    name: faker.person.fullName(),
    email: faker.internet.email(),
    phone: faker.phone.number(),
    experience: [
      {
        company: faker.company.name(),
        role: faker.person.jobTitle(),
        startDate: faker.date.past({ years: 5 }),
        endDate: faker.date.recent(),
        bullets: [
          `Led ${faker.number.int({ min: 3, max: 15 })} engineers to deliver ${faker.commerce.productName()}`,
          `Increased revenue by ${faker.number.int({ min: 10, max: 50 })}% through ${faker.hacker.verb()} initiatives`,
        ],
      },
    ],
    education: [{
      school: faker.company.name() + ' University',
      degree: faker.helpers.arrayElement(['BS', 'MS', 'PhD']),
      field: faker.helpers.arrayElement(['Computer Science', 'Engineering', 'Business']),
      year: faker.date.past({ years: 10 }).getFullYear(),
    }],
    skills: faker.helpers.arrayElements(
      ['JavaScript', 'TypeScript', 'React', 'Node.js', 'Python', 'AWS', 'Docker'],
      { min: 4, max: 8 }
    ),
    ...overrides,
  }
}

// Usage in tests
test('renders complete resume', () => {
  const resume = createMockResume({ name: 'Test User' })
  render(<ResumePreview resume={resume} />)
  expect(screen.getByText('Test User')).toBeInTheDocument()
})
```

Phased implementation plan

Week 1-2: Foundation

1. Install Vitest with React Testing Library
2. Configure test setup file with Zustand mocks
3. Write first unit tests for Zod schemas and utility functions
4. Set up basic GitHub Actions workflow

Week 3-4: Component and integration testing

1. Add MSW for API mocking
2. Test React Hook Form integrations
3. Test TanStack Query hooks with QueryClientProvider wrapper
4. Add Drizzle ORM test database setup
5. Write API route handler tests

Week 5-6: E2E and visual testing

1. Install and configure Playwright
2. Write critical path E2E tests (upload, edit, generate)
3. Add visual regression with Playwright screenshots
4. Configure parallel test execution in CI

Week 7-8: AI/ML validation

1. Implement metrics-based content validation
2. Create golden datasets for NER testing
3. Add AI-as-judge evaluation for complex quality checks
4. Write property-based tests for non-deterministic outputs

Ongoing: Maintenance and optimization

- Monitor coverage and enforce thresholds
- Update golden files when intentionally changing behavior
- Optimize CI caching for faster builds
- Add tests for new features before implementation (TDD)

This testing strategy provides comprehensive coverage while remaining cost-effective. The combination of Vitest's speed, Playwright's free parallelization, and tiered AI validation ensures your Next.js 15 application maintains quality as it scales.