
TECHNICAL WHITEPAPER

# Test Patterns for Microservices

## INTRODUCTION

As an architecture for building complex systems, microservices are gaining significant traction within the development community. While people are beginning to understand that it's not a panacea for all application architecture woes, applications that share challenges related to dependencies and scaling can benefit greatly from it.

As the adoption of microservices rises, understanding how to test them becomes a challenge. Testing the interfaces between services is difficult. But there's a robust way to test APIs that improves microservices releases. Learn more about a strategic approach to testing APIs that deliver the high quality, security, and performance you expect in our strategy guide.

In some ways, testing a microservices application is no different than testing an application built using any other architecture. Microservices use well-known technologies, such as REST or queues, for which the industry already has well-established testing tools and best practices. The unique challenge with microservices is the sheer number of services that make up an application, along with the dependencies between the services. In addition, each microservice still needs to function properly even when the other microservices on which they depend are unavailable or responding improperly.

## UNDERSTANDING MICROSERVICES PATTERNS

The first step in understanding microservices testing is having a common understanding of the common messaging patterns that outline their behavior.

The core principles behind microservices are that they are lightweight discrete services that run as individually deployable processes using whatever technology the team determines is optimal for their project. These microservices often get grouped under business functions and domains whose composition of components represents the system under test.

Microservices typically follow two patterns when interacting with each other:

» Orchestration

» Reactive (choreography)

Many microservices use a combined hybrid approach. In this paper, we outline some of the challenges that arise when creating automated tests for microservices that use these different patterns, and provide strategies for addressing those challenges. We will focus on tests for individual microservices as opposed to end-to-end tests of the entire application.

## ORCHESTRATION PATTERN

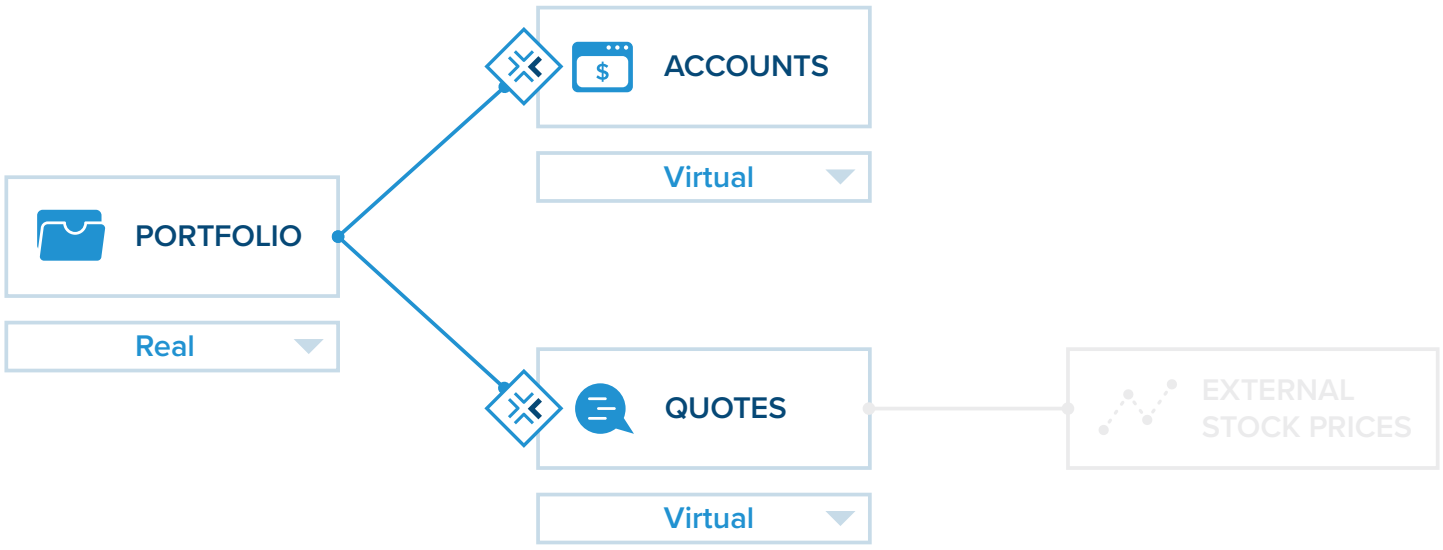
A microservice using orchestration will make one or more explicit calls to external services or dependencies. The calls typically use a synchronous request-response flow and will often access REST-based services. If the services need to be called in a specific order, calls to a subsequent service are not made until a response is received for a call to a prior service. Because one service explicitly calls another, they are tightly coupled.


