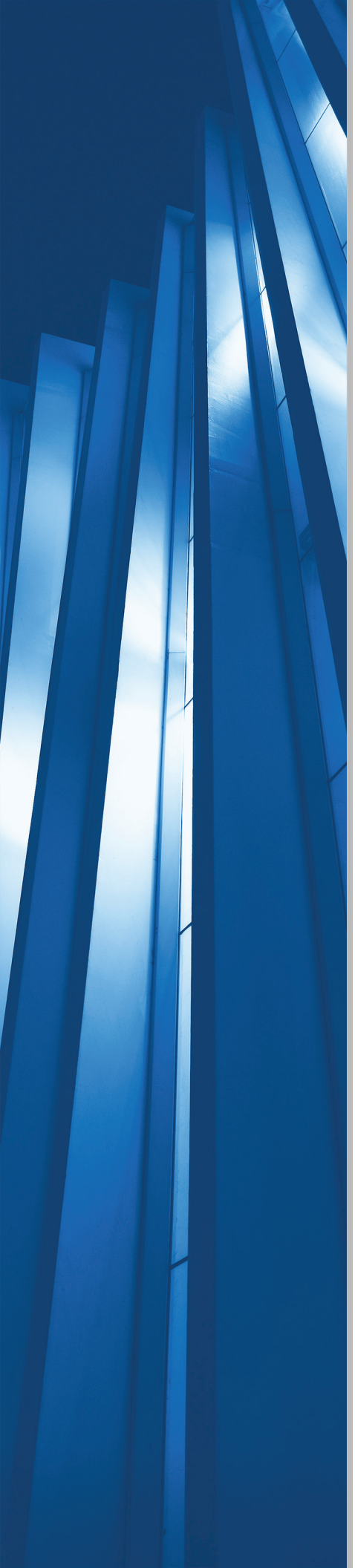


Event-Driven Microservices

The Architect's Guide to Building a Responsive, Elastic and Resilient Environment

Introduction

Many organizations migrate to microservices architecture for one reason: Agility. The ability to very quickly create and modify components in a way that offers bottom line business value is mission critical in a world where your competitors are a click away and time to market is everything. But the speed with which you develop components is just one piece of the puzzle.

How quickly can you integrate them with the rest of your system? How completely can you embrace innovative new techniques and technologies?

For your microservices initiative to be successful, your organization must realize that agility hinges on a holistic approach leveraging practices such as CI/CD, automated infrastructure, DevOps and agile development. An average microservices architect combines these concepts into a lean, mean development machine. A *great* microservices architect goes beyond that by appreciating and finding ways to overcome the undeniable – but *predictable* – pitfalls that accompany highly distributed systems.

The key to ensuring a microservice initiative's success is appreciating the significant risks or barriers to agility:

- Brittleness of distributed processing;
- Ecosystem integration challenges; and,
- A lack of service harmony.

If we don't respect the realities of distributed systems, we *will* repeat the failures of the past, like the SOA hype of the 2000s. Fortunately, there is one really powerful way to compensate for these realities: event-driven (also called reactive) architecture.

Jonathan Schabowsky, a senior architect in Solace's Office of the CTO, will explain the massive benefits of combining event-driven architecture and microservices, starting with the fact that decomposing applications, while in many ways beneficial, can make life a little more complicated.

The Service Decomposition Paradox

The theory of microservices is simple: achieve better agility, scalability and reusability by breaking monolithic applications into small, purpose-specific microservices.

The real-world implementation of microservices is, as always, more complicated than that. This is in large part because of the fallacies of distributed computing – a set of false assumptions that programmers and architects make when they enter the world of distributed applications. This list was written in 1994 by L. Peter Deutsch and others at Sun Microsystems, but it holds true today.

The Fallacies of Distributed Computing

- **The network is reliable.**
- **Latency is zero.**
- Bandwidth is infinite.
- **The network is secure.**
- Topology doesn't change.
- There is one administrator.
- **Transport cost is zero.**
- **The network is homogeneous.**

While all of these fallacies are relevant, those in **bold** are of special importance to the world of microservices. The smaller you make each microservice, the larger your service count, the more the fallacies of distributed computing impact stability and user experience/system performance. This makes it mission critical to establish an architecture and implementation that minimizes latency while handling the realities of network and service outages.

