Messaging Patterns for Event Driven

In a microservices architecture, each microservice is designed as an atomic and self-sufficient piece of software. Implementing a use case will often require composing multiple calls to these single responsibility, distributed endpoints. Although synchronous request-response calls are required when the requester expects an immediate response, integration patterns based on eventing and asynchronous messaging provide maximum scalability and resiliency. Some of the world's most scalable architectures such as [Linkedin](https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin) and [Netflix](http://techblog.netflix.com/2016/04/kafka-inside-keystone-pipeline.html) are based on event-driven, asynchronous messaging.

New Demands for Microservices Integration

While most of the requirements for integrating microservices still reflects existing [enterprise integration patterns](http://www.enterpriseintegrationpatterns.com/), their highly distributed nature creates new demands for decentralized messaging, based on [smart endpoints and dumb pipes](https://martinfowler.com/articles/microservices.html).  Instead of a central, unique integration bus, each group of microservices (usually within the same [bounded context](http://www.martinfowler.com/bliki/BoundedContext.html)) will choose its own messaging implementation, depending on the needs and characteristics of each use case.

Like [polyglot and decentralized persistence](https://martinfowler.com/bliki/PolyglotPersistence.html), decentralized polyglot messaging should be key in microservices architectures, allowing different groups of services to be developed on their own cadence. It also minimizes the need for highly coordinated, very risky big-bang releases. Best of all, a microservices approach allows developers dramatically more flexibility to choose the optimal messaging implementation for the job at hand.  Each use case will have its own specific needs, which may require different messaging technologies such as Apache Kafka, RabbitMQ or even event-driven NoSQL data grids, such as Apache Geode / Pivotal GemFire.

Asynchronous Messaging Patterns

Organizing different integration scenarios over a list of common patterns helps identify similar solutions and maximize reuse. Here are some best-of-breed asynchronous integration patterns for microservices, implemented with open-source solutions:

1) Event Firehose

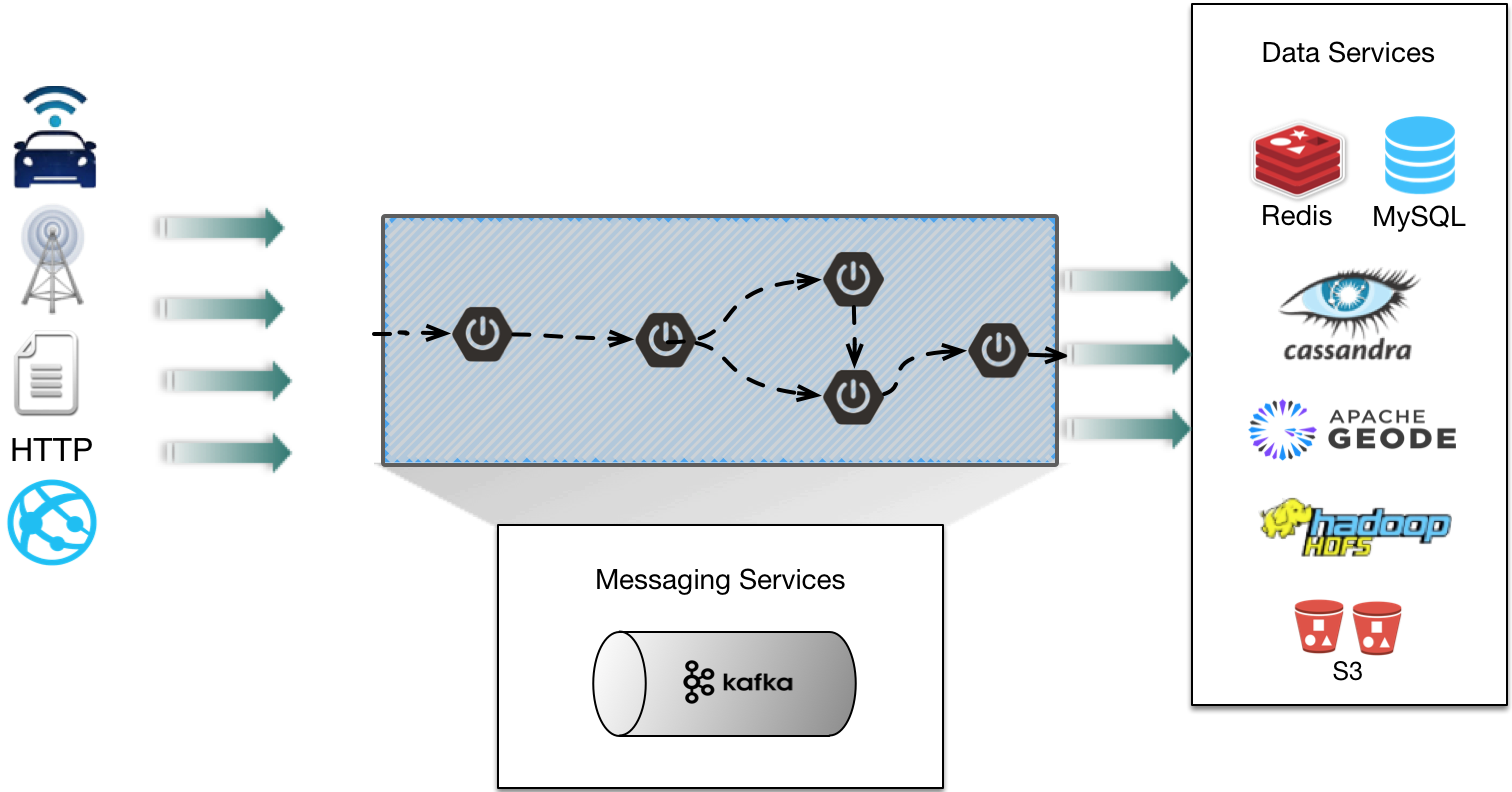
Thanks to IoT, social networks and real-time stream processing, event firehose use cases are becoming more common. The need is for highly scalable messaging, able to receive a very high number of events, coming from different sources, and delivering them to different clients over a common hub. Message consumers can consume the data as they wish, and re-read and replay messages on demand.  There's a many-to-many relation between message/event producers and consumers, where some of these consumers can be batch-based and others that prefer online stream processing.

For this use case, Apache Kafka's ability to scale to hundreds of thousands of events per second, delivered in partitioned order, for a mix of online and batch clients is the best fit. Kafka [was designed at Linkedin](https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin) as a producer-centric system centered around the [log abstraction](https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying), for ultimate scalability and performance for streaming scenarios. It's built to handle the explosion of both data events and specialized data systems.

Due to its intended simplicity, Kafka leaves it up to the clients to keep track of the state of the system (messages offset), partitioning and manually implement any routing.  From the message producer / publisher point of view, Kafka [can guarantee a message is persisted to the log](https://kafka.apache.org/documentation/#semantics) (not necessarily only once) and is replicated to multiple brokers for HA. Surprisingly, [keeping track of consumer position is one of the key performance points of a messaging system](https://kafka.apache.org/08/documentation.html#theconsumer), so Kafka's design leaves it up to the consumers to pull messages and keep track of their position in the log offset.

Since microservices architectures pattern calls for [smart endpoints and dumb pipes](https://martinfowler.com/articles/microservices.html), Kafka will do just enough for most application and system integration use cases. Your microservices endpoints should be smart enough to implement any intelligent routing and transformation of message enrichment by themselves.

This is what Spring Cloud Data Flow (SCDF) provides, complementing Kafka as a  [fundamental framework for building event-driven microservices](https://www.linkedin.com/pulse/cloud-native-architectures-rise-data-microservices-fred-melo). SCDF is based on open-source connectors and allows configurable message routing and transformation through a domain specific language (DSL), visual design and event-based processing. Pipelines built with SCDF are independent from the messaging transport implementation, leveraging Kafka, RabbitMQ or any of the available [standard transports (binders](http://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/index.html#_binders)) interchangeably.  Other existing stream processing solutions such as Kafka Streams and Storm can also work on top of Kafka, but at the expense of significant coding and a considerable departure from your cloud-native architecture model.



