nglogic

Microservices under X-Ray

Project Management, Technologies & Case Studies

How Top Companies Utilized Decentralized Microservices Development to Accelerate Software Products Delivery

Case Studies Included

AUTODESK · American Red Cross · singlecare · NewsDirect · Raise

Microservices under X-Ray

Project Management, Technologies & Case Studies

How Top Companies Utilized Decentralized Microservices Development to Accelerate Software Products Delivery

Case Studies Included

(Autodesk/AmericanRedCross/Raise/SingleCare/NewsDirect)

First publication, 2020

Publisher:

NG Logic LLC
400 Concar Dr
San Mateo, CA 94402

info@nglogic.com
+1 (888) 413 3806
https://nglogic.com/

Table of Contents

Introduction

Adrian Cockroft put Netflix on the map as a technology company after he implemented new and dynamic software architectures that were later (in 2011) coined as “microservices”.

> ***“You asked me what I’m most proud of. I think it’s, basically, that you can’t mention ‘cloud’ or ‘microservices’ or whatever without mentioning Netflix at some point now.”***
>
> *Adrian Cockcroft*
> *(former Netflix cloud architect)*
> *in an interview.*

While still a relatively new type of software architecture, microservices have gained popularity quickly and have been widely adopted by industry leaders such as Amazon and Uber.

By 2026, it is estimated that the global microservices market size will reach over $3 billion.

Microservice architecture makes businesses faster and more agile by structuring applications as collections of services that are reliable, independently deployable, and highly cooperative.

As Adrian Cockroft puts it:

> *"It's very hard to find well-written monoliths. Most of them are tangled balls of mud with all kinds of disgusting things going on inside that are broken in very odd ways that are hard to debug."*
>
> *Adrian Cockcroft*

*"But how can microservices help **my company** become a market leader?" you might ask.*

Well, here are some examples:

American Red Cross

- ✓ ARC is using Volunteer Connection, a large and complex web system, as a centralized place to manage all ARC volunteer tasks (including intake, training, and tracking opportunities and monetary contributions).
- ✓ NG Logic has partnered with Digital Cheetah, the vendor of the volunteer management system, to integrate their system with internal ARC systems (e.g. Hemasphere, RCView) as well as 3rd party SaaS providers (e.g. ClassMarker, SurveyMonkey, Cornerstone) and others.
- ✓ The integrations were implemented as microservices deployed on AWS cloud and developed in Python language. Each service has a back-office UI and uses internal REST API to communicate with the monolithic volunteer management system.
- ✓ With time, the ecosystem of microservices evolved: now it covers more functional areas and integrates tightly with the monolithic frontend. NG Logic developed a dedicated Single Sign On solution that allowed users to seamlessly navigate across monolith and microservice end user pages. Following that, NG Logic started to gradually migrate functionality of the monolith into the microservices architecture.
- ✓ Introduction of microservices in this case was a total success: it allowed multiple teams to work in parallel on the same product, new modules could be delivered two times faster than before, and the number of production issues dropped by 30%.

Autodesk

- ✓ NG Logic was engaged in a close cooperation with the Autodesk Education Community department and completed several projects, such as standalone web applications that integrated via TIBCO with the rest of the Autodesk eco-system.
- ✓ In 2017, the department planned a move to microservices architecture as part of a larger initiative within Autodesk. NG Logic was part of the transition process as subject matter experts in areas that NG Logic developers previously built web apps for.
- ✓ As part of the effort, NG Logic developed two microservices that consumed external APIs and exposed internal interfaces for other microservices. The services were related to school information management and license information management.
- ✓ The frontend of the education community app was integrated into a single web application and leverages NG Logic's microservices to perform backed tasks.

In the next chapters, we will take a deep dive into the "*microservices*" topic.

Here is a sneak peek:

- Microservices in a nutshell: overview, advantages and disadvantages
- Microservices design
- The right architecture
- Outsourcing to CEE
- Geographically distributed microservices development: detailed case study
  - Raise - developing and implementing a microservice-based architecture, creating new functionalities across the existing monolith
  - Singlecare - new microservice architecture leveraging Golang backend services, React frontend, open source database engines, and Kubernetes
  - Newsdirect - benefits of the microservices architecture without needing to invest in the overhead required for multiple microservices management
- Recommendations

Microservices in a nutshell

Overview

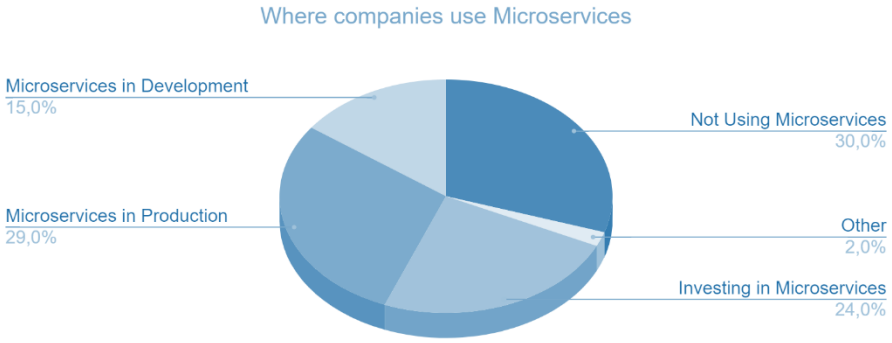
Microservices are an architecture pattern used in software development to build systems comprised of independent units. Microservices are commonly used to decompose a monolithic application into multiple independent services. The services are loosely coupled and can be developed and deployed independently and continuously. Each microservice functions as a separate piece of software that can be updated without impacting the rest of the system. This flexibility gives organizations the ability to adapt to changing market conditions and rapidly scale their products.

Microservices interact with each other via asynchronous communication, which enables the communication from one microservice to be sent to multiple receivers. This is important because it minimizes the communication between microservices and improves the resiliency and response time of the system, resulting in a better user experience.

Advantages of microservices

Scalability

Scalability is one of the greatest advantages of microservices. Since each microservice is a separate element, different components of an application can be scaled independently. This leads to better performance and increased development velocity compared to the monolithic approach, which requires the entire system to be updated every time a change is made.



SOURCE: APP. DEV. SURVEY

Microservices are ideal for large systems that utilize multiple platforms and devices and require frequent updates to meet market demands.

Business scalability is a key quality that many investors look for in start-up companies. A well designed, highly scalable company with ideas based on detailed market research attracts investors and brings capital.

A well-rounded product with well-designed scalability potential is a crucial part of every tech business model. At the core of every piece of software is its architecture, which should allow the product to easily and seamlessly reach its full potential. In many cases, microservices are the best way for start-ups to achieve scalability for their products and overall business models.

Failure isolation and resilience

A system with a microservice-based architecture is more resistant to failures. If one microservice fails, it does not lead to an entire system failure. Instead of bringing down the entire system, developers can isolate the issue while other services remain online.

Additionally, developers can build and deploy updates to applications without having to bring down and change the entire application. This results in a better user experience.

Faster deployments

Microservices work independently, so it is not necessary to change the code for an entire system to modify a

specific feature. Developers can change, test, and deploy individual components without impacting the entire system. This allows an application to be continually updated.

New features can be deployed quickly, and product time-to-market can be reduced. The ability to continually update and deploy new versions of a system/product is invaluable for companies who want to keep up with competitors and with the changing demands of a large user base. Fast deployments also enable quick bug fixes, which is critical for maintaining a good user experience.

Flexibility

Microservice-based systems are flexible; developers can easily add, remove, reconfigure, rearrange, rename, or replace functionalities on the fly. This is particularly important for large systems and results in better dependency handling. It is also easier to make changes with less risk of regressions.

Therefore, developers have the flexibility to try out new technologies on individual services; a service utilizing a specific programming language or technology can coexist with other services utilizing different technologies.
Microservice architecture also enables organizations to store data in multiple locations, which makes it possible to select custom storage types that best support each service.

Easy to maintain and understand

Applications are easier to build and maintain when they are split into small, individual components. Each microservice can be treated as a separate element with its own set of code. This enables more efficient resource allocation management and is time-efficient since multiple teams can work simultaneously on a single software project without stepping on each other's toes.

Managing small pieces of code is far easier than managing one massive piece of code, especially when dependencies are involved. It is easier for a developer to understand the functionalities within a system when each service is a separate component.

Microservice-based systems are also easier to maintain from a security perspective; since each service is isolated, security threats are typically easier to track down and do not pose a risk to an entire application.

Disadvantages of microservices

Complexity

Microservices are more complex than monolithic applications because they are comprised of many

components, each of which is a separate moving part. Each microservice has its own set of code, which means that multiple programming languages, tools, and technologies may be at play within the same system. Additionally, communication between microservices must be considered. The scale and complexity of a microservice-based system can rapidly rise with the addition of messaging middleware, separate databases, interfaces, etc. The complexity of microservice-based systems makes them more challenging to implement.

Expensive

Due to their complexity, microservices are often more expensive than monoliths. It takes more time and expertise to develop apps in a microservices architecture than in a monolith, so staff costs are higher.

Developers experienced in microservices are typically more expensive than regular developers because of their specific and advanced knowledge set. Depending on the project, multiple developers may be needed if expertise in different technologies or programming languages is required.

Microservices also require more work and expensive staffing from a maintenance and operations perspective. After a microservices architecture has been implemented, it requires maintenance by a site reliability engineer who

has experience with microservices orchestration systems such as Kubernetes, Docker Swarm, or Mesos.

