# Testing in Near-Production Shifted Left

## Leverage Past User Activity to Prevent API Regressions

Venky Ganti
Rahul Lahiri

**Continuous Testing driven by DevOps & Shift-Left**

Engineers taking on responsibility of test automation

**Microservices adoption increasing…**

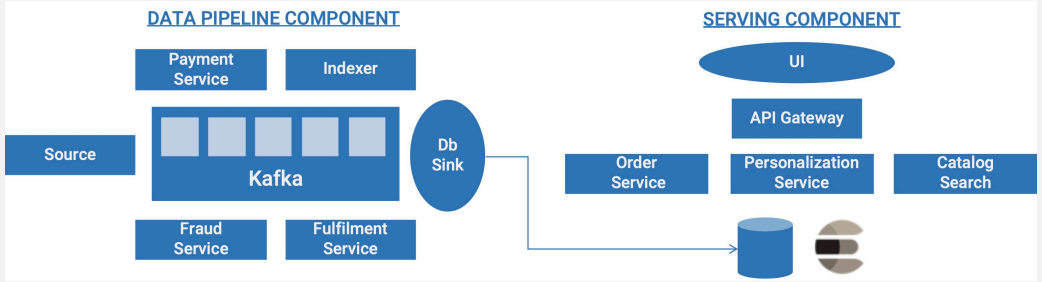
Variety of interactions need much higher functional test coverage

# Typical Testing Goals

TG1 – Preventing regressions to current usage

TG2 – Testing, debugging, and automation for new functionality

TG3 – Infrastructure testing: performance, resilience, security, …



# Current Testing Approaches

**Engineers must create automation**

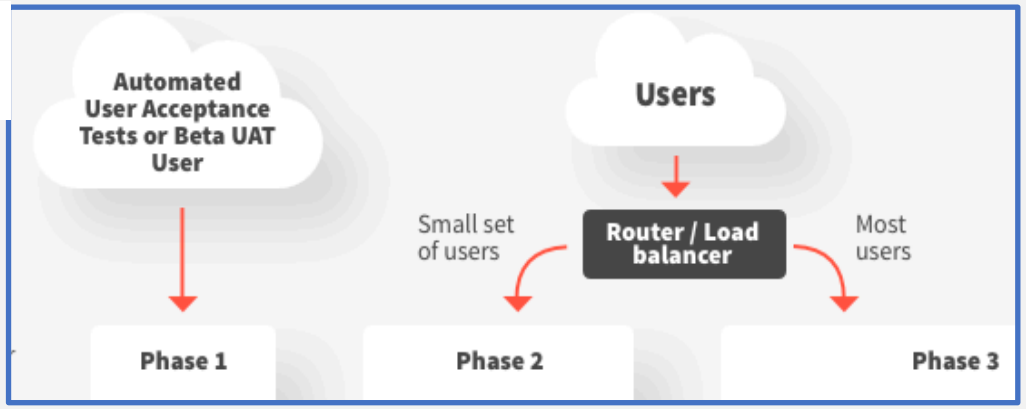
- Very demanding on engineering time (>25%)
- Limited regression test coverage due to resource constraints

- ❑ **Slows velocity**
- ❑ **Regressions in production**
- ❑ **Bugs reduce customer satisfaction**

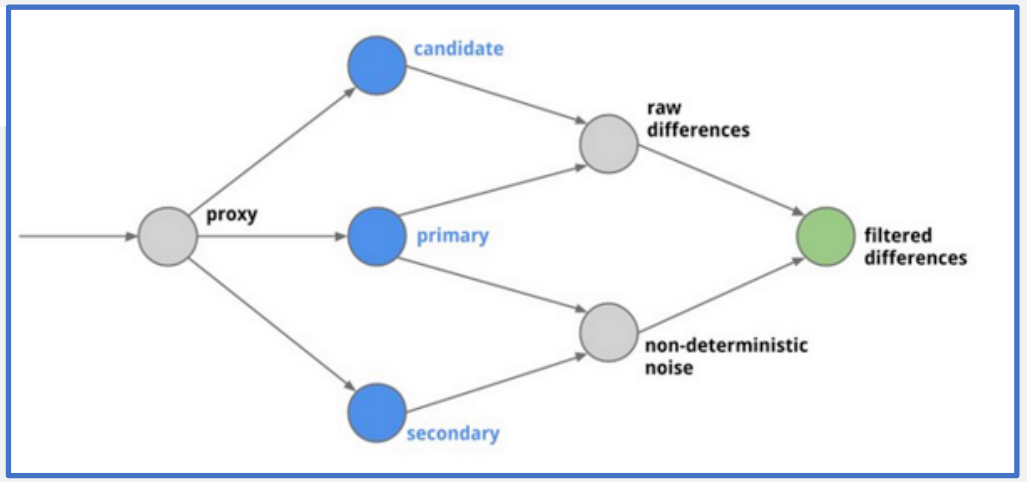
# Newer Approaches: Testing in Production