In this example, the Portfolio service keeps track of the stock positions for a user. When adding a position to the portfolio, the Portfolio service will call the Accounts REST service to ensure that the user has enough funds within his/her account. After the Accounts call returns, it will call the Quotes REST service to see the current stock price. Once the Quotes call returns, the Portfolio service can then complete its task of adding a new position.

Creating and executing tests for the Portfolio microservice is challenging for the following reasons.

- The Portfolio microservice has dependencies on the Accounts and Quotes microservices, which need to be deployed in the test environment along with the Portfolio microservice, but the team that is building the Portfolio microservice may not be the team building the other two microservices.
- The Quotes service has a dependency on a third-party service to retrieve realtime stock prices, and the data returned by that service is always changing. To make a stable test for the Portfolio service, the data returned by the Quotes microservice needs to be constant.
- Unexpected behaviors of the Portfolio service need to be tested, such as when the Accounts and/or Quotes services are unavailable, respond slowly, or respond with unexpected data. It is important to be able to make those services respond with different kinds of unexpected behavior to validate that the Portfolio microservice handles the error conditions properly.

One solution is to use service virtualization to simulate the responses of the Accounts and Quotes microservices. A service virtualization solution, such as Parasoft Virtualize, would enable you to define virtual versions of the Accounts and Quotes microservices and deploy them along with the actual instance of the Portfolio microservice. Virtualizing microservices is similar to virtualizing any other kind of service or application architecture.

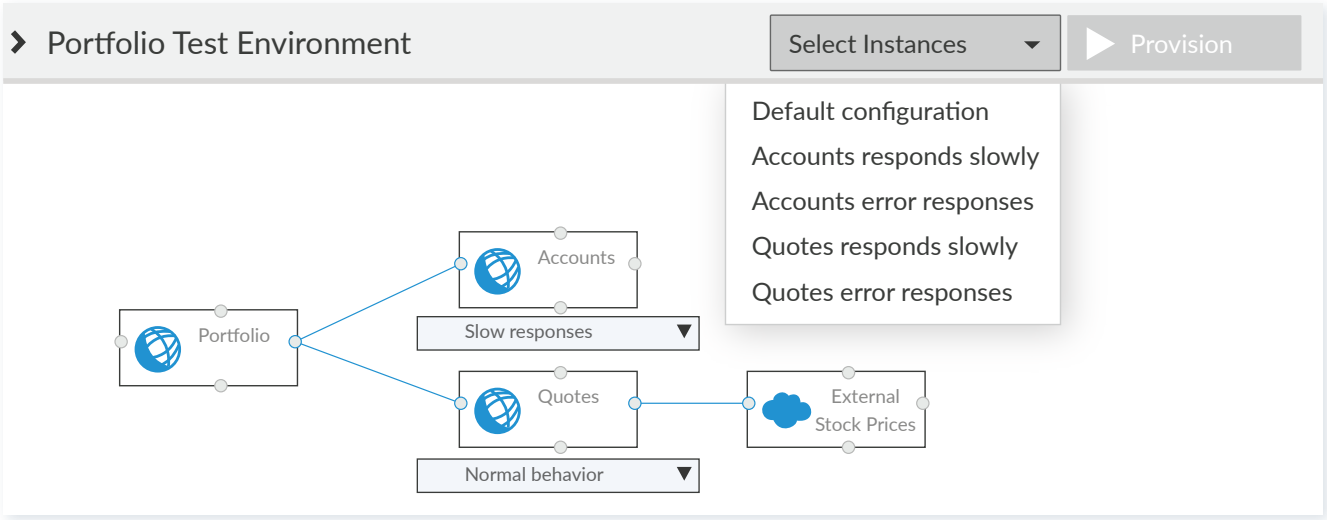



Once this is done, the Portfolio microservice can be tested independently of its two dependencies.