Much of the tooling related to microservices involve the use of CI/CD, automated infrastructure, DevOps and agile software development. A great example would be Pivotal Cloud Foundry, which gives developers an easy way to create, test, deploy and update microservices using modern, cloud-native techniques.

The challenge is that microservices require connectivity/data in order to perform their roles and provide business value, and data acquisition/communication has been largely ignored, so much so that the tooling has severely lagged behind. For example, API management/gateway products only support synchronous, request/reply exchange patterns, which exacerbates the challenges of distributed computing. And they don't have the ability to integrate with or acquire data from legacy systems.

At the same time, eventing/messaging tools have been stuck in the antiquated, non-agile world too, being incompatible with many of the guiding principles of microservices such as DevOps and self-service. But it's eventing/messaging that best deals with the idiosyncrasies of distributed computing and is the key to

unlocking the potential of microservices architecture.

As services become smaller and their purpose more singular, the potential for reusability increases, but that's contingent on the ability for services to collaborate.

In the days of SOA, you'd create massive monolithic services that directly implemented all facets of the use case as a set of services orchestrated using BPEL engines or an ESB. They weren't reusable and were difficult to scale because ESBs/BPEL orchestration moved too much logic into the network, leading to "dumb endpoints and smart pipes" that were expensive, complex and nearly impossible to troubleshoot.

Today, we recognize that the way to go is small, single-in-purpose microservices and a "smart endpoints, dumb pipes" approach to connectivity and communications. But that lingering question persists:

How can we enable service collaboration in order to offer business value without falling back to those failed monolithic or orchestration techniques?

The Integration Conundrum

Because all microservices need data for processing, and since 12-Factor Apps are stateless, data needs to come from somewhere. Acquiring data for greenfield systems is easy, but microservices almost always come to be as a side effect of digital transformation, modernization or the need to build new capabilities at a more rapid pace. So you're almost always dealing with an ecosystem of legacy systems; some will be modernized, others will remain as is for the foreseeable future.

The existing business ecosystem is the unavoidable burden that most businesses must deal with when they start their microservices journey. Most existing systems live on premises, while microservices live in private and public clouds. The ability for data to transit the often unstable and unpredictable world of wide area networks (WANs) is tricky and time consuming. Then we need to factor for the emergence of and specialization caused by IoT, mobile devices and big data. The volume and variety of these systems results in a very large upfront risk to microservices initiatives.

When you add it all up, there are impedance mismatches everywhere you look:

1. Updates to legacy systems are slow, but microservices need to be fast and agile;
2. Legacy systems use old communication mediums, but microservices use modern, open protocols and APIs;
3. Legacy systems are nearly always on premise and at best use virtualization, but microservices rely on clouds and IaaS abstraction;
4. IoT systems use highly specialized protocols, but most microservices APIs and frameworks don't natively support them; and,
5. Mobile devices may use REST but also require asynchronous communication, but most API gateways only support synchronous RESTful interactions.

How can an organization resolve all of these mismatches?

The Incredible Dancing Microservices

As I described earlier, the smaller the service, the less value it discretely offers the end user – value comes from orchestration. Historically, orchestration was handled by a central component like BPEL engines or ESBs, or, these days, API gateways.

I'll use a musical analogy to explain the problem with that approach.

Most composers create music by trial and error (just like a software developer). Their output? Musical scores containing sheets of music which will be played by each instrument. For this analogy each musician is like a microservice. If the composer is writing for a heavy metal band, there are only a few instruments (guitar, bass, drummer, vocalist) and a conductor isn't needed.

But a symphony orchestra consists of around one hundred musicians playing a wide range of instruments. In this instance, they absolutely need a conductor to ensure that every musician starts playing at the right moment, stays on beat, speeds up and slows down when necessary, plays louder or softer.

