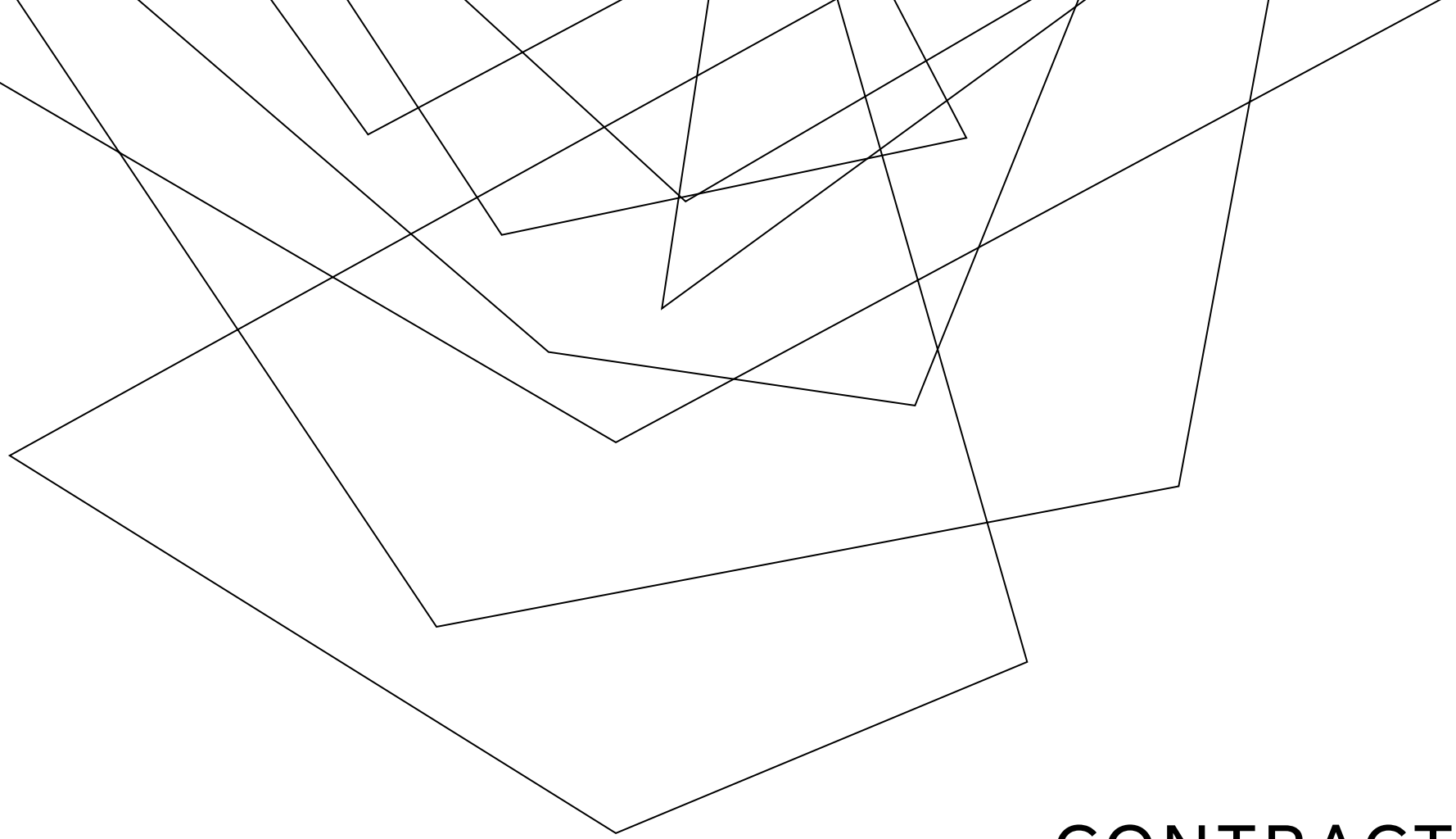