## Leverage ongoing user activity to validate new version

Canary




Diffy




BUT

- Late in the dev cycle
- Still requires upfront functional testing
- If traffic is low, we need time to converge
- High infrastructure cost

# Can we leverage past user activity to Shift-left Testing in Production?

## Massive Benefits

- ❑ 100% functional regression test coverage in CI pipeline
- ❑ High engineering velocity

## Main Challenge

Keeping state consistent between capture & test

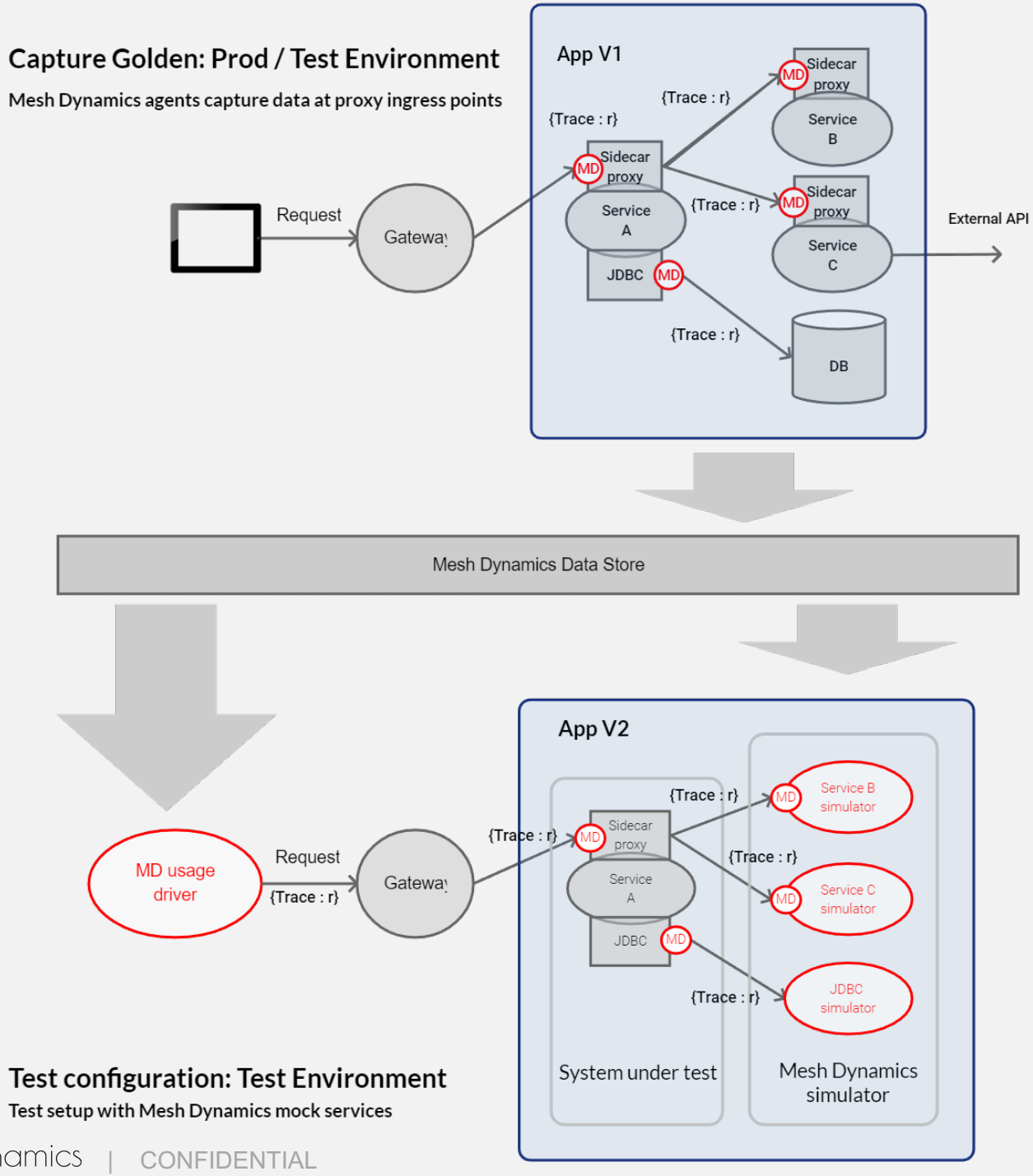
# Reconciling State Mismatch: Options

## Snapshot data just before capturing traffic

Capture must be uninterrupted.
High infra cost during testing.

## Simulate state with 100% fidelity

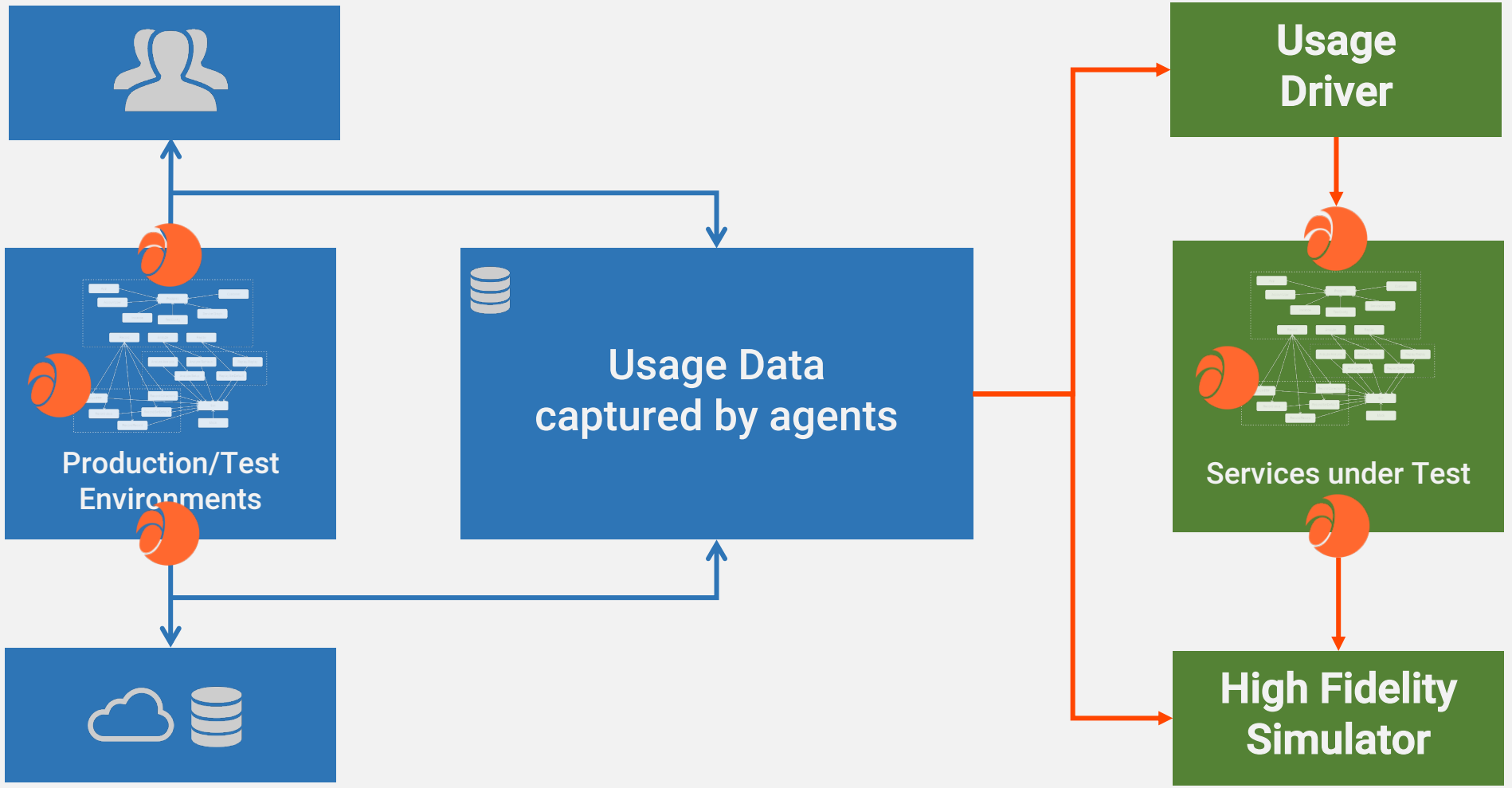
Is this even possible, especially with
- non-idempotent responses