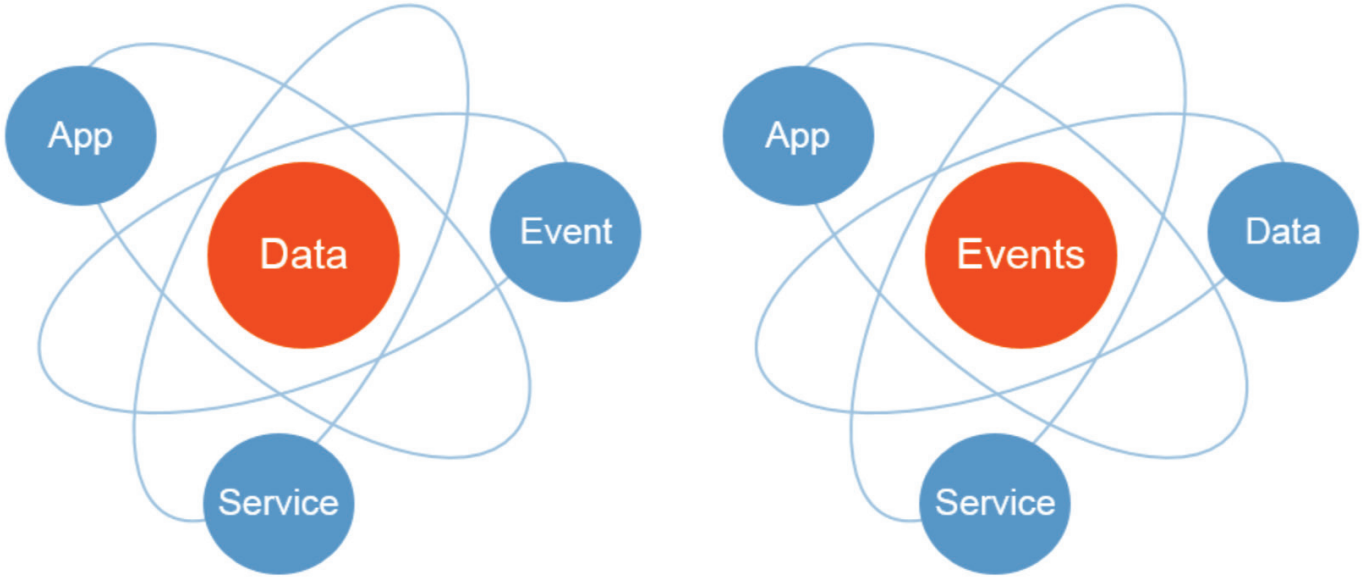
Unfortunately, if the conductor can't communicate with the musicians – or even just a section or potentially a single musician – things will quickly fall apart and the performance will suffer, if not be ruined.

Dancing is different. A choreographer listens to a song and creates a routine based on events in the music. The dancers may do completely different moves or steps from each other, but as long as it was choreographed together based on those audio queues (or events) then the routine will be a success. Even if someone messes up by doing the wrong step or losing the beat, the show can continue because each dancer is listening to the music for their specific *event* rather than being told what to do by an orchestrator.

To bring it back to microservices, a given microservice will perform a series of steps within the code which is an example of micro-orchestration. The input or output of a microservice is a data event which has domain significance. The key is that since the microservice is merely producing an event, it does not have knowledge of if or when it will be processed. Other services register their interest in an event or set of events and react accordingly. Just like a dancer executing the steps of the routine. Ironically, the common enabling theme between microservice execution and a dance move is the event (data event or audio event).

The Copernican Shift to Events

From Data-Centric to Event-Centric IT Priority – A Copernican Shift



Source: Gartner "CIO Challenge: Adopt Event-Centric IT for Digital Business Success," 30 May 2017, Yefim V. Natis, Tina Nunno, figure 2.

