



# Verifying Microservices Integrations



# TABLE OF CONTENTS

---

Executive Summary	3
Challenges with testing Microservices	4
The Modified Pyramid Approach	10
How Contract testing is the savior?	11
Two Approaches that you should not follow	14
Testing Microservices the HyperTest Way <ul style="list-style-type: none"><li>• <i>Benefits of testing microservices with HyperTest</i></li></ul>	15
Soundcloud's Ultimate Rescuer: <i>Contract tests</i>	20
Conclusion	21

## Executive Summary

As the world becomes increasingly digital, organisations are turning to microservices architecture to stay ahead of the curve. But with this newfound agility comes new challenges in the realm of testing. From complexity and inter-service dependencies to limited testing tools, the microservices landscape can be a complex and daunting.

However, with the right approach, testing microservices can be made simple, fast and scalable driving business success. This whitepaper will guide you on how to test microservices. By exploring the challenges and the corresponding solutions to them, we provide a roadmap for organizations seeking to harness the power of microservices while ensuring the stability and reliability of their systems.

We delve into the importance of incorporating testing at every stage of the development lifecycle, and leverage modern tools and techniques to help you navigate the complexities of testing multi-repo systems.

This guide talks about the ultimate approach to overcome the challenge of testing microservices, empowering you to stay ahead of the curve and achieve success in the digital age.

## Challenges of Testing Microservices

A microservices architecture is made up of separate services, each with its own data storage and deployment, testing becomes harder as the number of independent pieces increase.



**In a recent survey by TechBeacon, 37% of organizations said that testing was the hardest part of putting together their microservices.**

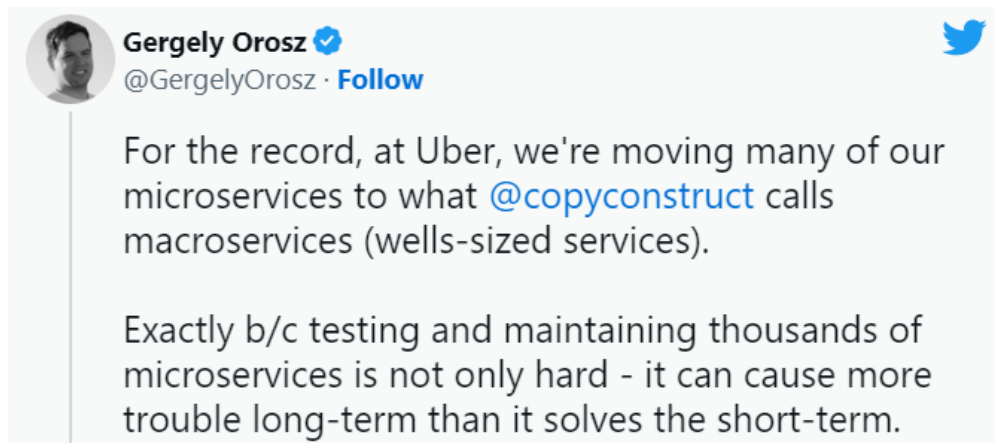
Since rapid development is inherent to microservices, teams must test each service separately and in conjunction, to evaluate the overall stability and quality of such distributed systems.

The broader the scope of a test, the more challenging it becomes to create, run, and maintain them. Agile teams worship speed, don't want release velocity to be slowed down by their testing approach.

Teams from companies like MAANG have already gone through the journey of building and implementing the right approach to test and maintain a multi-repo system. This paper brings the best practices from those approaches so that your teams can implement it without delay.



The ride-hailing service, **Uber**, also started out with a monolithic structure. Over time, it broke that monolith into more than 2000 micro-services. The transition had mixed repercussions for Uber.

