Microservice Architecture Patterns for Enterprise Applications Supporting Business Agility

PAVEL HRUBY, REA Technology
CHRISTIAN VIBE SCHELLER, NNIT

Would your organization like to achieve business agility and respond quickly to opportunities, but it takes too long to implement changes in your IT system? It might help to compose your application portfolio from smaller applications, communicating with each other using events. Patterns in this paper can help your organization address the following challenges:

• Your organization would like to respond quickly to market changes and opportunities. However, it takes weeks, even months, to identify the requirement, develop the software and deploy the new feature into production.

• Despite careful coordination, developer teams must wait too often for each other, which slows down progress. With the growing number of developers working on the system, developing new features becomes slower and more laborious.

This paper aims to provide a reader with a single direct path to microservice architecture that has been proven successful. The patterns can be used to decompose the monolithic client-server application into microservices; many patterns in this paper can also be used to guide a new business that wants to set up an IT system designed to support business agility.

Categories and Subject Descriptors: **•Information systems~Information systems applications~Enterprise information systems~Enterprise applications •Information systems~Information systems applications •Software and its engineering~Software organization and properties~Software system structures~Software architectures** •Social and professional topics

General Terms: Microservices; Business agility

Additional Key Words and Phrases: Wardley map; Antifragile

ACKNOWLEDGMENT

The authors would like to thank our shepherd Ernst Oberortner, for his valuable time on weekly shepherding calls and for his indispensable input, and the participants of the PLoP 2022 writers' workshop Joe Yoder, Marden Neubert, Paulo Marques, and Jô Choma for additional insights and for sharing experiences with this architecture. Three patterns in this paper were rewritten, many others have been improved, and the paper has gained a more natural flow by reordering the patterns, all thanks to your feedback.

1. INTRODUCTION

In the late 90s, after personal computers became broadly available, many business applications started using the client-server architecture, because the client-server systems have typically been cheaper and more lightweight than mainframes. A client-server is a software architecture where the application is distributed between a client and server components, communicating over a network.

Over time, the client-server applications became very complex. After many years of development, they typically have thousands of forms in their user interface and thousands of database tables. Despite applying architectural patterns and best practices of object-oriented design, these applications often consist of tightly coupled components, where every component depends on the others. Therefore, large client-server applications are often called monoliths.

Tight coupling makes new features increasingly difficult to develop because of the combinatorial effect of changes. The combinatorial effect of changes refers to a situation where the impact of a change depends on the

Author's addresses: Pavel Hruby; email: phruby@acm.org; Christian Vibe Scheller; email: cvs@reatechnology.com.

size of the change multiplied by the size of the system (Manaert and Verelst 2016). If the system is large enough, measured by the number of "components" and their relationships, implementing even simple changes could be very challenging due to system complexity. The combinatorial effect applies to systems in general, not just software systems. The "components" can be connected mechanical parts, the accounts in a chart of accounts, interconnected modules in a software application, and interconnected software applications in an application portfolio.

Because of the tight coupling in the internal structure of software monoliths, the whole system has to be deployed together, and the whole system has to be tested before deployment. Consequently, there could be only a few releases per year.

The development team in a client-server system is typically structured into a user interface team, database team, network team, and server team, matching Conway's law: ”Organizations, which design systems are constrained to produce designs which are copies of the communication structures of these organizations" (Conway 1968).

In such siloed organizations, many technicians and developers are working on multiple projects, and the wait time for tasks to be completed is often proportional to resource utilization. If each developer is busy, most tasks and requests wait in a queue instead of being worked on, which drastically reduces throughput (Kim et al. 2014).

To address the above challenges, the monolithic systems are decomposed (at the cost of rewriting code) into simpler applications, so that many teams can work independently on their application. They can be tested and deployed independently, and consequently, have independent release cycles. When deployed at a cloud provider, they achieve horizontal scalability and handle variable demand. These simpler applications are often called microservices.

The path to application modernization is described in eleven patterns, see Fig. 1.

The first pattern ANTIFRAGILE ORGANIZATIONS summarizes the business context of an organization seeking business agility. The following three patterns, USER JOURNEY, EVOLUTION STAGE, and BUSINESS CAPABILITIES illustrate how to understand and identify the stakeholder needs, the evolution stage of the existing components, and the required business capabilities of the new IT system.

The event-driven microservice architecture is the core section of this pattern language. As the MICROSERVICES are independent applications, the technology used to build them can vary from using general programming languages to using the serverless technology of a cloud provider. The patterns EVENT SOURCING, MICRO FRONT END, and SERVERLESS ARCHITECTURE summarize considerations on how to approach the design of microservices.

There are two archetypical ways of splitting the monolith into microservices: the EVOLUTIONARY TRANSFORMATION and the BIG BANG TRANSFORMATION, each requiring different architecture supporting the transformation.

The full benefits of the microservice architecture can only be achieved with automated infrastructure and DevOps practices, described in the DEVOPS and INFRASTRUCTURE AS CODE pattern.

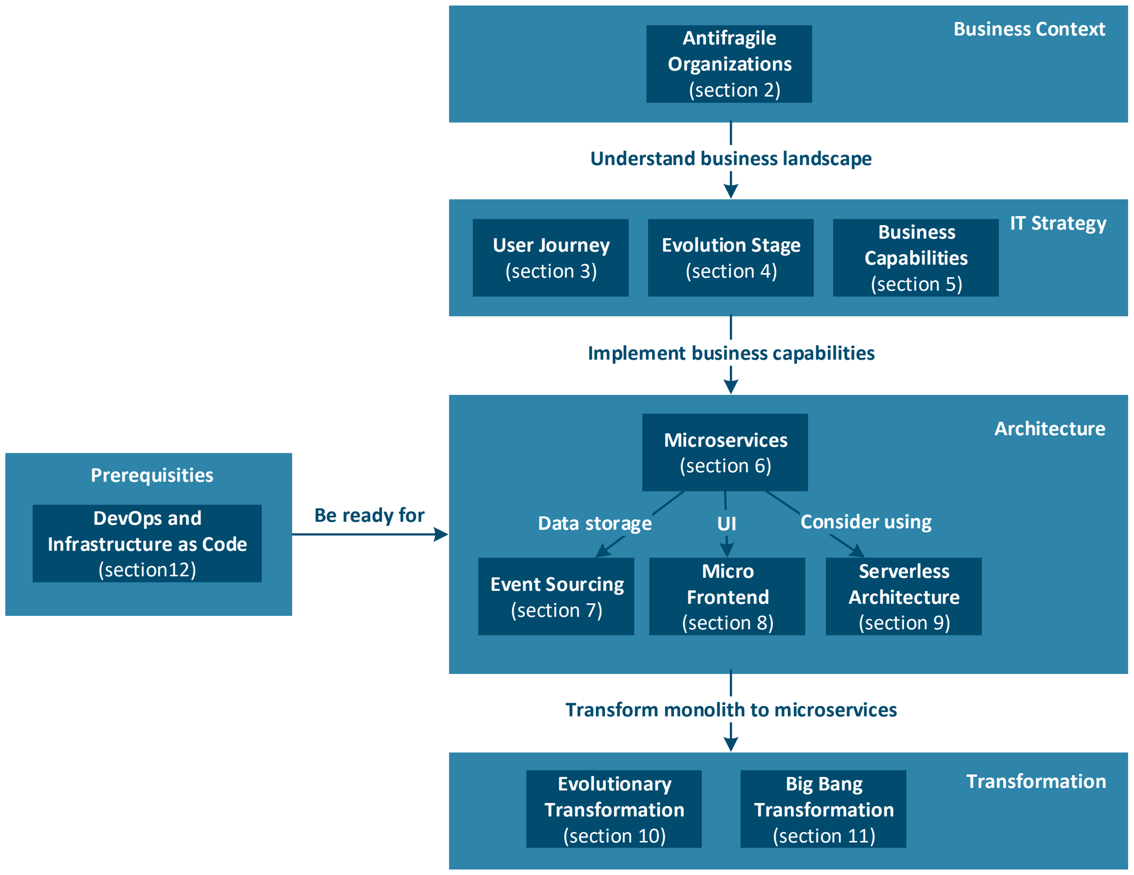
Business Context
Antifragile Organizations (section 2)
Understand business landscape
IT Strategy
User Journey (section 3)
Evolution Stage (section 4)
Business Capabilities (section 5)
Implement business capabilities
Architecture
Microservices (section 6)
Data storage
UI
Consider using
Event Sourcing (section 7)
Micro Frontend (section 8)
Serverless Architecture (section 9)
Prerequisities
DevOps and Infrastructure as Code (section12)
Be ready for
Transform monolith to microservices
Transformation
Evolutionary Transformation (section 10)
Big Bang Transformation (section 11)

Fig. 1. Pattern map – path to application modernization

2. ANTIFRAGILE ORGANIZATIONS

In summary, certain organizations grow after external shocks. What software architecture characterizes them?