Advanced solutions for monitoring, measuring, tracing, and debugging the system are also required. The increased cost of developers, increased time to develop applications, and need for more home-based staff are the primary factors that drive up the cost of microservices. Additionally, communication between microservices can marginally increase processing costs because each service requires its own CPU and runtime environment.

Require organizational maturity

Microservices are commonly used by organizations with large systems and mature management strategies; they can be difficult to implement and too complex or costly for start-ups who need to quickly get a minimum viable product (MVP) to market.

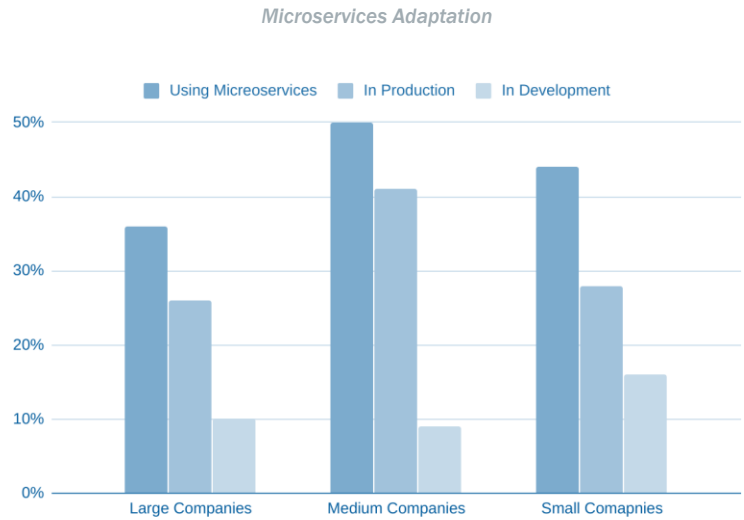
The microservices approach is most successful when an agile culture is in place that enables frequent communication and collaboration between multiple teams who are responsible for different services. The agile framework helps effectively manage the inherent complexity of microservices and ensures that developers understand the system as a whole.

Advantages	Disadvantages
Scalability	Complexity
Failure isolation and resilience	Expensive
Faster deployments	Require organizational maturity
Flexibility	
Easy to maintain and understand	

Microservices design

While there is no formal definition of what microservices design entails, microservice-based architectures do have some common characteristics.

First and foremost, services should be designed to function as individual units that can be replaced and upgraded independently; each unit should have a single, defined purpose. Services should be formed based on business needs and capabilities such as inventory management, user management, and delivery management.



SOURCE: APP. DEV. SURVEY

When designing services, developers should carefully define service boundaries and determine protocols required for communication between services. Typically, communication between services is achieved via REST

web services API calls. Decentralization also tends to be a key feature of microservices design.

Assigning a team to develop, deploy, maintain, and support each service is often one of the most successful approaches to microservices, especially when development teams are geographically distributed. Most approaches to microservices design are based to some extent on Conway's Law and domain driven design (DDD).

Conway's Law

Conway's Law was developed by Melvin Conway in 1967 and states that "any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure".

In other words, organizations develop software and systems that match their internal communication/organizational structure. The law assumes that for software components to interface and function properly, coordination and communication must occur between the developers who are responsible for the respective components.

Conway's Law is observable in software development; it is common for the software structure to end up mirroring the communication channels within the software development teams.

However, a better approach is to structure the development teams to mirror the system domains of a project. It is best to ensure that the team structure matches the optimal system structure for the domains being modelled.

Teams that are working on the specific domain or bounded context should ideally be in the same contractor, office, time zone, or communication unit. In other words, teams

should be split based on how business domains are defined and divided rather than by other factors such as developer availability and historical splits.

If an organization struggles due to communication issues between domains, utilizing cross-functional teams could be a solution.

For example, assume Team A is responsible for the frontend of a system and Team B is responsible for the backend. If the frontend and backend have difficulty communicating with each other, it is likely worthwhile to move someone from Team A to Team B and vice versa.

However, changing teams mid-project can be challenging, especially if the teams are structured based on different types of technologies. These challenges can be avoided from the start by utilizing Conway's Law to organize teams by domain instead of by other factors such as technology.

Taking this approach will naturally result in cross-functional teams with shared domain experience. Teams are less likely to hand off problems to other teams and more likely to internally troubleshoot and resolve system communication issues.

Domain driven design

One can see the influence of Conway's Law in domain driven design (DDD), a general framework for software design that is commonly used to design microservices. It involves defining the structure of a system and defining design patterns used to create the domain model.

The first step of designing an effective microservice-based architecture using DDD is to analyze the business domain in order to understand functional requirements of the system. The output of this step should include a description of the domain. The business domain can be decomposed into subdomains if needed.

After the domain has been analyzed, bounded contexts of the domain(s) should be defined. This is necessary because an entire business model is typically too complex to understand as a single unit. Bounded contexts allow developers to mark conceptual boundaries and separate them.

Each bounded context has its own ubiquitous language and contains a domain model that represents a subdomain of the larger system. Each bounded context represents an individual system function and should be assigned to a single team. Within each bounded context, contents need to be defined.

To accomplish this, DDD concepts such as entities, aggregates, value objects, and domain services should be used. A detailed description of how to define bounded context contents is outside the scope of this book.

Lastly, results from the previous steps should be used to identify the best microservices for the system. Aggregates should be assessed as potential microservices. Aggregates that are good candidates for microservices are loosely coupled, derived from a business need, and have good functional linkage. Domain services may also be good candidates for microservices.

When identifying microservices, several factors should be considered. Each service should have a single responsibility; a single microservice should not implement more than one bounded context (though a bounded context may be implemented by multiple microservices).

Depending on the overall architecture, the development teams may want to split a service by function (backend for frontend, data access, intercommunication, etc.) or keep all aspects of a service in a single microservice.

When making this decision, it is important to consider if splitting will cause the resulting services to be overly chatty; if this is the case, it is likely better to leave the functions as part of the same service.

Each microservice should be small enough that it can be built by a small, independent team that is responsible for all aspects of the microservice, including business model, design, architecture, implementation, deployment, and maintenance.

It is also important to ensure that microservices are not tightly coupled. Building small, loosely coupled microservices that can be managed by independent teams ensures that the services can evolve over time to meet business needs.

The right architecture

Implementing properly designed architecture is an important factor for achieving success with microservices. Desirable design elements include separate data storage, loose coupling, eventual consistency, and asynchronous communication.

Small organizations that do not have the resources needed to maintain the complex architecture associated with microservices may opt to use miniservices, which offer the benefit of scalability with less complexity.

Separate data storage

Each microservice should have its own data storage that contains information relevant to the microservice. It is important that microservices do not share the same database because it results in tightly coupled services; if the design or data structure of the database changes, all of the services that rely on the database have to be updated.

Therefore, a common database should not be the solution for integrating features or services. The data model of a service is considered internal and subject to change; other services cannot rely on it to implement inbound or outbound flows.

Instead, properly designed APIs or materialized views should be used. Using the APIs that are well documented and part of the official interface ensures that changes to the service's dependencies will not break the functionality. If the underlying service does not expose a data view required for implementing the functionality, consider building a materialized view inside the service based on incoming events from the services that are depended upon.

Since events are also part of the official API and changes to the API are properly managed, this approach will not result in a tight coupling that direct database access would cause.

Loose coupling

Loose coupling of services is key in a microservice-based architecture; services do not need to be aware of the existence of other services, and services should not rely on the availability of other services.

The goal of loose coupling is to minimize unnecessary interaction between services and reduce the risk that changes made to one service will negatively impact or change another service.

Loose coupling is important because it enables failure isolation in a system; if one service goes down, it does not bring the entire system down.

Distributed transactions, or transactions that span over multiple services, should be avoided in a microservice-based architecture. Distributed transactions are a form of strong coupling at the data model level and can lead to an increase in system failures.

While failures occur at the individual service level, they are a part of a larger sequence of actions that impact the entire system. If a single service permanently fails, compensating actions should be emitted by the service to initiate error conditions and cleanups in other related services.

For example, consider an account set-up process that requires actions in three different services. If an action fails in one of the services but is completed in the other two, the failed microservice should emit events that inform the system that the account has an unrecoverable error and needs to be cleaned up.

There are several ways to mitigate the risks of distributed transactions, including designing without the need for transactions, two-phase commit (2PC) protocol, and saga patterns. Designing a system without the need for transactions is effective, but not always achievable.

The 2PC approach utilizes a coordinator for distributed transactions.

- *In the first phase of 2PC, each service that needs to contribute data records its data to a log. If the service is unsuccessful, it emits a failure message; if it is successful, the service emits an OK message.*

- *The second phase of 2PC does not begin until all services respond OK. After this occurs, the coordinator sends each service commit instructions. The services respond to the coordinator when the commit has been successfully implemented. If a service fails, the coordinator instructs all services to roll back the transaction.*

Saga patterns are architectural patterns used to execute transactions involving multiple services. Each service in a sequence of transactions (i.e. saga) is dependent on the successful execution of the previous service/transaction. If a transaction fails, the saga will execute a compensating transaction to mitigate the failure.

Eventual consistency

Keeping distributed system data consistent at all times is often a complex task that comes at the cost of system performance and resiliency. To mitigate this risk, the microservices architecture should embrace eventual consistency to achieve high system availability.

The eventual consistency paradigm is very different from what developers and businesses use when working with monoliths. In a monolith, user actions or state transitions are implemented as single transactions that are applied after a process has finished successfully; transactions are rolled back in the event of an error. Once the transaction is committed, the effects of it are immediately visible for all other components of the system. This approach is called strong consistency.