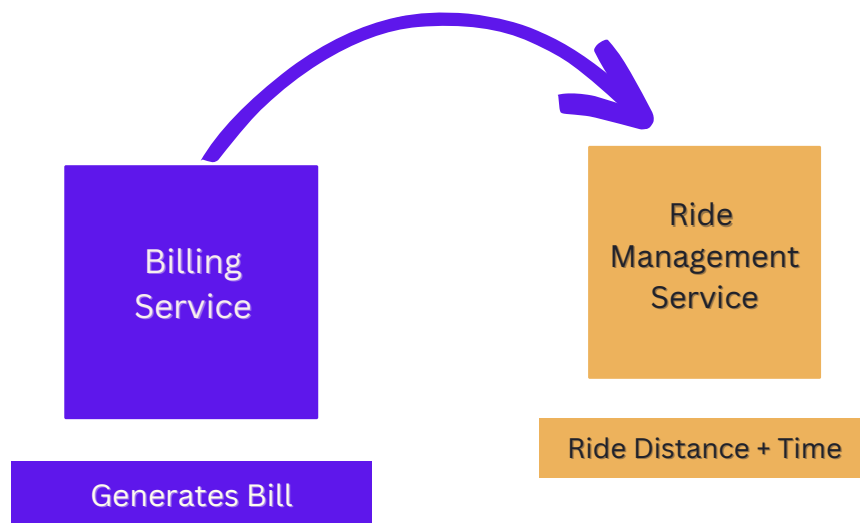


Implementing micro-services the right way is a lot of hard work, and testing adds to that challenge, because of their sheer size and complexity. Let's understand from Uber's perspective the challenges they had with testing their multi-repo system.

## Inter-service Dependency

Each individual service is dependent on another for their proper functioning. More the services, higher is the number of inter-service communications that might fail.

In this complex web of inter-service communications, a breakdown in any of the services has a cascading effect on all others dependent on it



Creates an inter-dependency where Billing service can't function without Ride Management Service.

Calls between services can go through many layers, making it hard to understand how they depend on each other. If the nth dependency has a latency spike, it can cause a chain of problems further upstream.

## Finding the root cause of failure

When multiple services talk to each other, a failure can show up in any service but the cause of that problem can originate from a different service deep down.

Doing RCA for the failure becomes extremely tedious, time-consuming and high effort for teams of these distributed systems



*For instance, engineers had to work through around 50 services across 12 different teams in order to investigate the root cause of the problem.*



Uber has over 2200 microservices; in its web of interconnected services; if one service fails, all upstream services suffer the consequences. The more the services, the more difficult it is to find the one that originated the problem.



## Unexpected Functional Changes

Uber decided to move to a distributed code base to break down application logic into several small repos that can be built and deployed with speed.

Though, this gave teams the flexibility to make frequent new changes, but at the same time increased the speed at which new failure are introduced.

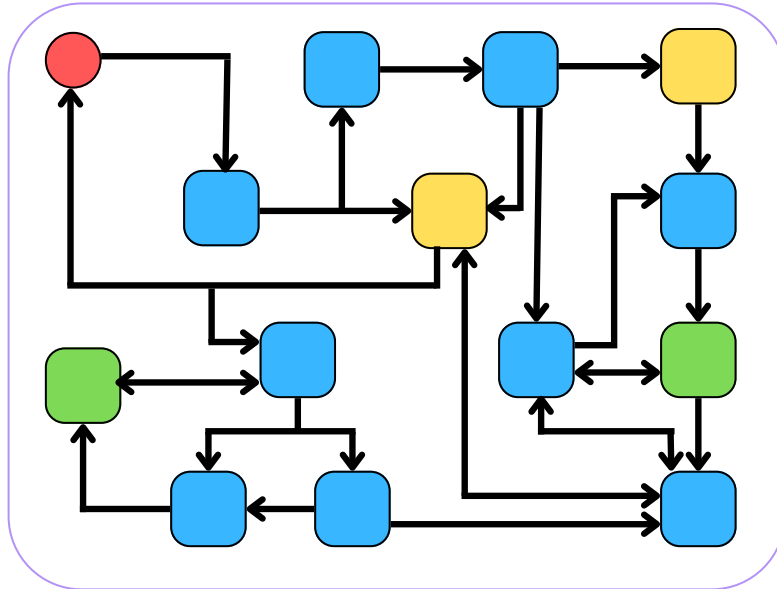
“

A study by Dimensional Research found that the average cost of an hour of downtime for an enterprise is \$300,000, highlighting the importance of minimizing unexpected functionality changes in microservices.

”

So these rapid and continuous code changes, makes multi-repo systems more vulnerable to unintended breaking failures like latency, data manipulation etc.