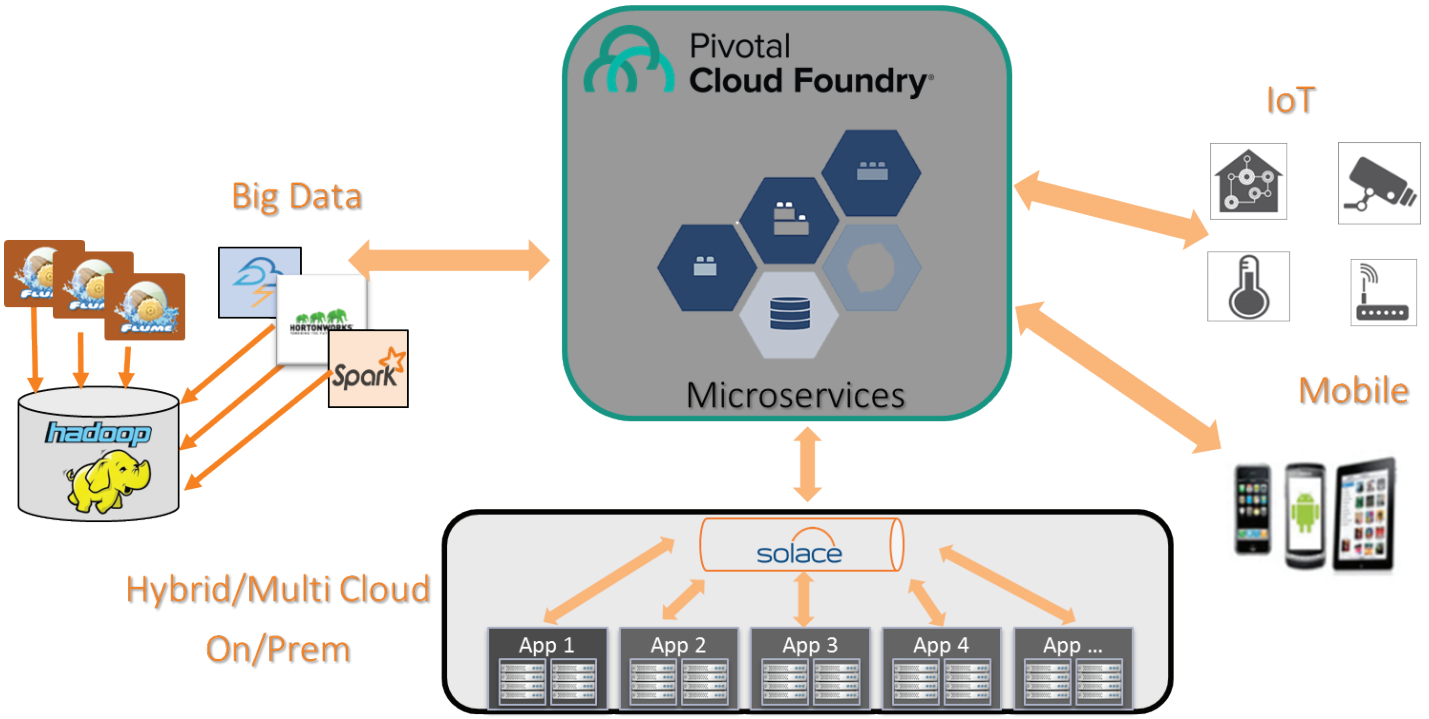
Copernicus was a Renaissance-era mathematician and astronomer who formulated a model of the universe that placed the Sun rather than the Earth at the center of the universe. Game-changing stuff at the time.

Similar to the mistakes that astronomers made before Copernicus, many architects and technologists are obsessed with the idea that *data* is the center of the computing universe. This traditional viewpoint is based on the belief that data is job one, and that once it's preserved it will be viewed, updated and deleted using command style request/response interactions.

The reason for this extreme focus on data is simple: all interactions with databases are performed by command-style interactions! The problem with this viewpoint is that enterprises end up making a bunch of microservices with databases at their core, leading to many stove-piped application enclaves that can't share data in a fast, flexible manner. In other words, money was spent to create a different type of monolithic application, one that is doomed to the same lack-of-agility fate.

So what should be at the center of the microservices universe? That's easy: Events.

If we are to exponentially increase agility through microservices, we need to replace our static, stove-piped, monolithic thinking with a desire to get the right event to the right service at the right time.



Consider your own body. We constantly react to and act upon events that arrive via our senses of touch, vision, taste, hearing and smell. This events relay information that's stored in our memory, replayed in our minds, and acted upon if necessary in the form of some action that produces new events for our universe to react to.