In contrast, when using eventual consistency, developers cannot assume that the effects of a transaction are immediately visible throughout all the microservices. The support for eventual consistency needs to be built into the application from the start.

For example, if a user initiates an action in the Reporting microservice that results in a document being created in the Repository microservice, the system cannot assume that the document will be created right away. Instead, the request to create the document will be queued, and a

handle could be used to refer to the document while it is being created.

This process should be reflected on the frontend part of the application, e.g. by a spinning wheel icon next to the document.

Asynchronous communication

Event-driven asynchronous communication is the main mode of communication that should be used between microservices. This type of communication is facilitated by events that are published to a common event bus. Microservices emit messages that reflect their internal state, and other system components may act upon these messages. The messages are considered part of the API, and any changes to the API are governed by strict rules.

Synchronous calls (e.g. GRPC, REST) should be avoided and reserved only for flows that require immediate feedback, such as an acknowledgement or passing back a result. Synchronous communication is not desirable because it creates strong dependency between microservices.

For a service to function, another service must be online, not overloaded, and available via a network link. Asynchronous communication is the cornerstone of resiliency that microservices bring to the table. It enables failure isolation and minimizes impacts to the system; only business functions related to the service are affected. If one module goes down, events are queued and processed when the module is fixed.

Miniservices

A pure microservices approach calls for implementation based on the single responsibility principle; each microservice should be responsible for a single business function. However, this may lead to a rapid increase in the number of services, events, and compensating actions the system needs to handle.

It will also lead to an increase in infrastructure and monitoring complexity. If a company is not yet ready to handle the complexity of an entirely microservice-based system, a miniservice approach could be a good solution that offers the benefit of scalability with less complexity.

The miniservice approach conforms to the same architectural guidelines as microservices in terms of intra service communication. However, it decomposes a

system into fewer services, each of which covers more scope (e.g. multiple bounded contexts) than a microservice. Less services results in a less complex system.

The technologies behind microservices

In the early days of microservices adoption there were not many tools and technologies that addressed the issues that developers face with microservices development. Soon new ideas, frameworks, stacks, and tools started to flourish; most were conceptually good, but not stable enough for serious production use. The APIs and interfaces lacked widespread community support and were changed too often, which made upgrades a pain.

Subsequently, the task of choosing the right technologies for a new system became an important part of project planning. Extensive experience was required to avoid pitfalls, such as vendor lock-in or the need to support abandoned projects in-house. Fortunately, the market is more mature now and even though the industry is constantly innovating, there are technologies that have been battle tested and are solid choices for projects. However, each project is different and the tools and technologies always need to be evaluated and chosen based on the specific project requirements, not generic recommendations.

Containers and orchestration

Docker has become a de facto standard for managing dependencies, configuring, and deploying applications. Its low footprint compared to running virtual machines along

with a quicker turnaround time make it a straightforward choice. As microservice-based systems consist of multiple components, Docker helps keep all the configuration details and dependencies under control. An added benefit is that almost all developers are familiar with Docker, so no additional time is needed to introduce the technology; teams can be immediately productive.

Once all the components are packaged as containers, a container orchestrator comes into play. The purpose of a container orchestrator is to keep track of the container lifecycle, distribute the workload, and manage the platform from a single point. Kubernetes is quickly becoming a widely adopted standard for managing containers and infrastructure, and it plays an important role in microservices projects. Kubernetes allows developers to easily set up and run a local development environment and multiple testing/staging environments by reusing much of the configuration (sometimes augmented by infrastructure provisioning tools like Terraform, Cloud formation, or Rancher). This significantly reduces the time developers and admins need to spend setting up the environments and resolving issues.

A service mesh is a technology specific to microservice-based systems that enables high-level concepts like discovering services, securing communication channels, balancing advanced loads, and observing traffic, transactions and workloads. Istio is an increasingly used service mesh, despite the high learning curve and stability

issues it used to have some time ago. Utilizing a service mesh brings clear benefits to the engineers, because they do not need to reinvent the wheel nor include the code performing service mesh tasks into each microservice.

In addition, further data processing and visualization tools make a big difference for developers and site reliability engineers. Products like Prometheus, Kibana, and Grafana allow them to gather metrics from various sources, display them on dashboards, and analyze them in a convenient way. Such products can be augmented or replaced by cloud hosted solutions like Datadog, Newrelic, and others.

Development Technologies

Go

Golang (Go) was created by Google as an infrastructure management language around 2007 and has gained a lot of traction as the tech market transitions to cloud and microservices architecture. While discussing details of the language is outside the scope of this book, the following is a condensed list of its advantages that exhibit why it is a good choice for microservice-based systems and why it makes a lot of buzz in the community:

- ✓ Compiles by default to one binary in order to avoid dependency hell, makes deployment less complex and platform teams happy
- ✓ Compiled to native code instead of bytecode, which enables performance comparable to the fastest languages out there (e.g. C++) and results in reduced costs on cloud environments
- ✓ Developer friendly (built-in garbage collector, fast compile cycles, built-in web server and REST API framework), so the developers can work efficiently
- ✓ Simplicity: typically, there is only one way to implement a specific concept in the language. This makes handovers of code (e.g. to replace a developer) much easier compared to other languages that impose less discipline on developer
- ✓ Build-in efficient concurrency concepts which makes leveraging modern CPUs with dozens of cores easy.
- ✓ Supported and financed by a large organization (Google) and backwards compatibility with previous versions
- ✓ Very good support for testing code, which makes it easy for developers to test code

GRPC

GRPC/Protobuf is a protocol and format for exchanging data across applications with binding to all popular languages. It is a natural choice for synchronous microservices communication (replacing JSON/REST) for the following reasons:

- *Strong typing, type checks, and generated stubs make enforcing contracts easier than with free form protocols*
- *Since the underlying protocol is binary, the processing of messages is typically 6 times faster than JSON based messages.*
- *GRPC relies on HTTP/2.0, which comes with performance improvements and is widely adopted in the other blocks of the microservices stack. This allows proxies and load balancers to understand and redirect traffic across the infrastructure in an optimal way.*

Kafka

Kafka is a distributed event streaming database that is often used as a messaging platform for asynchronous microservices communication. Its advantages include:

- *High performance and low latency*
- *High resiliency and scalability*
- *Supported by Apache and wide community of enthusiasts*

Nevertheless, leveraging Kafka in microservice projects brings its own challenges both for developers and site reliability engineers. There are alternatives that might be better suited to a project (e.g. Pulsar); the decision to use specific tools should be based on detailed evaluation of the project requirements.

Micro-frontends

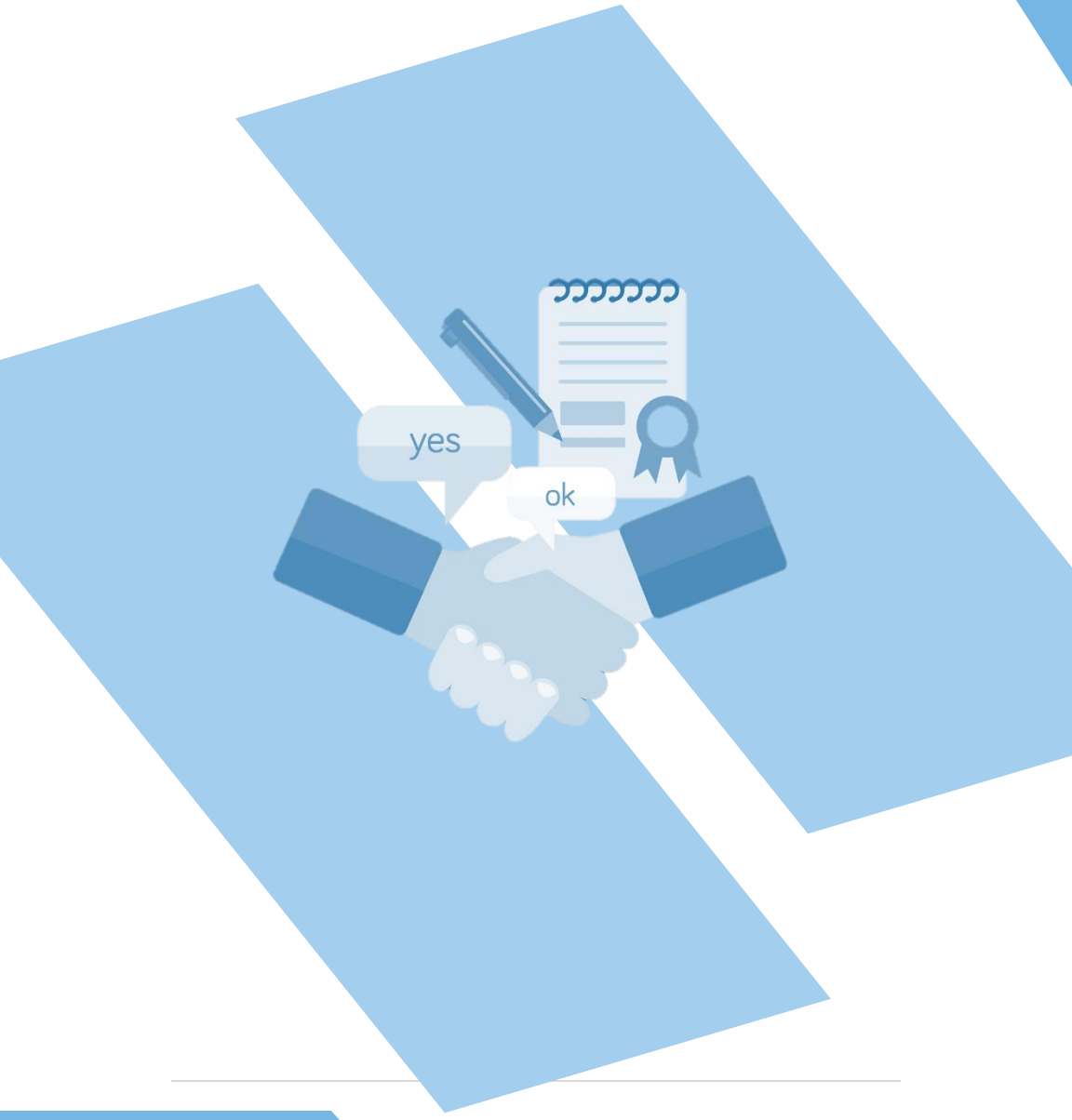
While this is not a specific technology per se, it is worth mentioning in this chapter. Micro-frontends are the frontend's answer to microservices; they allow the frontend of an application to be developed in a similar way as the backend microservices. In this approach, the frontend app is split into several modules that can be

independently developed by engineering teams, either frontend focused or cross functional, The distinct advantages of this approach include:

- Simpler, decoupled codebase
- Independent deployment of modules
- Autonomous teams

The frontend apps developed by each team are typically composed using HTML custom elements.

Microservices outsourcing



The microservices approach enables easy outsourcing of individual system parts to external vendors. However, many companies start off with a monolithic system architecture.

In these cases, conversion to a microservice-based architecture is the first step toward implementing a microservices approach. When outsourcing a system conversion or parts of an already established microservices-based system, it is critical to have a communication plan and build a team with the appropriate expertise.

Communication

Good communication is critical for successful outsourcing, because the external vendor or team needs to understand how the current system works and be able to work productively with a home-based team towards a shared vision.

If the system is already properly built using microservices, the communication process is often simplified because there is no need to understand the whole system – only the bounded context (i.e. microservice) the external team will be working on. In the best-case scenario, enough documentation will exist to successfully train the new

team on system concepts, development guidelines, platform and infrastructure approach, internal and external APIs, authorization etc. However, even if good documentation is in place, communication remains an important factor for successful completion of a project. Apart from understanding the system itself, an external team needs to understand the overall goal of the project, milestones within the project, and the vision for how the end product will be used. Appropriate communication channels are critical to resolve vague project requirements and effectively collaborate during development.

It is more common for existing system documentation to be fragmented, lacking, or nonexistent. In these cases, good communication is even more critical because an existing internal team who is familiar with the system needs to transfer their knowledge to a new external team. The new team must proactively investigate the system and achieve an understanding of it. If documentation is limited, it is necessary to proactively manage the project on the client side and ensure that appropriate technical expertise is available. While an outsourced team will have technical expertise, is important that someone on the home team be able to effectively communicate and collaborate with the new team on technical issues to achieve desired project results. If such expertise is not available in-house, it is important to look for an external partner who can both execute the work and function as a technical consultant.

Unfortunately, many internal client teams are not keen on sharing insider knowledge with contractors or third-party teams. Such clients often limit knowledge-sharing to critical pieces of information needed to complete a specific task, which makes it difficult for an external team to understand the big picture of a project and effectively execute work. When clients limit knowledge-sharing, it may be related to a fear of being replaced by an external team or losing positions to people in low-cost countries. Therefore, it is important for management to reassure existing teams that their jobs are secure and clearly and honestly state the reasons for bringing an external partner on board. These reasons may include demand that exceeds current team capacity or an urgent need for a new functionality.

Even if good communication channels are in place, external teams may face some resistance simply because they are newcomers. External teams should be prepared to figure things out on their own and drive communication and task completion. To help deal with such issues, it is important to have good project management on the vendor side. It is preferable to have a project manager in place who has prior experience on similar projects; a good project manager is critical to maintain team morale and ensure successful completion of a project, especially when working with clients who are challenging to communicate with.

Good communication throughout a project helps ensure quality, reduce rework, and save time and money. Frequent check-ins, collaboration, and product testing streamline the development process and should be integral parts of any product development effort. By maintaining and effectively managing open channels of communication, outsourced teams and home teams can work together to guarantee a strong and functional final product.

Down the road, issues and blockages can arise. With the right communication and project management skills, they can be overcome.

Let's look at some examples

Blockages	Answers
Fragmented or nonexistent system documentation.	The client side should proactively manage the project on the client side and ensure that appropriate technical expertise is available.
Internal client teams are not keen on sharing insider knowledge with contractors or third-party teams.	The external team needs this knowledge to understand the big picture of a project and effectively execute work. The team is not there to evaluate the code quality, but to improve and build on top of current functionalities.
Fear of being replaced by an external team or losing positions to people in low-cost countries	Management must reassure existing teams that their jobs are secure and honestly state the reasons for bringing an external partner on board (e.g. demand that exceeds current team capacity or an urgent need for a new functionality).
Resistance in front of newcomers	It is important to have good project management on the vendor side and good communication skills. The message is that the new team is there as a team extension.

Expertise required

It is important to understand the type of expertise required for a project when selecting an external vendor. Therefore, researching potential vendors is a critical step to ensure successful execution of a project. The design process for a global or single microservice-based architecture is smoother when working with an experienced vendor. Additionally, a vendor with a proven track record of working on projects with microservice-based architecture is able to more quickly and effectively onboard new teams. Familiarity with the microservices infrastructure/platform (e.g. container orchestration, service mesh) helps quickly introduce a new team to the deployment process.

It is crucial to find a vendor who has hands-on experience with microservices. Since microservices are a relatively new approach to architecture design, many developers have only theoretical knowledge of how they are used. To find a good vendor, hiring managers should look for experience and skills related to data modelling, API development, asynchronous and event-driven programming, orchestration technologies, cloud solutions, and Docker software. Familiarity with modern trends is also important; if a vendor uses outdated methods or workflows, it can have a negative impact on the end product and user experience.

Many vendors have the necessary skills to build a microservices architecture, so relevant industry-specific experience is a good way to differentiate a good vendor from a great vendor for a particular industry or business need. The best vendor for a project will not only have the skills needed to develop microservices; they will also have prior experience with similar projects in the same industry. Industry experience is important because it allows developers to quickly understand project needs and develop efficient solutions. If project case studies from a vendor are available, they are a good way to measure the expertise and experience of the vendor when making hiring decisions.

Potential vendors should also be assessed on their culture and ability to provide long-term support. Since communication is key to a successful project, it is important to find a trustworthy vendor who is a good culture fit with the home team.

For example – many companies prefer to use an agile approach to implement microservices; having a small agile team associated with each independent microservice is a quick and effective way to develop a product. If an agile approach is desired, it is important to find a vendor who has experience with the agile management style. It is also ideal to form a long-term partnership with a vendor because switching vendors in the middle of a project can negatively impact project cost, quality, and schedule. It is preferable to work with a

vendor who is well-established in the market and can see a project through from start to finish, including post-launch support.

What to analyze when selecting your vendor?

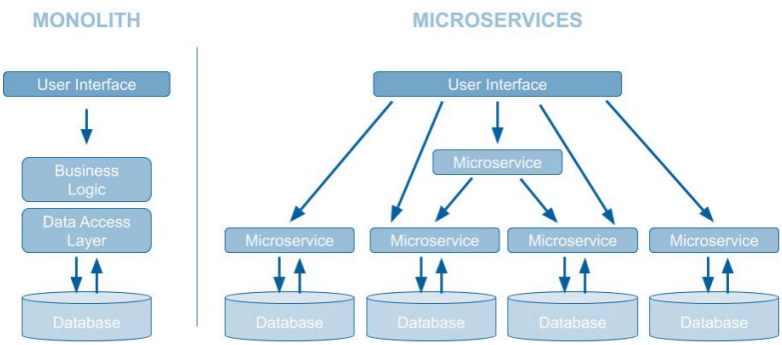
- ✓ The design process for a global or single microservice-based architecture is smoother when working with an experienced vendor.
- ✓ A vendor with a proven track record of working on projects with microservice-based architecture is able to more quickly and effectively onboard new teams.
- ✓ Familiarity with the microservices infrastructure/platform (e.g. container orchestration, service mesh) helps quickly introduce a new team to the deployment process.
- ✓ Familiarity with modern trends has a positive impact on the end product and user experience.
- ✓ Hiring managers should look for experience and skills related to data modelling, API development, asynchronous and event-driven programming, orchestration technologies, cloud solutions, and Docker software.
- ✓ Relevant industry-specific experience is a good way to differentiate a good vendor from a great vendor for a particular industry or business need.
- ✓ Make sure to analyze vendor case studies to measure the expertise and experience of the vendor when making hiring decisions.
- ✓ Make sure that the vendors' company culture is a good fit. If an agile approach is desired, it is important to find a vendor who has experience with the agile management style.

Monolith to microservices conversion
the process

"Monolith first"

It is typical for a team to start with a monolithic architecture when building a new product, according to the principle "monolith first". In the first stage of new product development there are many unknowns, including but not limited to business requirements, market traction, architecture approach, and performance bottlenecks. It is in a company's best interest to develop a minimum viable product (MVP) as quickly and cost-effectively as possible to test the product and resolve the many unknowns. Building an MVP using the monolith approach is desirable because it takes less time and resources than a similar project using the microservices approach.