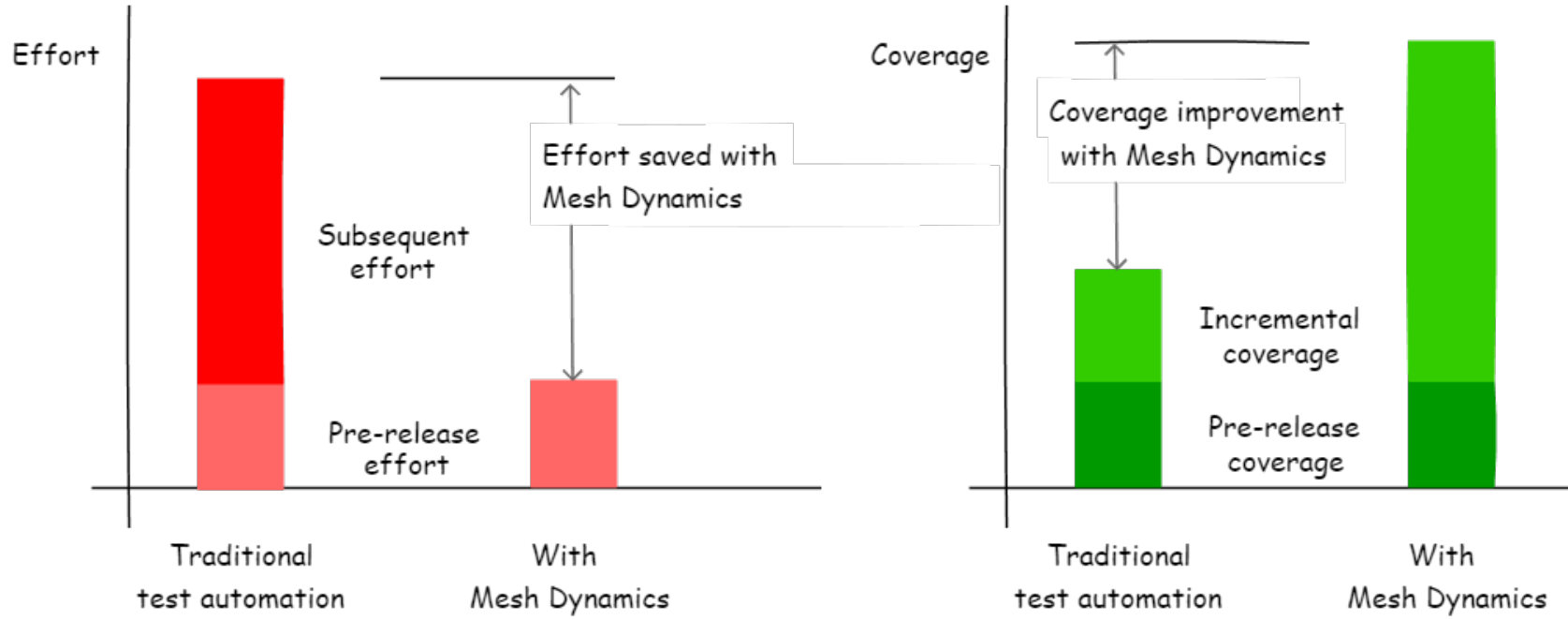

# Open-Tracing across services to the rescue! Enables simulation with 100% fidelity

# Testing in Near-Production Shifted Left with low effort

## Prevent API regressions with 100% coverage of usage

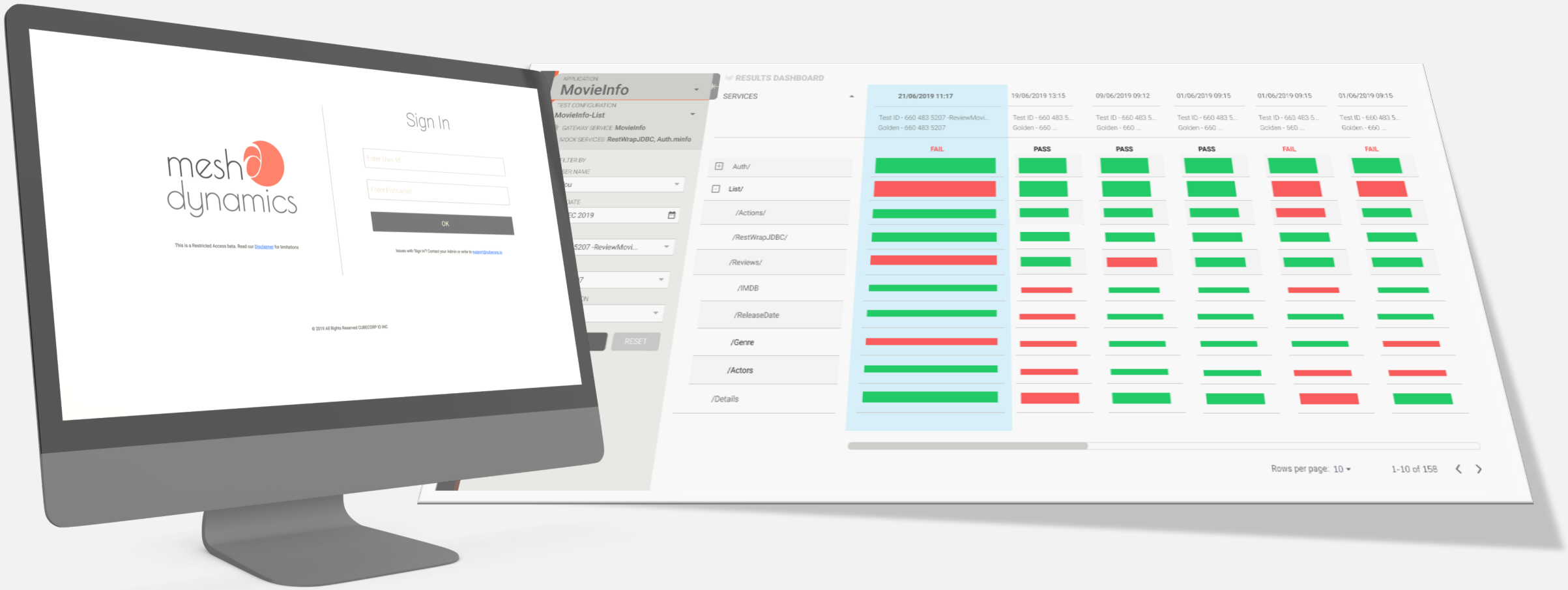



# High Velocity and Quality with Low Engineering Effort



# Demo

## Testing with Mesh Dynamics



# Conventional Process

1. Define several use cases, e.g.

```
Test results page API returns summary results of last 20 test runs
```

2. Enumerate several test cases for each use case, e.g.

```
• Did the API request return an OK status?
• Was the count of results set returned 20 or less?
• Did the actual data sets returned match the number?
```

3. Create test data for each test case, e.g.

```
• Create table with the right schema
• Populate at least 20 rows with representative data
• Populate 1st and 10th row with data to indicate failure
```

4. Write assertions for each test case, e.g.

```
tests["response code is 401"] = responseCode.code === 401;tests["response has WWW-Authenticate header"] = (postman.getResponseHeader('WWW-Authenticate'));var authenticateHeader = postman.getResponseHeader('WWW-Authenticate'), realmStart = authenticateHeader.indexOf ('"',authenticateHeader.indexOf("realm"))+1, realmEnd = authenticateHeader.indexOf ('"',realmStart), realm = …
```