## Difficulty in Localizing the Issue

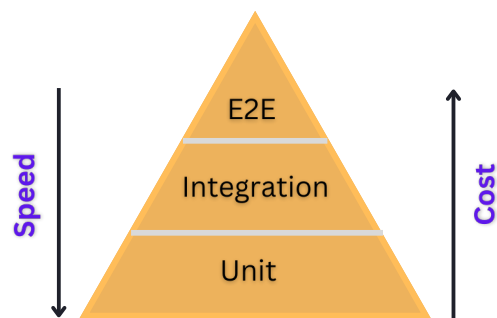


Each service is autonomous but when it breaks, the failure it triggers can propagate down and far, with damaging effects.

This means the failure can show up elsewhere but the trigger could be several services upstream. Hence, identifying and localizing the issue is very tedious, sometimes impossible without the right tools.



## Modified Test Pyramid for testing Micro-services



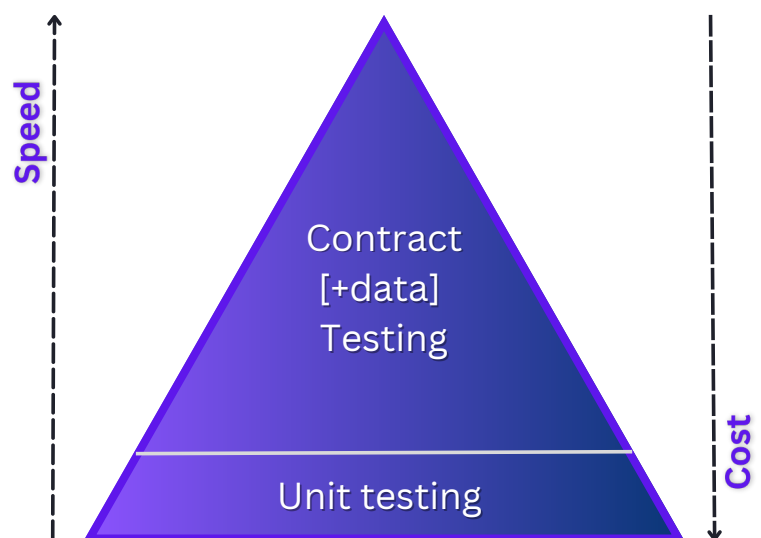
**Mike Cohn's Test Pyramid**

The traditional "Test Pyramid" suggests balancing unit, integration, and end-to-end tests. Unit tests are focused on individual components. Integration tests test how multiple components work together, and end-to-end tests test the entire system from a user's perspective.

This test pyramid approach needs to be modified for testing microservices. E2E tests need to be completely dropped. Apart from taking a long time to build and maintain, E2E tests execute complete user-flows every time on the entire application, with every test.

This requires all services under the hood to be simultaneously brought up (including upstream) even when it is possible to catch the same kind and the same number of failures by testing only a selected group of services; only the ones that have undergone a change.

This approach of selecting and testing only a single service at a time is faster, cheaper and more effective, and can be easily achieved by **testing contracts [+data] for each service independently**.



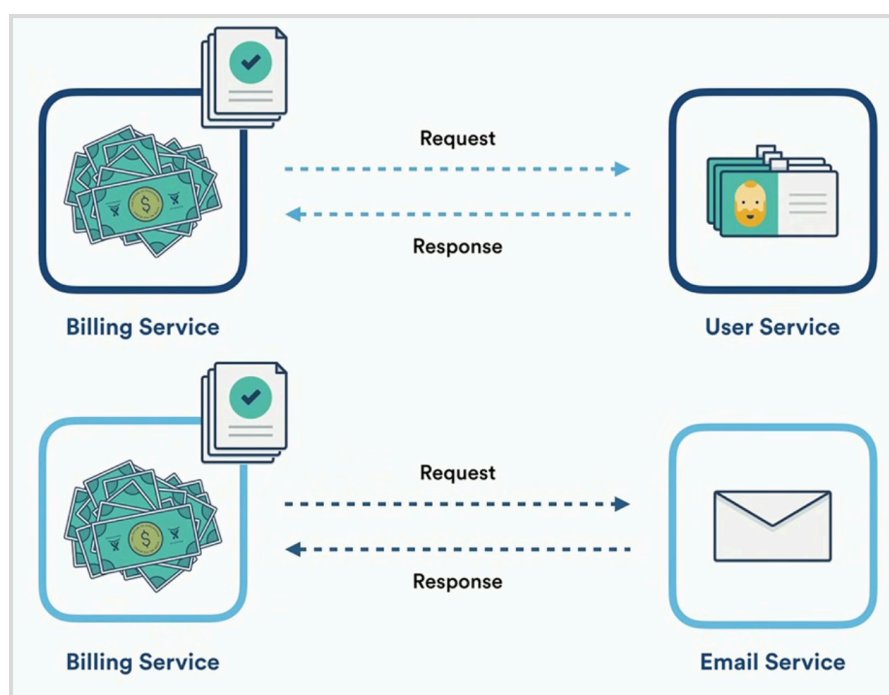
**Modified Pyramid for testing complex Microservices**

