Contract Testing for Internal APIs

Published on April 7, 2024

Webstack Builders

Table of Contents

When Service A depends on Service B's API, how do you ensure changes to B don't break A? Integration tests are slow and can be flaky. Documentation drifts from reality within weeks. Manual coordination between teams doesn't scale past a handful of services.

I've seen this play out more than once. A User Service team ships what they consider a minor change: renaming `userId` to `user_id` in their response payload to match their new coding standards. They updated their OpenAPI spec. They ran their own tests. Everything passed.

Three services broke in production.

The Order Service, Shipping Service, and Analytics Service all consumed that field. Nobody had checked with them. The OpenAPI spec was correct for the **new** behavior, but the consumers were still expecting the old field name. The deployment happened on a Friday afternoon. The on-call engineer spent the weekend coordinating rollbacks and emergency patches.

Contract testing would have caught this at PR time. Before the User Service merge, automated tests would have run against the actual consumer expectations, not just the provider's idea of what consumers need. The build would have failed with a clear message: "OrderService expects field 'userId', but response contains 'user_id'."

Consumer-driven contracts flip the traditional testing model. Instead of the provider defining what it offers and hoping consumers adapt, consumers define what they need and providers verify they can deliver it. The contract becomes a shared artifact that both sides test against, ensuring compatibility before code leaves the developer's machine.

> **INFO**
>
> Consumer-driven contracts catch breaking changes at PR time, not in production. The provider can't ship changes that break consumers because the consumer's expectations are encoded in automated tests.

The Problem with Traditional Integration Testing

The instinct when services start breaking is to add more integration tests. If Service A calling Service B fails in production, write a test that spins up both services and verifies the call works. Problem solved, right?

This approach collapses under its own weight as services multiply.

Why Integration Tests Fail at Scale

Integration tests require **all** services to be running simultaneously. For a simple three-service chain, that means coordinating databases, message queues, and network connectivity across all three. Add a fourth service, and the coordination overhead grows. By the time you have 20 services, the "integration test environment" has become a full-time job for someone on the platform team.

The problems compound from there. Shared test databases accumulate garbage data from previous runs, causing tests to fail for reasons unrelated to the code change. Port conflicts appear when two developers run tests simultaneously. Network timeouts introduce flakiness that erodes trust in the test suite. Developers stop running integration tests locally because they take too long, pushing the feedback loop to CI where it's even slower.

When an integration test fails, debugging becomes archaeology. A failure in Service X might be caused by a change in Service Y, but the stack trace only shows where the exception was thrown, not where the problem originated. Reproducing the failure locally requires spinning up the entire environment, which often behaves differently than CI.

#	Testing Approach	Speed	Reliability	Isolation	Catches Breaking Changes
1	Unit tests	Fast	High	Complete	No (mocks hide reality)
2	Integration tests	Slow	Low	None	Yes (but late)
3	E2E tests	Very slow	Very low	None	Yes (in production-like)
4	Contract tests	Fast	High	Complete	Yes (at PR time)

Comparison of testing approaches for API compatibility.

The fundamental problem is that integration tests conflate two concerns: verifying that your code works correctly, and verifying that your code is compatible with its dependencies. Contract tests separate these concerns. Unit tests verify correctness. Contract tests verify compatibility. Neither requires spinning up the entire world.

The Documentation Lie

Teams often try to solve the compatibility problem with documentation. “We’ll maintain an OpenAPI spec, and consumers will code against it.” This works for about three months.

openapi-drift.yaml

```yaml
# What the OpenAPI spec says
paths:
  /users/{id}:
    get:
      responses:
        200:
          content:
            application/json:
              schema:
                type: object
                properties:
                  userId:      # Spec says userId
                    type: string
                  email:
                    type: string

# What the implementation actually returns
# {
#   "user_id": "123",  # Implementation uses user_id
#   "email": "test@example.com"
# }

# The spec was written once and never updated
# Consumers who trusted the spec are now broken
```

OpenAPI specs drift from implementation without enforcement.

The spec was written when the API was first built. Six months later, someone refactored the response format to match a new coding standard. They updated the code, ran the tests, and shipped. The OpenAPI spec stayed frozen in time because updating it was “on the backlog.” Consumers who built against the documented spec are now broken.

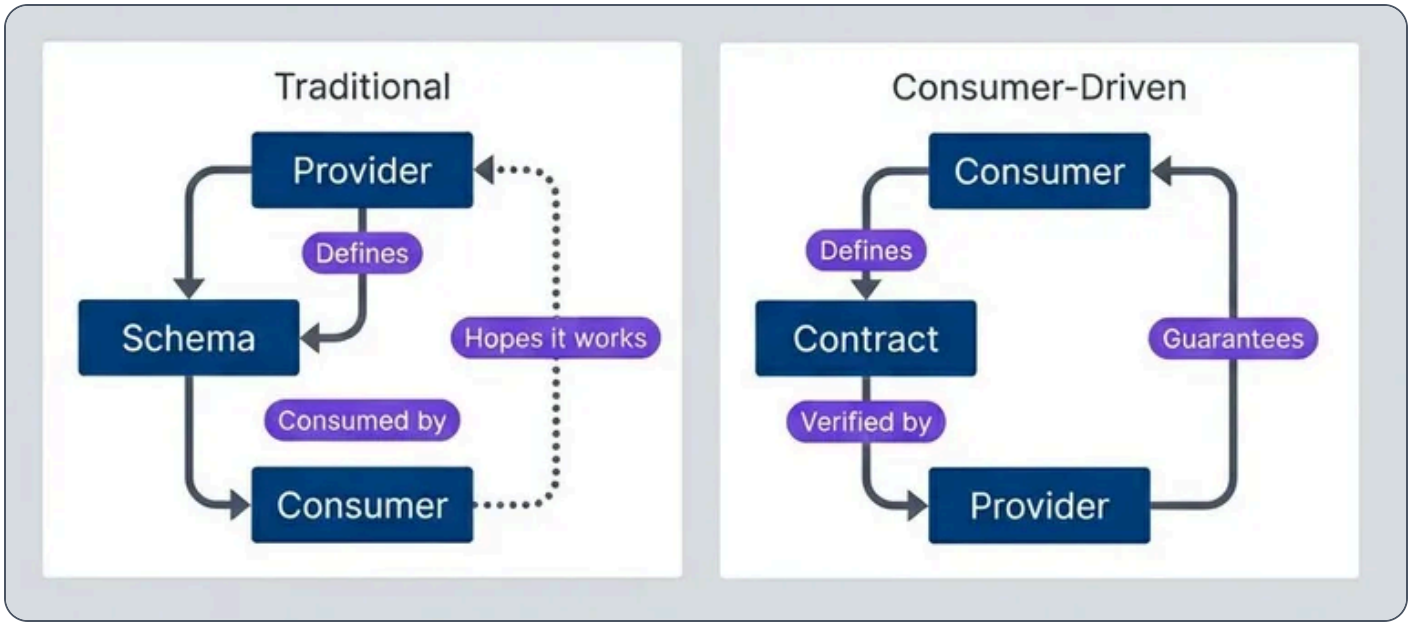
> **⚠ WARNING**
>
> Documentation without enforcement is fiction. Schemas describe intent; contract tests verify reality.

Consumer-Driven Contracts Explained

The Core Concept

Traditional API testing puts the provider in charge. The provider defines a schema, publishes documentation, and consumers build against it. If the provider changes the API, consumers find out when their code breaks — often in production.

Consumer-driven contracts invert this model. The **consumer** defines what it needs from the provider and encodes those expectations in a contract. The provider then verifies it can satisfy that contract. Both sides test against the same artifact, so compatibility is verified before either side deploys.



Traditional vs consumer-driven contract model

The flow works like this: First, the consumer writes a test that defines its expectations. “When I call `GET /users/123`, I expect a response with a `userId` string and an `email` string.” That test generates a contract file describing the expected interaction.

The contract gets stored somewhere accessible to the provider — usually a Pact Broker, though a file share or git repository works for simpler setups. The provider then runs verification tests that fetch these contracts and replay the interactions against the real implementation. If the provider’s response doesn’t match what the consumer expects, the test fails.

> **INFO**
>
> Pact is the most widely adopted tool for consumer-driven contract testing. The project includes language-specific libraries for writing contract tests, a mock server for consumer testing, and the Pact Broker — an open-source registry for storing and versioning contracts. We’ll dive deep into Pact’s architecture and usage in the next section.

The key insight is that both sides now have a shared source of truth. The consumer knows the provider will deliver what it needs because the provider’s CI pipeline verifies it. The provider knows exactly which fields each consumer depends on, making it safe to change or remove anything that isn’t in a contract.