Monolith vs Microservices



It is common for the MVP to be built by a small team of senior developers that can quickly deliver the product to market. Therefore, one of the most important advantages of microservices – the ability to work on multiple system components in parallel – is typically not utilized. Building a system with microservice-based architecture is more complex than building a system with monolithic architecture; it requires more upfront planning, more advanced architecture, and a common platform. The complexity required for microservices can offset the benefits of microservices in the short term and can affect the MVP release date.

Through the process of developing and releasing the MVP, developers and product managers learn a lot about their business needs and what they are trying to build. The benefit of using a monolith during initial product development is that changing the code impacts the entire internal architecture; solutions can easily be tested and updated. It is far easier to plan, implement, and release groundbreaking changes to one large system than to multiple independent and interconnected systems.

The experience gained during development of a version 1 (V1) product is invaluable. The original developers learn all the stages of product design and implementation and can help put new teams on the right track and avoid pitfalls during design of version 2 (V2). Lessons learned from market research and user feedback are extremely useful

when developing V2. Likewise, results of issue processing and retrospectives on the effectiveness of team communication can provide valuable insight to a new team taking on development of V2. As the product grows and more advanced functionalities are added, it becomes more challenging to manage all the dependencies using a monolithic architecture.

Transitioning to microservices

Added complexity makes it more difficult to scale services, develop and deploy new features, and innovate using a monolithic architecture. Monoliths utilize a single technology stack and single codebase; adding, removing, or updating a functionality impacts the entire system, and it is nearly impossible to leverage new technologies or tools when developing new services.

When companies decide to move to a microservice-based architecture, prep work should be done prior to migration. Before designing the system architecture, the domain should be mapped out and its boundaries and shortcomings understood. There is a good chance, especially if the system is overblown, that the system is not comprehensive enough in its current form. A well-performed mapping will make the structure easier to understand, and only after properly mapping the system should the new architecture be designed. During design, developers should consider whether the monolith will coexist with microservices. If so, the new architecture

must be compatible with the monolith so the two can effectively communicate. A plan should be established for how to migrate the system to microservices or incorporate microservices into the existing system. The plan should establish the sequence of migration and services to be added, removed, or replaced.

One of two paths can be taken when transitioning to microservices. An existing monolith can be decomposed piece by piece until the entire thing has been migrated into separate microservices. Alternatively, new functionalities can be added as new microservices on top of a pre-existing monolith. The latter approach keeps the old system intact and builds a new infrastructure around it. It is the best approach if the current system is bulky or old and not well understood. Building on top of an existing system is also desirable if it is not economically sensible to invest in the system (e.g. a system with a limited user base). For such cases, it is preferable to build new functionalities around the monolith.

Regardless of which path is taken, it is crucial to establish a communication channel for information exchange between the old monolith and the new microservice system to ensure the system works properly. The recommended way of establishing communication is to make the monolith microservice-aware and have it “play nicely” with the new entities; this process involves introducing new internal APIs as well as event generation and consuming code. Additionally, the monolith should be

containerized and corresponding code should be written to make it compatible with new architecture requirements (e.g. logging, configuration management, metrics, health checks). However, executing these steps is not always possible and depends on the monolith's original structure. In such cases, it may be necessary to use previous API calls (e.g. REST/JSON/XML calls) and separate operational procedures and teams to keep the old code up and running.

One of the first steps when transitioning to microservices is to separate the modules related to new functions from the main code and build them as separate subprojects that communicate with the monolith via a well-defined API. It is important that the API in the monolith be designed in a way that will enable future splits of the monolith into smaller services; the API needs to reflect the bounded contexts in the monolith. The API should be designed and preferably implemented by experienced developers who have a solid understanding of the product domain. The API serves as the entry point that unlocks the functionality provided by the monolith and allows new microservices to integrate with it. It also allows the company to start exploring opportunities related to outsourcing.

Once an API is established, developers can start decomposing the monolith into smaller services. This is valuable when it becomes difficult to deliver changes quickly and reliably to the monolith module or functional area. Having existing internal API coverage built for other

microservices allows developers to have a good understanding of how the API for new microservices should be structured. It also makes monolith decomposition easier because developers can reuse existing API calls for newly decomposed parts.

A microservice-based system is especially valuable for performance-sensitive online products. When rapidly scaling up an online product, the amount of traffic the system can handle must be considered. It is important to design the appropriate architecture early on. Since each microservice scales horizontally with ease, it is simple to increase capabilities for systems getting large traffic either on a regular basis or in spikes. Incorporating microservice-based architecture to extend the functionality of a system is a low entry barrier solution for scalability since it does not require redesign of the entire monolith from the ground up. New solutions can be added on top of existing code and immediate benefits can be gained from the improved performance of new functions.

Outsourcing considerations

In today's fast-moving world, having a flexible business model that can overcome market fluctuations is one of the main indicators of a company's potential scalability. The ability to freely add or reduce resources when needed is a highly desirable feature of any company, and it is one reason why outsourcing has become a highly popular form

of organizational development in recent years. Being able to reduce or scale-up a development team depending on project peaks enables a company to rapidly react to emerging demands.

Outsourcing allows a company to rapidly onboard talented developers to projects, which makes it possible to rapidly scale a product and business. If the outsourced partner has experience with similar projects, they are able to quickly pick up the development. As a result, a company can immediately start reaping the benefits of new functionalities and releases to production with less consumption of core resources. Over time, an outsourced partner will continue to learn the product domain and be able to deliver subsequent microservices even faster and with increasingly less time required from the home team. If an outsourced partner has offshore development centers, it may also be less expensive than in-house development.

Decomposing a monolith requires excellent familiarity with the product domain and good understanding of the APIs and data model. It is crucial for outsourced developers to understand the scope of the monolithic system and how it functions and communicates to effectively decompose it into microservices. This familiarity and understanding are most easily gained by working on less complex assignments related to the project. While it is possible to bring an outsourcing partner onboard for the decomposition process, it is advisable that he or she has

prior experience building smaller microservices for the project.

An alternative approach is for the internal team to focus on the monolith migration while the outsourced teams work on other microservices. While this is a perfectly sensible approach, it foregoes the benefits of having more than one team possess expertise on the system code. Including outsourced teams as part of the migration process enables a company to leverage a partner's experience in microservices architecture and ensures that enough development resources are available.

It is important to keep in mind that outsourcing work may result in less company control over the project, particularly over some functionalities and parts of the product. To minimize this risk, proper product management must be in place. Technical documentation should be written and delivered so that the in-house team can take over if needed. Building up a microservice-based architecture may get complex and overwhelming at times. Companies should choose a partner with significant experience and a proven track record who has been on the market for a while.

Outsourcing to CEE

Since microservice-based architecture can be developed by independent teams working on different functionalities, the ability to outsource it is in its nature.

Outsourcing this kind of development comes with many benefits such as lower cost and access to talented developers. Therefore, many companies have chosen to outsource microservices work to Central Eastern Europe (CEE).

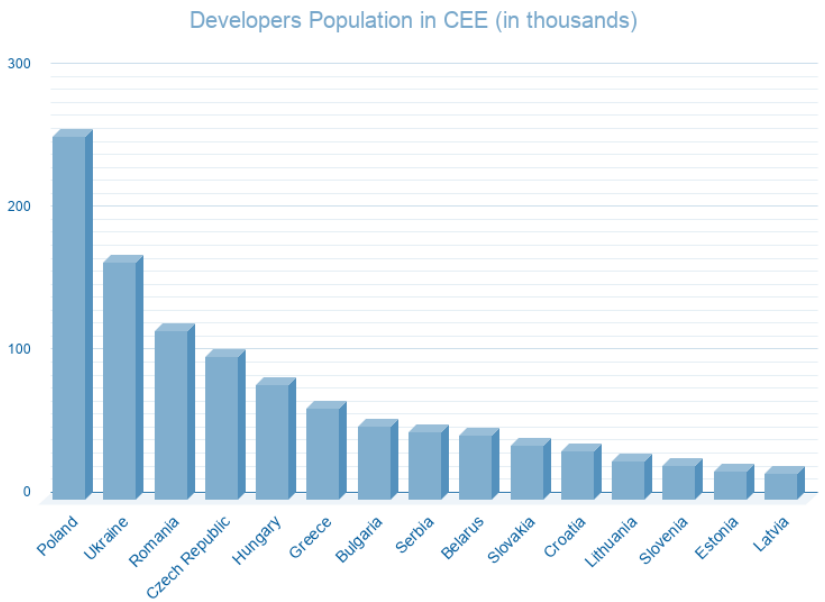
Both global corporations and start-up companies are bringing their assets to the region; over 2,000 Shared Service Centers (SSCs) are open in the region.

The SSCs employ 640,000 people, many of whom are highly qualified engineers who bring innovative product development strategies to the region.

Advantages of outsourcing to CEE

For many companies, lowering project costs is one of the major factors that makes outsourcing appealing. Cost of living and average salaries are much lower in CEE than in the US. However, due to high demand, the salaries in the IT sector are steadily rising. During the last 10 years, Poland and other countries in the region have seen an influx of investment from overseas as more outsourcing

centers and software houses have been established. While it does not have the lowest pricing, the region offers the perfect balance between the cost and overall quality of services; companies get the best service for the least cost. In less expensive locations like South-East and Eastern Asia, the quality of services is more likely to suffer due to language barriers or significant time-zone differences. Apart from staff costs, using an outsourced team also reduces costs associated with assets such as additional office space and hardware.