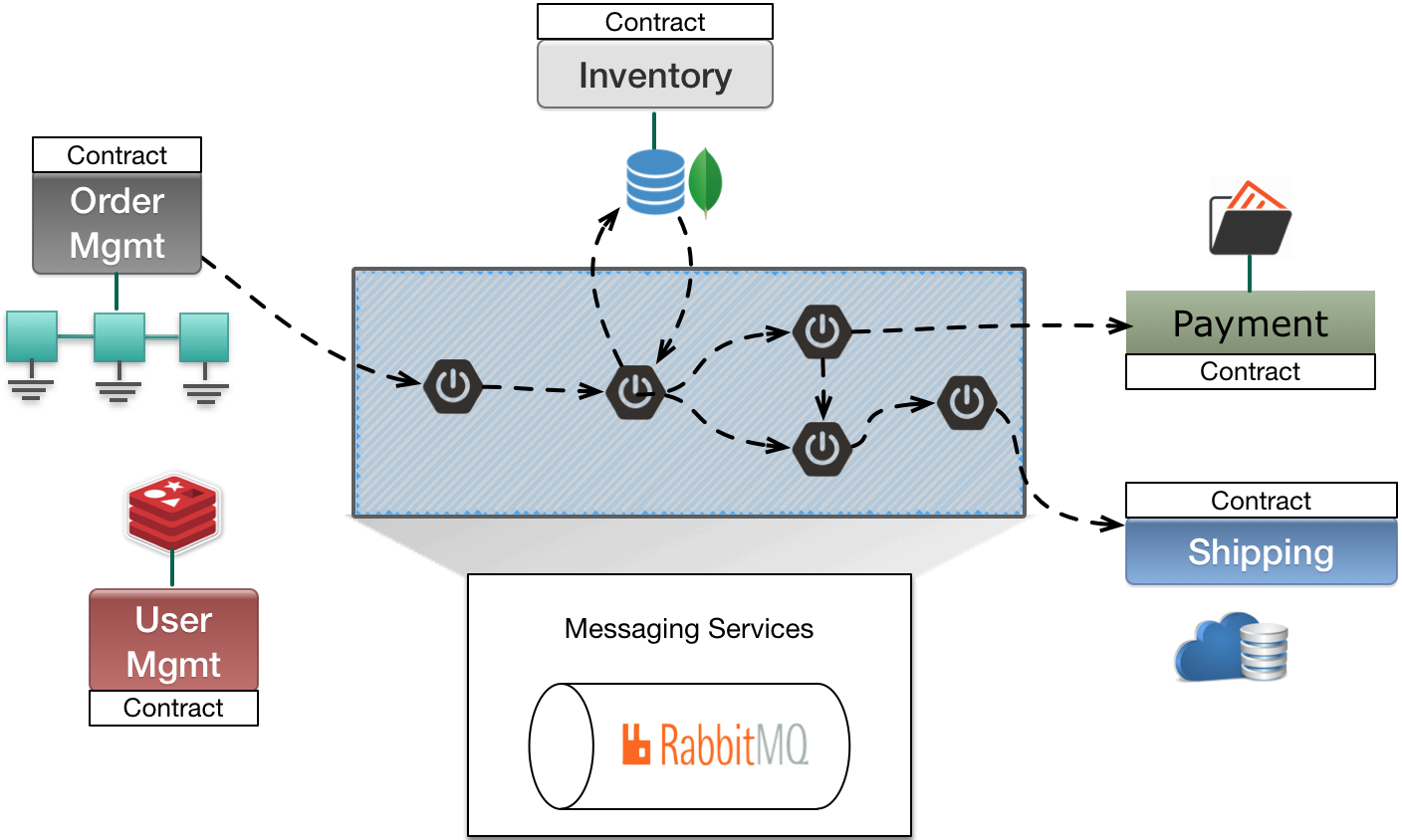
2) Asynchronous Command Calls

Composing microservices atomic calls into complex flows often requires proper orchestration over asynchronous actions. These are usually local integration use cases, connecting related microservices that must exchange messages with a delivery guarantee. The messaging layer in this use case has substantially different needs from an event firehose, since its messages are point-to-point (queues instead of topics). This usually requires a delivery guarantee and most are short-lived (albeit still asynchronous), conversational. It's a traditional broker-centric use case, reliably connecting endpoints through asynchronous communication. The communication flows through atomic messages exchanged between parties, instead of a constant stream of events potentially handled by multiple processes.

This pattern is better implemented by a lightweight messaging platform such as RabbitMQ, as described [by Martin Fowler](https://martinfowler.com/articles/microservices.html). RabbitMQ [scales incredibly well with a small system