DIVIDE AND CONQUER:

An Overview of Microservices in Financial Technology

PROVENIR

DIVIDE AND CONQUER:

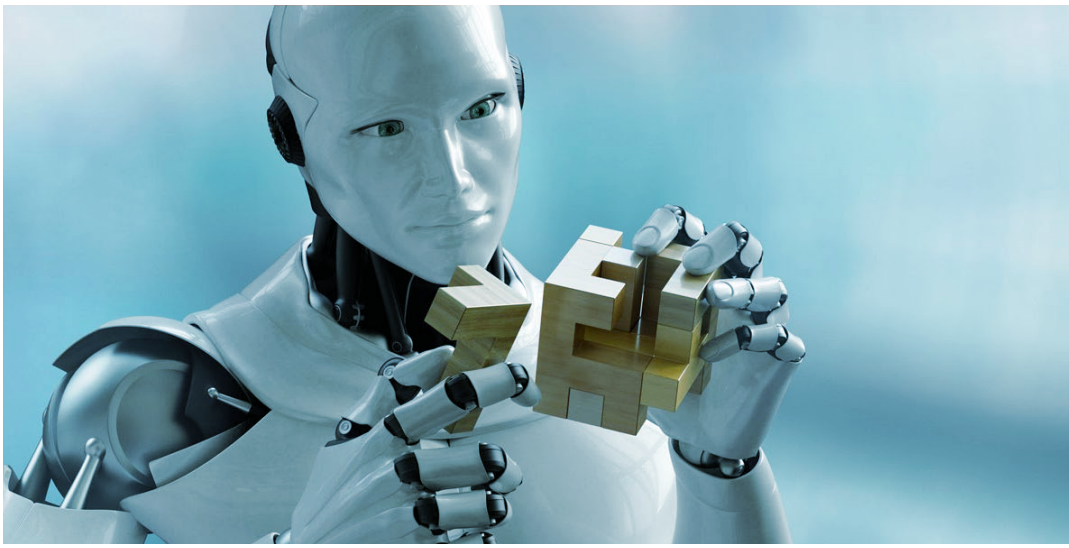
An Overview of Microservices in Financial Technology

The intention:
software systems become more easily evolvable, scalable, and maintainable.

The Microservices concept is becoming an increasingly popular architectural pattern in modern systems. A cult-classic on the back of Netflix's widely-promoted work within the concept, it has become the new-tech poster child for organizations – from growth-stage startups to the enterprise – who are seeking more agility, scalability, and independence.

The notion, closely related in concept to Service Oriented Architecture (SOA) that was made popular by a Wells Fargo case study in the 1990's, is often presented in contrast to a monolithic architecture. That is to say, Microservices takes what could have traditionally been a single monolithic application and decomposes that application into multiple, loosely coupled, and autonomous services. The intention: software systems become more easily evolvable, scalable, and maintainable.

While microservices are infiltrating organizations of all kinds, this paper will focus on microservices within the financial services industry. We discuss the strategic challenges facing the sector and consider how an architectural shift to microservices could play into its evolution. We will also evaluate the challenges involved in architecture transformation, specifically from Microservices point of view. Are they the easy win they are said to be or do they require more substantial operational and organizational changes along the way?



THE FINANCIAL SERVICES OUTLOOK

External Challenges

Financial services is a highly competitive industry that is riddled with high barriers to entry. Challengers historically found it difficult to break through drastically low margins and tightening regulations. However, large enterprises that once dominated the market are now facing disruption from smaller, leaner fintech companies that are eating away at the value chain one discrete bite at a time. Typically marked by technological agility, specialization, and customer-centric UX, new challengers in financial services present a serious outside threat to established players. In fact, PWC Global has estimated that 28% of all incumbent Banking and Payments business is at risk in the wake of the FinTech movement.

Internal Challenges

This shift leads us to the realization that legacy presents an increasing risk to these large organizations. Comparably ancient technologies and rigid software architectures can make maintenance and change over time costly and extremely challenging. Management of these architectures is further complicated by high acquisition activity in the industry (the silver lining of that FinTech movement we mentioned), thereby introducing incompatible systems. And then, there's the rise of multi-channel financial services that presents a whole slew of additional integration trials.

To remain competitive, financial services firms are reconsidering cumbersome architectures and transforming them into something more adaptable. In fact, a recent survey of financial institutions found that ~85% of decision makers consider their core technology to be too rigid and slow. Consequently, ~80% are expected to replace their core banking systems within the next five years. Could this be where microservices plays the *Deux es machina* of the banking saga, swooping in to save the day?

WHAT ARE MICROSERVICES?