## How Contract testing is the savior?

Microservices have a consumer-provider relationship between them. In a consumer-provider, one microservice (the consumer) relies on another microservice (the provider) to perform a specific task or provide a specific piece of data.

The consumer and provider communicate with each other over a network, typically using a well-defined application programming interface (API) to exchange information.

This means the consumer service could break irreversibly if the downstream service (provider) changes its response that the consumer is dependent on.



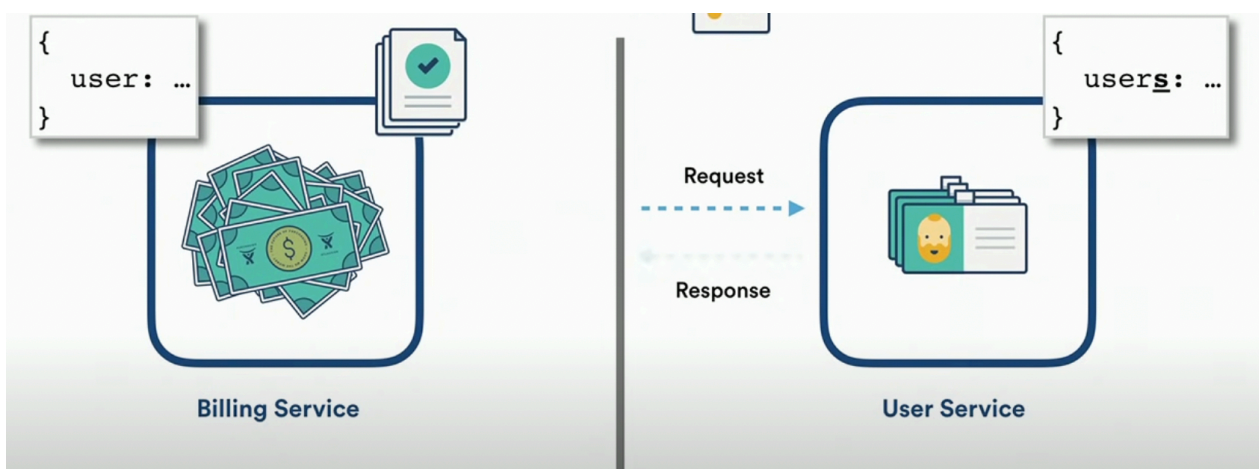


But why do these failures happen? When the provider service:

- breaks its user contract that changes its response schema
- changes the data even when the contracts stays intact

Consider an application that has two services in this relationship i.e. Billing service that asks the user service for details every-time it creates an invoice for payment.

The billing service send the request, the user service invokes *<Getuser>* method and send all the details back as response to the Billing Service (consumer).