What Goes in a Contract

Contracts should test API shape, not business logic. This distinction trips up teams new to contract testing.

A contract verifies **structure**: the request path, HTTP method, required headers, response status code, and the shape of the response body. It answers questions like “does the response have a `userId` field that’s a string?” and “does a 404 response include an `error` object?”

A contract does **not** verify business logic: it shouldn’t check that user 123 has a specific email address, or that creating a user with duplicate email returns a particular error message. Those are integration test concerns. The consumer shouldn’t know or care about the provider’s database state.

In Scope	Out of Scope
Request path and method	Specific data values
Required headers	Database state
Response status codes	Exact error messages
Response body structure	Performance characteristics
Field types (string, number, array)	Security rules

Contract scope boundaries.

This boundary keeps contracts stable. Business logic changes frequently — error messages get reworded, validation rules evolve, edge cases get handled differently. API structure changes rarely, and when it does, it's usually a breaking change that consumers need to know about.

> ✓ SUCCESS
>
> Contract tests verify the API contract, not business logic. They answer "does the response have a userId field that's a string?" not "does user 123 exist?"

Who Drives API Evolution?

The name "consumer-driven contracts" raises an obvious question: if consumers drive the contracts, who drives the API? Are consumer teams dictating the provider's backlog? How do API developers evolve their services if they don't control what they're implementing?

The short answer: "consumer-driven" describes where contracts come from, not who designs the API.

Providers Still Design the API

Provider teams design, implement, and evolve their APIs exactly as they always have. They decide endpoint structure, field names, data types, versioning strategy — all of it. Consumer-driven contracts don't change that.

What changes is **visibility**. Before CDC (Consumer-Driven Contracts), a provider team might rename a field and discover three weeks later (in a staging environment, or worse, production) that two other services depended on the old name. With CDC (Consumer-Driven Contracts), they discover it immediately: the contract tests fail, the Pact Broker shows exactly which consumers use that field, and the conversation happens before any code deploys.

The Workflow for API Changes

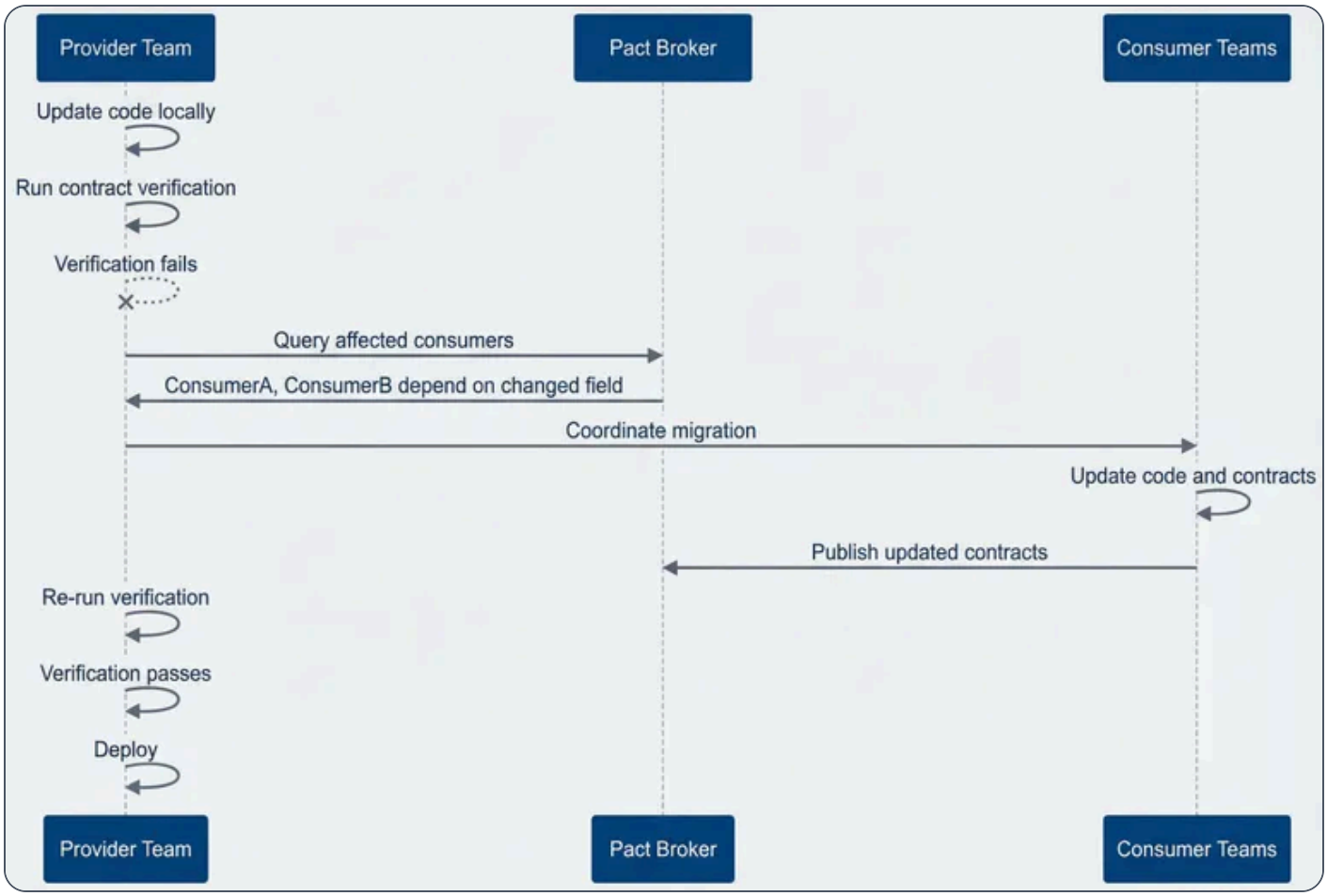
API evolution follows one of two patterns depending on whether the change is additive or breaking:

Additive changes (new endpoints, new optional fields) require no coordination. Providers add the capability, deploy it, and consumers adopt it when ready. As consumers start using the new feature, they add interactions to their contracts. The provider already satisfies these contracts — that's what "additive" means.

Breaking changes (removing fields, renaming fields, changing types) trigger a conversation. When a provider needs to make a breaking change, the workflow is:

- Provider team updates their code and runs contract verification locally
- Verification fails for consumers using the affected fields
- Pact Broker shows exactly which consumers depend on the changing contract
- Provider team coordinates with affected consumer teams
- Consumer teams update their code and contracts
- Once all consumers have migrated, the provider deploys

The diagram below shows this coordination in practice. Notice that the provider never deploys until verification passes against every affected consumer — the Pact Broker acts as the single source of truth for who depends on what.



Breaking change coordination workflow with Pact

This isn't providers waiting on consumers' permission — it's providers having **evidence** of who they need to coordinate with. Before CDC (Consumer-Driven Contracts), step 3 was "check the wiki, ask around in Slack, hope you found everyone." With CDC (Consumer-Driven Contracts), step 3 is "query the Pact Broker."

Contracts Capture Usage, Not Requirements

Think of consumer contracts as usage telemetry, not requirements documents. A consumer contract says "we currently call this endpoint and expect these fields." It's not a feature request or a demand — it's a statement of fact about the integration.

When a consumer needs something the provider doesn't offer, that's a normal product conversation: file a ticket, discuss in sprint planning, negotiate priority. The consumer can even write a contract for the **expected** behavior before the provider implements it — Pact handles this gracefully with "pending pacts" (covered in the

Scaling section).

The key mindset shift: contracts aren't constraints that providers must satisfy. They're visibility into what consumers actually depend on, so changes can be coordinated instead of hoped-for.