The next challenge is to configure different environments for different cases, such as when the Accounts and Quotes services exhibit expected and unexpected behaviors. Let's say that the team wants to test how the Portfolio service behaves when either the Accounts service or the Quotes service responds slowly or responds with error conditions. This may require running at least five different sets of tests, each of which has a different environment configuration:

| TEST RUN | ACCOUNTS SERVICE CONFIGURATION | QUOTES SERVICE CONFIGURATION |
|---|---|---|
| Normal behavior | Normal behavior | Normal behavior |
| Accounts slow response time | Responds slowy | Normal behavior |
| Accounts error responses | Responds with errors | Normal behavior |
| Quotes slow response time | Normal behavior | Responds slowly |
| Quotes error responses | Normal behavior | Responds with errors |

For each test run, the environment needs to be put into the correct configuration before the tests for that configuration can run. In this example, we end up with at least five different test runs, each of which have their own configuration for the environment. The Environment Manager module within the Parasoft CTP (Continuous Testing Platform) can manage these different environment configurations:




What we have described in this example is not specific to a microservices architecture. The same kinds of problems arise in service-oriented architectures in general, as well as in monolithic applications that may depend on only a handful of services. In a microservices architecture, however, the number of dependent services significantly increases. In this simple example with only three microservices, we can already see that the number of different environment configurations can increase quickly when trying to test different application states. In a microservices environment with tens or hundreds of services, the ability to create, manage, and programmatically switch between different environment configurations for different test scenarios is very important.

### MANAGING API CHANGES IN ORCHESTRATED MICROSERVICES

As teams evolve their microservices, it is inevitable that API changes will be made to the services. A key problem that arises with API changes is how to understand the effect of those changes on consumers of the services. The industry has been moving toward contract testing as a solution for this problem. A little-noticed problem, however, has emerged that relates to API changes: how to efficiently update test scenarios and virtual assets within the testing infrastructure to reflect the updated APIs.

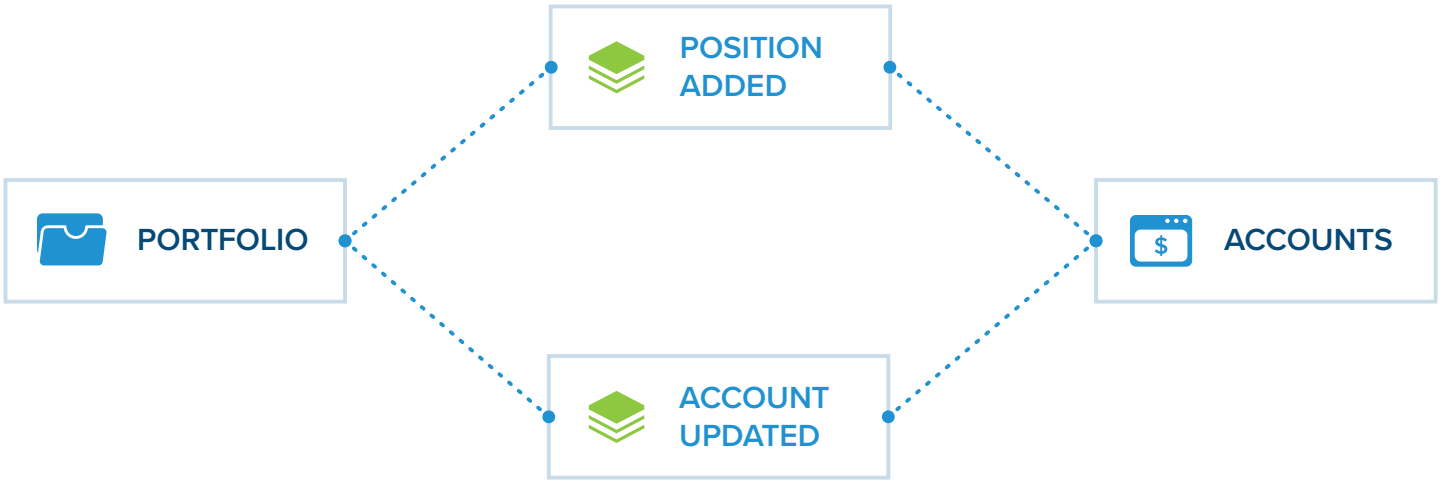

When a team modifies the API for a microservice they are building, any tests that validate that microservice need to be updated based on the changes in the API. Conversely, if virtual services are used to simulate dependent microservices and an API for one of those dependent microservice changes, the virtual services for the dependent microservice must be updated to reflect the changes in the API.

Going back to our previous example, let's say that the API for the Portfolio microservice changes: fields are added to and/or removed from some of the payloads. All tests for the Portfolio microservice will need to be updated to send the updated set of fields. Furthermore, if the API for the Accounts microservice changes, then the associated virtual services for the Accounts service will need to be updated to return the proper set of fields.