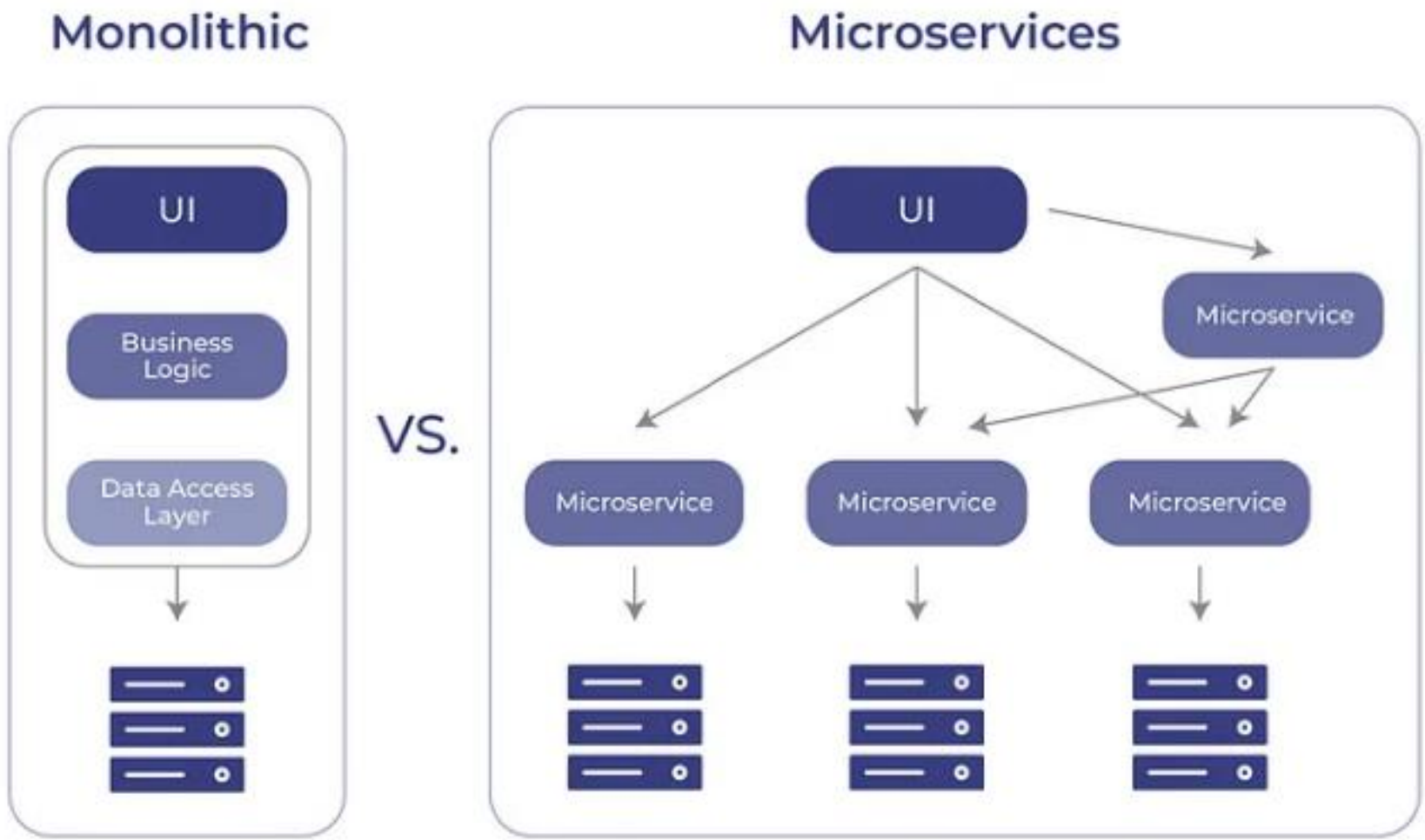