For those of you who are just catching up, let's take a step back for a second and set the stage. Microservices are an architectural pattern in which a software system is split up into distributed services. These services tend to be autonomous, loosely coupled, and independently deployable, each running a unique process to serve a business outcome. Whereas a monolithic architecture crams an entire application into one codebase, a microservice architecture tends to distribute by business functionality. For example, functionalities like credit checks or funding could live in different microservices, each developed and managed by a different team.

Think of it this way:
If a monolithic system is akin to kids riding a school bus, microservices are super-children riding unicorns.

THE ADVANTAGES OF MICROSERVICES

A Microservices approach comes with many benefits over the monolith. And, while microservices are far from a silver bullet (we will touch on the challenges in the next section), the architectural style can promote agility and speed to market for those who do make the transition responsibly.

Let's take a look at some key advantages of microservices:

- Independently Deployable
- Resilient
- Scalable
- DevOps Oriented
- Polyglot

Illustrations sourced from Freepik

By breaking an application up into microservices, your teams can develop and release each service independent of another.

Independently Deployable

By breaking an application up into microservices, your teams can develop and release each service independent of another. This is a big deal when it comes to product or feature roll-outs. Teams no longer have to treat the software system as an unwieldy whole whenever they want to roll out a new feature.

In the risk field, this notion behaves the same way. A data scientist interacts with a risk model in the same way a developer might change a login service. The data scientist can make changes to models or business logic, and those changes are validated, tested, and pushed back into the application.

If you're starting to get the idea that microservices also means organizational change that more closely aligns your development teams with your operational teams, you're right. We will touch on that in the aptly named 'DevOps' section.

Resilient

Because a microservice is autonomous and loosely coupled, the failure of one service tends to happen in isolation of the rest of the system. The logic goes something like this: In a monolith, everything exists on one "circuit" (think of it as a string of those old fairy lights). If something fails, the entire system goes down. However, microservices exist like independent, battery-operated light bulbs. If one goes out, it has no impact on the remaining lights. This concept of fault tolerance is a huge driver in financial services, where downtime can have detrimental reputational and regulatory consequences.

Consider this in a banking application, where a mission critical piece of functionality like payment processing would continue to function, even if something unrelated like a credit check microservice failed. Traffic can then be routed to a new instance because microservices tend to be stateless by design.

A microservice based system offers unique scalability because parts of an application scale independently.

Scalable

If microservices receive praise for nothing else, they have undoubtedly been hailed as the liberator of scalability. A microservice based system offers unique scalability because parts of an application scale independently. If there is a heavy load on a payment system, for example, it can be scaled on its own leaving the rest of the system untouched.

By contrast, the monoliths of old tend to be stateful. Thus, scalability issues are solved:

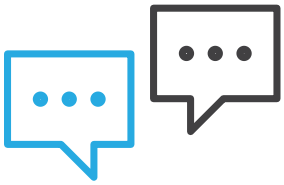
1. Vertically, by throwing more compute resources at them, which ends up becoming expensive and limited in success.
2. Horizontally, by making use of sticky sessions, something else which can be quite brittle in the event of failure.

Stateless microservices don't encounter either of these challenges. In fact, it becomes very trivial to scale as needed horizontally.

DevOps Oriented

Conway's Law promotes the idea that a team will produce a design reflective of its communication structure. Depending on your present organizational structure, this could be a lesson or a warning. That is to say, if your developers don't mingle with operations outside of the annual company party, you're in for some challenges.

The microservices-related advantage here lies in the organization of teams around business functionality rather than technical concerns, leading to more efficient application design. Remember when we talked about the Data Scientist who pushes R models like a developer pushes Java? In a DevOps or BizDevOps model, that Data Scientist could sit, stand, or walk on a treadmill desk next to your QA engineer. This cross-functional team structure is breaking down the silos that are all too familiar with financial services, and are widely known to cause inefficiencies and lackluster customer experience.



Polyglot

Because a monolith has a single codebase, it also has one language. Introducing a new language, a challenge that many an acquisition onboarding process has encountered, would be a daunting task that could result in significant migration work. However, with the microservices approach, services can be authored in various languages. For example, you might write domain centric services in a language like Java, and network or system services in a language like Go. If they need to communicate with one another, they do so via language-agnostic means such as HTTP.

The same principle also applies to persistence models. While a monolith is backed by a single database, typically relational, microservices often leverage a single database for each service. This opens up exciting possibilities for NoSQL in a climate where financial institutions frequently make use of unstructured data for risk-related decisioning, or where storing credit bureau data in a data lake could introduce tremendous cost savings.