Many teams use OpenAPI, RAML, or another service definition to describe the APIs for their microservices. When service definitions are used, the Change Advisor module within Parasoft SOAtest and Parasoft Virtualize can automatically detect which APIs have changed, and then automatically refactor existing functional tests or virtual services to update them with any new and/or removed fields in the API. Teams can create updated versions of their service definitions and use Change Advisor to understand the impact of the changes on their tests and virtual services before making the changes. Once changes are made, Change Advisor makes it quick and painless to update existing assets to reflect the changes within the microservices.

## REACTIVE (CHOREOGRAPHY) PATTERN

One of the primary goals of a microservices architecture is to create independent components. As a result, deploying, scaling, and updating the services will be easier. This goal is not completely realized, however, when using the orchestration pattern because individual microservices have direct dependencies on other microservices. A way to solve this is to use the choreography pattern, also known as "reactive" or "event-driven" microservices. In this pattern, microservices do not directly reference each other. Instead, they push messages onto event streams to which other microservices have subscribed.




In this example, let's say the Portfolio service has been instructed to add a stock position. Rather than calling the Accounts service directly, it publishes an event to the "Position Added" event stream. The Accounts microservice has subscribed to that event stream so it gets the notification. It checks to make sure that the user has enough funds in their account. If so, it reduces the amount of funds in the users account and publishes an event to the "Account Updated" event stream. If the user does not have enough funds in their account, then it may publish an error event to a different event stream (not shown for simplicity of the example). The Portfolio microservice is subscribed to the "Account Updated" event stream, and when it sees the event posted by the Accounts microservice, it then updates its portfolio based on the confirmation from the Accounts service.

The asynchronous communication in this type of architecture introduces the benefit that the services are highly decoupled from each other – instances of each service can get replaced, redeployed, or scaled without the other microservices caring about them. The tradeoff is that the asynchronous nature of the events makes it harder to understand how the system will execute and what the flow of events will be. Depending on the order or kind of events that are produced, the system could behave in unexpected ways. This is known as emergent behavior, and is an inherent challenge in the development and testing of choreographed microservices.

There are different asynchronous messaging patterns that fall under the broader category of event-driven microservices.

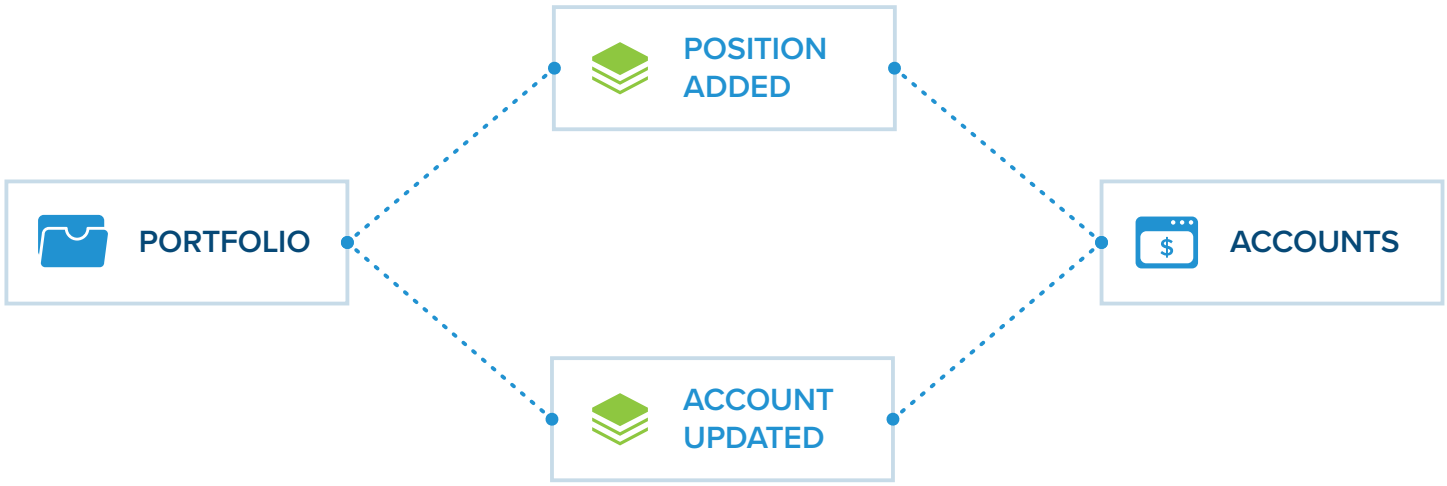
» Asynchronous command calls

» Event firehose

### ASYNCHRONOUS COMMAND CALLS

The asynchronous command calls pattern is used when microservices need to be orchestrated using asynchronous actions—where one microservice needs to call another microservice asynchronously, while guaranteeing that the second microservice received the message. In this pattern, messages are typically exchanged using queues. A common framework used in microservice architectures to implement this pattern is RabbitMQ.