During the COVID-19 pandemic in 2020 – 2022, many organizations were under pressure and their performance decreased. After adapting to the initial shock, a portion of companies returned to near-normal performance, but there were some companies, which performance, surprisingly, significantly increased. Several research and consulting firms started to analyze what makes these companies different and called them "antifragile organizations". Antifragility is a capability of an organization to emerge stronger from external shocks (Murray et al, 2020).

What characterizes an antifragile organization?

A UK-based technological and business research company Leading Edge Forum developed the Business Resilience Maturity Model that categorizes organizations at four levels of maturity (Murray and Aron 2017):

- Fragile – organizations that fail quickly after a shock
- Brittle – organizations that can operate normally for a short term after a shock, but sooner or later fail
- Resilient – organizations that experience a significant dip in performance, but can adapt to changes and return to near-normal performance
- Antifragile – organizations that grow after a shock. They do not only respond to shocks but seek and embrace them.

Antifragility has been observed to be related to microservice architecture.

Analysts of Leading Edge Forum (Murray 2020a, Daniel 2020) observed that antifragility is related to the microservice architecture: “Moving to smaller increments of functionality within a microservice-based ecosystem is a key facilitator of continuous access to functionality during the time of shock” (Murray 2021). “Many of the aspects of antifragility are facilitated by IT-related capabilities, such as a move to the cloud, DevOps, microservices, and architecting for more modular business models” (Daniel 2020).

The microservice architecture is necessary, however, not sufficient for an organization to become antifragile. Building an antifragile organization involves activities across the whole organization; it includes organizational change such as establishing mechanisms for working from anywhere, availability of resilient IT infrastructure, and adapting continually through experimentation. Among these activities are also deliberately introducing shocks, as an organization needs to be exposed to shocks to become antifragile and explore alternative opportunities in the organization’s ecosystem.

An important factor in enabling antifragility is understanding the patterns of USER JOURNEY and the EVOLUTION STAGE of components in the organization’s application portfolio. It has been observed that at the beginning of the COVID-19 pandemic in March 2020, organizations with a Wardley map adapted better than organizations without it (Daniel 2020b). Wardley map is described in the pattern EVOLUTION STAGE.

Benefits: An antifragile organization can better handle “black swans,” unexpected and unpredictable events with potentially severe consequences. Microservice architecture is an enabler to achieve this goal (Murray 2017).

Challenges: Microservices alone are not sufficient to make an organization anti-fragile; it also requires collaboration between IT and other parts of the organization. “There are cases in which antifragility will be costly, extremely so.” (Humble, 2013).

3. USER JOURNEY

In summary, application modernization projects have little chance of success if the needs of all of their stakeholders are not fully understood.

Migration to the microservice architecture must happen for the right reasons, which is often increased business agility and resiliency of the organization. One of the first things to understand in application modernization is what value it brings to the users, customers, developers, and all other stakeholders including business partners and government agencies. In other words, it is imperative to fully understand the needs and expectations of all stakeholders.

How to describe all the needs of all stakeholders?

The needs of users and other stakeholders can be determined in a variety of ways, for example, as user stories, use cases, by describing personas, and to a certain degree also by business modeling techniques such as business model canvas, SWOT (Strengths, Weaknesses, Opportunities, and Threats) analysis, Porter's Five Forces Framework. All techniques have different advantages and they fit different purposes. In the context of application modernization, we need a way of analyzing user needs, which would also identify the business capabilities of a new solution satisfying these needs.

A possible solution is a method called user journey.

User journeys describe the steps for completing a specific task within a system. Fig. 2 illustrates a buyer’s user journey in an e-shop: *Search for a product*, *Add to a basket* the selected product, specify *Delivery details*, provide *Payment*, and *Receive confirmation*. These steps can be matched by the business capabilities of an existing and future software system, see the BUSINESS CAPABILITIES pattern.

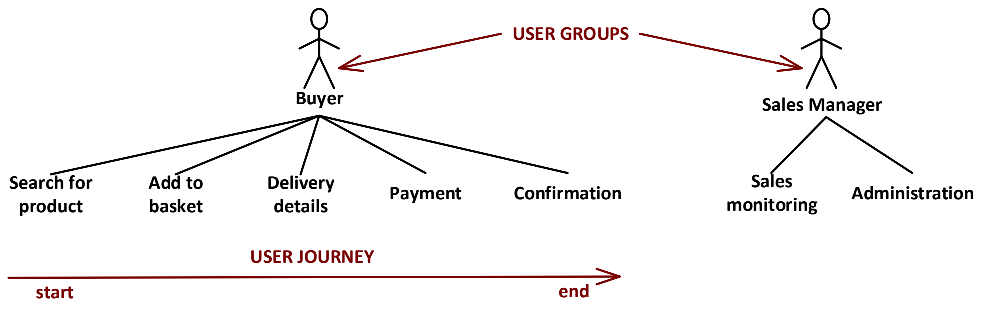

Fig. 2. User journey

To identify all BUSINESS CAPABILITIES of the system, the user journeys should be created for all user groups representing the stakeholders, such as buyers, financial institutions, sales managers, employees, shareholders, tax office, etc.

Benefits: The user journey specifies vertical partitioning of user needs, and focuses attention on things that are important for the stakeholders (TPXimpact 2020). It helps to determine the scope of MICROSERVICES, see also the BUSINESS CAPABILITIES pattern.

Challenges: Creating a user journey often requires the assistance of a UX professional, as creating user journeys requires a deeper and different skill set than creating informal user stories or use cases. Some regulatory restrictions and standards require a more thorough requirements management process, providing traceability between requirements, implementation, and tests.

4. EVOLUTION STAGE

In summary, components in an early stage of development have a different modernization path than custom-built solutions, products, and commodities.

Application modernization requires a significant effort and can only be achieved over a long period of time, broken down into smaller steps, each of which addresses a specific piece of functionality. Organizations must decide how to “slice the elephant” and where to focus modernization efforts.

Which applications and systems to focus on during software modernization?

An application portfolio usually consists of applications in various stages of evolution; such as applications under development in the genesis stage characterized by frequent changes, more stable custom applications, products and standard components provided by external vendors, and commodities, typically provided as a service.

When modernizing a software system, the evolution stage determines the modernization approach.

- The components in the genesis stage can be developed as microservices directly.
- The custom-built components, such as large client-server monoliths after many years of development, whose architecture resembles a “Big Ball of Mud” (Foote and Yoder 2003). Such components need to be transformed into microservices. It is the focus of this paper. For completeness, monoliths are not necessarily bad if they are well structured and could be modified quickly without much effort.
- If the custom-built client-server monolith contains functional areas covered by existing standard applications that could be consumed as a service, they are candidates for outsourcing. For example, if the client-server application contains a CRM functionality, it should rather be replaced by a standard CRM solution, than build a CRM microservice from scratch.

- Components in the commodity stage should be outsourced (unless your organization is itself a provider of these commodities); typically, they will be consumed as services of a cloud provider. Another exception to this rule is when your organization has developed a commodity that the rest of the market recognizes and acquires as a custom-built or product component – this is an excellent business opportunity; your organization should become a provider of this commodity and take advantage of the economy of scale.

The solution is to focus on large custom-built applications and systems.

The evolution stage of software and hardware components and applications, together with user needs see the USER JOURNEY pattern, can be visualized using a technique called Wardley mapping (Wardley 2017). The Wardley mapping is an IT strategy method developed by Simon Wardley (Leading Edge Forum, 2022, Mossier 2022). The map places software components in two dimensions: The vertical dimension represents user visibility – components more visible to the stakeholders are higher up on the map. The horizontal dimension represents the evolution stage: genesis stage, custom build, product, and commodity. The connections represent dependencies between components, thus forming a value chain; the components higher up on the map depend on the connected components lower on the map.

The components in a Wardley map should be understood informally, these "components" can represent any economic resource that an organization and its partners have under their control.