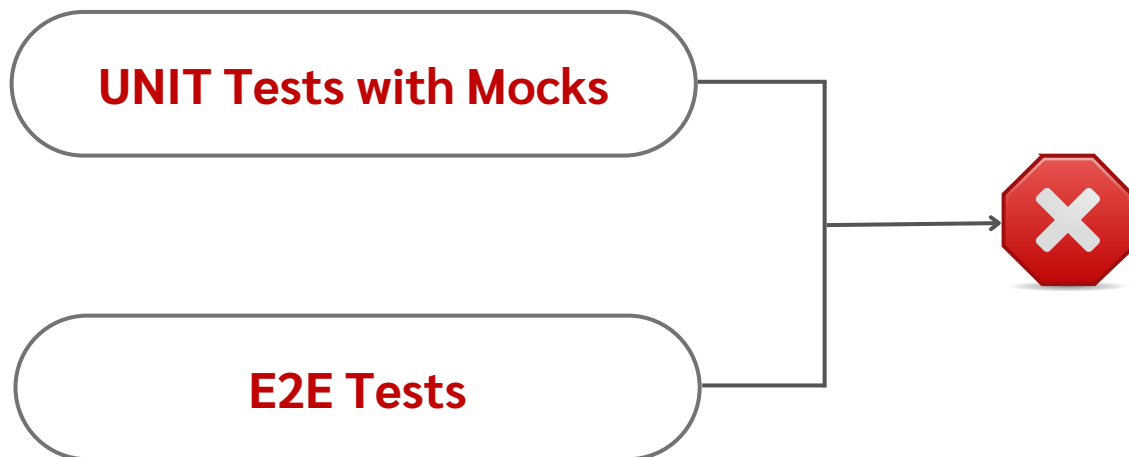
An example of a breaking change for the billing service is when the User service updates or changes name of the id from User to Users. This looks like an innocuous, small change but breaking the user contracts breaks the response for the consumer service triggering a catastrophic failure.


So any change in API contract (or the data) in the provider / downstream service can break the dependent service (here Billing).

The billing service could worse break in production if it is unable to handle the new response by the user service, crashing in front of the users.



## Two approaches that you **should NOT** consider



 **UNIT tests with mocks:** Mocks are not trustworthy, specially that devs write themselves. Static mocks that are not updated to account for changing responses could still miss the error in our example because;

- they don't test the SUT with the dependencies
- the more you mock, the less you can trust the results

 **E2E tests: Extremely difficult to write, maintain and update.**

An E2E test that actually invokes the inter service communication like a real user would catch this issue. But cost of catching this issue with a test that could involve many services would be very high, given the time and effort spent creating it.

- imprecise because they've such a broad scope
- needs the entire system up & running, making it slower and difficult to identify the error initiation point

# Testing Microservices the HyperTest Way

Integration tests that tests contracts [+data]:

## 1 Testing each service individually for Contracts:

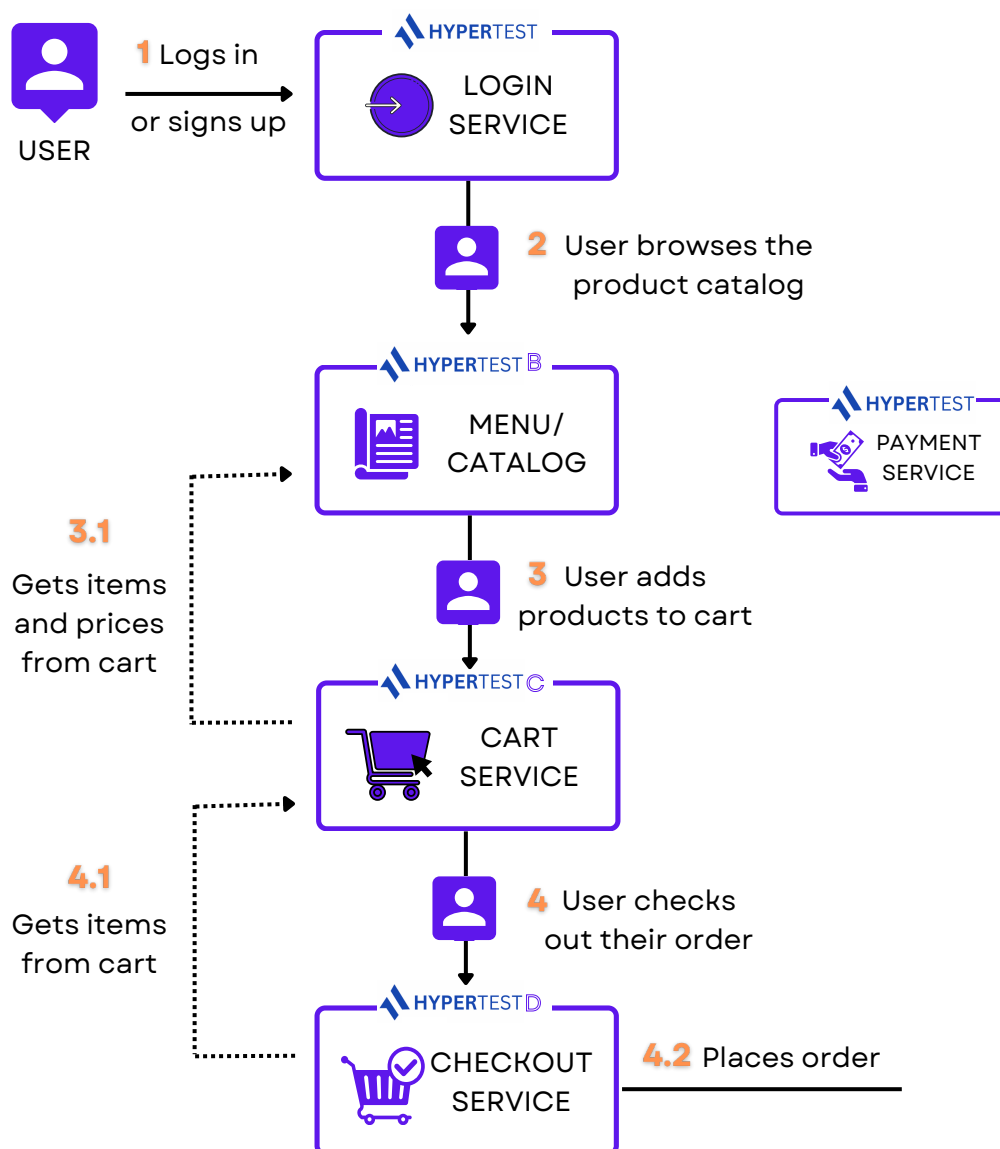
In our example the consumer service can be saved from failure using simple contracts tests that mock all dependencies like down-streams and db for the consumer

- **Verifying (testing) integrations** between consumer and provider by mocking each other i.e. mocking the response of the provider when testing the consumer, and similarly when testing the provider mocking of the outgoing requests from the consumer.
- **But changing request / response schema** makes the mocks of either of the services update real-time, making their contract tests valid and reliable for every run.
- **This service level isolation** helps test every service without needing others up and running at the same time.

Service level contract tests are much simple to maintain than E2E tests, only when respective mocks are smartly updated for every service under test

## 2 Build integration tests for every service using network traffic

If teams find it difficult to build tests that generate response from a service with pre-defined inputs, there is a simple way to test services one at a time using HyperTest Record and Replay mode.