Pact: The De Facto Standard

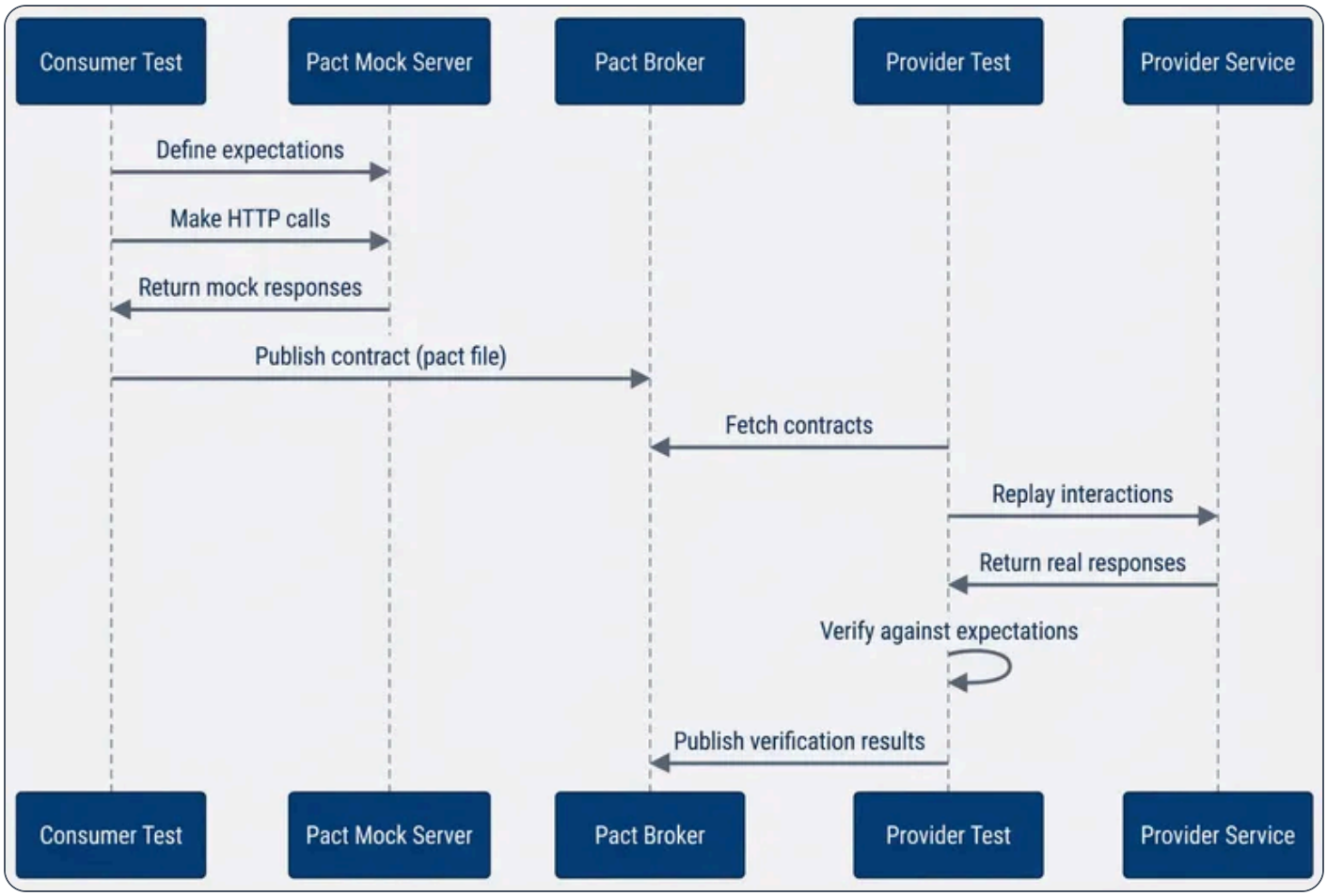
Pact Architecture

Pact consists of a component you deploy and a component you run locally or in CI.

The component you deploy is the **Pact Broker** — a central registry that stores contracts and tracks compatibility between service versions. It's distributed as a Docker image (`pactfoundation/pact-broker`) and runs as a single container backed by PostgreSQL. In Kubernetes, that's typically a Deployment with one replica plus a database (or a managed PostgreSQL service). If you're using GitOps with Argo CD or Flux, the broker deploys like any other stateless service. Resource requirements are modest — it's essentially a Ruby web app with a database.

The component you run locally and in CI is the **Pact library** for your language. Pact has official libraries for JavaScript/TypeScript, Java, Go, Python, Ruby, .NET, and others. These libraries provide two capabilities: a mock server for consumer tests, and a verifier for provider tests.

Here's how the pieces fit together:



Pact workflow from consumer test to provider verification

On the **consumer side**, you write Pact tests that describe what your service expects from its dependencies. These tests run against a mock server that the Pact library spins up locally — no network calls to real services. You define the request you'll make and the response you expect. The mock server returns that response, and your test verifies your code handles it correctly. When the test passes, Pact generates a "pact file"—a JSON document describing every interaction your consumer expects.

That pact file gets published to the Pact Broker using the `pact-broker` CLI, typically as an automated CI step after tests pass.

On the **provider side**, verification tests fetch contracts from the broker, replay each recorded interaction against the real running provider service, and compare actual responses to what consumers expect. If the provider's response doesn't match, the test fails. Results get published back to the broker, completing the compatibility record.

The broker's killer feature is webhooks. When a consumer publishes a new contract, the broker can trigger the provider's CI pipeline via webhook to run verification immediately. This closes the feedback loop: a consumer change that would break the provider gets caught within minutes, not after deployment.

Consumer Side: Writing Pact Tests

A consumer Pact test has three parts: defining what you expect, exercising your code against the mock server, and letting Pact generate the contract.

The example below shows a typical consumer test using the JavaScript/TypeScript Pact library. The OrderService (our consumer) needs to fetch user details from the UserService (the provider). Instead of calling the real UserService, the test defines what response it expects and runs against a Pact mock server.

consumer-pact-test.ts

```
import { PactV3, MatchersV3 } from '@pact-foundation/pact';

const { like, eachLike, regex, datetime } = MatchersV3;

const provider = new PactV3({
  consumer: 'OrderService',
  provider: 'UserService',
});

describe('User Service Contract', () => {
  it('returns user details for valid user ID', async () => {
    await provider
      .given('a user with ID 123 exists')        // Provider state
      .uponReceiving('a request for user 123')
      .withRequest({
        method: 'GET',
        path: '/users/123',
        headers: { Authorization: like('Bearer token') },
      })
      .willRespondWith({
        status: 200,
        body: {
          userId: like('123'),                   // Type matcher
          email: regex(/^\S+@\S+$/, 'a@b.com'),  // Pattern matcher
          roles: eachLike('admin'),              // Array matcher
```

```
        },
      });

    await provider.executeTest(async (mockServer) => {
      const client = new UserServiceClient(mockServer.url);
      const user = await client.getUser('123');
      expect(user.userId).toBeDefined();
    });
  });
});
```

Consumer-side Pact test with matchers for flexible verification.

When this test passes, Pact generates a contract file (typically JSON) that captures the interaction:

orderservice-userservice.json

```
{
  "consumer": { "name": "OrderService" },
  "provider": { "name": "UserService" },
  "interactions": [{
    "description": "a request for user 123",
    "providerState": "a user with ID 123 exists",
    "request": {
      "method": "GET",
      "path": "/users/123"
    },
    "response": {
      "status": 200,
      "body": {
        "userId": "123",
        "email": "a@b.com",
        "roles": ["admin"]
      },
      "matchingRules": {
        "body": {
          "$.userId": { "match": "type" },
          "$.email": { "match": "regex", "regex": "^\\S+@\\S+$" },
          "$.roles": { "match": "type", "min": 1 }
        }
```

```
        }
      }
    }]
  }
```

Generated pact file showing the contract artifact.

The `matchingRules` section is where matchers get encoded — the provider verification will use these rules rather than exact value comparison.

The `.given()` clause specifies a "provider state"—a precondition that the provider must set up before running verification. The provider will implement a state handler for "a user with ID 123 exists" that seeds its test database appropriately. This keeps the contract focused on structure while letting the provider control its own test data.

The `.withRequest()` and `.willRespondWith()` clauses define the contract itself. Notice the matchers like `like()`, `regex()`, and `eachLike()`—these are critical. Instead of asserting exact values, matchers verify **types and patterns**. The contract says "userId must be a string" rather than "userId must be '123'". This flexibility prevents brittle tests that break when test data changes.

Inside `provider.executeTest()`, you exercise your real client code against the mock server. This verifies that your code actually handles the response shape you've defined. If your client expects `user.id` but the contract says `userId`, the test fails here — before you ever publish the contract.

Understanding Pact Matchers

Matchers are what make contracts flexible instead of brittle. Without matchers, you'd be asserting exact values: "userId must equal '123'". That breaks as soon as the provider uses different test data. Matchers let you assert **types and patterns** instead: "userId must be a string."