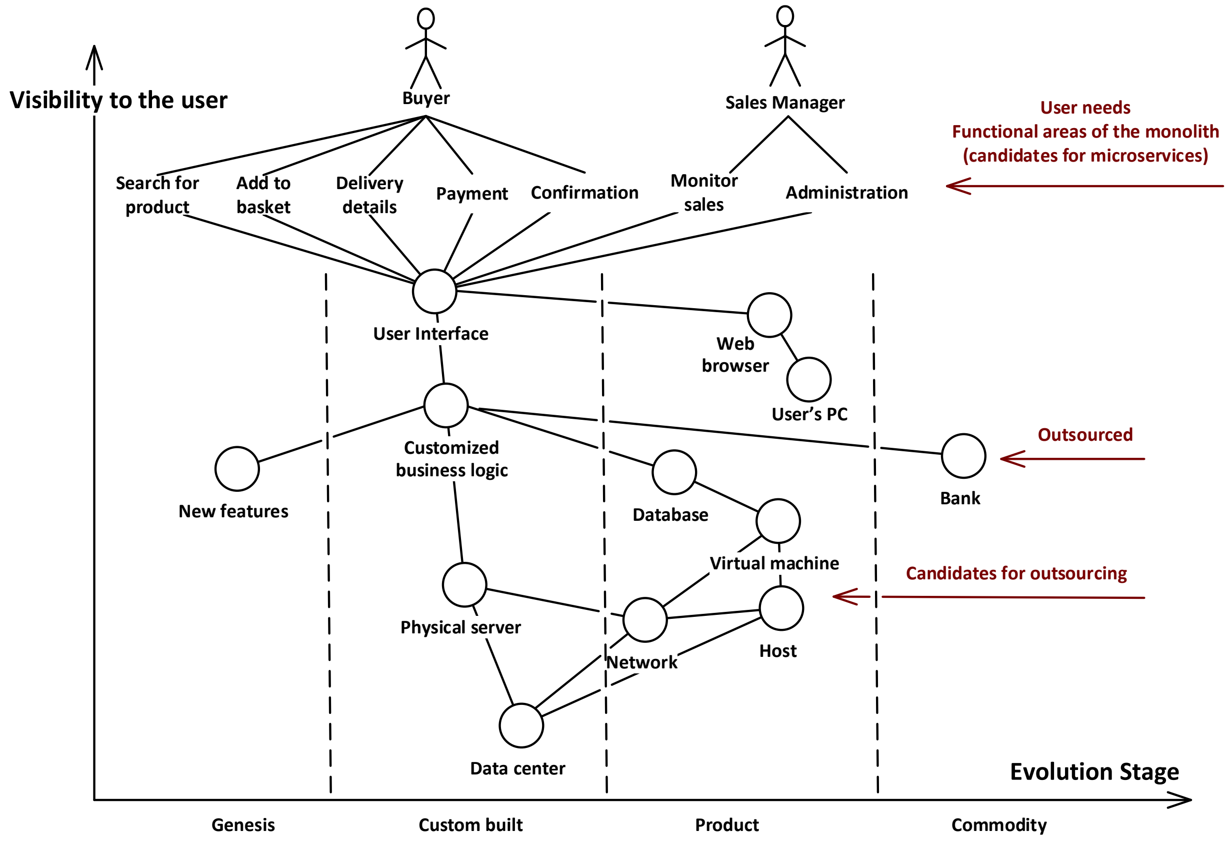

Fig. 3. A Wardley map of a client-server application

Wardley map can be created practically for any software system. Fig. 3 is an example of a Wardley map illustrating a web-based client-server application. The *User Interface* runs in a *Web browser* that runs on a *User's PC*. The application in a browser requires *Customized business logic*, and a *Database* to run. The application provides several features to its users, a *Buyer* and *Sales Manager*, such as *Search* for a product, *Add to basket*, *Delivery details*, *Payment* and *Confirmation of sale*, and *Sales monitoring*. *Customized business logic* is connected to a *Bank* providing financial services. The *Database* runs on a *Virtual machine*, while the *Customized* *business logic* component runs on a *Physical server*, which needs a *Network* and a *Datacenter*. The organization also develops *New features*, currently in the genesis stage.

The map in Fig. 3 is for illustration purposes and is not complete. Other stakeholders have their needs as well; senior management needs business agility; developers need to decrease the lead time from development to deployment; operations people need stability and decrease the risk of changes in the production environments. To get the complete picture, you should create additional Wadley maps describing the needs and user journeys of all identified stakeholders and how their needs are satisfied.

The evolution stage of each component determines many aspects of IT development, such as the required skills and mindset of the team. People who like experimenting, usually best fit the tasks on the components in the genesis stage, while people that understand the economies of scale better fit the tasks related to commodities. The evolution stage also determines the suitable development method, such as agile methods, embracing change, are best suited for the components in the genesis stage, while Lean is best suited to the products and commodities, where minimizing waste, stability and efficiency are most valuable (Leading Edge Forum 2022).

Benefits: Wardley map determines the overall software engineering approach in application modernization. The evolution stage of an application focuses attention on software components and services that should be modernized and those that should be outsourced and consumed as a service.

Challenges: Creating a map takes time. Nevertheless, it is an iterative process, and discussions about the map add the most value. Outsourcing encompasses many new tasks from the selection of the SaaS provider to budgeting and getting approval from senior management to purchasing licenses.

5. BUSINESS CAPABILITIES

In summary, the application should be partitioned around business capabilities, instead of technical concerns.

Understanding the business capabilities influences the architecture of a microservice application, described in the MICROSERVICE pattern, to a much larger degree, than in traditional client-server applications. Client-server applications are structured into layers, typically the presentation layer, business logic layer, persistence layer, and database layer, implementing the Layered Architecture pattern (Richards 2015). In the microservice architecture, the modular structure of the solution is different and is determined by business capabilities.

Client-server applications are typically developed by teams of specialists for each layer, such as user interface specialists, database specialists, etc., and consequently, changes in the application require careful planning and coordination among these teams. Due to high coupling and dependencies among application components, changes require redeployment of the whole application, which means that implementing features needed by business users is cumbersome and time-consuming, and can be released only a few times per year.

To mitigate these challenges, a new way of structuring the components in the modernized architecture is needed, so that its architectural components represent business capabilities, which could be developed and released independently.

How to determine the business capabilities of a new system?

There are many ways to determine business capabilities, such as a business capability map (Ardoq 2022), use cases, continuous business model planning (VDMBee 2022), requirements engineering methods, and to a certain degree also by Osterwalder's business model canvas (Osterwalder and Pigneur 2010). In the context of application modernization, we need to consider both user needs and the evolution stage of the modernized components.

User journey steps are initial candidates for business capabilities.

The USER JOURNEY pattern describes the steps of a user for completing a specific task within a system, which are good initial candidates for the business capabilities of the existing and new solutions. The EVOLUTION STAGE and the Wardley map help to select which capabilities should be fulfilled by microservices in the new modernized system.

If the application is in a domain for which there exists a well-established ontology, the ontological categories provide additional insights for the partitioning of business capabilities.

As an example, there exist several ontologies for the business domain, including REA - Resources, Events Agents (Hruby, Kiehn, Scheller 2006) and POA - Possession, Ownership, Availability (Scheller and Hruby 2016). These ontologies provide the insight, that the business capabilities could be partitioned into the management of transactions, economic resources, economic agents, policies, contracts, etc. Alternatively, the partitioning could be done horizontally, so that each business process (i.e., the duality relationship in the REA ontology) has its own microservice.

The task of determining the size and scope of business capabilities and consequently MICROSERVICES is complex and usually undergoes several iterations. As microservices implementing business capabilities must be isolated from each other (no shared database, etc., see the MICROSERVICES pattern), partitioning business capabilities usually require the attention of a software architect.