One specific incarnation of this pattern is when one microservice needs to publish an event for a second microservice to process and then wait to read a "reply" event from that second microservice. This behavior was described above when introducing the microservices choreography pattern.




In this example, a REST API call tells the Portfolio microservice to add a position. The Portfolio service posts an event to the Position Added queue for the Accounts microservice to process, and then waits for the Accounts service to post a reply event to the Account Updated queue so that the REST API call can return data received from that event. There are two different ways to configure a test scenario for this example.

1. Create an environment with the necessary queues where the Portfolio service is deployed but the Accounts service is not. Because the Accounts service is not deployed, the test scenario will need to simulate the behavior of the Accounts service by posting the expected event from the Accounts service at the appropriate time. A Parasoft SOAtest test scenario will be constructed with two tests.

   - Execute the Portfolio service's REST API.
   - Post the event from the Accounts service.

   The tests need to be configured to run concurrently so that the event from the Accounts service posts while the Portfolio service is listening for the event.

2. Instead of simulating the Accounts service by using a test to post its event, it may be helpful to build a reusable virtual service that listens for events that are posted to the Position Added queue and posts a resulting event to the Account Updated queue. This virtual microservice would then be reusable across multiple different test scenarios that might need it.

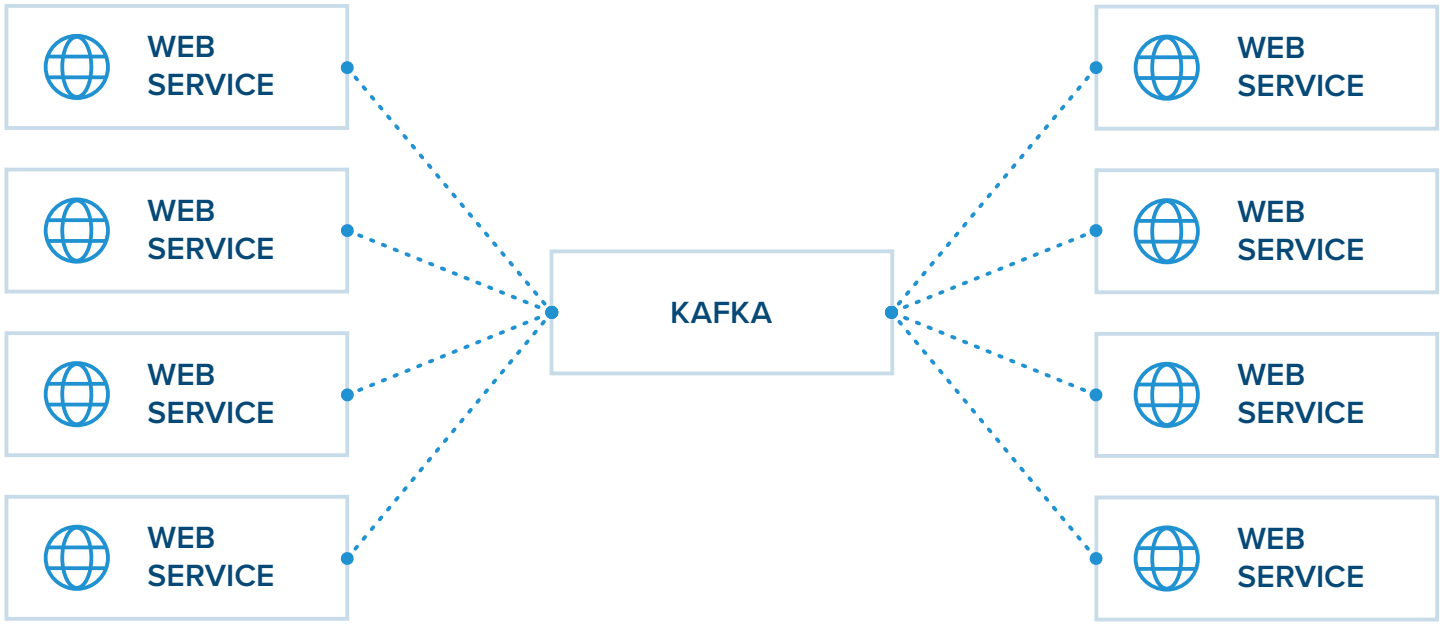
The first approach is simple and makes a self-contained test asset that has no additional external dependencies on test infrastructure. The second approach is reusable and is a closer simulation of the real behavior of the system. The second approach, however, has the cost of building, deploying, and managing a separate virtual asset.