Every matcher serves a dual purpose. During consumer tests, the example value gets returned by the mock server so your code has realistic data to work with. During provider verification, the matcher rule gets applied to the actual response to verify it conforms to the contract.

pact-matchers.ts

```typescript
import { MatchersV3 } from '@pact-foundation/pact';

const {
  like,           // Type matching
  eachLike,       // Array with type matching
  regex,          // Regex pattern matching
  datetime,       // DateTime format matching
  integer,        // Integer type
  decimal,        // Decimal type
  boolean,        // Boolean type
  string,         // String type
  nullValue,      // Null value
  uuid,           // UUID format
  email,          // Email format
  ipv4Address,    // IPv4 format
} = MatchersV3;

// Examples
const matcherExamples = {
  // like(): Match type, use example for mock
  userId: like('abc-123'),
  // Contract: must be a string
  // Mock returns: 'abc-123'

  // eachLike(): Array where each item matches type
  items: eachLike({
    id: like('item-1'),
    price: decimal(19.99),
  }),
  // Contract: array of objects with id (string) and price (decimal)
  // Mock returns: [{ id: 'item-1', price: 19.99 }]

  // regex(): Match against pattern
  status: regex(/^(pending|completed|failed)$/, 'pending'),
  // Contract: must match regex
  // Mock returns: 'pending'

  // Nested matching
  metadata: like({
    createdAt: datetime("yyyy-MM-dd'T'HH:mm:ss'Z'"),
    updatedBy: like('user-123'),
```

```
      tags: eachLike('important'),
    }),
  };
```

Pact matchers for flexible contract definitions.

The table below summarizes the most commonly used matchers. In practice, `like()` handles 80% of cases — use it as your default and reach for more specific matchers when you need tighter validation.

Matcher	Use Case	Contract Meaning
`like(example)`	Most fields	Same type as example
`eachLike(example)`	Arrays	Array with items of same type
`regex(pattern, example)`	Enums, formats	Must match regex
`datetime(format)`	Timestamps	Valid datetime in format
`integer(example)`	Counts, IDs	Integer number
`decimal(example)`	Prices, rates	Decimal number
`uuid()`	Identifiers	Valid UUID format

Common Pact matchers and their use cases.

INFO

Matchers are the key to maintainable contracts. Use `like()` for most fields — it verifies structure without coupling to specific values.

Provider Side: Verification

Provider verification is the other half of the contract testing loop. While consumer tests generate contracts, provider tests verify the real service can satisfy them.

The verification process works like this: the Pact verifier fetches contracts from the broker (or reads them from local files), replays each interaction against your running service, and compares actual responses to the contract expectations. If all matchers pass, the verification succeeds. If any response doesn't match, you get a detailed diff showing exactly what went wrong.

provider-pact-test.ts

```
import { Verifier } from '@pact-foundation/pact';
import { app } from '../src/app';

describe('User Service Provider Verification', () => {
  let server: Server;

  beforeAll(async () => {
    // Provider service port (not the Pact Broker port 9292)
    server = app.listen(3001);
  });

  afterAll(async () => {
    server.close();
  });

  it('validates the expectations of OrderService', async () => {
    const verifier = new Verifier({
      provider: 'UserService',
      providerBaseUrl: 'http://localhost:3001',

      // Pact Broker configuration
      pactBrokerUrl: process.env.PACT_BROKER_URL,
      pactBrokerToken: process.env.PACT_BROKER_TOKEN,

      // Or local pact files
      // pactUrls: ['./pacts/orderservice-userservice.json'],

      // Provider states setup
      stateHandlers: {
```

```
30          'a user with ID 123 exists': async () => {
31            await seedDatabase({
32              users: [{ id: '123', email: 'user@example.com', name: 'John Doe' }],
33            });
34          },
35          'no user with ID 999 exists': async () => {
36            await clearDatabase();
37          },
38        },
39
40        // Publish results
41        publishVerificationResult: true,
42        providerVersion: process.env.GIT_COMMIT_SHA,
43        providerVersionBranch: process.env.GIT_BRANCH,
44      });
45
46      await verifier.verifyProvider();
47    });
48  });
```

Provider-side Pact verification with state handlers.

The `stateHandlers` object is where you implement the provider states that consumers reference in their `.given()` clauses. When the verifier encounters a contract with `.given('a user with ID 123 exists')`, it calls your state handler to set up the test data before replaying that interaction. This keeps test data management on the provider side where it belongs — the consumer just declares what precondition it needs.

The `publishVerificationResult` flag tells Pact to report results back to the broker, completing the compatibility matrix. The `providerVersion` should be your git commit SHA so you can track exactly which code version verified which contracts.

Provider States

Provider states bridge the gap between contracts and test data. A contract says "when I request user 123, I expect these fields." But where does user 123 come from? The provider needs to set that up — and that's what state handlers do.

provider-states.ts

```
const stateHandlers: StateHandlers = {
  'a user exists': async () => {
    await db.users.create({ id: 'test-user', email: 'test@example.com' });
  },

  'a user with ID {id} exists': async (params) => {
    await db.users.create({
      id: params.id,
      email: `user-${params.id}@example.com`,
    });
  },

  // State with cleanup
  'the system has no users': async () => {
    await db.users.deleteAll();
    return () => {
      // Teardown function (optional)
      console.log('Cleaned up empty state');
    };
  },

  'user {userId} has {count} orders': async (params) => {
    // Parameterized state with multiple entities
    const user = await db.users.create({ id: params.userId });
    for (let i = 0; i < params.count; i++) {
      await db.orders.create({ userId: user.id, orderId: `order-${i}` });
    }
  },
};
```

Provider state handlers for test data setup.

State handlers support three patterns. Simple states like `'a user exists'` take no parameters — they set up a known fixture. Parameterized states like `'a user with ID {id} exists'` extract values from the state description, letting consumers specify exact IDs when they need predictable responses. Complex states combine multiple parameters to set up richer scenarios.

The naming convention matters. When a consumer writes `.given('a user with ID 123 exists')`, Pact matches it against your `'a user with ID {id} exists'` handler and passes `{ id: '123' }` as the params object. Keep state descriptions readable — they show up in the Pact Broker UI and serve as documentation.

> **⚠ WARNING**
>
> Provider states should be minimal — just enough to satisfy the contract. Don't build elaborate fixtures; contract tests verify structure, not business logic.

Schema-Based Alternatives

Not every team needs Pact. Schema-based approaches offer simpler tooling when you don't need the full consumer-driven workflow — especially for public APIs with unknown consumers or teams already maintaining OpenAPI specifications.

OpenAPI-Based Contract Testing

OpenAPI-based tools flip the script: instead of consumers generating contracts, you maintain a specification and validate both sides against it. Several tools take your OpenAPI spec and generate tests automatically:

Prism (from Stoplight)	Acts as a mock server and validation proxy. Point it at your OpenAPI spec and it validates requests/responses in real-time.
Dredd	Generates and runs tests from your spec, comparing actual API responses to documented examples.
Schemathesis	Takes a property-based testing approach, generating hundreds of test cases from your schema to find edge cases.

These tools work well when you already maintain an OpenAPI spec and want to catch drift between the spec and the implementation. But they test against the specification, not against what consumers actually depend on. The table below compares the trade-offs across approaches.

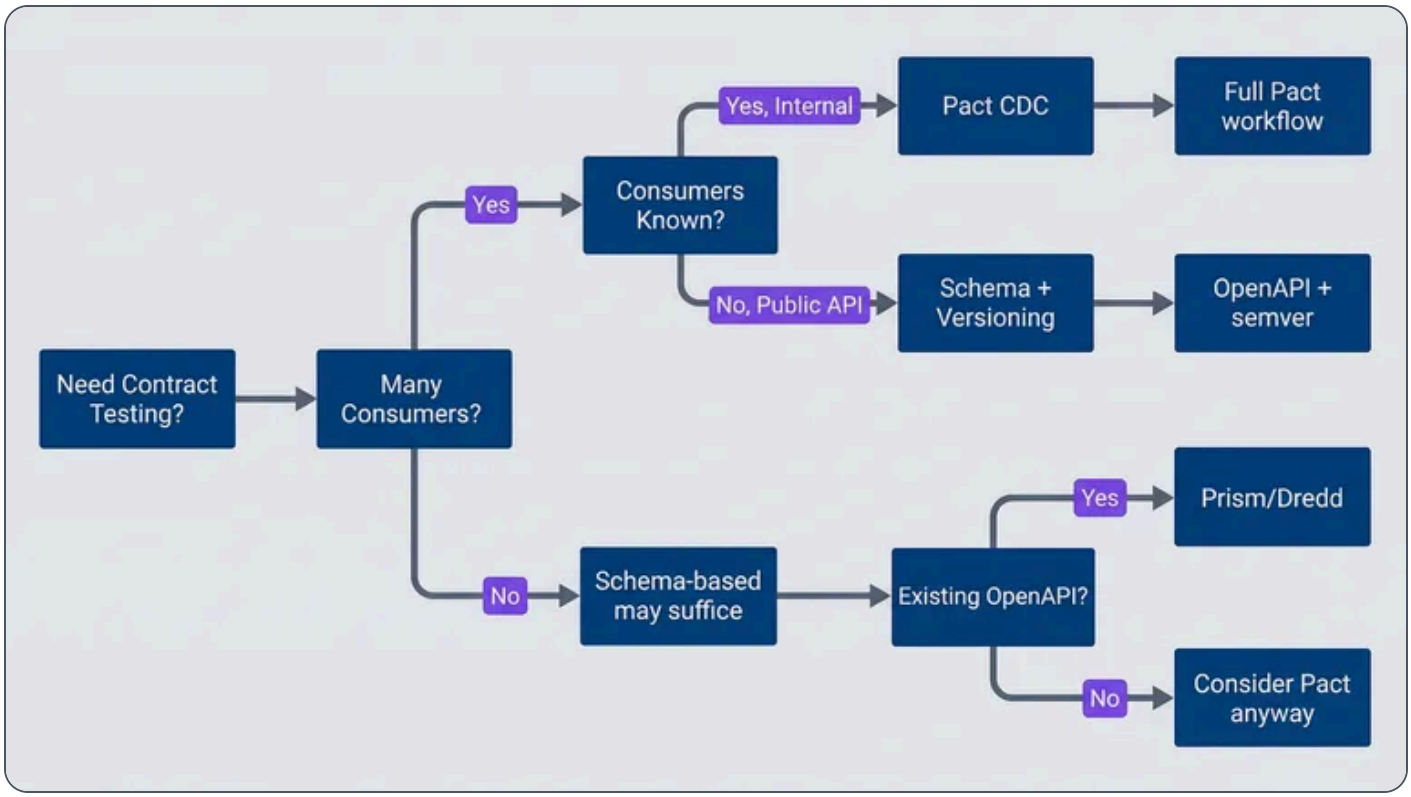
Approach	Contract Source	Strengths	Weaknesses
Pact CDC	Consumer tests	Tests actual consumer needs; fine-grained matchers; built-in broker	Requires buy-in from all teams; learning curve
OpenAPI tools	OpenAPI specification	Uses existing specs; simpler mental model	Specs drift from reality; tests provider, not consumer needs
GraphQL Schema	GraphQL schema	Schema is the contract; breaking change detection	GraphQL-specific