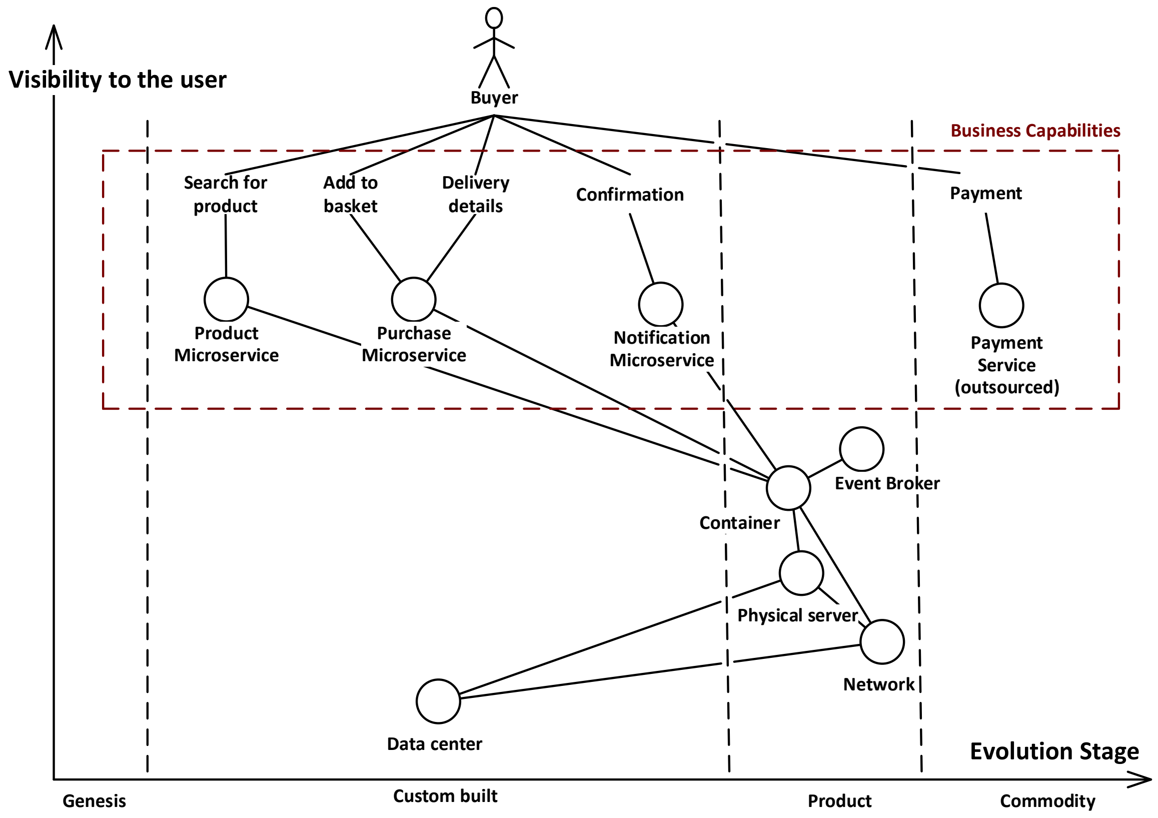

Fig. 4. Business capabilities

Fig. 4 illustrates an example of a user journey with business capabilities *Search for a product, Add to basket, Delivery details, Confirmation,* and *Payment*. We can decide to implement these business capabilities using the microservices *Product, Basket*, and *Notification*, and outsource the *Payment*.

Benefits: Wardley map helps structure microservices according to business capabilities instead of technical concerns.

Challenges: Strangling the monolith, that is, identifying business capabilities of the monolith that can be extracted as a microservice is not a simple task and usually requires assistance from a software architect.

6. MICROSERVICES

In summary, microservices are loosely coupled applications, with their user interface, business logic, and storage, communicating with other microservices using events. The event broker transmits events between microservices.

Before applying this MICROSERVICE pattern, you identified the components in the application portfolio that are candidates for application modernization, for example, using the Wardley map described in the EVOLUTION STAGE pattern. The candidates for application modernization are typically custom-built large monoliths with thousands of tables, whose internal architecture can be described as a big ball of mud (Foote and Yoder 2003). It should be noted that there exist small, well-structured monoliths, where changes can be implemented quickly and which do not need to be modernized because of business agility.

In the architecture of large custom-built monolithic components, it takes often too long from identifying the requirement, through software development, to the deployment of the new feature into production. Teams of specialists often must wait too often for each other, which slows down progress. It applies mainly to large monoliths with thousands of tables, discussed in the introduction section.

The organization cannot respond quickly enough to market changes and opportunities, because it takes too long to develop new features.

Traditionally, client-server systems have been structured according to technical concerns – a user interface, business logic, database, network, infrastructure, security, etc. According to Conway's law, the development organization of a client-server system is then typically structured into a user interface team, database team, network team, infrastructure team, security team, etc. Implementing and delivering a new feature requires meticulous coordination between these teams, with long lead times, as described in (Kim 2014).

This pattern deliberately does not discuss all different interpretations and approaches of the microservice architecture, such as SOA, monolithic user interface, and synchronous communication, because the purpose of this paper is to provide a reader with a single direct path to microservice architecture that has been proven successful. A reader interested in both problematic and successful approaches to microservice design, can consult the book (Newman 2021).

The solution is to rewrite custom-built monolithic components into smaller and simpler applications, each with its own user interface, business logic, and storage, that communicate with other applications using asynchronous events.

The new applications must be small enough for a self-coordinated team can work on the application independently from other teams. The applications can be tested and deployed independently, and consequently, have independent release cycles These smaller and simpler applications are called microservices.

> *Some authors define microservices differently, for example, as described in Microsoft blog What is Cloud Native (Microsoft 2022a). In this definition, microservices do not have a user interface on their own;*

instead, the architecture has a monolithic user interface, see Figure 1-4 in the blog. "The microservice approach … segregates functionality into independent services, each with its logic, state, and data".

Our position is that the monolithic user interface shared among different microservices increases coupling, which decreases the benefits of the microservice architecture outlined in the introduction section. Loosely coupled components must be loosely coupled not only in the back end but also in the user interface, to gain the benefits of the microservice architecture. The known use of our approach is more aligned with the microservice architecture described in (AWS 2022a), where the user interface, hosted on Amazon CloudFront, is part of the microservice.

The event broker is an essential element of the microservice architecture, which distinguishes our interpretation of microservices from others. Other literature allows microservices to communicate directly with each other, or via an enterprise service bus (ESB) by synchronous calls, but the result is less loosely coupled architecture, such as in the case of Service-Oriented Architecture (SOA). It reduces the main benefit of the microservice architecture, which is business agility.

We can distinguish three types of microservices, see Fig 5:

- **User interaction microservices,** with their own user interface. Typical examples are transactional applications implementing each single business capability, such as product catalog and shopping basket.
- **Autonomous microservices**, without their own user interface. These microservices perform automation tasks and only respond to the events received from the event broker. Examples of such microservices are intelligent agents, which are small applications simulating user behavior according to a certain algorithm, or microservices encapsulating policies and rules in case management systems. Other examples are data warehouse microservice, archiving microservice, and microservices providing combined database views for reporting.
- **Integration microservices**, receiving and providing input from the external systems using the microservice's API. A special case of this type of microservice is an adapter, discussed in more detail in the EVOLUTIONARY TRANSFORMATION pattern.

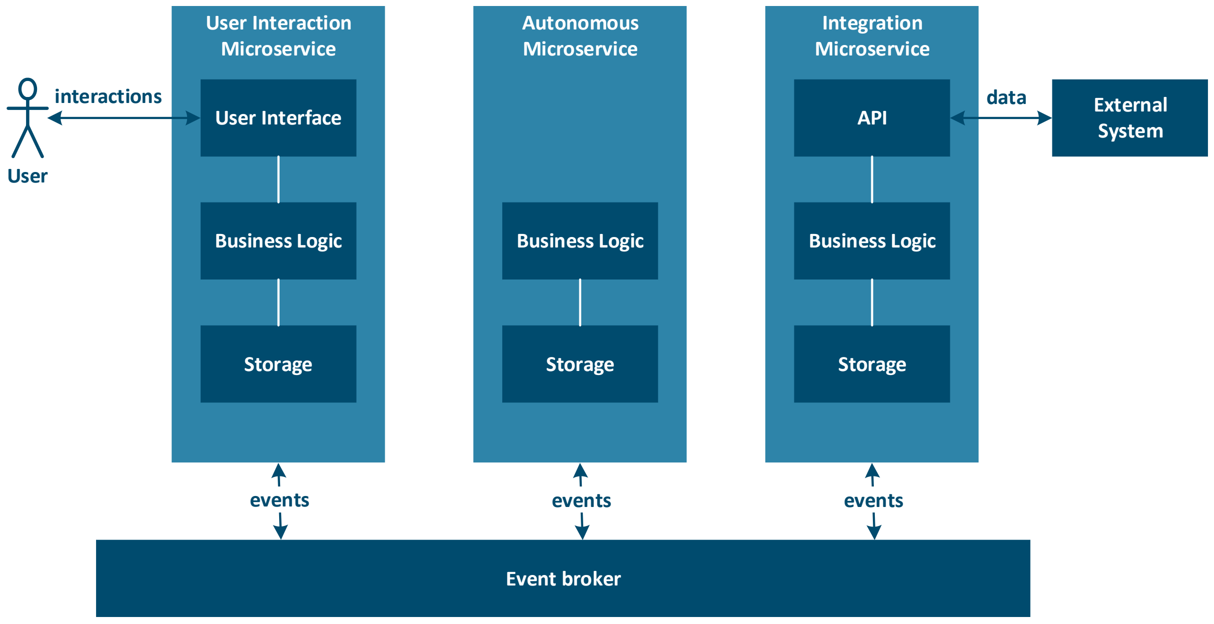

Fig. 5. Microservices are small applications

Each microservice implements its own storage of generated and received events, see the EVENT SOURCING pattern for more details. Own storage is essential for achieving loose coupling between microservices. Creating bundles - microservices sharing the features of the three archetypes might be tempting as it reduces duplication of code and data, however, we must be aware that by allowing bundles we are entering a slippery slope to reintroducing a monolith.