CONTRACT TESTING

In today's rapid development environment, deployments have shifted from occurring every few days **to happening every few seconds. This is especially true in microservices architectures, where applications consist of numerous interconnected services.**

Frequent changes in one microservice can significantly impact others, making it crucial to identify issues early. Addressing these challenges during the code development phase is essential to avoid costly problems after deployment.

In this presentation, we will explore effective strategies for managing these complexities.


Monolithic
UI
Business Logic
Data Access Layer
VS.
Microservices
UI
Microservice
Microservice
Microservice
Microservice

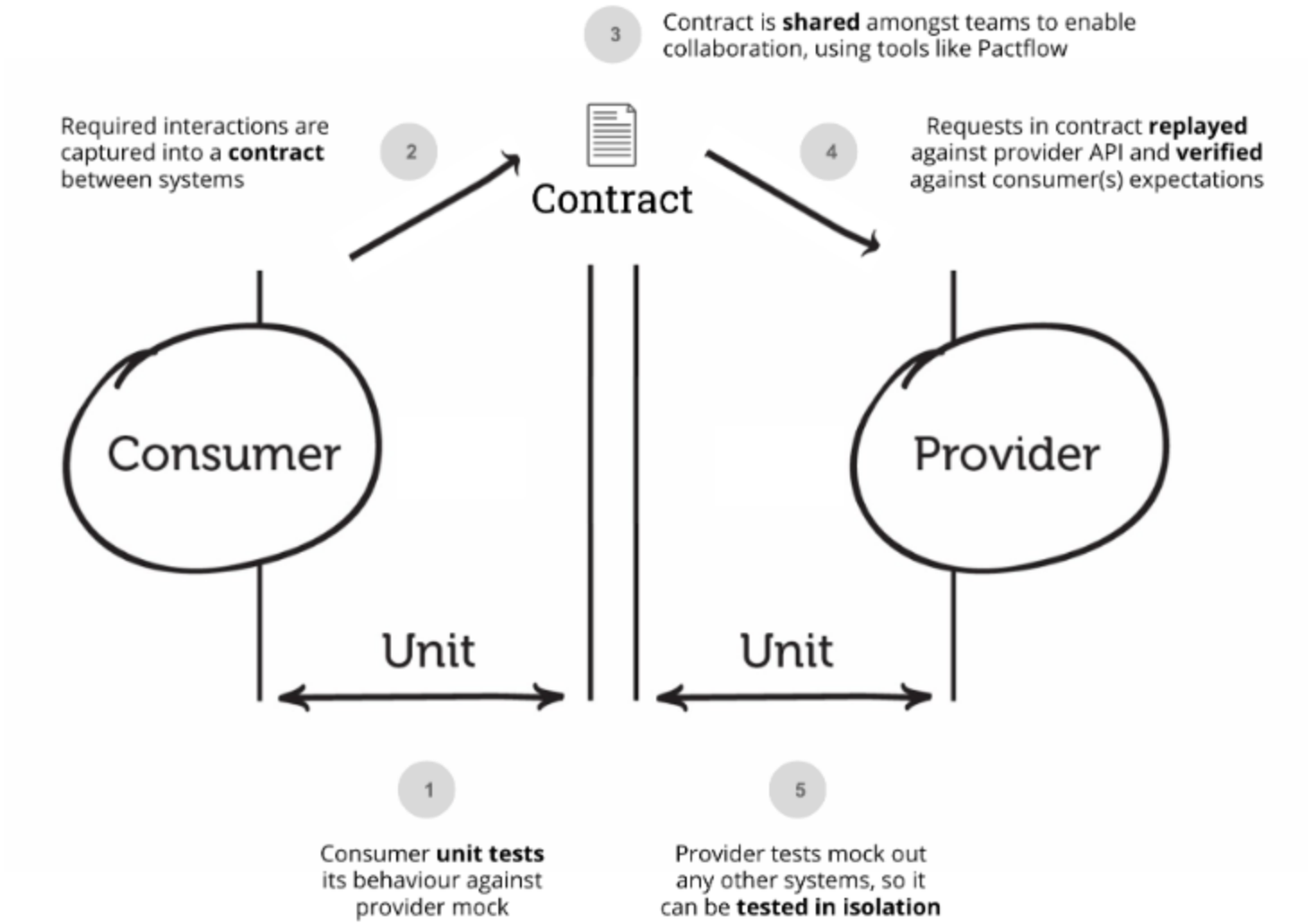
CHALLENGES OF INTEGRATION TESTING (END-TO-END TESTING) ON MICROSERVICES

- **Complexity of Dependencies**: Microservices rely on other services, databases, and external APIs, making it hard to test interactions effectively.
- **Data Management**: Each microservice may have its own database, requiring careful setup of test data and handling synchronization issues.
- **Service Isolation**: Testing services independently while simulating dependent services requires mocking or stubbing many interactions.
- **Environment Setup**: Replicating a production-like environment with all necessary services can be resource-intensive and time-consuming.
- **Test Flakiness**: Network issues, timeouts, and service unavailability can lead to intermittent test failures.
- **Scalability**: As the number of microservices grows, it becomes difficult to maintain and scale a comprehensive end-to-end test suite.
- **Test Execution Time**: Full end-to-end tests across multiple services take time, slowing down the development cycle.

WHAT IS CONTRACT TESTING?

Contract testing is a methodology for ensuring that two separate systems (such as two microservices) are compatible and can communicate with one other. It captures the interactions that are exchanged between each service, storing them in a contract, which then can be used to verify that both parties adhere to it. requiring both parties to come to a consensus on the allowed set of interactions and allowing for evolution over time.

What sets this form of testing apart from other approaches that aim to achieve the same thing is that each system can be tested independently from the other and that the contract is generated by the code itself, meaning the contract is always kept up to date.



how contract testing works

WHY CONTRACT TESTING EXISTS?

It exists to help with integration testing – the process by which we build confidence that a system works as a whole.

In a distributed system, integration testing is a process that helps us validate that the various moving parts that communicate remotely – things like microservices, web applications, and mobile applications – all work together cohesively.

There are many types of integration testing, but the most common approach is called "**end-to-end integrated testing**," which involves all the components being deployed together in a real environment – one that closely resembles production – and running a battery of test scenarios against it.

PROS OF CONTRACT TESTS

Contract tests generally have the opposite properties to integrated e2e tests.