Thinking in an event-driven manner turns organizations into a sensory element in the universe of computing. New events sensed by web/mobile applications, IoT sensors, or legacy systems of record are forwarded to the eventing/messaging platform which distributes them to the microservices platform. Just like how our sensory system fires events to the central nervous system for interpretation. In the world of microservices, we need a central nervous system that can:

- be receptive to different stimuli;
- provide fast, rock-solid reliable transport; and,
- adapt to changes in event reception.

Not adopting event-driven thinking will forestall the success of digital transformation and microservices initiatives by increasing costs and decreasing productivity. So what are the patterns and approaches that will enable you and your organization to achieve success with events at the center of your IT universe?

Realizing the Agility Provided by a Modern Central Nervous System

As Gartner states in their August 2017 report titled *Business Events, Business Moments and Event Thinking in Digital Business*, "Application leaders engaged in digital transformation initiatives must add 'event thinking' to their technical, organizational and cultural strategies."

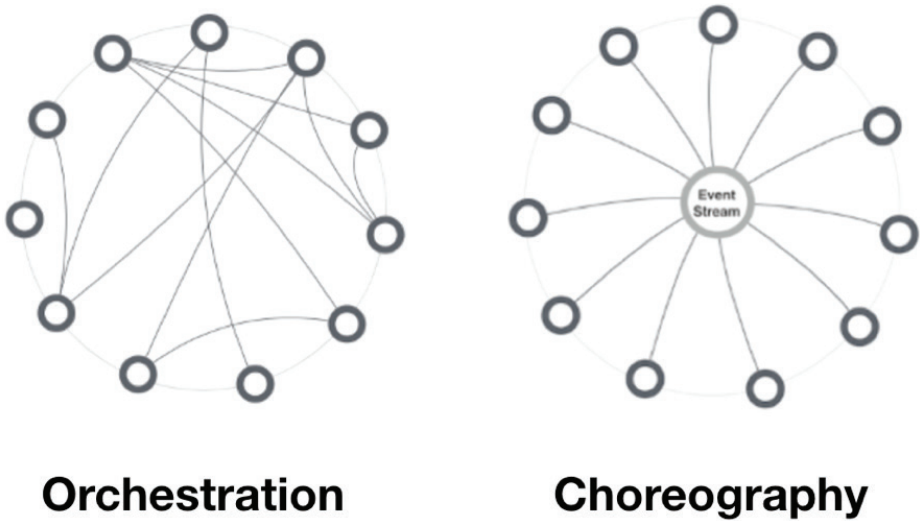
That is a bold statement, but obviously I couldn't agree more. The goal of this paper, after all, is to lay out the actionable steps you must take to realize the agility of event-driven microservices.

Think Event-Driven and Choreograph Service Execution

The first step towards adopting the event-driven mindset is to change the way you think about designing and architecting solutions. Initially the tendency is to think about all interactions between services as a series in a sequence of request/reply service calls. In fact, if you or your team uses the terminology of "invoking," "requesting," or "calling" then it is a sure sign you are still thinking in the paradigm of command style.

Instead, try these: "What events should my service process?" and "What events will my service emit?"

Once you adopt event-driven thinking, you need to make the shift from orchestration to choreography.



It is common for architects to think in terms of "service A will call service B which will call service C" and then implement that model through a chain of invocations (a->b->c) or by creating an orchestrator service such that x->a, then x->b, then x->c.

Both approaches will cause chaos when the realities of distributed computing kick in, especially when you start to scale.

Source: Gartner "Business Events, Business Moments and Event Thinking in Digital Business," 4 August 2017, Yefim N. Natis

The alternative is to follow the philosophy of choreography. Going back to the analogy of dance, services should react to changes in their environment just like a dancer reacts to musical cues. The benefits are immense:

- **Better agility**. Agile development teams are more independent and are significantly less impacted by changes to other services.
- **Services are smaller/simpler**. Each service is not required to have complex error handling for downstream service or network failures.
- **Less service coupling**. Services have no knowledge about the existence of other services.
- **Enables fine-grained scaling**. Each service can be independently scaled up or down based on demand. This ensures a good user experience and is less wasteful of compute resources.
- **Easy to add new services**. Due to less coupling, a new service can come online, consume events and implement new functionality without changes to any other service.