The microservice architecture solves several problems of the monolithic architecture:

- Each microservice is developed by a small full-stack team independently of the others, so many teams can simultaneously work on the whole system.

As microservices are independent of each other, they can be deployed frequently, whenever needed.

- Each microservice can scale out independently of other microservices. It helps reduce overall costs, because only microservices under load are scaled out, instead of the entire system.

The microservice architecture solves the challenges outlined above, but at a cost, that should not be underestimated. There are special requirements for the microservice design due to the distributed computing model:

- **Commutativity**: the microservices must correctly process the events received in the incorrect order, so that "changing the order of the events does not change the result" (Wikipedia 2022a). Receiving events in an incorrect order can occur in situations when one of the microservices is under a heavy load and generates events at a slower pace than other microservices at a normal load.
- **Idempotence**: the microservices must correctly process a single event received more than once, so that "receiving an event multiple times does not change the result beyond the initial application" (Wikipedia 2022b). A microservice can resend events after a microservice has been restored after a failure or maintenance. Also, rollback causes the events to be resent because data storage of different microservices is not in a single transaction scope.
- **Race conditions**: A race condition denotes a situation when the system's behavior is dependent on the sequence of uncontrollable (usually external) events, and one or more of the possible behaviors is undesirable. For example, an ordering microservice accepts a customer order after the CRM microservice has deleted or inactivated the customer. In the microservice architecture, the race condition is not a technical but a business problem and should either be solved by business logic or passed to a user for resolution.
- **Data redundancy**: because each microservice must function as independently as possible of the others, and because microservices cannot rely on shared storage, different microservices contain duplicated data.
- **Eventual consistency:** as each microservice can change its data, and inform others about the changes using asynchronous events, for some time after the change, the data duplicated between different microservices will be inconsistent. "Eventual consistency guarantees that all data will be eventually updated to the last updated value if no new updates are made" (Wikipedia 2022c).
- **Conflicts:** different microservices can update the duplicated data while in an inconsistent state. This problem cannot be eliminated completely due to the CAP theorem (Wikipedia 2022d), and errors (usually very rare) must be resolved at the business level. For example, in a booking application, a double booking may occasionally occur. The risk of conflicts can often be mitigated by having good business boundaries between microservices; so that microservices represent separated BUSINESS CAPABILITIES.
- **Integration complexity:** the microservice architecture requires a new component, the event broker.
- **Event size limit:** The size of the events allowed by the event broker determines the applicability of the microservice architecture. For example, Kafka accepts events up to 1MB by default. Documents, videos, and other media files are too large to be transmitted as events. In such situations, the microservice architecture is not applicable. A solution is a different architecture, where the documents and media files are stored in central storage communicating with other components via an API. The central storage is not a microservice - it does not communicate only by events with other microservices, but also via an API. The central storage is also a single point of failure. Other business capabilities, such as a catalog, playlists, and history can be implemented as loosely coupled microservices, communicating via events containing only links to the documents and media files.
- **Partitioning of business capabilities.** Each microservice represents a business capability. When strangling a monolith, a new microservice must be isolated from the rest of the system, including data,

business logic, and the user interface. Although the USER JOURNEY and EVOLUTION STAGE patterns provide initial candidates, there is not a simple solution for partitioning microservices; it is a task for a software architect.

- **Single sign-on,** as a single authentication and authorization should suffice for multiple microservices.
- **Consolidated logging** is necessary for development and troubleshooting purposes - developers should be able to examine the events in the log, consolidated using, for example, a correlation ID.

There are solutions to some of these challenges, for example, commutativity and idempotence of microservices can be addressed by the EVENT SOURCING pattern.

Benefits: The microservice architecture solves some problems of the monolithic client-server architecture, such as development by independent teams, independent deployment, and independent scaling.

Challenges: The distributed computing model poses challenges such as commutativity, idempotence, race conditions, data redundancy, eventual consistency, conflicts due to updates in an inconsistent state, integration complexity, event size limit, and necessary are also single sign-on and consolidated logging.

7. EVENT SOURCING

In summary, the sequence of changes is the source of truth, instead of the current state of a database.

The microservice architecture is a distributed computing model, which imposes certain challenges to microservice design. In particular, microservices must be designed to correctly process duplicated events, and events received in an incorrect order. These properties are called idempotency and commutativity.

Microservices must be designed for idempotency and commutativity.

There are several engineering solutions to achieve idempotency, such as (AWS 2023) and different solutions to achieve commutativity, an overview is, for example, in (LWN 2015). There is a pattern that solves both, called event sourcing. Event sourcing provides additional benefits such as increased performance, resilience, additional possibilities for testing, and defect tracking.

A solution to both idempotency and commutativity is the event sourcing pattern.

Event sourcing is a way of persisting application state as a sequence of events, each representing a change in the application state. The events become the source of truth (hence the name of this pattern). It is possible at any time to delete the application state, and reconstruct it by replaying the events in the event store (Fowler 2016), see Fig 6.

If a duplicated event is detected in the event store, the microservice can ignore it; in this way implementing the idempotency property. The microservice can sort the events in the event store and process them in the correct order, in this way implementing the commutativity property. The order can be chronological or based on a version of the updated resource.

The number of events in the event store grows over time. To limit its size, a microservice can at specified time intervals create a snapshot of an application state, and eventually, delete or archive all events older than the snapshot. Idempotency and commutativity can no longer be guaranteed beyond the date in the past after which the events have been deleted. A similar mechanism is known from financial accounting as a fiscal year closing. This mechanism stores the balance at the end of the fiscal year, and all entries from the previous year are deleted.

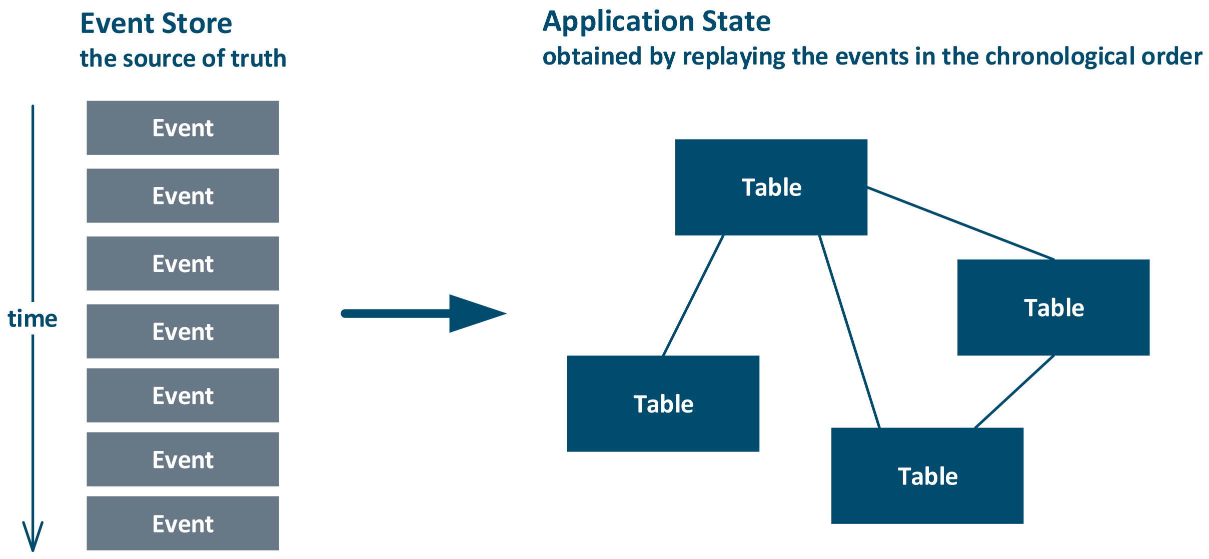

Fig. 6. Event sourcing

Event sourcing has many additional advantages both for business users and for developers, such as increased performance – when an event arrives, a microservice only inserts the event to its event store, and updates its application state later, when needed, as illustrated in (Microsoft 2022b). Replaying the events until a certain past date allows reconstructing the application state that was in the past. Event sourcing allows for simulations of various business scenarios (the application stops writing to the event store), without impacting the rest of the system. When developing and debugging business logic, developers can inspect the event store. Testers can easily reconstruct the initial and any other state of the application.