OVERCOMING THE CHALLENGES OF MICROSERVICES

Although vastly beneficial in the right environment, microservices are not a silver bullet. In fact, they add a lot of additional complexity over the monolith. Let's look at some of the key challenges to consider and highlight some best practices to work around said problems:

- The Distributed Monolith
- Cultural Resistance
- Regulations
- Monitoring
- Deployment
- Distributed Tracing

The Distributed Monolith

The distributed monolith, while a problem of execution rather than concept, is often seen anti-pattern in microservice. This occurs when microservices are so tightly coupled that they must always be deployed and tested together as a whole. For example, if you have to run a slow and time-consuming set of end-to-end tests to validate a change to a single service, then a lot of the benefits of microservices are negated.

Compound this problem if that single change causes a ripple effect, requiring subsequent changes to multiple downstream services. The two most important strategies to avoid this are:

1. Clearly Defined Service Boundaries

By encapsulating a business functionality within a single service, you can define a boundary that reduces network hops and dependencies on other services. Reducing dependencies, thus reducing network hops, makes it easier to reason about and test a change before releasing it.

2. Consumer-driven Contract Testing

When dependency between services is critical, consumer driven contract (CDC) testing reduces reliance on end-to-end tests, leaving them as basic user journeys. CDC works a lot like test driven development (TDD) but at the architectural level, allowing you to validate functional collaboration between services through their independent component tests. And Voilà! You have shaken that dependence on the end-to-end testing tier.

"It's about the people as much as it's about the tech."

– DANIEL BRYANT
7 DEADLY SINS OF MICROSERVICES

Cultural Resistance

In his smart *7 Deadly Sins of Microservices*, Daniel Bryant says "It's about the people as much as it's about the tech." As such, one often overlooked consideration when adopting microservices lies within the organizational culture. Culture clashes are nothing new in technological change, and many an article stands on the soapbox of change management in software and system implementation. Since change management is a white paper in itself, suffice it to say that heightened tension exists in firms where the status quo equals an organization's identity.

Before moving to microservices, plan and conduct a formal change management process to avoid any unnecessary cultural obstacles. Start by asking yourself: Is there any skepticism or resistance to microservices? Do the architects believe it makes sense, and do the business "get" the advantages to the extent that they're actively supporting the shift? Will the organization buy-in to the structural changes that happen in support of a microservices architecture?

Regulations

Because microservices tend to be elastic and ephemeral, they are naturally suited for cloud computing where resources are programmatically provisioned on demand. However, certain financial data tends to be strictly regulated, making public cloud storage tricky. Whether by external or internal regulation of financial institutions, this produces a regulatory blocker that prevents a software system from following a cloud-based microservice architecture.

Need an approach to get around this? Simple; do not store the data. For example, if you are developing a cloud based microservice system that integrates with a core banking system, then data ownership can lie with that core banking system. This data-as-a-service approach means that, as long as the data is never copied, then the microservice based system will also be compliant.

You can take advantage of a user interface (UI) that visualizes your microservice application to watch that automated masterpiece do its thing.

Monitoring

If you have ever seen a shell game play out in a heist movie, you can relate to the monitoring challenges that microservices can present. When dealing with monolithic applications, you typically interact with one deployment and one database. In the movie, this is the jewel thief in the red jacket fleeing the museum; he's easily identifiable. In the same way, monitoring the monolith is simple, with only a few hosts and log files to analyze.

With the microservices architecture, there tends to be many services and databases that are elastic, scaling and disappearing as required. This is where hundreds of jewel thief decoys in red hoodies flood the streets, presenting more red jackets to complicate the scene. In microservices, it becomes harder to keep tabs on everything that is happening within the system and to pinpoint where something is breaking.