SOURCE: INFOSHARE.PL

Access to talented developers is another major advantage of outsourcing to CEE. There are over one million developers in CEE. That equates to 1.3 developers for every 100 people in the labor force in the region. Due to a mostly free and widely available regional higher education system that attracts the best candidates, the region has many quality engineers who are highly valued in the job market across the world. Each major city typically has two public universities with computer science departments and multiple private schools.

There are 11 countries in the CEE region, with Poland producing the most engineers of all. Poland has over 250,000 IT developers, which is a quarter of the total number of developers in the region. The high utilization of the CEE workforce on complex IT projects has resulted in a large pool of experienced IT specialists who are able to manage and build the most complex projects.

Poland in particular has long been recognized as having a problem-solving culture. Poland has produced many of history's most exceptional mathematicians, including Alfred Tarski (emigrated to the US before WWII and then worked at University of California at Berkeley), Benoit Mandelbrot (born in Poland, researched and coined the fractal term), Stefan Banach (creator of functional analysis), and Stanisław Ulam (one of the designers of nuclear and thermonuclear weapons in the Manhattan project). The exceptional quality of Polish engineers is supported by Hackerrank, one of the largest online

candidate screening services. The Hackerrank leaderboard for best developers by country can be seen here:

Which Country Has the Best Developers?

Ranked by Average Score Across All HackerRank Challenges

Rank	Country	Score Index	Rank	Country	Score Index
1	China	100.0	26	Netherlands	78.9
2	Russia	99.9	27	Chile	78.4
3	Poland	98.0	28	United States	78.0
4	Switzerland	97.9	29	United Kingdom	77.7
5	Hungary	93.9	30	Turkey	77.5
6	Japan	92.1	31	India	76.0
7	Taiwan	91.2	32	Ireland	75.9
8	France	91.2	33	Mexico	75.7
9	Czech Republic	90.7	34	Denmark	75.6
10	Italy	90.2	35	Israel	74.8
11	Ukraine	88.7	36	Norway	74.6
12	Bulgaria	87.2	37	Portugal	74.2
13	Singapore	87.1	38	Brazil	73.4
14	Germany	84.3	39	Argentina	72.1
15	Finland	84.3	40	Indonesia	71.8
16	Belgium	84.1	41	New Zealand	71.6
17	Hong Kong	83.6	42	Egypt	69.3
18	Spain	83.4	43	South Africa	68.3
19	Australia	83.2	44	Bangladesh	67.8
20	Romania	81.9	45	Colombia	66.0
21	Canada	81.7	46	Philippines	63.8
22	South Korea	81.7	47	Malaysia	61.8
23	Vietnam	81.1	48	Nigeria	61.3
24	Greece	80.8	49	Sri Lanka	60.4
25	Sweden	79.9	50	Pakistan	57.4

HackerRank

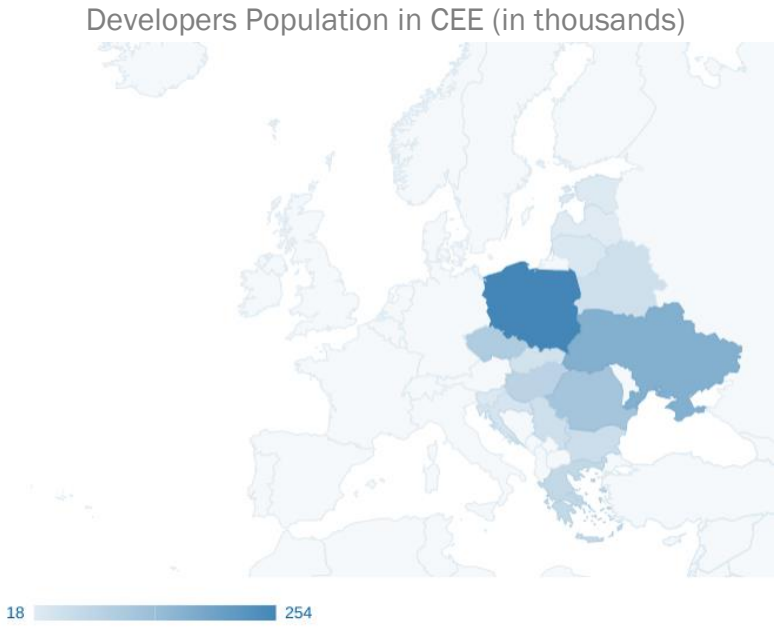
A similar culture to the US also makes CEE a desirable outsourcing location. CEE has overlapping time-zones with the US, a similar cultural background, and a population that mostly speaks fluent English.

It has been 30 years since the Communist Block disappeared, and a lot has changed during this time in CEE societies, especially in Poland. Central and Eastern Europeans now share similar values with Western Europeans and Americans. Most people are ready to work hard to catch up with more advanced countries, and they tend to be highly motivated, self-contained, honest, and transparent. It can be difficult to find similar values in other outsourcing areas such as South and Far-East Asia. While there may be minor cultural differences in CEE that need to be understood for successful cooperation, getting on the same page is usually a very quick and painless process. People in CEE also embrace western culture in terms of entertainment, events, etc. and are easy to relate to. Chances are that a US-based team will be able to chat with a CEE team about the most recent Netflix series or music release. While this is not a critical factor for a successful project, it can improve the communication and sense of community across the teams.

Outsourcing is particularly advantageous for small companies who do not have a significant amount of in-house expertise. Utilizing an experienced external vendor reduces trial and error and often leads to improvements in home team performance by allowing the home team to focus on key tasks without the distraction of non-core activities.

Challenges of outsourcing to CEE

While there are many benefits of outsourcing to CEE, it does not come without challenges. Fortunately, most of the challenges can be addressed with good project management. When looking at the big picture, the pros of outsourcing to CEE outweigh the cons.



SOURCE: INFOSHARE.PL

Geographical/time zone difference is the most obvious challenge of outsourcing to CEE. The time zone in CEE is six to nine hours ahead of the US. For the East Coast,

there are three to four hours of common work time. For the West Coast, there is a narrower window; developers are typically wrapping up for the day while US workers are just starting their work day. To some extent this is a good arrangement, because work that was completed by the CEE team during their work day is ready to review at the beginning of the US work day. This can result in more efficient feedback loops and the ability to progress work more quickly. If needed, developers are typically willing to participate in evening calls if scheduled in advance (or unplanned in case of emergencies). A shift-based arrangement is also a viable option; some developers are willing to adjust their work day to the US time zone or take shifts doing so within the team.

It is possible that communication challenges may exist due to a language barrier, but this gap is quickly closing. Most developers communicate effectively in English, and many are fluent. The prevalence of the English language in the technical world has made it necessary for developers to frequently use and learn it. Reading and writing technical documents is an easy task for most developers. If there are communication challenges, they are easily resolved by appointing one or several people who are responsible for communicating with the client and ensuring that all team members are aware of client expectations.

Some companies have concerns about outsourcing because they fear they will have less control over the

development process and delivery. This is often viewed as risky since critical parts of internal processes and product development are handled by engineers and specialists who are not employees of the company. However, there are several ways to mitigate this type of risk. A company can introduce strict controls, metrics, and/or permissions to more tightly manage the project. Alternatively, a company can integrate outsourced developers with their internal team. This will often involve assigning a trusted and experienced internal employee to supervise the outsourced team. The latter approach is typically the best solution because it often leads to improved communication and a good team dynamic.

Advantages	Challenges
✓ *Lower cost software development means lower entry barrier*	✓ *Need for additional preparation and adequate project management*
✓ *Higher quality of engineers than in other affordable locations*	✓ *Time-zone shift might cause issues for some project types*
✓ *Access to a large pool of highly qualified developers*	✓ *Less control over project if not done right*
✓ *Each major CEE city typically has at least one public university with computer science departments and multiple private schools*	✓ *Communication barrier might appear with some partners*
✓ *During the last 10 years, Poland and other countries in the region have seen an influx of investment from overseas as more outsourcing centers and software houses have been established*	
✓ *CEE has overlapping time-zones with the US, a similar cultural background, and a population that mostly speaks fluent English*	

Jurisdiction differences: CEE vs. US

Differences in jurisdiction must be considered when outsourcing to CEE. Different nations will have different legal frameworks, laws, and regulatory requirements that may not align. It is crucial to properly navigate legal issues to avoid pitfalls on a project; this is most easily accomplished by partnering with a US-based entity who has experience with the CEE legal system.

Overview

Many regulations in CEE have roots in the communist era, which makes the entire justice system dated and complex. Is can be challenging even for local companies to stand before the court. For US companies that are accustomed to a completely different justice system, navigating through the complicated legal structure quickly becomes time and resource consuming.

In CEE, even simple lawsuits can take many years to resolve; complex lawsuits can take decades. While drawn-out lawsuits are a risk in all countries, the probability of experiencing one in CEE is far higher than average. Business related lawsuits are up to three times longer than the global average in some CEE countries.

There are two significant differences between US and EU law that impact work with CEE development teams – privacy law and intellectual property/copyright law. Both are significantly more complex and less liberal in the EU than in the US. If EU regulations are not followed, issues are likely to arise down the line. However, successful transfer of personal information about CEE staff and transfer of intellectual property created by development teams (applications and code) are critical for outsourced development efforts to be effective. Transferring this information and intellectual property, especially when developers are spread across different countries, is no small task. Each developer and the intellectual property he or she create may be under different jurisdictions with unique legal regulations. Each software component may need to be addressed in a separate court under the local law in the event of intellectual property claims; this adds another layer of complexity to an already complex legal system.