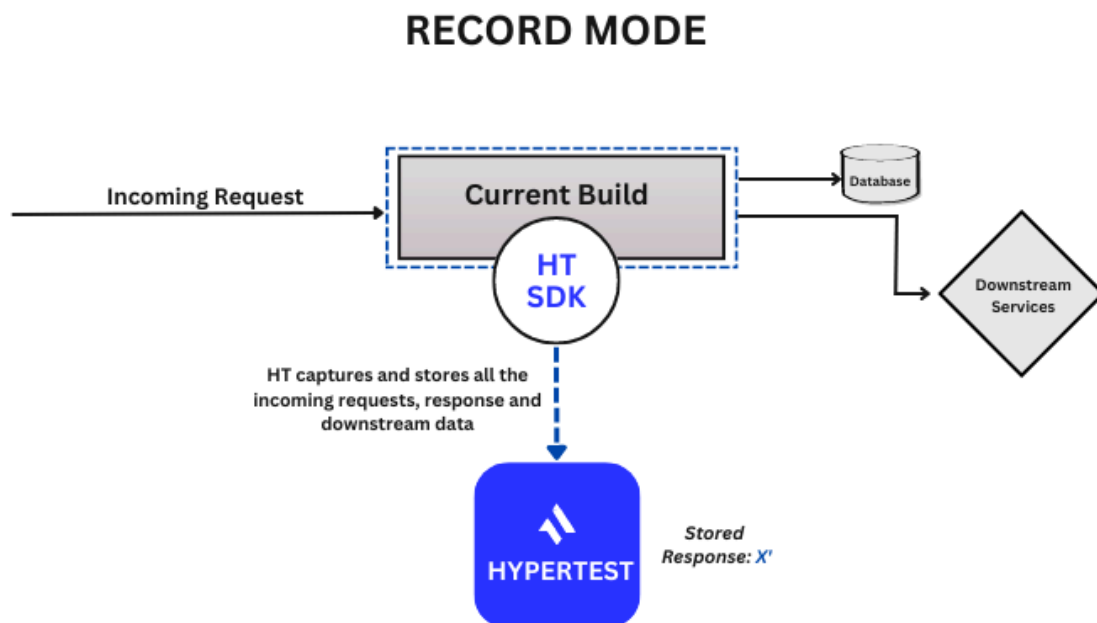


Test service integrations with smart record and replay of HyperTest

If teams want to test integration between services, HyperTest sits on top of each service and monitors all the incoming traffic for the service under test [SUT].

Like in our example, HyperTest will capture all the incoming requests, responses and downstream data for the service under test (SUT). This is Record mode of HyperTest.

This happens 24x7 and helps HyperTest build context of the possible API requests or inputs that can be made to the service under test i.e. user service.

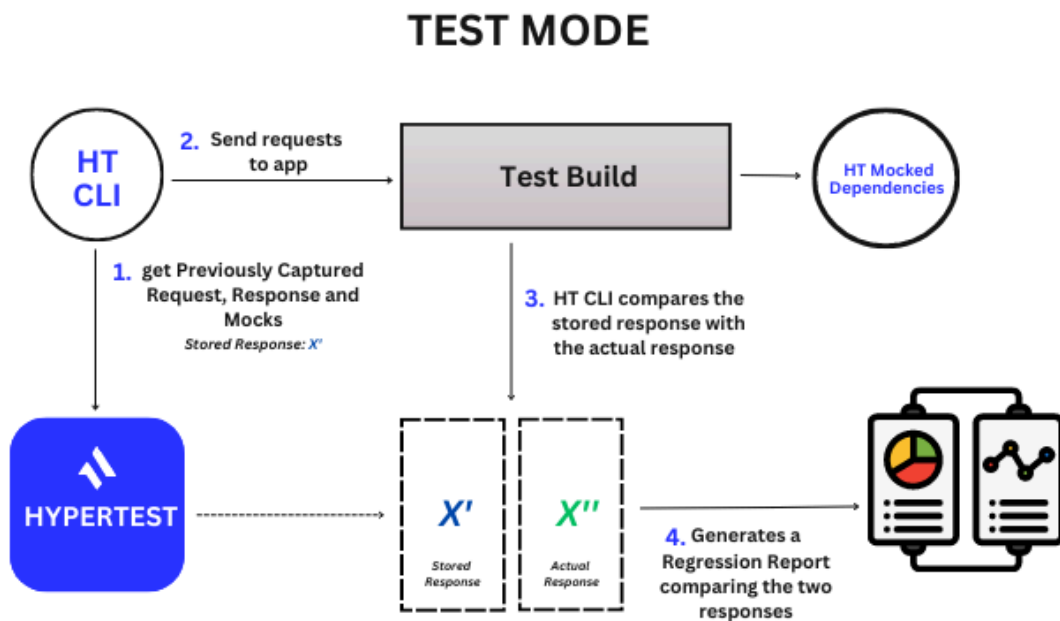




HyperTest then tests the SUT by replaying all the requests it captured using its CLI in the Test Mode.

These requests that are replayed have their downstream and database calls mocked (captured during the record mode). The response so generated for the SUT ( $X''$ ) is then compared with the response captured in the Record Mode ( $X'$ ).

Once these responses are compared, any deviation is reported as regression. A HyperTest SDK sitting on the down stream updates the mocks of the SUT, with its changing response eliminating the problem of static mocks that misses failures.