Benefits: A way to implement idempotency and commutativity. Increased performance. Easier debugging and testing.

Challenges: Event sourcing can be expensive in terms of CPU usage. It is an unfamiliar programming model for mainstream developers because traditional design practices, such as normalized relational databases, no longer apply. Data duplication is no longer a problem to fix; on the contrary, data duplication is a natural characteristic of microservice architecture.

8. MICRO FRONTEND

In summary, a microservice's user interface occasionally needs to display part of the user interface of another microservice.

The MICROSERVICE pattern specified microservices as small applications, each of a specific business capability, with its own user interface, business logic, and storage. Business users, while using a specific microservice, would like to also access the business capabilities of another microservice.

A microservice often needs to display part of the user interface from another microservice.

For example, consider two microservices, Sales Order and Product. A Sales Order microservice encapsulates the business transaction and a Product microservice encapsulates the Product business capabilities. While working on a Sales Order, users would also like to see an image of a Product. The product is handled by another microservice, but users do not want to switch to the Product user interface while working on a Sales Order.

The architecture should make it possible without increasing coupling between microservices.

The solution is a concept called micro frontend.

A micro frontend is a section of a microservice's user interface, embedded in the user interface of another microservice, see Fig. 7.

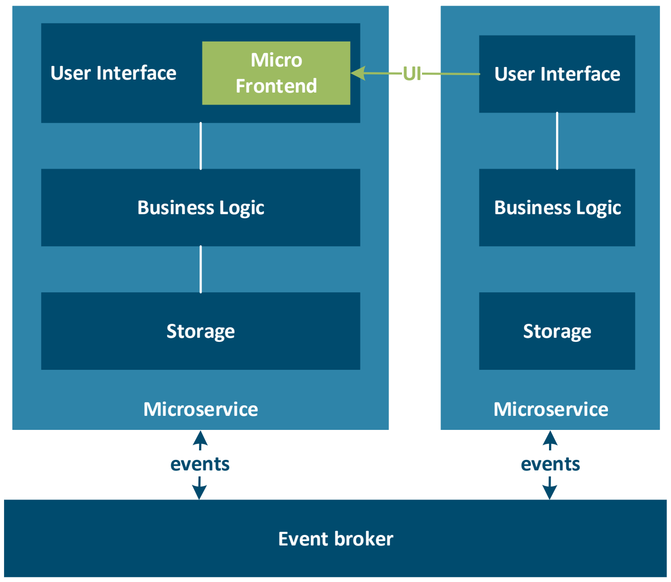

Fig. 7. Micro frontend

Fig. 8 illustrates an example of two microservices, *Product* and *Basket*. The user interface of the *Product* microservice contains an element *Add to basket*, which is the micro frontend of the *Basket* microservice. Clicking it opens the main user interface of the *Basket* microservice. Similarly, the user interface of the *Basket* microservice contains an image of the product, which is a micro frontend of the *Product* microservice. The data (the image) is supplied by the *Product* microservice and clicking it opens the main user interface of the *Product* microservice.

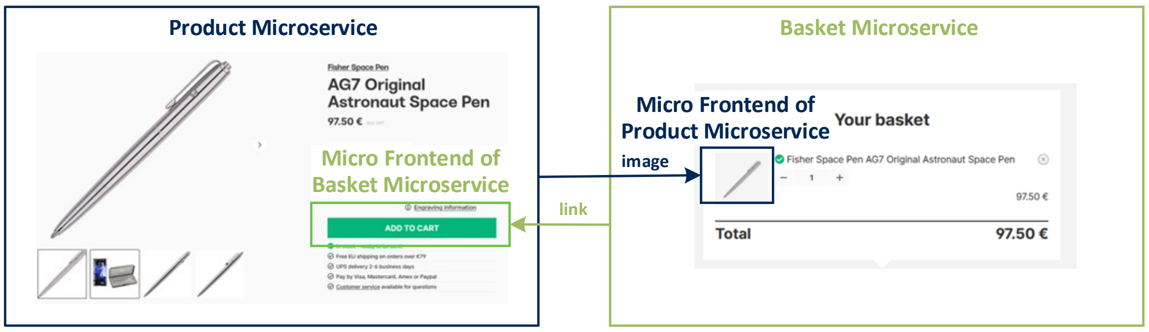
Product Microservice
Fisher Space Pen
AG7 Original Astronaut Space Pen
97.50 €
Micro Frontend of Basket Microservice
Engraving Information
ADD TO CART
Free EU shipping on orders over €79
UPS delivery 2-6 business days
Pay by Visa, Mastercard, Amex or Paypal
Customer service available for questions
link
Basket Microservice
Micro Frontend of Product Microservice
image
Your basket
Fisher Space Pen AG7 Original Astronaut Space Pen
1
97.50 €
Total
97.50 €

Fig. 8. Examples of micro frontends

Benefits: The micro frontend provides integration at the user interface level, while maintaining loose coupling between microservices. Technical details of how to implement the integration can be found, for example in (Jackson 2019).

Challenges: The micro frontend might have a different response time than the host microservice, caused by a heavy load, network latency, and other factors, which might influence user experience.

9. SERVERLESS ARCHITECTURE

In summary, the serverless architecture uses native managed services of a cloud provider, instead of deploying applications in virtual machines or physical servers.

Typically, microservices are deployed in containers and can run both on-premises data centers and hosted by any cloud provider. Microservices deployed in the containers allow for taking advantage of certain limited benefits of cloud computing, such as horizontal scalability and automated infrastructure. However, cloud provider also offers many useful managed services that provide additional benefits.

How to operate a microservice architecture without taking care of the underlying infrastructure?

Hosting containers with microservices on-premises or in the cloud has associated activities with configuring and patching the infrastructure. These activities do not directly contribute to the business benefits of the end users. As cloud providers provide these activities as services, and they already became commodities, these activities are candidates for outsourcing, see the EVOLUTION STAGE pattern.

The solution is a "think serverless first" mindset.

The "think serverless first" is a mindset of embracing using native managed services of a cloud provider whenever possible, such as building microservices from Microsoft Functions or AWS Lambda, illustrated in (AWS 2022b). The serverless architecture eliminates the infrastructure management tasks and is usually cheaper because the cloud provider can utilize the economy of scale. "Managed services can often save the organization hugely in time and operational overhead" Grey (2022). This architecture also appeals to the developers (Roy 2022, Kazimierczak, 2022), because they can often implement a solution only by configurations, which makes the developers more productive.

The serverless architecture has several drawbacks, real or perceived. The serverless architecture often implies vendor lock-in; applications heavily dependent on a cloud provider's managed services will be challenging to transfer to another cloud provider, or back to premises. Another potential problem is that the cloud provider might retire the managed services the application depends on or introduce backward-incompatible changes that will require an upgrade of the application. Some cloud providers such as Microsoft are trying to mitigate these concerns, for example by Azure multi-cloud and hybrid cloud solutions eliminating the reliance on a single cloud provider. However, overall complexity increases, making the solution less resilient, and security and governance become more complicated.

Benefits: Usually cheaper than deploying a microservice in a container. Easier to use for developers.

Challenges: The cloud provider might retire the managed services the application relies on. Often vendor lock-in.

10. EVOLUTIONARY TRANSFORMATION

In summary, the monolith is integrated with the event broker of the microservice architecture. Newly developed microservices gradually replace the functionality of the monolith.

Decomposing a client-service monolith into microservices can take up to several years (Franz 2022). The question is, can users start using the new microservices as soon as they are developed, tested, and deployed, or it is necessary to wait until the whole project is finished?

Users would often like to use new microservices as soon as they are ready.