Privacy law

The US and EU have vastly different approaches to privacy law and sensitive data. In the US, data protection is liberal and less strict than in the EU. Only a limited set of sensitive personal data such as social security numbers

and credit card numbers are legally protected against unauthorized access or acquisition. In contrast, data privacy law in the EU is stricter and is regulated across the entire union by the General Data Protection Regulation (GDPR).

The GDPR is a legal framework that sets guidelines for the collection and processing of personal information from individuals who live in the EU. Data processing includes any possible interaction with the data – from collection to erasure or destruction. Every data interaction is strictly defined by GDPR regulations.

In the EU, any data that can be directly or indirectly associated with a living individual is defined as "sensitive data". This definition is broad in scope; sensitive data may include but is not limited to IP addresses, racial or ethnic origin, political opinions, religious or philosophical beliefs, and trade union membership. Since so much qualifies as sensitive data, data processing is much more complicated in the EU than in the US. The EU requires companies who process personal data to have defined data processing procedures, policies, and security measures in place. Additionally, there must be a person on staff who oversees data processing and ensures compliance with GDPR requirements. The GDPR is also stricter than the US when it comes to handling data breaches. The GDPR requires companies who have experienced a data breach to document the facts related to the breach and take remedial action to prevent a recurrence.

The GDPR stipulates that sensitive data can only be transferred from the EU to countries deemed to have adequate data protection laws; the EU does not recognize the US as a country that meets this requirement. To overcome this dilemma and more easily enable global business, Privacy Shield was introduced in 2016. Privacy Shield is an agreement between the EU and the US that allows the transfer of personal data from the EU to the US; participating companies in the Privacy Shield program must be established as having adequate data protection. Privacy Shield enables Non-EU companies to receive personal data of EU residents, and thus it is an important tool for any US business that interacts with EU consumers or staff. In addition to Privacy Shield, it is also important for US businesses to have a robust and professionally directed GDPR compliance plan.

Processing sensitive data is critical for any US company who has EU employees. However, dealing with all the legal issues of working directly with EU developers may be daunting to many companies. Finding a US partner that can manage the process is a much easier approach than working directly with EU/CEE based developers or companies. The approach avoids direct data processing and complicated paperwork, which can be delegated to a US partner who has experience handling complicated EU and country specific regulations.

Intellectual property/copyright law

The EU and US have completely different approaches to intellectual property and copyright law that originate from historic roots. European copyright arose from the humanistic and creative movement of the French Revolution (1789-1799), during which the first modern copyright laws were formulated. US copyright arose from English legislation, specifically the Statute of Anne (also known as the Copyright Act 1710), and it was aimed to improve education and knowledge circulation by protecting the rights of publishers to ensure they would receive payment. EU copyright law focuses directly on the originator of the work in question, while US copyright law focuses on exploitation rights and any potential financial implications.

The EU has a unified law across the union called the Rome II Regulation. The Rome II Regulation recognizes the lex loci principle regarding intellectual property – i.e. the local law of the country where the property was established applies. This means that different laws may apply in different countries and in some cases even different parts of the same country (e.g. Germany). When working with a group of developers located in different countries, different jurisdictions and laws may apply. The US has no such regulations; it is up to federal courts to make policies based on federal law. While this may lead to legal

uncertainty at the federal level, there are not any differences from state to state.

EU legal requirements for copyright transfer are complex and stricter than those in the US. In the US, there is a concept called “work made for hire”, which means that the party for whom a piece of work was created is considered the author and owner of the work. The EU does not recognize any such concept (with rare exceptions); the creator of the work is always considered the owner, and the party ordering the work cannot claim it as their own without giving the owner credit. The EU also requires that intellectual property transfer contracts define all possible uses for a piece of copyrighted work. This requirement is particularly problematic in software development since possible uses are difficult to predict and emerge over the course of development.

Forming a partnership with a US entity makes it far easier for a company to handle potential copyright claims. The company can continue to work in accordance with US regulations without having to deal with complicated and diverse intellectual property systems in different countries. Any legal issues can be handled by the partner with no implications to the company.

Why choose a US partner?

While privacy and copyright law differences are most pronounced when outsourcing software development, there are many other legal discrepancies between the US and the EU. Gaining a full understanding of jurisdictional differences and the potential issues arising from them may be a daunting task for many organizations. Thankfully, there is an easier solution. An organization can partner with a US-based company with development centers in CEE. Working with a US partner who has access to CEE developers is the ideal way for US companies to reap the benefits of outsourcing work to CEE developers without having to spend valuable time and resources dealing with complicated European and country-specific regulations.

This approach ensures that the organization is working with a US company for legal contracts and agreements. Intellectual property rights and principles of trade are based on US law. The outsourcing company internally handles any local CEE/EU law complexities or disputes that arise. This is beneficial to all parties, because the outsourcing company likely has local CEE/EU branches, familiarity with the law, and experience with how to avoid law pitfalls.

Geographically distributed microservices development:
case studies

Raise

The #1 gift card buyback and sell platform, valued at $1 billion by the New York Times in 2016

Raise is a Chicago based start-up that was launched in 2013 by George Bousis. Raise uses an e-commerce platform that enables third party sellers to sell new or used gift cards on a fixed-price online marketplace. Now it is the #1 gift card buyback and sell platform; in 2016, Raise was valued at $1 billion by the New York Times. Raise's entire system was originally based on a Ruby monolith, but the company wanted to adopt a Golang-based microservice architecture and React.js frontend. Raise brought in NG Logic to help develop the system, and offshore developers successfully transferred knowledge to the local team during implementation of critical new functionalities like the "Instant Order".

Cooperation

NG Logic was engaged to help develop and implement a microservice-based architecture. This entailed creating new functionalities across the existing monolith, a frontend, and a platform. To effectively execute the scope of work, cooperation and communication with the home team was needed. However, the initial reaction by existing developers was cold and passive aggressive. It was difficult to get information regarding what needed to be

done and how to do it. The home team was afraid of losing influence and jobs to the newcomers, and some of the home team employees left shortly after.

The offshore team was able to overcome the initial setbacks by persistently pushing toward the goals set by the business and working hard to understand the current system. Ultimately, the team was able to bring value to the project, and the first piece of new functionality was delivered on time and with minor friction.

Raise was highly satisfied with the initial delivery, and two more offshore team members were quickly brought on board. NG Logic developers became an important part of the team and Raise continued to be happy with the performance of offshore team members. With time, NG Logic was entrusted with a six-month project to convert part of the monolith into a microservice-based architecture. A cross-functional Polish sub-team was formed and worked together on the design, implementation, and testing of the new system. The team was responsible for directly communicating to business stakeholders, reporting progress, and asking questions to mitigate roadblocks. The project was successful; it was delivered on time and brought significant improvements to the end user experience. The team was praised for its performance.

NG Logic continues to work with Raise to maintain and update their platform. Today, 75% of the frontend and

50% of the backend developers are Polish, and Polish team members are highly regarded by the US developers. The Polish teams are trusted with a high amount of responsibility, such as performing important platform migrations and upgrades. New hires are often sourced from Poland due to the talented pool of individuals available and shorter recruitment timeframes. An added benefit of offshoring recruitment efforts is that no additional work is required from the in-house developers.

Lessons learned

✓ **Good communication is critical**

Working with Raise reinforced how important it is to have good communication between an outsourced team and an existing home team. A company's home-based development team may initially have concerns and fears about bringing an outsourced team on-board – fear about losing their jobs, concern about the quality of engineers being brought in, etc. These fears and concerns may be magnified if they have had previous negative experiences working with far east teams, which often do not produce high quality work. It is important for management on both the client and vendor side to address these fears and concerns. Management should ensure that the existing team understands the purpose of bringing external resources to the project, and it should be reinforced that external resources are not intended to replace or reduce

the headcount of the local team but to bring additional value to the project.

✓ **Teams should work together**

Raise had a complex existing system that was challenging to understand without taking advantage of knowledge transfer between new and existing teams. To promote project efficiency, new teams should be allowed to work alongside existing engineers. This enables new teams to rapidly gain an understanding of the system and project requirements. It also is invaluable from a team-building perspective. Requiring new and existing teams to work closely for one to three months gets them used to working together and builds a foundation for trust, communication, and personal relationships.

✓ **Team autonomy results in increased productivity**

Allowing the outsourced team to become independent resulted in a significant productivity boost. Once trust has been established, it is in a company's best interest to give outsourced teams some level of autonomy. Giving a team full ownership and responsibility for a specific component, functionality, or microservice allows the team to streamline their development efforts more efficiently. However, it is critical to make sure the team has all necessary resources and is cross-functional. A product owner and QA procedures should be in place. Metrics and key performance indicators should be used to measure team performance and hold team members accountable for their work.

✓ **Time zone differences can be overcome**

Over the course of working with Raise, it was observed that time zone differences became less problematic in the long run. Over time, the home and outsourced teams began to learn each other's habits and availability patterns. Soon enough they were either actively or subconsciously adjusting their communication to suit the other team's availability. Additionally, the amount of on-going communication decreased over time as the new team accumulated more knowledge. Remote developers were usually flexible and willing to respond during their evening if needed.