5. Test scripting for each test

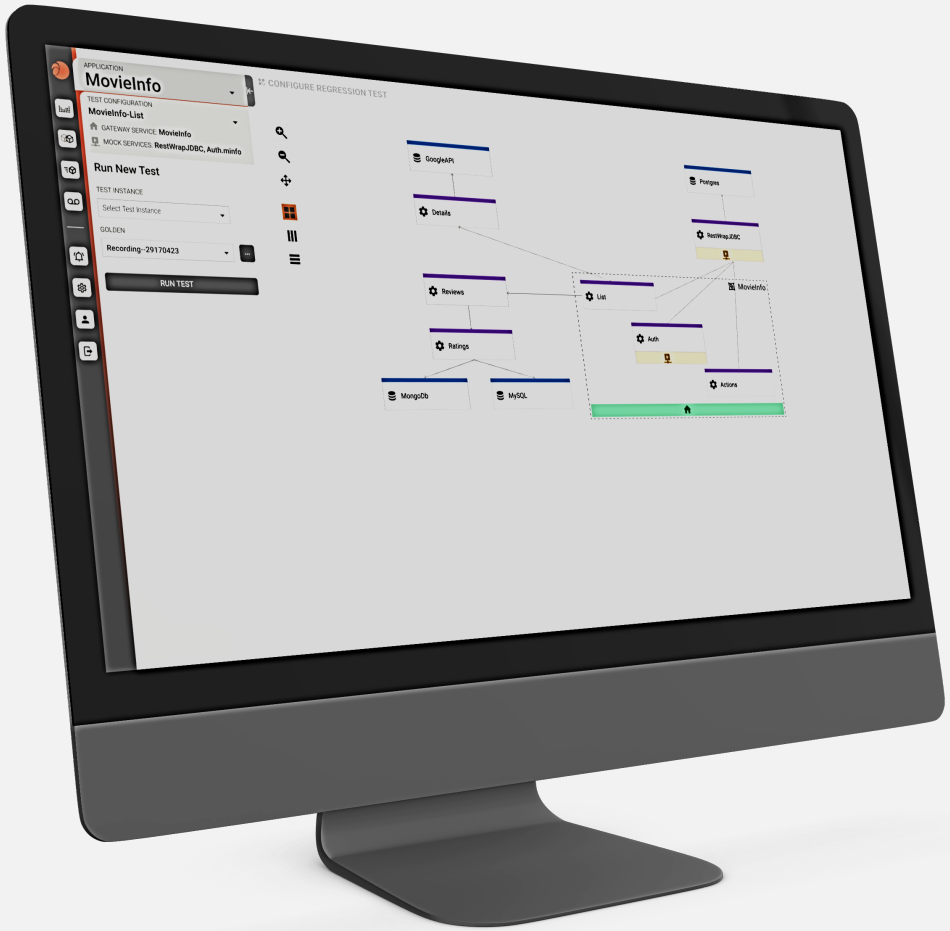
```
• Create input request
• Call test results service
• Identify data items to validate
• Add asserts for data items to validate
```

# Mesh Dynamics

Replaced by:

- Application usage capture
- Automated assertions
- Replay driver

# Testing with Mesh Dynamics



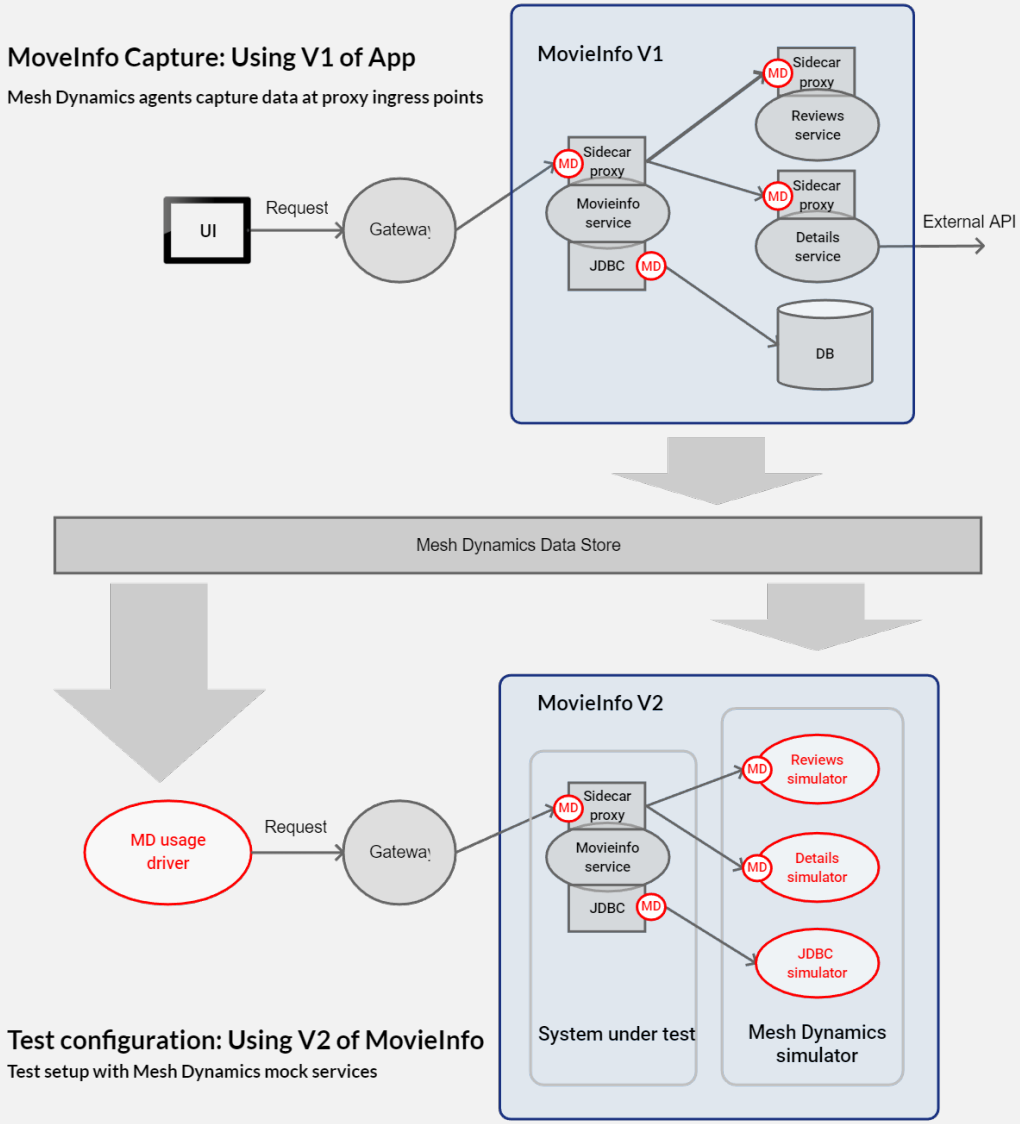

1. Create test
2. Run test + review test results
3. Update goldens

# Mesh Dynamics Demo Setup

Microservices application instrumented with OpenTracing libraries and MD listeners