This litany of benefits doesn't come for free; there is no such thing as a free lunch.

Consistency of state then becomes an area of focus, because a service, being temporarily down, means that the event state changes may not be processed immediately. Fundamentally, how do we deal with this negative side effect?

Embrace Eventual Consistency

Eventual consistency is the idea that consistency will occur in the future, and it means accepting that things may be out of sync for some time. It's a pattern and concept that lets architects remove costly XA transactions from the mix. It's the job of the eventing/messaging platform to ensure that these domain change events are never lost before being appropriately handled by a service and acknowledged.

Some think the only benefit of eventual consistency is performance, but the real advantage is the decoupling of the microservices since individual services are merely acting upon events that they are interested in.

Databases + CQRS

The thought process regarding using events typically leads to an interesting question: If data is no longer the center of my universe, where do I now persist these events?

Databases take our thinking back to command (Create, Read, Update, Delete) style interactions. The database dilemma is extremely interesting, and using a pattern called Command Query Responsibility Segregation (CQRS) can provide large benefits. The key is that CQRS is not an architecture; it's a simple pattern that can help to enable event driven-architecture since, yes, the events at the center of our universe must be persisted somewhere.

Ultimately, that somewhere will be a database for most cases.

So what is CQRS, exactly?

Let's explore a trivial banking use case as an example of what we are talking about with CQRS. Traditionally we would have an Account service deal with all account interactions. Its API would be defined as:

```
AccountService
{
Public void createAccount(name)
Public Account getAccount(name)
Public void debitAccount(name, amount)
Public void creditAccount(name, amount)
Public AccountList getInactiveAccounts()
Public AccountList getOverdrawnAccounts()
}
```

The CQRS pattern simply separates this single service into two different and independently scalable services:

```
AccountChangeService
{
Public void createAccount(name, acctMetadata)
Public void debitAccount(name, amount)
Public void creditAccount(name, amount)
}
```

And

```
AccountReaderService
{
Public Account getAccount(name)
Public AccountList getInactiveAccounts()
Public AccountList getOverdrawnAccounts()
}
```

The execution of commands and queries is fundamentally different. For example, commands and queries are always scaled differently because they occur at different rates and have different overhead penalties.

So why should we combine event-driven architecture with the simple CQRS pattern, and what are the benefits?

Consider this: typically with databases you start with a data design and data models. The decisions in that design affect all upstream services and processing because it has to work within the constraints of the data model. This thinking puts data squarely into the center of the architecture rather than the event.

If you separate command actions such as create, update and delete from query, you essentially have segregated events which change domain state from the queries which do not. It is these events which move your thinking back into events being the focus for the architecture and is also the most natural way to model the domain.

The huge advantage is that you can easily siphon these domain change events into new microservices (easily adding new capabilities and features) or into the world of big data (where analytics can be performed and new discoveries made).

An example of a new feature for our bank example would be implementing a marketing campaign for a new bank-offered credit card. The use of EDA and CQRS makes it easy to consume account creation events and market the bank's credit card to the new customer based off the acctMetadata (e.g. opening balance, address, job, etc.).

What you do sacrifice with CQRS is consistency, but you enhance both performance and availability and, as mentioned earlier, embracing *eventual* consistency alleviates this concern for most use cases.

Integrate Event-Driven Architecture

The enterprise is chock-full of events and data stores that contain game-changing potential. It's really important that you not let the world outside of your event driven-microservices architecture pull you back into command-style interactions or worse than that (shudder) batches. This philosophy is straightforward for systems and devices which are already event-driven, but not so for database-centric systems.

One way of helping microservices coexist with the legacy world of data-centric systems is to implement change data capture (CDC) on the underlying databases. An example of this technique with Oracle databases is to leverage Golden Gate, a CDC utility. As events occur they are written to the database. By capturing these changes and treating them as events you can leverage them in our microservices platform. This eliminates any impact on existing systems, and the need to make costly code changes. Most of the work here is transforming CDC events into the domain data structure.