Contract testing approaches compared.

The fundamental difference: Pact tests what consumers actually use, while schema-based tools test what the specification says. If your spec is always accurate and complete, schema-based works great. If your spec drifts (as specs often do), you're testing against documentation, not reality.

When to Use Each Approach

The decision tree is simpler than it looks: if you know your consumers and can coordinate with them, Pact pays dividends. If you're building a public API or can't get consumer teams to write tests, schema-based is more practical.



Decision tree for choosing contract testing approach

For a quick reference, the table below pairs typical architectural and team scenarios with their ideal testing approach.

Scenario	Recommended Approach
Internal microservices, known consumers	Pact CDC
Public API, unknown consumers	OpenAPI + semantic versioning
GraphQL API	GraphQL schema + Apollo Studio
Small team, few services	Pact or schema-based both work
Large org, many teams	Pact with Pact Broker
Legacy API, can't change provider	Schema-based validation only

Scenario	Recommended Approach

Contract testing approach by scenario.

For most internal microservice architectures, Pact is the stronger choice. It catches the exact breaking changes that affect your specific consumers — not theoretical breakages based on a spec that might be outdated.

CI/CD Integration

Contract testing delivers value when it's wired into your deployment pipeline. The goal: make it impossible to deploy incompatible services. This section shows how to set up Pact Broker, configure consumer and provider pipelines, and use `can-i-deploy` as your deployment gate.

Pact Broker Setup

If you're running locally or want a quick Docker Compose setup for development, here's a minimal configuration. For production, you'd typically deploy this to Kubernetes (as shown in the Architecture section) with proper secrets management.

docker-compose-pact-broker.yaml

```yaml
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: pact
      POSTGRES_PASSWORD: pact
      POSTGRES_DB: pact
    volumes:
      - pact-db:/var/lib/postgresql/data

  pact-broker:
    image: pactfoundation/pact-broker:latest
    ports:
      - "9292:9292"
    environment:
      PACT_BROKER_DATABASE_URL: "postgres://pact:pact@postgres/pact"
      PACT_BROKER_BASIC_AUTH_USERNAME: admin
```

```
        PACT_BROKER_BASIC_AUTH_PASSWORD: admin
        PACT_BROKER_ALLOW_PUBLIC_READ: "true"
      depends_on:
        - postgres

volumes:
  pact-db:
```

Docker Compose setup for Pact Broker.

This gives you a running broker at `http://localhost:9292`. The `PACT_BROKER_ALLOW_PUBLIC_READ` setting lets you browse contracts without authentication — useful for development, but disable it in production.

Consumer CI Pipeline

Consumer pipelines do three things: run contract tests (generating pact files), publish those pacts to the broker, and check if deployment is safe. The publish step tags contracts with the git SHA and branch, so the broker knows exactly which code version created each contract.

consumer-ci.yaml

```yaml
# .github/workflows/consumer-ci.yaml
name: Consumer CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Run unit tests
        run: npm test

      - name: Run contract tests
        run: npm run test:contract
        env:
          PACT_BROKER_URL: ${{ secrets.PACT_BROKER_URL }}
          PACT_BROKER_TOKEN: ${{ secrets.PACT_BROKER_TOKEN }}

```

```
      - name: Publish contracts
        if: github.ref == 'refs/heads/main'
        run: |
          npx pact-broker publish ./pacts \
            --consumer-app-version=${{ github.sha }} \
            --branch=${{ github.ref_name }} \
            --broker-base-url=${{ secrets.PACT_BROKER_URL }} \
            --broker-token=${{ secrets.PACT_BROKER_TOKEN }}

      - name: Can I Deploy?
        run: |
          npx pact-broker can-i-deploy \
            --pacticipant=OrderService \
            --version=${{ github.sha }} \
            --to-environment=production \
            --broker-base-url=${{ secrets.PACT_BROKER_URL }} \
            --broker-token=${{ secrets.PACT_BROKER_TOKEN }}
```

Consumer CI pipeline with contract publishing.

The `can-i-deploy` check at the end queries the broker's compatibility matrix. If the provider hasn't verified this consumer version yet (or verification failed), the check fails and blocks deployment. This is the safety net that prevents deploying consumers that would break against production providers.

Provider CI Pipeline

Provider pipelines verify contracts and report results back to the broker. The key difference from consumer pipelines: providers listen for webhook events. When a consumer publishes a new contract, the broker can trigger the provider's CI to verify immediately — no waiting for the next scheduled build.

provider-ci.yaml

```
# .github/workflows/provider-ci.yaml
name: Provider CI

on:
  push:
  pull_request:
  # Webhook from Pact Broker when new contracts published
```

```yaml
    repository_dispatch:
      types: [pact-contract-published]

  jobs:
    verify:
      runs-on: ubuntu-latest
      steps:
        - uses: actions/checkout@v4

        - name: Run unit tests
          run: npm test

        - name: Verify contracts
          run: npm run test:contract:verify
          env:
            PACT_BROKER_URL: ${{ secrets.PACT_BROKER_URL }}
            PACT_BROKER_TOKEN: ${{ secrets.PACT_BROKER_TOKEN }}
            GIT_COMMIT_SHA: ${{ github.sha }}
            GIT_BRANCH: ${{ github.ref_name }}

        - name: Can I Deploy?
          run: |
            npx pact-broker can-i-deploy \
              --pacticipant=UserService \
              --version=${{ github.sha }} \
              --to-environment=production \
              --broker-base-url=${{ secrets.PACT_BROKER_URL }} \
              --broker-token=${{ secrets.PACT_BROKER_TOKEN }}
```