footprint](https://blog.pivotal.io/pivotal/products/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine) and doesn't require the consumer application to control the messaging consumption state like Kafka. It powers some of the world's largest scale use cases, like [Instagram's feed](https://www.youtube.com/watch?v=E708csv4XgY).

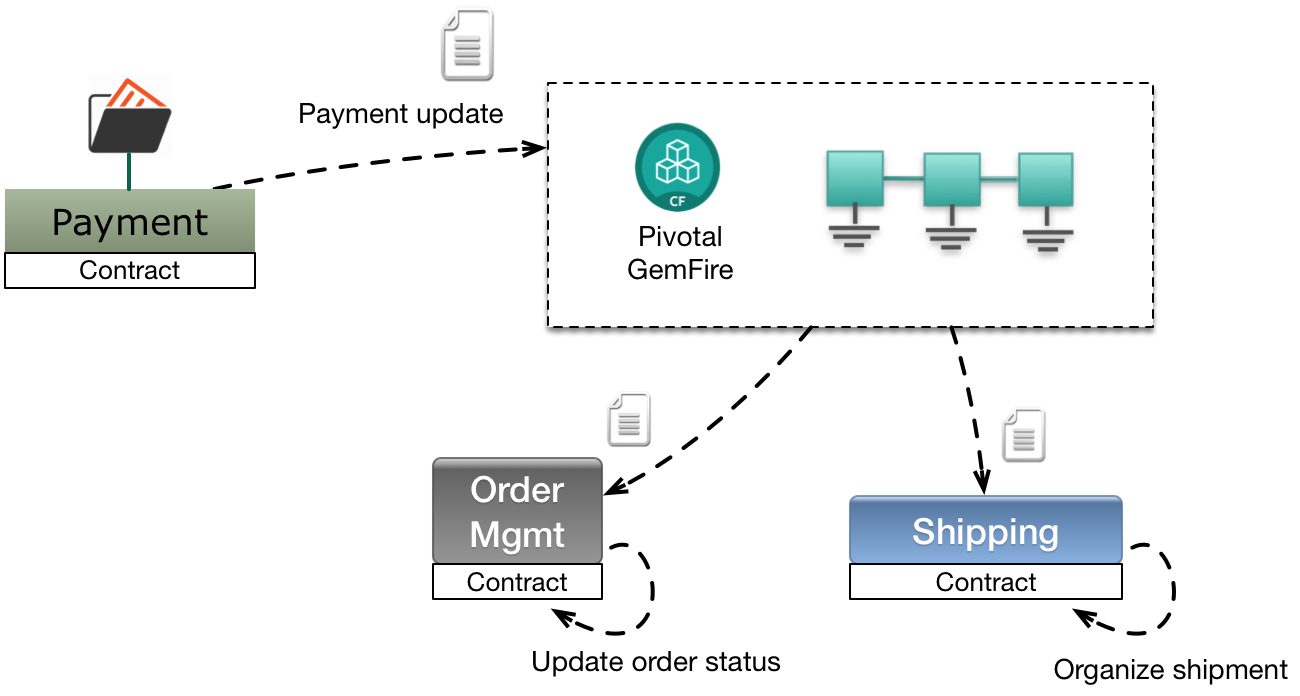
However, RabbitMQ's hidden secret for integrating microservices in a cloud-native architecture is the Pivotal Cloud Foundry [service broker and tile](http://docs.pivotal.io/rabbitmq-cf/1-7/). Certainly one of the most neglected [fundamental characteristics](https://martinfowler.com/articles/microservices.html) of microservices architectures is infrastructure automation, or the ability to fully and repeatedly build, deploy and operate microservices through continuous delivery pipelines. The Pivotal Cloud Foundry tile for RabbitMQ allows automated install, updates and scaling for multiple cloud environments and can be fully integrated into [continuous delivery tools](https://concourse.ci/) so you can focus on building software and not automating services.