SingleCare

The healthcare discount services provider who partners with 35k+ pharmacies - a success story leading to a 20% increase in website conversion rates

SingleCare is a healthcare discount services provider that lets customers compare costs of prescription drugs free of charge. It was founded in 2014 by Rick Bates. SingleCare has partnerships with over 35,000 pharmacies (including CVS, Target, Walmart, and Walgreens) and helps customers save up to 80% on 50,000 drugs.

SingleCare's original system was a .NET application that was designed and implemented in a way that required a significant maintenance effort. As SingleCare became more successful, more consumers began to use their platform. The system was unable to handle the influx of users, which resulted in severe performance bottlenecks. Adding new functionalities to the existing system was difficult, time consuming, and resulted in multiple regressions due to architecture issues. (e.g. lack of separation of concerns). To add to the problems, the technology being used was generating substantial licensing costs. SingleCare needed a solution, and NG Logic was brought in to help them transition to a microservice-based system.

Cooperation

The first step of the project was a detailed assessment of SingleCare's current platform, the current market situation, and the client's plans for future expansion. NG Logic developers, together with the client engineering team, used the output of this assessment to determine a path forward. It was jointly decided that the best solution was to replace the current system with a new one based on microservice architecture.

The new architecture was designed leveraging Golang backend services, React.js frontend, open source database engines, and Kubernetes as the container orchestrator. The main design goals were aimed at mitigating bottleneck issues, enabling scalability, and improving the overall quality of the system to reduce regressions and improve the user experience. Design goals included proper separation of domains and concerns and horizontal scaling to support planned workloads. The design was also intended to give SingleCare the ability to scale up the velocity of new feature delivery as required by business needs and the ability to run the current and new systems side by side during the transition period. Successful implementation of the design goals resulted in improved system performance and a reduction in system errors. Ultimately, the new system design led to an improved user experience and an increase of approximately 20% in website conversions.

Part of the development process also entailed the establishment of common contracts for interservice communication (Protobuf definitions). After the plan and timeline was created to develop the new system and gradually phase out the old one, the new system was created and implemented by multiple remote teams around the world. Each team was assigned a specific domain/microservice to develop and maintain. To improve communication and optimize the development process, a single Platform Team was formed consisting of Devops resources from all the locations. In addition to eliminating the need for on-call shifts (the engineers are on call during their working hours), this strategy assured that developers had access to a site reliability engineer in their time zone who spoke their language.

The outsourced team brought significant expertise to the project that resulted in the creation of a robust and effective microservices-based system design. The NG Logic team was quickly recognized as having the most knowledge and experience required to design and develop the system. Throughout the project, outsourced engineers reliably delivered high quality products with minimal production issues.

Lessons learned

✓ **Utilize DDD and common contracts**

All development teams, especially geographically dispersed teams, are most effective when domain driven design principles are applied and common contracts are agreed on by all teams. Before any development steps took place, the offshore team identified several system domains that were divided between two sub-teams. Each sub-team consisted of resources located in the same time zone with the expertise required to design and implement their assigned domains. Using DDD helped ensure efficient execution of work and appropriate division of responsibilities. Common contracts were also critical to achieve consistency on the project. The teams all collaborated to produce a common approach for implementing the new system. Together, the teams developed standard documentation outlining creation and maintenance of APIs, code standards, test coverage, and microservices requirements such as logging, metrics, and tooling.

✓ **Establish coding and design standards**

One of the first steps when working with any client, including SingleCare, should be to reach a mutual agreement between all teams on coding standards and design patterns that should be used in the development process. In the initial stages of the project, some of the teams contributed code that did not meet the standards of other teams. This issue was resolved by holding a

meeting across the teams and architects to discuss and establish common coding practices and approaches. This was followed by a requirement for all teams to have their code reviewed and accepted by at least one member of another team before it could be merged into the system. Over time, these reviews resulted in a common understanding and the establishment of code standards that enabled the teams to work separately without cross-team code reviews.

✓ **Use automated system testing**

Another important takeaway was recognizing the usefulness of automated end to end testing to help quickly verify system performance. Over the course of the project, extensive end to end tests were developed that leveraged Cypress framework and allowed quick verification of the consistency of the system before release. Since there were multiple teams contributing to the system, this mechanism was useful for identifying changes that could cause regressions. The continuous integration system was able to quickly identify offending changes and the person/team that needed to fix the build.

News Direct

The cutting-edge content and news distribution platform

News Direct is a cutting-edge content and news distribution platform built for the demands of today's strategic communications. It is a startup founded by Gregg Castano (longtime president of Business Wire) in 2019 and headquartered in Norwalk, CT. News Direct aims to help PR, Corp Comms, and IR professionals bring their media outreach efforts to the next level. The company prides itself on redefining the news distribution industry, and their platform was designed to reinvent content delivery, re-engineer workflow, revitalize metrics and ROI, and modernize security and pricing. NG Logic was brought in to design the architecture of the platform and develop the main components of the system. Throughout the process, offshore developers cooperated closely with News Direct's US-based team (Golang developers and devops).

Cooperation

In addition to development requirements, the project had several non-functional requirements including high security and high availability. Therefore, it was decided to use multi-instance deployment with a central controller component. The backend was split into multiple services to match the domain of the project, and the services were

implemented using either Golang or Python/Django based on the suitability of each technology for the needed functions. For example, the complex REST-based service that handles user, permission, and content management was implemented in Django Rest Framework. Internal admin pages were implemented using Django Admin, and performance-sensitive parts used Go programming language. The services communicate via a REST-based API for the React.js frontend; on the backend, the services communicate internally using a combination of GRPC and Protobuf messages over AWS queues. The offshore team also developed a full end-to-end testing suite in Cypress that enabled issue identification early in the development process.

NG Logic worked with News Direct when they were still a small start-up company with limited resources to manage a complex system. Instead of going with a pure microservice-based approach in which each microservice has a single responsibility, the offshore team opted to use a miniservices approach. The application was designed with several larger components that encapsulate large or multiple related domains (like user and content management, publication, conversion, etc.). This allowed News Direct to reap the benefits of the modular architecture without needing to invest in the overhead required for multiple microservices management, monitoring, communication, diagnostics, etc.

News Direct was highly satisfied with the delivery efficiency of the remote team, who successfully achieved all project goals such as reinventing content delivery and modernizing security. The remote team implemented a specialized WYS/WYG content editor that streamlined the process of creating professional press releases. In addition, a set of services was designed and implemented to reliably publish press releases to multiple targets, including the largest press agencies such as AP and Bloomberg. Developers also incorporated high security design elements in the platform such as two factor authentication and data encryption. News Direct was so impressed with the remote team's performance that the offshore team grew from six to eleven people over the course of the project.

Lessons learned

✓ **Upfront assignment of services**

Initially, there was no clear assignment of responsibility for the modules/services, and all the developers were working as a single team. This was helpful in the short term and facilitated communication and collaboration in the architecture design phase. However, in the long run it was difficult to maintain common coding standards and a common approach to software delivery. These issues were eventually resolved by assigning responsibility for services to teams based on their expertise and geolocation. To avoid such issues, there should be a plan in place to assign services to teams early in the project.

✓ **Adaptation to project scope changes**

The user experience (UX) design was not finalized until late in the project because the business team and UX designers were researching multiple approaches. To keep the project on schedule, the architecture development team started their work based on educated assumptions and discussions with the business stakeholders. As the UX design progressed, it began to drift from the development team's assumptions. Some of the architecture design concepts did not align with the new UX requirements. The team adapted quickly to the situation, assessed the UX requirements, and proposed changes to simplify the overall architecture and bring it into alignment with the UX requirements. These changes were approved by the product owner, added to the backlog, and implemented.

The changes primarily manifested in the types of flows across microservices. The developers were originally using asynchronous communication to increase resiliency and reliability of the system. However, some of the user interactions with the system required immediate feedback to the user about the outcome of his or her action. Consequently, the underlying APIs were modified to support synchronous communication channels that matched the UX approach.

Regular technical meetings across multiple teams were critical for successfully adapting to changes in the project requirements. If a team member felt that the architecture

was not appropriate for a new or changed requirement, there was a space to discuss the issue and agree on the required modifications to keep the architecture consistent.

Recommendations

Oh, my... blink twice if you are still reading this.

It was a tough read, but hey, who says microservices are easy to digest?

It is a fact: microservices are here to stay and to make businesses more flexible and scalable. A wide range of companies that differ in size, maturity, and business model seem to benefit from adopting elements of microservices into their systems. By gradually moving the architecture from a monolith approach to a microservice-based approach, large companies with established systems can benefit from improved time to market and performance immediately, without a full overhaul of the system. Alternatively, startups that need a scalable IT system that can be quickly built and extended should consider leveraging microservices architecture from the start.

When planning to move to microservices architecture, a good strategy is crucial for success. Do not just buckle up, start your engine and go full speed ahead. The first step should always be to review the current system or the new

system requirements, identify the pain points or non-functional requirements, and investigate if microservices architecture can solve those challenges. Based on the review, an architecture approach action plan can be established.

Finding a reliable partner who can help migrate to microservices can be beneficial for companies that have no experience with microservices and would like to reinforce their internal team.

It has been demonstrated by many case studies that microservices-based projects can be successful using an outsourcing model and even geographically distributed development teams. Outsourcing microservices development comes with many benefits including access to a larger talent pool, more affordable rates, and quick staffing.

Now, if you want to move further with a professional, we would love to hear from you.

NG Logic LLC

400 Concar Dr
San Mateo, CA 94402

info@nglogic.com
+1 (888) 413 3806

https://nglogic.com/