It's easier to integrate modern systems and devices. For example, IoT devices are inherently event-driven, social media platforms are stream-enabled, and even many JEE applications utilize message driven beans (MDB) which can easily be exploited. The key is to avoid allowing integrations to lead you back into the non-event-driven world.

Utilize Events to the UI

Now that you understand the power EDA can bring to backend microservices processing architecture, you should consider bringing that same power to the user experience.

Let's face it, while AJAX provides a user experience that seems asynchronous to the user, under the covers it's just polling web resources. If the polling interval is too long, user experience suffers and they cause unnecessary load on the system by attempting to refresh.

Conversely, if the polling interval is too short, the chance that an update has actually occurred is low and again resources are wasted. As more and more users make use of this web application, they will dramatically increase the traffic hitting these update services (most returning no new result) and cost the business more money.

A better approach would be to use an asynchronous protocol such as MQTT or WebSocket. The web application opens a connection, subscribes to the data the user wants, and waits for events to stream to the browser. User experience is better because events are arriving in real-time, and it will save your business money as bandwidth and compute resources aren't wasted on pointless requests.

Events as the Cornerstone of DR

Let's look at the airline industry for a prime example of struggling to respond to IT disasters.

Within the last few years, tens of thousands of passengers have been stranded not due to terrorism, geopolitics, or disease outbreak, but by the cascading effect of IT failures. In many of these cases, the system was designed improperly. In others, their disaster recovery systems were ill conceived or took too long to do their thing.

This relates directly to event-driven microservices. Events can be easily replicated to other active or disaster recovery sites so systems are always in sync and ready for action. Historically, this was done at a data store level where each database type used its own replication mechanism.

This was complex as it involved many components and different strategies. It was also really expensive because in many cases the same event was stored in multiple locations. A great side effect of designing your system around events instead of data is enabling your business to continue executing through potentially disastrous situations as though nothing had happened.

Use Vendor Agnostic Standard API or Wireline Protocols for Events

It may seem appealing to leverage cloud provider eventing/messaging offerings such as AWS's SQS/SNS, Google Cloud Pub/Sub, Azure Service Bus, or proprietary solutions for event transport like Confluent Kafka, IBM MQ, and TIBCO EMS. There are multiple problems with this approach:

- **Lock-in**. Problems arise when for technical or business reasons you want to move away from the solution. This requires significant investment to undo the mistake as nearly all services and integrations are affected.
- **Lack of APIs**. One of the key benefits of microservices is that it does not dictate what programming language is used. Proprietary APIs and wirelines typically only support a small subset of languages, thus limiting agility and choice.
- **Inhibited Innovation**. Another benefit of microservices is the ability to use new technologies and techniques as industries and products evolve. Being locked in to a vendor's proprietary APIs can mean that your locked out of innovation since the cost to switch is so high. Sounds like the problem with monolith applications!

When choosing an implementation for your eventing/messaging platform it's important to make sure it supports standard APIs like Spring, JMS, NMS, Paho and Qpid and/or wirelines such as MQTT and AMQPv1.0.

Overcoming the Barriers Between You and Event-Driven Microservices

EDA has tremendous potential, as long as architects and developers embrace the mindset required for success. The challenge is the lack of tooling to implement EDA in the context of microservices.

The tooling is not modern, performant, open or stable. It also cannot be deployed in the configuration dictated by business requirements such as on premise and/or in a variety of public clouds. They don't provide federation, synchronization or recovery across wide area networks (WANs). Finally, API management/gateway solutions are simply not the right tools for event-driven microservices. Here's why:

- **Lack of Modernization.** Messaging itself has been around a long time. Fifteen years ago, there was no agile development — waterfall ruled the day and messaging infrastructure was managed by specialized teams. Today, the concept of DevOps and its associated agility means that developers cannot wait for messaging infrastructure to be installed, configured and data made available. Everything must be automated, DevOps-friendly and self-service. This means that while messaging is the right tool, the tools are no longer compatible with the agility needs and requirements of today. This is magnified by the promises of speed, scale and the agility of microservices architecture.
- **High, Unpredictable Latency.** As monolithic applications are broken into discrete services there is a point where latency and performance will decrease, impacting user experience and your ability to satisfy SLAs. The need for the eventing/messaging platform to be as low latency as possible has direct implication on the ultimate success of the implementation. When this occurs, the initial reaction is to reverse course and construct larger, more monolithic, less reusable and agile services to decrease network hops and thus latency. This step completely undoes many benefits of microservices but, in many cases, the enterprise is locked in due to proprietary APIs and wirelines and therefore takes the path of least resistance.
- **Proprietary Protocols and APIs.** Years ago there weren't any standard wirelines or protocols, so IBM, TIBCO and other enterprise messaging companies implemented their own. Today, they do so because they (correctly!) worry that open standards will increase competition and enable their customers to easily abandon their product as competitors innovate and differentiate their offerings. Most vendors implement, at most, a single standard wireline or open API, thereby constraining the languages in which microservices can be implemented and excluding the larger ecosystem of events such as those from IoT devices. This reduces the overall value and agility that microservices provide and results in integration shims intended to unify data movement.
- **Poor Stability.** No service will ever achieve 100% availability and no system will operate perfectly at all times. Bugs within service, network outages, and poison messages are all real examples of how services can slow down or tax the microservices ecosystem. When these off-nominal events occur, we as architects expect that our foundational services, such as messaging, weather the storm, buffer the requests, and ensure no data loss. It turns out the reality is that many messaging systems work well until these scenarios occur. In other words, they let us down when the going gets tough. Ironically, this is where we needed them the most!
- **Deployability Risk.** The cloud providers all have messaging/eventing platforms that can't be deployed in their competitor's clouds or on premises where many businesses maintain core business functions. Legacy messaging systems cannot easily be deployed into cloud environments and some simply cannot because of their use of multicast. Deployability is an important consideration given today's climate of ever-evolving cloud strategies. The pain to migrate can be large when the decision to move towards hybrid and/or multi-cloud architectures occur.

- **WAN Weakness.** The use of hybrid cloud and multi-cloud architectures are extremely common for both economic and continuity of operations. All of these strategies require events to be propagated across WANs in a reliable, secure and performant way. Since legacy messaging products were originally used in datacenter environments, they respond poorly to the common WAN attributes of jitter and slow round-trip times.
- **API Management/Gateways - Wrong Tool.** The API management/gateway space is booming. This component is necessary because in many cases B2B and web applications need to integrate using web-friendly APIs such as REST/HTTP. These interactions, where they represent domain change events, must be quickly transformed into events for downstream processing. API management and gateways simply do not attempt to enable the interactions between event-driven microservices, nor do they enable many asynchronous protocols for streaming events to web applications.

Conclusion

While event-driven microservices may seem difficult initially, they are the future of most microservices and IT strategies. Choosing the right eventing/messaging platform is one of the most critical steps in the path to realizing the vast benefits of microservices.

Architects and developers simply need a platform that has been engineered to thrive in today's modern, rapidly-evolving world. Engineered as a modern eventing platform, Solace can be deployed in every cloud and platform as a service, and can easily span WANs. It supports DevOps automation and provides an almost completely self-service experience for developers. Solace is the only stable and performant solution that fits the unique needs of microservices architects.

About The Author

Jonathan Schabowsky is a senior architect in Solace's Office of the CTO. His expertise includes architecting large-scale, mission critical enterprise systems in various domains, such as for the FAA, satellite ground systems (GOES-R) and healthcare. Recently, Jonathan has been focused on the use of event-driven architectures for microservices and their deployments into platform-as-a-services (PaaS) running within public clouds.

About Solace

Solace develops the world's leading, enterprise grade message brokers, deployable as an appliance, software and as a service. Solace message brokers provide a single messaging layer for all your apps and microservices, in all your environments, enabling performant, stable and secure data movement across your hybrid cloud. Learn more.

A few of our customers

Our featured partners

linkedin.com/company/solacedotcom