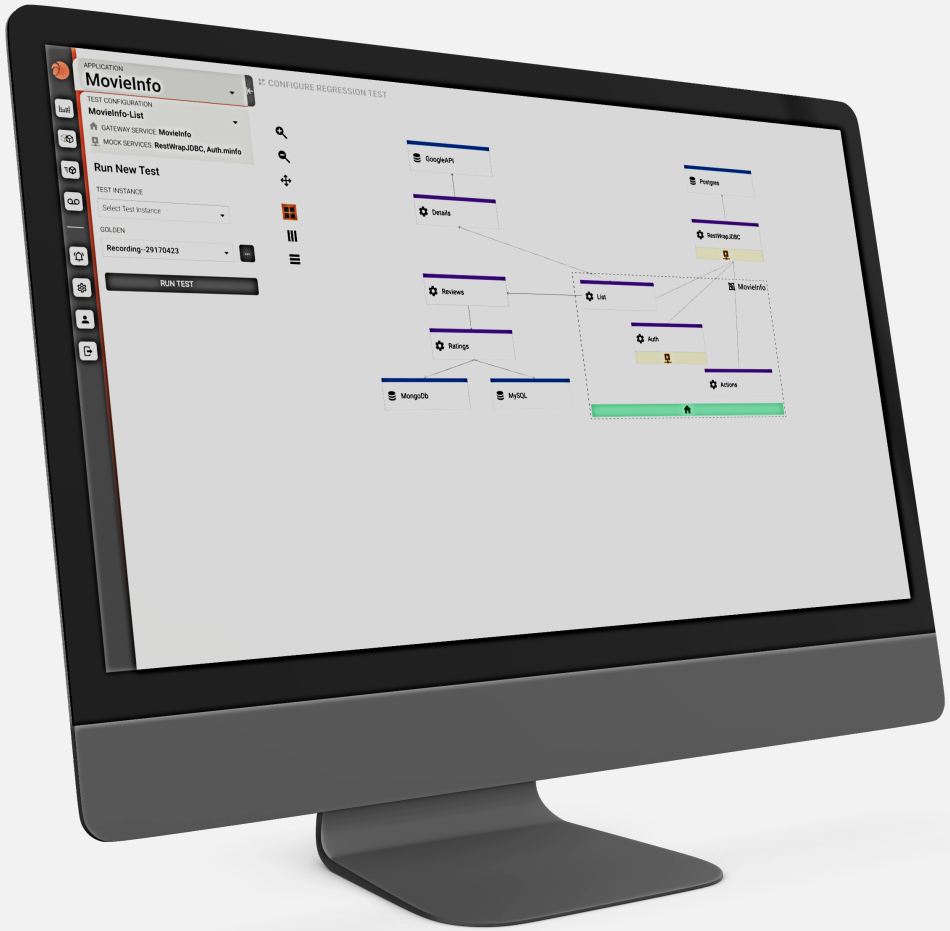
Step 1: Use V1 of MovieInfo app to capture usage

Step 2: Test V2 of MovieInfo app by replaying captured traffic



# Demo

## Testing with Mesh Dynamics




1. Create tests
    - Actual usage becomes test cases
    - No manual test case enumeration
    - No test data preparation
    - No test scripting
2. Run test + review test results
    - Review differences
    - File bugs + Notify people
3. Update goldens
    - No assertion scripting
    - No traditional test maintenance
    - Test maintenance through data updates

# Benefits

## Testing with Mesh Dynamics

### Conventional Process

1. Define several use cases, e.g.

```
Cube test results page API returns the summary results of the last 20 test runs
```

2. Enumerate several test cases for each use case, e.g.

```
• Did the API request return an OK status?
• Was the count of results set returned 20 or less?
• Did the actual data sets returned match the number?
• Validate the data structure of the response
```

3. Create test data for each test case, e.g.

```
• Create table with the right schema
• Populate at least 20 rows with representative data
• Populate 1st and 10th row with data to indicate failure
```

4. Write several test assertions for each test case, e.g.

```
tests["response code is 401"] = responseCode.code === 401;tests["response has WWW-Authenticate header"] =
(postman.getResponseHeader('WWW-Authenticate'));var authenticateHeader = postman.getResponseHeader('WWW-
Authenticate'),     realmStart = authenticateHeader.indexOf('"',realmStart),     realm = authenticateHeader.indexOf("realm")) + 1 ,
realmEnd = authenticateHeader.indexOf('"',authenticateHeader.slice(realmStart,realmEnd),
nonceStart = authenticateHeader.indexOf('"',authenticateHeader.slice(nonceStart,nonceEnd),
authenticateHeader.indexOf('"',nonceStart),     nonce = authenticateHeader.slice(nonceStart,nonceEnd);
postman.setGlobalVariable('echo_digest_realm', realm);postman.setGlobalVariable('echo_digest_nonce', nonce);
```

5. Test scripting for each test

```
• Create input request
• Call test results service
• Identify data items to validate
• Add asserts for data items to validate
```

mesh dynamics

1. Comprehensive regression test of microservices
2. Low engineering effort
   - No extensive test case enumeration
   - No test data preparation
   - No test scripting
   - No test maintenance
3. Based on actual usage