- They run fast, because they don't need to talk to multiple systems.
- They are easier to maintain. You don’t need to understand the entire ecosystem to write your tests.
- They are easy to debug and fix, because the problem is only ever in the component your testing – so you generally get a line number or a specific API endpoint that is failing.
- They are repeatable.
- They scale; because each component can be independently tested, build pipelines don’t increase linearly/exponentially in time.
- They uncover bugs locally on developer machines. Contract tests can and should run on developer machines before pushing code.

WHAT IS PACT?

Pact is a contract testing tool used to ensure reliable communication between services in a microservices architecture. It helps verify that services (like APIs) interact correctly by defining and enforcing "contracts" between a service provider (the API) and a service consumer (another service or client).

Key points about Pact:

- **Consumer-driven contracts**: The consumer (e.g., a front-end or another service) defines the expectations for the provider (e.g., an API), creating a "contract."
- **Contract verification**: The provider must adhere to this contract, and Pact verifies this by running tests to ensure both sides are compatible.
- **Decoupled development**: Pact allows consumer and provider teams to work independently while ensuring integration points are tested.
- **Early detection of issues**: It helps catch potential integration issues early, before they reach production, by validating expectations during development and testing phases.

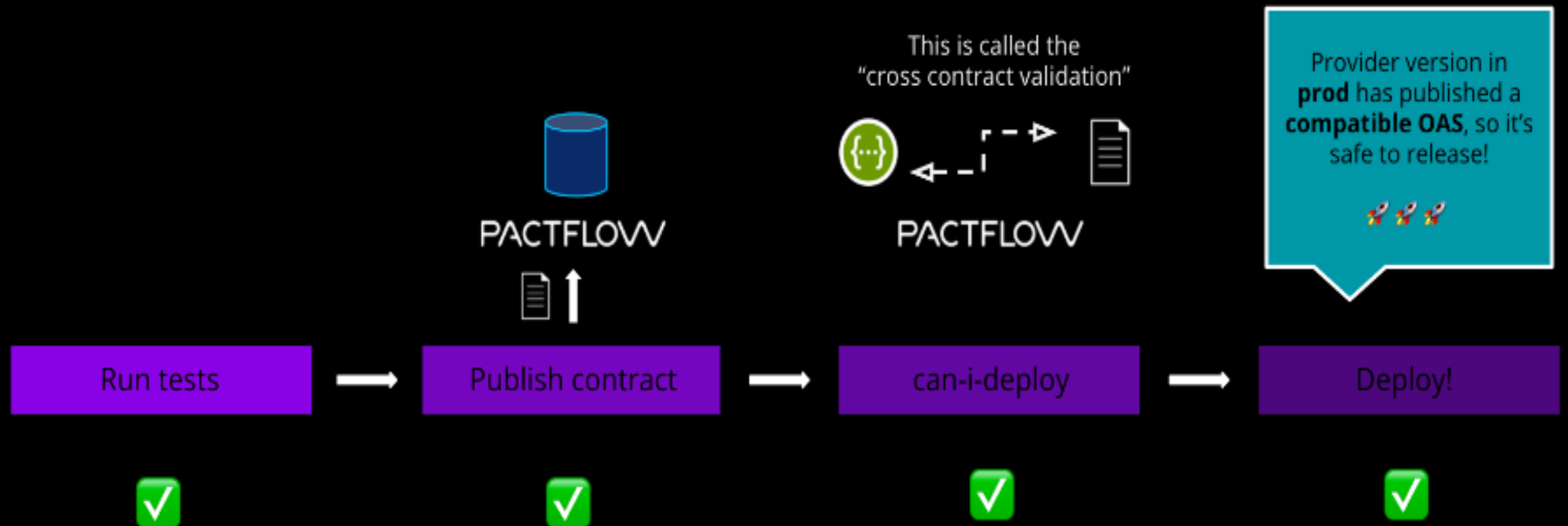
By focusing on the contract between services, Pact minimizes integration failures and makes testing in distributed architectures more efficient.

PACT FLOW

- Two services, a consumer and a provider, are interacting via a REST API, with the expectation of receiving a specific status code and response.
- Use Pact as a mock provider to avoid directly interacting with the real service
- Write tests on the consumer side, mocking the expected data from the provider. Cover all scenarios, specifying the expected status and response values.
- Pact automatically generates a contract file (JSON) with all the interactions from the test runs. and uploads it to the PACT Broker.
- The provider consumes this contract file from the Pact Broker. All interactions are executed on the actual provider service, and the result is compared with the expected outcome.

This approach ensures that any changes made by the provider to their API are immediately flagged during development. Issues are highlighted, indicating that local changes may break the consumer, as the contract test has failed.

Consumer pipeline: when provider is in production



PACTFLOW

Provider pipeline

First run