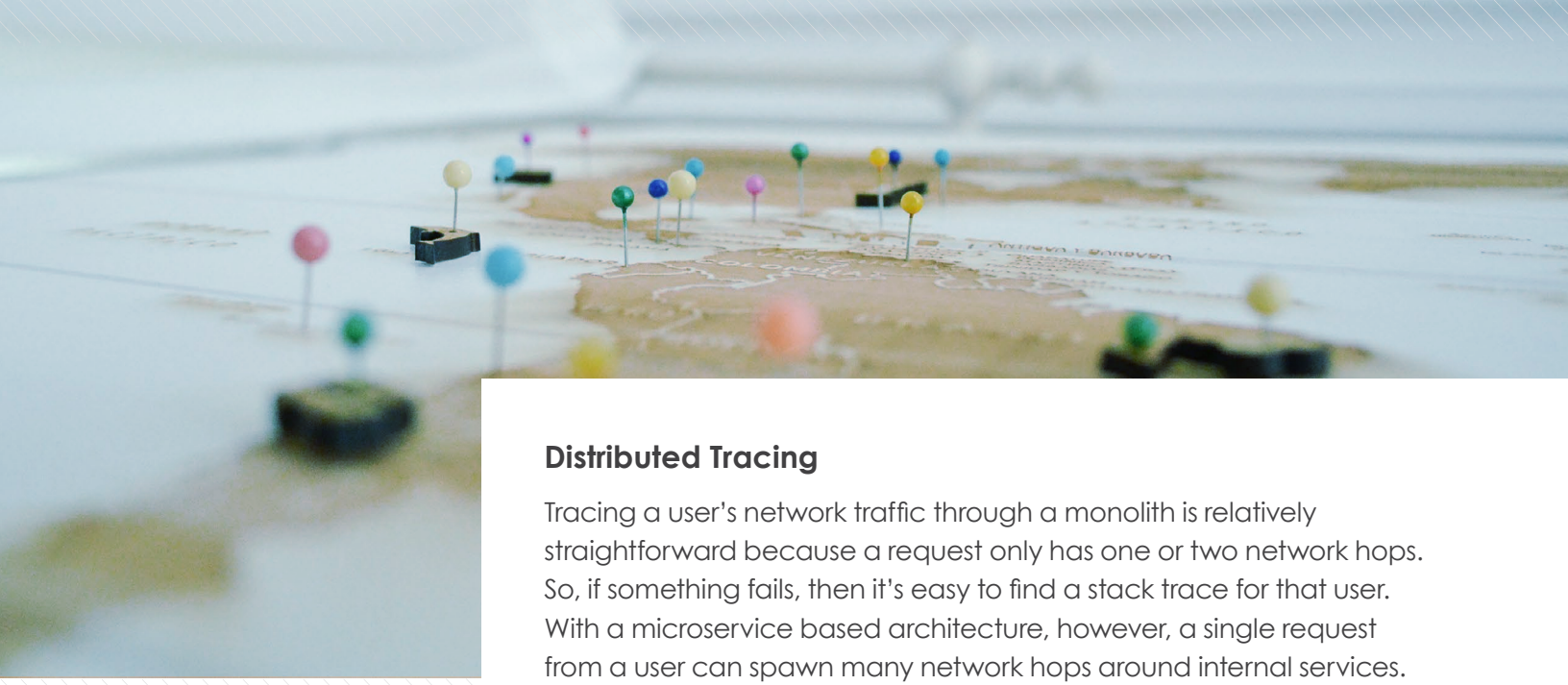
Monitoring is where leveraging the proper tools becomes necessary. If you can automate monitoring, logging and the aggregation you can significantly simplify the process. Even better, you can take advantage of a user interface (UI) that visualizes your microservice application to watch that automated masterpiece do its thing.

Deployment

The microservices approach presents many different types of services, often written in different languages and with different dependencies. Various services may require different versions of Python, some might be in Java and be dependant on the JVM, and some might be a simple Go binary.

Though we've all met a birthday party magician who could prove otherwise, every coin has two sides. This is microservices' double-sided coin in that the best-suited technology for a service can be used (advantage), but that each of these must be installable and runnable in an environment (challenge).

This is why container based technologies such as Docker have become so popular. They act as a wrapper around an application, containing libraries and runtime settings as if a dedicated host. Now, developers can ship a fully executable container, simplifying deployment to a single operational concern.



Distributed Tracing

Tracing a user's network traffic through a monolith is relatively straightforward because a request only has one or two network hops. So, if something fails, then it's easy to find a stack trace for that user. With a microservice based architecture, however, a single request from a user can spawn many network hops around internal services. Therefore, tracing a user through the system requires navigating loads of different log files and piecing them together.

To avoid this headache-inducing exercise, dig into the use of correlation IDs for user requests. Correlation IDs identify a cohesive journey across all internal services and allow distributed tracing tools to monitor these journeys visually.

CONCLUSION

The microservices concept continues to gain popularity, showing itself as a clear means to accelerate software evolution. In financial services, the agility itself answers many of the industry's questions around digital transformation and scalability in the face of constantly changing customer, business, and regulatory requirements. Though firms are likely to experience increased operational overhead during the shift to microservices – particularly in areas such as building, deployment, and monitoring – these challenges can be mitigated over time with the right planning, tools, and patterns.

Any architectural change is subject to the needs of a particular organization, but microservices are a sensible choice for financial services firms who are seeking structural reform to solve real business problems. Responsible adoption could mean distinct flexibility for the long-haul.

MODERNIZE YOUR RISK SOFTWARE WITH PROVENIR FOR MICROSERVICES

For organisations that have adopted a Microservices style of architecture, or are looking to do so, Provenir supports the decomposition of business processes providing the capability to develop and expose business functions as discrete services.

WWW.PROVENIR.COM

CONTACT US TO LEARN MORE