HyperTest updates all mocks for the SUT regularly by monitoring the changing response of the down streams / dependent services

## Benefits of testing Microservices with HyperTest

1

Service level contract tests are easy to build and maintain. **HyperTest builds or generates these tests in a completely autonomous way**

2

The provider can make changes to their APIs **without breaking upstream services**

3

Reduces the need for developers to talk to each other and coordinate, **saving time and unnecessary communication**

4

HyperTest localizes the root cause of the breaking change to the right service very quickly, **saving debugging time**

5

**Very easy to execute**, since contract[+data] tests can be triggered from the CI/CD pipelines

## SoundCloud's Ultimate Rescuer: Contract testing

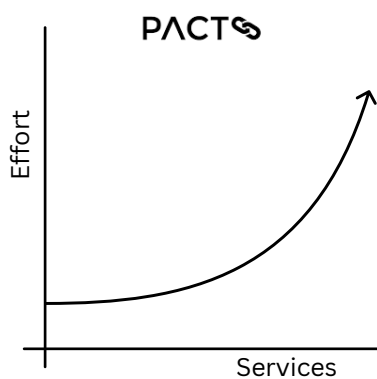
When SoundCloud, which is the leading music streaming service in US, made a shift from mono to micro, it allowed their teams to dramatically increase their release velocity. However, they learned quickly that if a service's API were to be changed in a way that broke existing code, it may cause a chain reaction of failures.



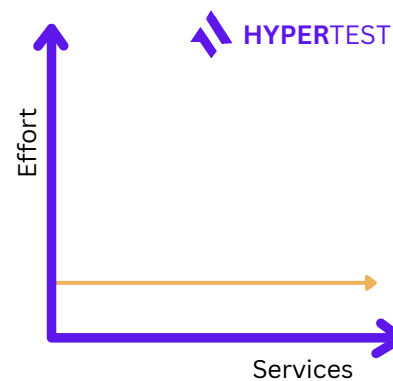
**To mitigate this risk, SoundCloud implemented consumer driven contract tests.**

With these tests, consumer services could define their interactions and expectations with provider services, and providers verified these contracts every time the service was built.

With over 300 services in place, SoundCloud took benefit of PACT based contract testing to avoid breakage and maintain a stable and reliable system.



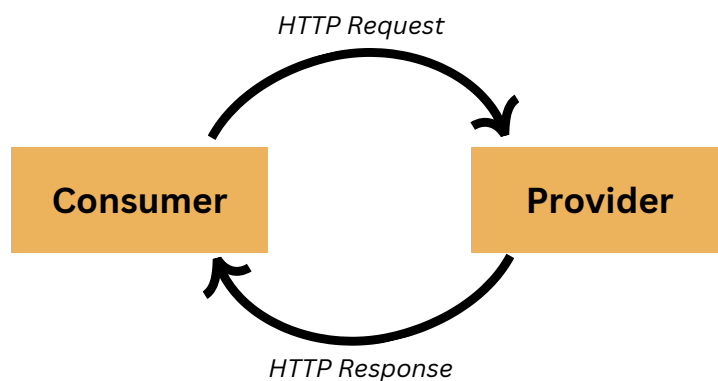
With PACT contract testing, the effort required to test your services will gradually increase as you expand PACT to more services.



With HyperTest, the effort required to test your services will always be same, whether you have 5 services or 500.

## Conclusion

As teams look to release fast and on-demand, devs find it harder to focus on either unit test coverage or E2E tests. Since one has a narrower scope while the other is slower and high on maintenance.



Contract [+data] tests are-the optimal solution to test distributed systems. These service level contract tests are simple to build and easy to maintain, keep the microservices in a '**releasable**' state.

This strategy produces the right results without the need to invest in expensive teams or test suites. Additionally, they can be integrated with the CI / CD pipelines, sitting nicely with the needs of the release process.



“

We have recently upgraded our code framework. **And by running one instance of HyperTest, we got the first-cut errors in less than an hour**, which could have taken us a few days.

”

**Vibhor G.**  
VP - Engineering



Revamp your testing strategy with HyperTest's contract testing approach to quickly pinpoint and resolve the root cause of any failure.

Don't let bugs go unnoticed - learn more about our solution here:

<https://hypertest.co>

Request a Demo >

# Want to Unlock Full Document

Click here



 **DOWNLOAD**