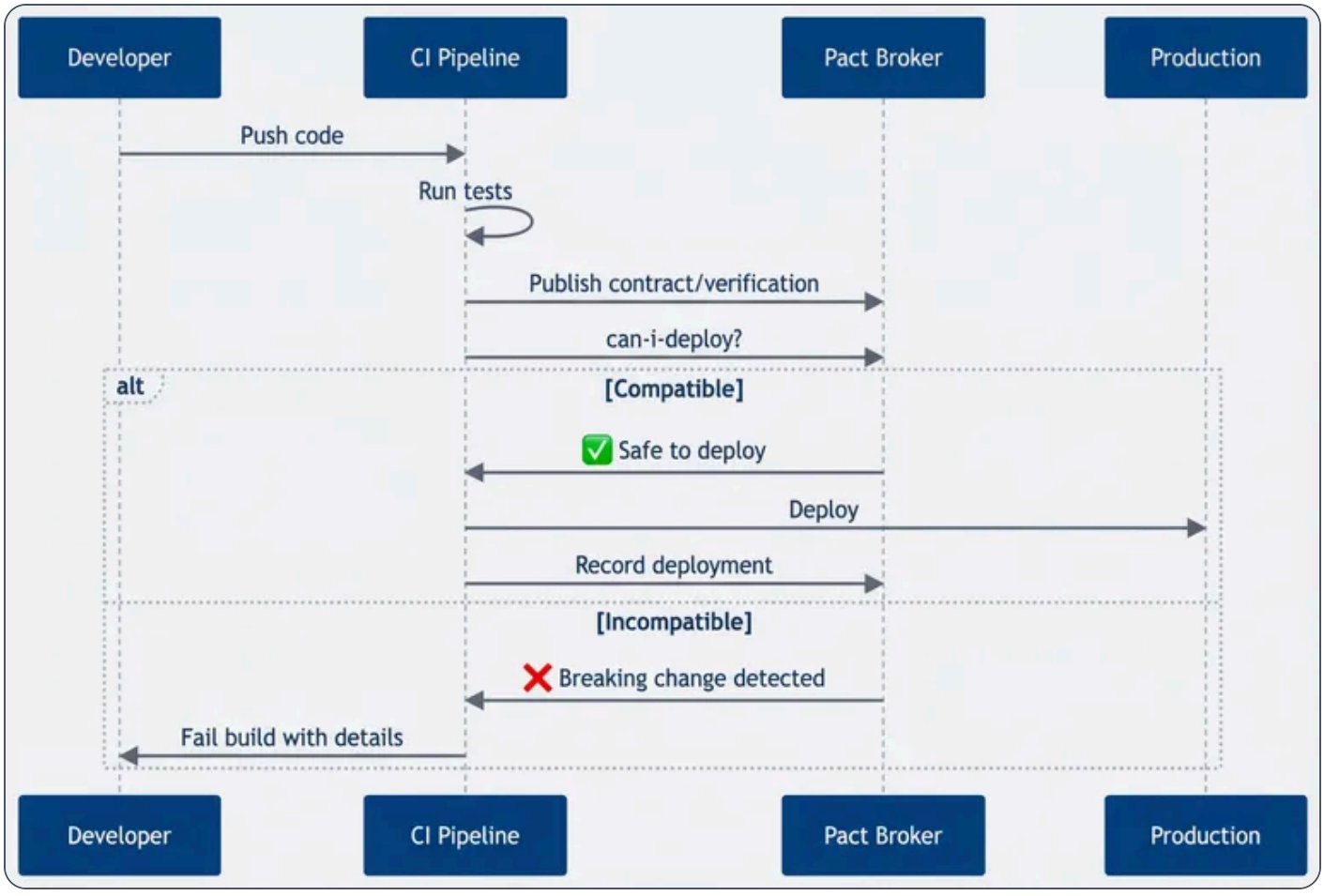
Provider CI pipeline with contract verification.

The `repository_dispatch` trigger is how Pact Broker webhooks reach GitHub Actions. Configure the webhook in the broker to POST to `https://api.github.com/repos/{owner}/{repo}/dispatches` with the event type `pact-contract-published`. This creates a tight feedback loop: consumer publishes contract → broker triggers provider CI → verification results appear within minutes.

> ✓ SUCCESS
>
> The `can-i-deploy` command is the key to safe deployments. It checks if the current version is compatible with all services in the target environment before allowing deployment.

The Can-I-Deploy Flow

The `can-i-deploy` command ties everything together. It queries the Pact Broker's compatibility matrix to determine if a specific version of a service can safely deploy to a target environment. The broker tracks which versions are deployed where, and which contracts have been verified — so it can answer "will this deployment break anything?" with certainty.



Can-I-Deploy flow preventing incompatible deployments

When `can-i-deploy` passes, the pipeline records the deployment with `pact-broker record-deployment`. This updates the broker's view of what's running in each environment, so future compatibility checks are accurate. The entire flow — publish, verify, check, deploy, record — creates an audit trail of every service version that ever ran in production.

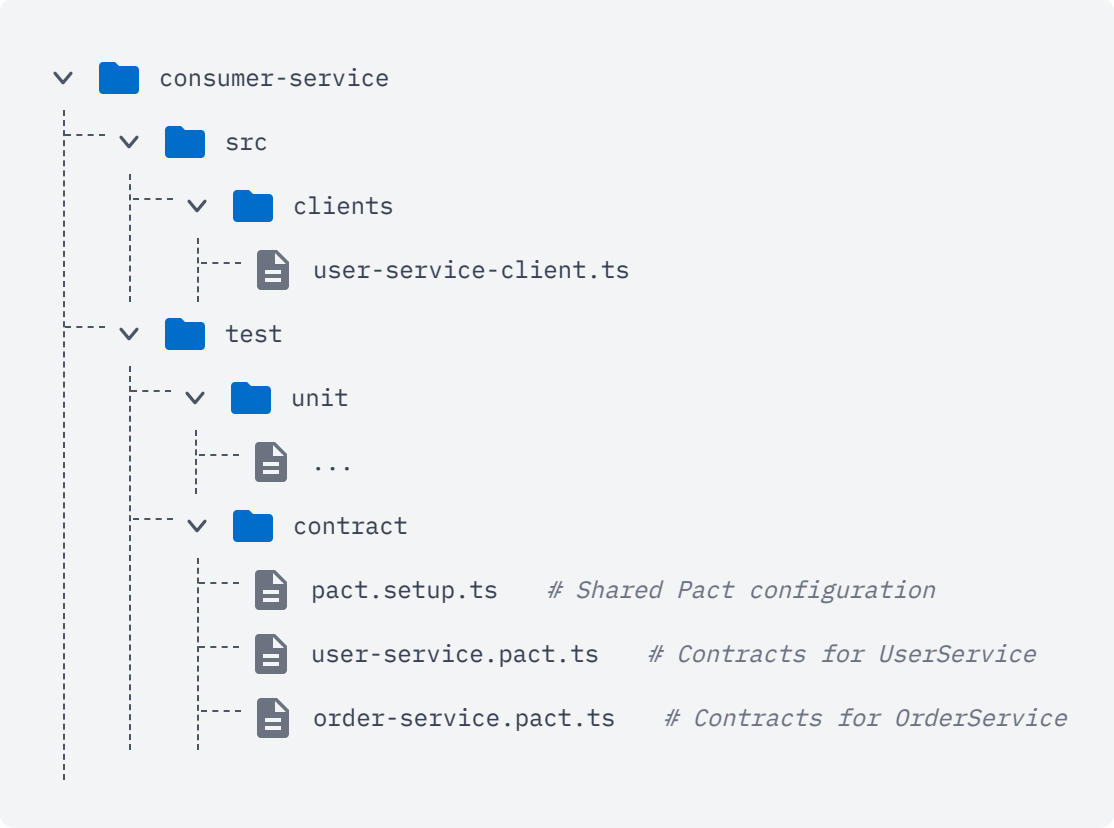
Test Organization

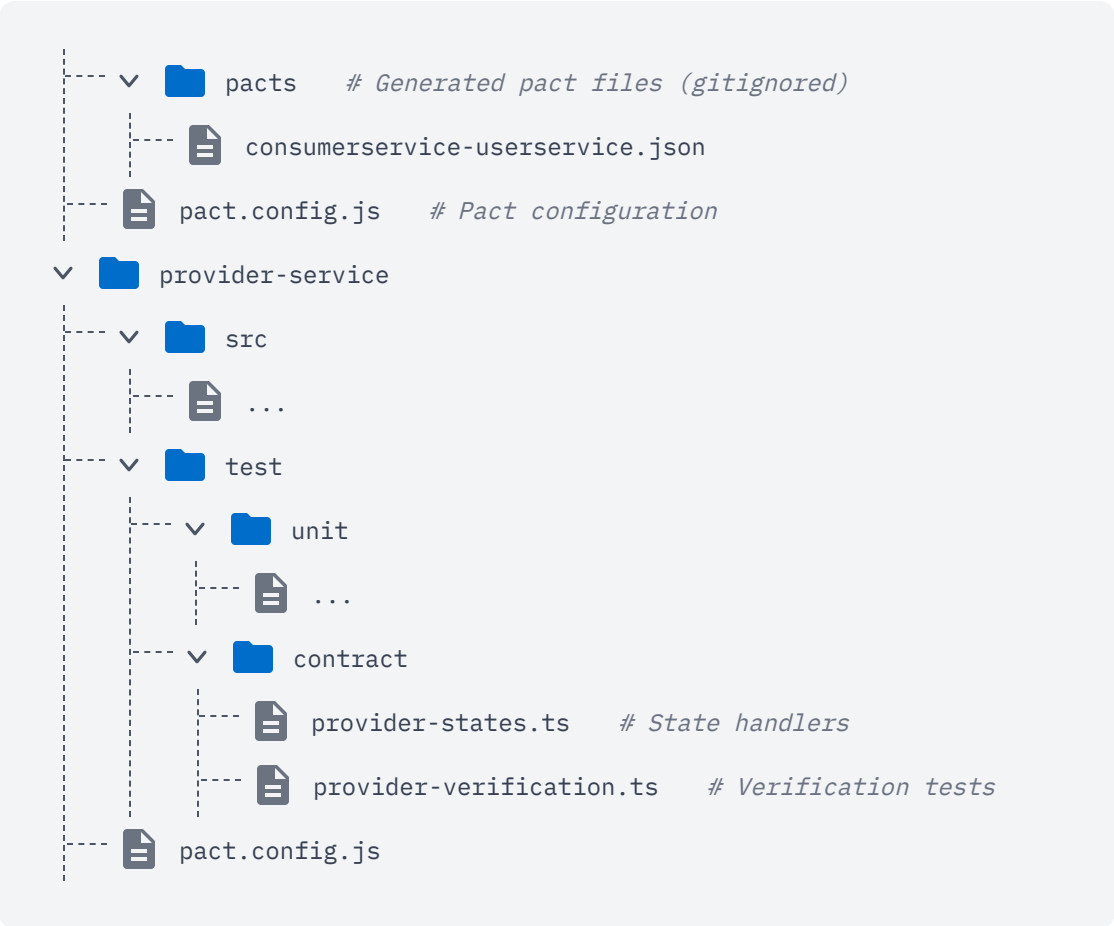
As your service count grows, so does your contract test surface. A three-service system has manageable complexity; a twenty-service system can drown in test files if you don't establish conventions early. This section covers project structure and maintainability patterns.