New microservices initially cover only a small part of the business capabilities of the old system. It will take several months, even years until the new system is fully functional. Is there a way to allow users to use new microservices and the legacy system in parallel?

A solution is called evolutionary transformation.

In the evolutionary transformation (Stiller 2019), the old system must be modified to consume and raise events and communicate with the event broker; see the Adapter component in Fig. 9.

The users can choose to use the newly developed microservices, and they can also continue using the existing application. As new microservices will provide additional user benefits, such as modern user interface and new functionality, users are expected to prefer new microservices and gradually use the old application less often, so it could be phased out, see Fig 9.

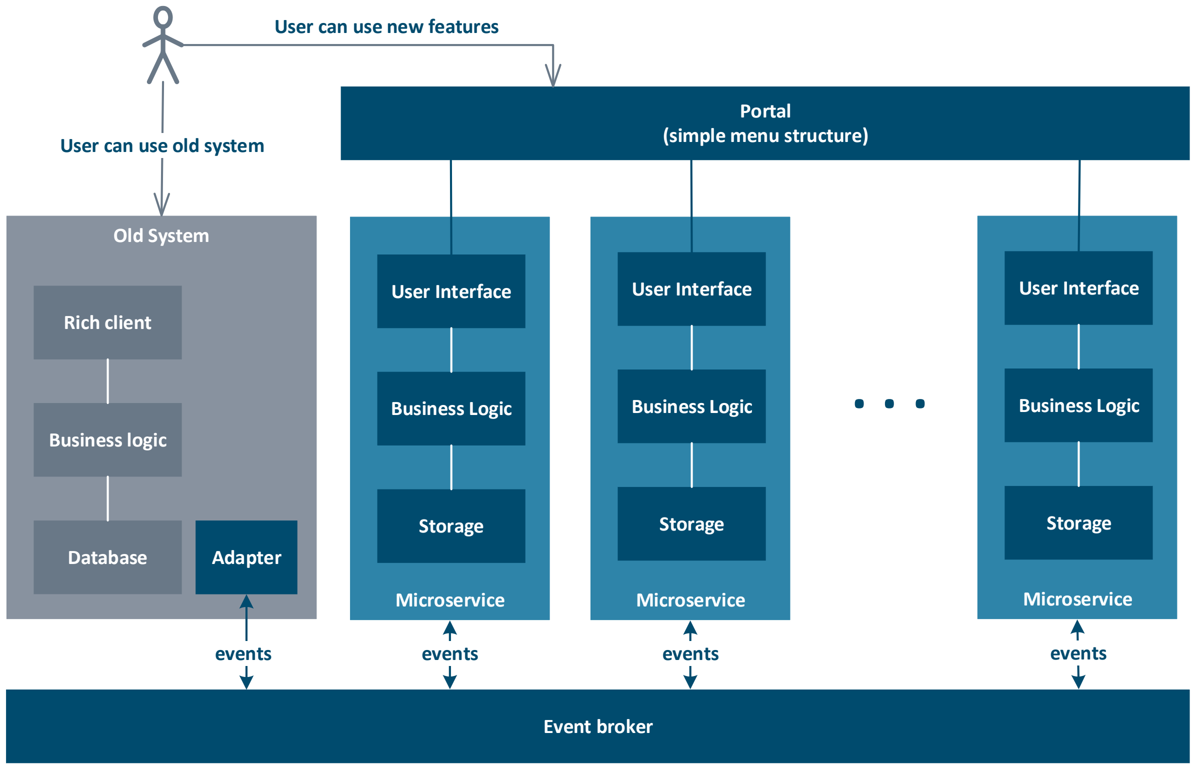

Fig. 9. Architecture supporting the evolutionary transformation

Developing an adapter might be challenging, depending on the architecture of the old client-server application. The old system's API can usually consume events from the event broker. If the old system can generate events, it can communicate with the event broker directly. If this is not possible, the events could be a result of a periodic batch job exporting changes in the state of the old application. Another possible solution is to set up triggers on Insert, Update, and Delete statements in a database and derive the events from the changes in database records.

Compared to the BIG BANG TRANSFORMATION, the evolutionary approach allows for receiving feedback from the users early. If there are unforeseen inherent problems in the architecture, they get revealed early. The evolutionary approach also improves the return on investment due to the early adoption of the parts of the new system.

Benefits: Minimizes the project risk by allowing it to fail early. Improves return on investment by early adoption.

Challenges: Requires building an adapter for the old system, which is a waste, as the old system will be retired.

11. BIG BANG TRANSFORMATION

In summary, the new system will not be used until it is fully completed.

There are cases when EVOLUTIONARY TRANSFORMATION is not possible or desirable. An alternative is the big bang approach, the users use the old system, until the new system is completed.

What if evolutionary transformation is not possible or desirable?

There could be contractual requirements from the customer, the old system is sufficiently small to finish the transformation within a time that can justify waiting, such as in 6-9 months. For longer periods it would be hard to keep the old system without any changes, and changes to the old system mean changes to the specification of the new system.   It may also be the case that the conceptual model of the new system is vastly different from the conceptual model of the old system. This may make it difficult to continuously map between the conceptual model of the new system and the conceptual model of the old system, as is necessary when events must be shared between the systems via the event broker.

The solution is the big bang transformation.

In the big bang transformation, the users use the old system until the new system is completed and fully tested. Then the data is migrated from the old system to the new system, the users start to use the new system, and the old system is retired. The architecture supporting this transformation is in Fig. 10.

The big bang transformation is typically cheaper, as it does not require changes in the old system (i.e., building the adapter). Furthermore, some organizations prefer to manage their IT projects in this way.

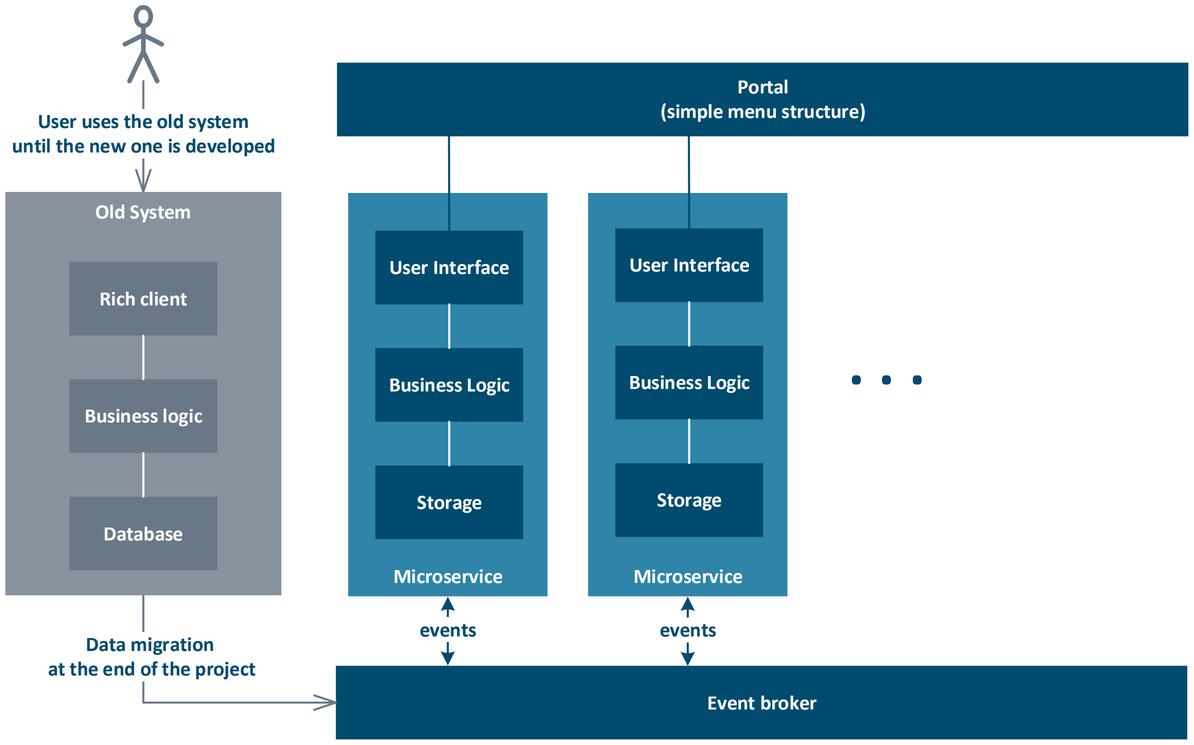

Fig. 10. Architecture supporting the big bang transformation

A combination of these two archetypical approaches is certainly possible, for example, your organization could choose the evolutionary transformation as the overall approach, but some parts of the monolith must be transformed like a big bang (Yoder and Merson 2020, Newman 2021).