As with Kafka, RabbitMQ is one of the [standard transports (binders](http://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/index.html#_binders)) for SCDF. Users can use SCDF's visual design to create integration pipelines that fully abstract the underlying messaging implementation, while leveraging RabbitMQ's performance, scalability, and reliability.



3) Data Events Exchange

Some microservices integration scenarios can be solved by simply handling lifecycle events from data persisted in a data store. In this scenario, one or more microservices subscribe to data change events directly from a NoSQL store and are notified upon data changes. Those notifications can include new data being persisted, existing data being modified or deleted. Unlike the other patterns previously mentioned, events are triggered out of data operations and the message payload is the updated data itself. This considerably simplifies event-driven models when system operations should follow data updates.



By subscribing to [data lifecycle events](http://gemfire.docs.pivotal.io/geode/developing/events/chapter_overview.html), microservices acting as clients of a Pivotal GemFire or Apache Geode cluster will have their listeners triggered by changes in the persisted data. Delivering data events directly from where they are stored is [used extensively in capital markets](http://www.slideshare.net/Pivotal/wall-street-derivative-risk-solutions-using-geode), when extreme low latency is a must.  It also powers use cases such as [China Railways ticketing system](https://blog.pivotal.io/pivotal/case-studies/china-railway-corp-for-chinese-new-year-chunyun), where ticketing events can trigger actions in other distributed components, such as fleet logistic adjustments and the repricing of remaining seats.

While this pattern can be useful, it requires all components to agree on the context and format of the data being exchanged. Architects should be careful not to introduce unwanted coupling between microservices that exchange data events, by protecting their boundaries and clearly diving responsibilities over a [bounded context](https://martinfowler.com/bliki/BoundedContext.html).

Like RabbitMQ, GemFire also has a [Pivotal Cloud Foundry Service Broker and tile](https://content.pivotal.io/blog/introducing-gemfire-for-pivotal-cloud-foundry) for a fully automated operational experience on multiple clouds.

Conclusion

This is not intended to be an exhaustive catalog of asynchronous integration patterns for microservices, but rather a look at  common scenarios for cloud-native architectures. There's no single solution for all use cases, and embracing decentralized messaging allows more flexibility, faster iterations and better resiliency.

As with polyglot persistence, enterprises should define their internal standards for decentralized polyglot messaging based on reference architectures goals and requirements. Each new product adoption comes with its own costs and challenges, and automating operations on multiple clouds becomes mandatory.  Companies should standardize on few common patterns, implemented using reusable best-of-breed solutions over a [cloud-native platform](https://pivotal.io/cloud-native).

Next Steps

For trying out microservices messaging patterns on a cloud-native development sandbox environment, [PCFDev](https://pivotal.io/pcf-dev) is the easiest way to get started. It includes a fully managed RabbitMQ service broker, along with Redis, MySQL and Spring Cloud Services.

If you're looking for a fully managed Cloud Foundry experience, [Pivotal Web Services](https://run.pivotal.io/) is free to signup and  features a RabbitMQ (Cloud AMQP) offering on its [marketplace](https://console.run.pivotal.io/marketplace), along with dozens of other services.

For a full-fledged cloud-native platform that supports all these microservices patterns, check out [Pivotal Cloud Foundry](https://pivotal.io/platform).