Structuring Contract Tests

Keep contract tests separate from unit tests — they serve different purposes and often have different dependencies. The recommended structure puts contract tests in their own directory with shared setup extracted into common files.

Project Structure:





Example project structure for consumer and provider services, showing where contract tests, provider states, Pact configuration, and generated pact files typically live.

The `pacts/` directory holds generated JSON files and should be gitignored — these are build artifacts, not source code. One contract file per provider keeps things organized: `user-service.pact.ts` contains all interactions with the User Service, regardless of how many endpoints you call.

Extract shared configuration into a setup file that all contract tests import:

pact-setup.ts

```
// test/contract/pact.setup.ts
import { PactV3 } from '@pact-foundation/pact';
import { resolve } from 'path';

export function createPactProvider(providerName: string): PactV3 {
```

```
    return new PactV3({
      consumer: process.env.PACT_CONSUMER || 'OrderService',
      provider: providerName,
      dir: resolve(__dirname, '../../pacts'),
      logLevel: process.env.PACT_LOG_LEVEL || 'warn',
    });
}

// Shared matchers for consistency
export const commonMatchers = {
  timestamp: () => datetime("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"),
  uuid: () => regex(
    /^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}$/,
    'a1b2c3d4-e5f6-7890-abcd-ef1234567890'
  ),
  email: () => regex(
    /^[\w.-]+@[\w.-]+\.\w+$/,
    'user@example.com'
  ),
};
```

Shared Pact configuration and matchers.

The `commonMatchers` object is where you'll see the biggest maintainability wins. When every contract test defines its own UUID regex, you get subtle inconsistencies. When they all import from a shared file, you change the pattern once and it propagates everywhere.

Keeping Tests Maintainable

Contract tests accumulate technical debt like any other code. These guidelines keep the debt manageable:

Guideline	Why It Matters
One interaction per test	Easier to debug; clearer failure messages
Use matchers, not literals	Contracts verify structure, not specific data
Minimal provider states	Contract tests aren't integration tests

Guideline	Why It Matters
Share common patterns	Consistency across contracts
Version contracts with code	Contract changes = code changes
Review generated pacts	Catch overly specific or missing matchers

Guidelines for maintainable contract tests.

The "one interaction per test" rule deserves emphasis. When a test file has ten interactions in a single `describe` block, failures become cryptic. When each interaction is its own test case, the failure message tells you exactly which API call broke.

> **⚠ WARNING**
>
> Contract test suites grow with your service dependencies. Without organization, you'll end up with hundreds of similar tests. Extract patterns, use shared setup, and review periodically for redundancy.

Common Pitfalls

Contract testing has a learning curve. Teams new to Pact often make the same mistakes — over-specifying contracts, ignoring error cases, or building elaborate provider states. Here's how to avoid the most common traps.

Testing Too Much

The most frequent mistake: treating contract tests like integration tests. Contract tests verify API shape, not business logic. When you hard-code exact values instead of using matchers, you create brittle tests that break whenever test data changes — even though the API contract is perfectly stable.

over-specified-contract.ts

```
// X BAD: Over-specified contract
```

```
  await provider
    .uponReceiving('get user')
    .withRequest({
      method: 'GET',
      path: '/users/123',
    })
    .willRespondWith({
      status: 200,
      body: {
        userId: '123',                    // Exact value!
        email: 'john@example.com',        // Exact value!
        name: 'John Doe',                 // Exact value!
        createdAt: '2024-01-15T10:30:00Z', // Exact value!
      },
    });
  // This test is brittle - any data change breaks it

  // ✅ GOOD: Properly specified contract
  await provider
    .uponReceiving('get user')
    .withRequest({
      method: 'GET',
      path: regex(/^\/users\/\d+$/, '/users/123'),
    })
    .willRespondWith({
      status: 200,
      body: {
        userId: like('123'),
        email: regex(/^[\w.-]+@[\w.-]+\.\w+$/, 'john@example.com'),
        name: like('John Doe'),
        createdAt: datetime("yyyy-MM-dd'T'HH:mm:ss'Z'"),
      },
    });
  // This test verifies structure, not specific data
```

Over-specified vs properly specified contracts.

The bad example will fail if the provider returns a different user or formats the timestamp differently. The good example passes as long as the response has the right fields with the right types — which is exactly what a consumer cares about.

Missing Error Cases

Happy path tunnel vision is the second most common pitfall. Consumers parse error responses too — they display validation messages, handle 401s by redirecting to login, retry on 503s. If your contract only covers 200 responses, you're leaving error handling untested.

error-case-contracts.ts

```
// Don't just test happy paths!
// Consumers depend on error response shapes too
describe('Error contracts', () => {
  it('handles 400 Bad Request', async () => {
    await provider
      .given('invalid request data')
      .uponReceiving('a request with invalid data')
      .withRequest({
        method: 'POST',
        path: '/users',
        body: { email: 'not-an-email' },
      })
      .willRespondWith({
        status: 400,
        body: {
          error: like('Validation failed'),
          details: eachLike({
            field: like('email'),
            message: like('Invalid email format'),
          }),
        },
      });
  });

  // test cases for handles 401 Unauthorized, etc.
});
```

Contract tests for error responses.

At minimum, test 400 (validation errors), 401 (authentication), 404 (not found), and any error codes your consumer explicitly handles. The contract doesn't need to cover every possible error — just the ones where response shape matters to the consumer.

Provider State Complexity

Provider states should be the simplest possible setup that satisfies the contract. When state handlers grow complex — spinning up elaborate fixtures, seeding production-like data, managing cleanup — you've drifted into integration test territory.

	State Description	Problem / Implementation
❌	"production-like database with 1000 users"	Slow, brittle, not what contract tests are for
❌	"user 123 with orders 456, 789 and payment method ending 4242"	Too specific, couples consumer to provider data model
✅	"a user exists"	Create one user with auto-generated data
✅	"user has orders"	Create user with 2 generic orders
✅	"unauthorized"	No setup needed, just don't auth

Provider state complexity pitfalls.