A variation on this pattern is a microservice that listens on a queue for an incoming event, processes the event, and then publishes a follow-up event on a different queue for one or more other microservices to process.


In this example, the Invoice microservice is the service that needs to be tested. The Payments service publishes an event to the Payment Processed RabbitMQ queue for the Invoice service to pick up. The Invoice microservice reads the event from the queue, creates an invoice, and then publishes an event to the Invoice Created queue to direct the Email microservice to send an email to the customer with the invoice. To create a test scenario for the Invoice microservice, a test environment would be configured that contains two RabbitMQ queues and the deployed Invoice microservice. Using the RabbitMQ custom transport included in the IoT/Microservices Pack from the Parasoft Marketplace, a Parasoft SOAtest test scenario is constructed that publishes a payment-processed event to the Payment Processed queue. The scenario then subscribes to the Invoice Created queue to validate that the proper invoice created event gets published in response by the Invoice service.

### EVENT FIREHOSE

The event firehose pattern is used when different sources produce a very high number of events that need to be delivered quickly to different consumers over a common hub. In this pattern messages are exchanged via topics, while in the asynchronous command calls pattern previously discussed the messages are exchanged via queues. A common framework that is used to implement this pattern is the Apache Kafka framework.



Let's say that we want to test a single microservice that subscribes to a Kafka topic, processes the events it receives, and then publishes its results to a second Kafka topic.


In this example, we have a Forecast microservice that subscribes to a Weather Data topic that collects lots of different weather data from many different sources. It then processes that data to update its forecast model and publishes forecast data to the Forecast Data topic. In this case, we need to validate that the Forecast service publishes the expected events to the Forecast Data topic for a specific set of Weather Data events. This would be done by configuring a test environment that has the two Kafka topics and the deployed Forecast service.

We would construct a test scenario using Parasoft SOAtest and the Kafka custom transport included in the IoT/Microservices Pack from the Parasoft Marketplace. The test scenario would first publish the necessary events to the Weather Data topic and then subscribe to the Forecast Data topic to verify that the correct forecast data events were published by the Forecast service. Multiple different test scenarios would need to be constructed to verify the different types and order of events that could be expected to be handled by the Forecast microservice.

This is a relatively simple test scenario. The fact that the Forecast microservice is decoupled from the other microservices has the fortunate side effect that the test for the Forecast service is also decoupled from the microservices. In this case you don't need to set up a complex environment with virtual services. You can simply just create test scenarios that publish events and verify that the correct events are created in response.

## CONFIGURING TEST ENVIRONMENTS

Many microservice teams have adopted a CI/CD process for building, testing, and deploying containerized microservices to automate the process and decrease the risks associated with deploying updates. In this process, a container image that contains the microservice is automatically created and deployed into a test environment, often managed by Kubernetes or a Kubernetes-based distribution like OpenShift where the microservice can be validated before it is pushed into end-to-end tests and finally into production.

We have seen that service virtualization is useful and even essential for creating microservice test scenarios. When test environments are based on technologies like Docker or Kubernetes, the tools used to create and deploy virtual services need to fit nicely into those environments. Virtual services need to be highly componentized and easily deployable for the same reasons that the microservices they simulate are componentized. However, traditional service virtualization providers don't meet this need due to the applications being monolithic, centralized, and used by multiple teams.

To make service virtualization work in these environments, you need to create containerized virtual services that can be easily deployed. To create a containerized virtual service, you would take a base image that contains Parasoft Virtualize and all its dependencies, and layer it with another image that contains all virtual asset configurations for the virtual service. The new image for the virtual service would be deployed as a container into the Docker/Kubernetes environment along with the container for the microservice under test and all its (virtualized) dependencies.

## CONCLUSION

The messaging patterns and associated test patterns discussed in this paper are not new, but the need to use these patterns has grown significantly as microservices become more common and more applications adopt a microservices paradigm. Parasoft's SOAtest, Virtualize, and CTP enable teams to create and deploy test scenarios for their microservices with maximum flexibility—ensuring the high quality and reliability of their microservices.

## TAKE THE NEXT STEP

Request a demo to see how your development team can more easily create and deploy automated tests for microservices that use different patterns.

### ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives — security, safety-critical, Agile, DevOps, and continuous testing.