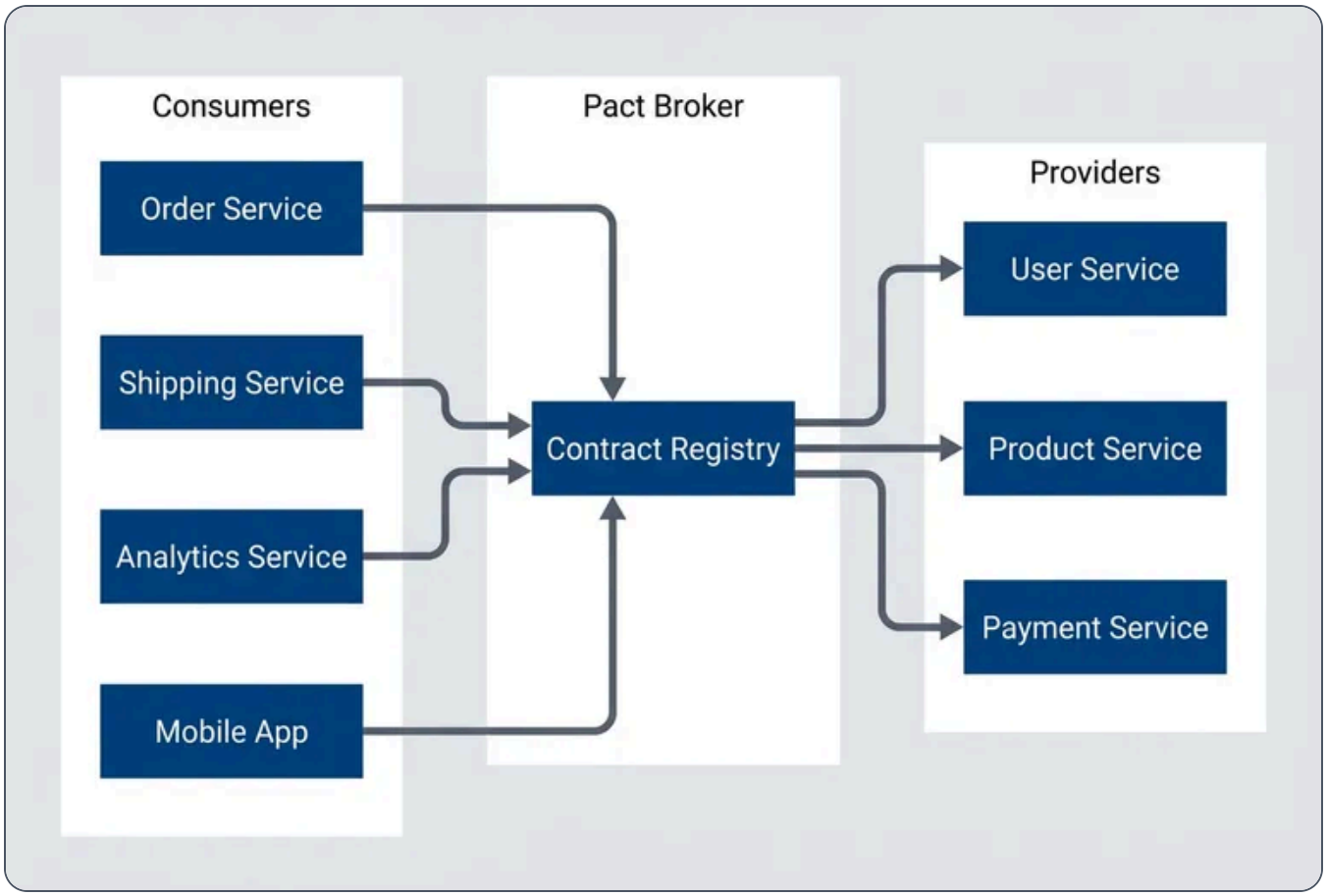
If your state handler takes more than 5 lines, question whether you're testing the right thing. Contract tests verify "given X exists, the API returns this shape." They don't verify "given a complex business scenario, the API behaves correctly"—that's what integration and E2E tests are for.

Scaling Contract Testing

Contract testing shines at scale. While integration test complexity grows quadratically with service count (every service potentially talks to every other service), contract test complexity grows linearly (each service maintains contracts with its direct dependencies). But scaling requires deliberate strategies.

Managing Many Contracts

The Pact Broker becomes your central nervous system for contract management. It tracks every contract, every verification, and every deployment across your entire service mesh.



Contract testing at scale with Pact Broker as central registry

As services multiply, you'll need these scaling strategies:

#	Strategy	What It Does
1	Environment management	`record-deployment` tracks which versions run where, so `can-i-deploy` knows what to check against
2	Webhook triggers	Auto-trigger provider verification when consumers publish new contracts — no waiting for scheduled builds
3	Pending pacts	New contracts don't fail provider builds until providers explicitly support them
4	WIP pacts	Include work-in-progress contracts in verification for early feedback

#	Strategy	What It Does
5	Matrix view	Pact Broker UI visualizes compatibility across all services at a glance

Strategies for scaling contract testing.

The pending pacts feature deserves special attention. Without it, adding a new consumer interaction immediately breaks the provider build — even before the provider team knows about the change. With pending pacts enabled, new contracts are "pending" until the provider verifies them at least once. This gives provider teams time to implement support without blocking consumer development.

> **INFO**
>
> Pending pacts prevent the "chicken and egg" problem: new consumers can publish contracts without breaking provider builds, while providers gradually add support.

Beyond HTTP

Contract testing isn't limited to REST APIs. Pact supports gRPC and asynchronous messaging, covering the full spectrum of service communication patterns.

gRPC Contract Testing

For gRPC services, Pact V4 uses a plugin system to handle Protocol Buffers. The contract captures the same consumer expectations — just expressed in terms of protobuf messages rather than JSON.

grpc-contract-test.ts

```
import { PactV4 } from '@pact-foundation/pact';

const provider = new PactV4({
  consumer: 'OrderService',
  provider: 'UserService',
  pluginDir: './pact-plugins',
```

```
});

describe('UserService gRPC Contract', () => {
  it('returns user for GetUser', async () => {
    await provider
      .usingPlugin('protobuf')
      .given('user 123 exists')
      .uponReceiving('GetUser request')
      .withPluginContents({
        'pact:proto': './protos/user_service.proto',
        'pact:content-type': 'application/grpc',
        request: { user_id: matching(type, '123') },
        response: {
          user_id: matching(type, '123'),
          email: matching(regex, /^.+@.+$/, 'user@example.com'),
        },
      })
      .executeTest(async (mockServer) => {
        const client = new UserServiceClient(mockServer.url);
        const user = await client.getUser('123');
        expect(user.email).toContain('@');
      });
  });
});
```

gRPC contract testing with Pact V4 plugin.

Event-Driven Contracts

For message queues and event buses, Pact's `MessageConsumerPact` tests asynchronous contracts. The consumer defines what message shape it expects; the provider verifies it publishes messages matching that shape.

message-contract-test.ts

```
import { MessageConsumerPact, synchronousBodyHandler } from '@pact-foundation/pact';

const messagePact = new MessageConsumerPact({
  consumer: 'OrderService',
```

```
    provider: 'UserService',
    dir: './pacts',
  });

  describe('User Created Event Contract', () => {
    it('handles user.created event', async () => {
      await messagePact
        .given('a new user is created')
        .expectsToReceive('a user.created event')
        .withContent({
          eventType: 'user.created',
          data: {
            userId: like('123'),
            email: like('new@example.com'),
            timestamp: datetime("yyyy-MM-dd'T'HH:mm:ss'Z'"),
          },
        })
        .verify(synchronousBodyHandler(async (message) => {
          const handler = new UserCreatedHandler();
          await handler.handle(message);
        }));
    });
  });
```

Asynchronous message contract testing.

The same principles apply: consumers define expectations, providers verify they can deliver, and the Pact Broker tracks compatibility. Whether your services communicate via HTTP, gRPC, or message queues, contract testing catches breaking changes before deployment.

Conclusion

Consumer-driven contract testing solves the coordination problem that plagues microservice architectures. Instead of discovering API incompatibilities in staging (or worse, production), you catch them at PR time — before code merges, before deployment, before the Friday afternoon incident.

The key insights:

- **Contracts capture what consumers actually use**
  not what providers think they provide. This focus on real dependencies eliminates false positives from unused fields.
- **The Pact Broker is your safety net.**
  The `can-i-deploy` command answers "will this deployment break anything?" with certainty, not hope.
- **Contract tests verify structure, not business logic.**
  Use matchers to specify types and patterns; leave exact values to integration tests.
- **The investment scales well.**
  While integration test complexity grows quadratically with service count, contract testing grows linearly.

Getting started is straightforward:

> **INFO**
>
> **Getting Started Checklist**
>
> - [x] Pick one consumer-provider pair with a stable, well-understood integration
> - [x] Write 2-3 consumer contract tests covering your most critical interactions
> - [x] Deploy Pact Broker (Docker Compose for dev, Kubernetes for production)
> - [x] Add provider verification tests with state handlers
> - [x] Wire both into CI: publish on consumer builds, verify on provider builds
> - [x] Add `can-i-deploy` checks before deployments
> - [x] Expand to additional service pairs

Once that first pair is working, expanding to other services follows the same pattern.

> ✓ SUCCESS
>
> Contract tests are fast, reliable, and catch breaking changes at PR time. They don't replace integration tests entirely, but they eliminate the majority of "works on my machine, breaks in production" API compatibility issues.

The team that renamed `userId` to `user_id` would have caught that breaking change in their PR checks. The deployment would have failed `can-i-deploy`, the compatibility matrix would have shown exactly which consumers depended on the old field name, and Friday afternoon would have stayed quiet.

https://www.webstackbuilders.com/articles/consumer-driven-contract-testing-pact-internal-apis