Benefits: Often cheaper than the evolutionary approach

Challenges: Long time, often several years, until the users can use the new system

12. DEVOPS AND INFRASTRUCTURE AS CODE

In summary, the infrastructure as code and DevOps practices are two essential prerequisites for the successful implementation of the microservice architecture.

The ANTIFRAGILE ORGANIZATIONS pattern shows that the microservice architecture is necessary, though not sufficient, for a large organization to become antifragile. The microservice architecture enables rapid response to business opportunities by modifying existing microservices and developing and deploying new microservices at the speed the business demands. However, new microservices require the infrastructure to be provisioned and modified at the same speed.

Business agility and antifragility are not possible without agile infrastructure provisioning.

If the developers need to wait weeks, even months for a new server, organizations are unable to respond quickly to market changes and opportunities. Likewise, the developers embrace change as the fundamental agile principle (Beck and Andres 1999). However, changes represent risks in service delivery, and minimizing these risks is the fundamental goal of operations management. If the developers following agile development must constantly contend with the resistance to changes inherent in operations, organizations are unable to respond quickly to changes and market opportunities.

The solution is called DevOps and the infrastructure as code.

Infrastructure as code automates the infrastructure provisioning and enables practices such as immutable virtual machines and containers. DevOps practices automate the process from the code pushed to a source code repository until deployment to the production environment, including CICD pipeline and unit, integration, and acceptance tests.

Developing infrastructure as code on-premises and introducing DevOps practices requires time and IT skills. Today practically every cloud provider offers services for configuring the infrastructure as code and DevOps practices. The decision of whether to develop them on-premises or consume them as a service from a cloud provider is discussed in the EVOLUTION STAGE pattern. If an organization considers infrastructure as code a commodity, it should be outsourced to a cloud provider, unless there are business reasons not to.

The reasons against outsourcing infrastructure to a cloud provider could be requirements for absolute reliability and absolute security, such as in nuclear power plants and similarly critical production facilities. Such very restricted environments and some government agencies are isolated from the outside world and implementing DevOps practices with frequent changes to the production is not desirable for production stability and security reasons (Nielsen 2021).

DevOps and the infrastructure as code are not the only prerequisites for successful microservices implementation. The right mindset, and the right team structure, organized around business capabilities, continuous feedback from the production environment, monitoring, and intelligence from incidents and operations, are essential to successfully implementing the microservice architecture.

Benefits: Increased deployment frequency decreases the lead time for changes and the time to restore service, and significantly decreases the number of failures (Google 2021).

Challenges: Setting up automated DevOps and Infrastructure as Code takes time and experience. It is typically easier to configure and use them in the cloud environment than in on-premises environments.

13. CONCLUSION

It is important to modernize for the right reasons. The Wardley map visualizes the needs of all stakeholders, such as customers, end-users, developers, IT operations, and management, how well IT services meet these needs, and what IT modernization steps an organization needs to take to meet these needs and grow after a shock.

Monolithic client-server applications can be modernized by replacing the application functionality with microservices – loosely coupled applications, that each has its own user interface, business logic, and storage, which communicate through events. Microservices deployed at a cloud provider also gain the benefits of automated infrastructure and horizontal scalability.

Many patterns in this paper can also be used by new organizations that are designing their IT systems from scratch to support business agility.

REFERENCES

Ardoq. 2022. Accelerate Your Business and IT Planning With Business Capabilities. Retrieved 28. January 2023 from https://content.ardoq.com/accelerate-your-business-and-it-planning-with-business-capabilities

AWS. 2022a. *Microservices architecture on AWS*. Retrieved 8 February 2022 from https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/simple-microservices-architecture-on-aws.html.

AWS. 2022b. *Amazon Serverless microservices*. Retrieved 8 February 2022 from https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html

AWS. 2023. Making retries safe with idempotent APIs.Retrieved 27 January 2023 from https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/

Kent Beck, Cynthia Andres. 1999. *Extreme Programming Explained: Embrace Change* (Addison-Wesley 1999).

Melvin Conway. 1968. How do committees invent? *Datamation: 28-31*. http://www.melconway.com/Home/pdf/committees.pdf

Krzysztof Daniel, Carl Kinson C, Caitlin McDonald C, Bill Murray. 2020. *Shock Treatment: Developing Resilience & Antifragility*. Leading Edge Forum, October 2, 2020, https://leadingedgeforum.com/insights/shock-treatment-developing-resilience-antifragility/ and https://leadingedgeforum.turtl.co/story/shock-treatment-developing-resilience-and-antifragility/page/1

Krzysztof Daniel. 2020b. Presentation for the DXC Architecture Guild.

Brian Foote, Joseph Yoder. 2003. Big Ball of Mud, https://www.researchgate.net/publication/2938621_Big_Ball_of_Mud

Martin Fowler. 2016. Event Sourcing. *WOW! Nights, March 2016*. https://www.youtube.com/watch?v=aweV9FLTZkU&t=145s

Joseph Franz, Solution Architect at DXC. 2022. *Personal communication*

Google. 2021. *State of DevOps 2021*. Retrieved 8 December 2021 from https://cloud.google.com/devops/state-of-devops

Tom Grey. 2022. *5 principles for cloud-native architecture—what it is and how to master it*. Retrieved 13 February 2022 from https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it

Pavel Hruby, Jesper Kiehn, Christian Scheller. 2006. Model-driven design using business patterns. Springer 2006.

Jez Humble. 2013. On Antifragility in Systems and Organizational Architecture. *Continuous Delivery*. https://continuousdelivery.com/2013/01/on-antifragility-in-systems-and-organizational-architecture/

Cam Jackson. 2019. *Micro Frontends*. Retrieved 10 February 2022 from https://martinfowler.com/articles/micro-frontends.html

Czeslaw Kazimierczak, System Architect at DXC. 2022. *Personal communication*

Gene Kim, Kevin Behr, George Spafford. 2014. *The Phoenix Project, A Novel About IT, DevOps And Helping Your Business Win*. NBN Tradeselect., Kindle edition, location 5505,

Leading Edge Forum. 2022. *Wardley Mapping, Learn How to Stimulate Future Ideas and Strategies*. Retrieved 10 February 2022 from https://learn.leadingedgeforum.com/p/wardley-mapping/?product_id=1606147.

Susanne Kaiser. 2020. Preparing For a Future Microservices Journey Using Wardley Maps. *DDD Europe 2020*. https://www.youtube.com/watch?v=csjaxNGF5LQ

Herwig Manaert, Jan Verelst, Peter de Bruyn. 2016. Normalized Systems Theory, From Foundations for Evolvable Software Toward a General Theory for Evolvable Design, Koppa Media

LWN. 2015. Creating scalable APIs. Retrieved 27 January 2023 from https://lwn.net/Articles/633538/.

Microsoft. 2022a. *What is Cloud Native?* Retrieved 8 February 2022 from https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition

Microsoft. 2022b. *CRUD, only when you can afford it*. Retrieved 8 February 2022 from https://docs.microsoft.com/en-us/archive/blogs/maarten_mullender/crud-only-when-you-can-afford-it-revisited.

Ben Mosier. 2022. *Wardley Mapping: Strategy for the Self-Taught*. Retrieved 13 February 2022 from https://learnwardleymapping.com/

Bill Murray. 2020. Building an Antifragile Organisation, *CEO Today*, October 21, 2020. https://www.ceotodaymagazine.com/2020/10/building-an-antifragile-organisation/.

Bill Murray, Dave Aron. 2017. *Rethink Risk Through The Lens of Antifragility*. https://leadingedgeforum.com/media/1983/rethink-risk-through-the-lens-of-antifragility.pdf

Bill Murray. 2021. Senior Researcher and Advisor at Leading Edge Forum. *Personal communication*

Sam Newman. 2021. *Building Microservices - Designing Fine-Grained Systems*. O'Reilly Media. 2021

Anders Holte Nielsen. 2021. Cyberattack at Demant, Contingency that works in practice. *Morning briefing at DevoTeam, Denmark*, 14. October 2021

Alexander Osterwalder, Yves Pigneur. 2010. Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers, Wiley.

Mark Richards. 2015. *Software Architecture Patterns*, Layered Architecture, O'Reilly Media, Inc. Retrieved 28 January 2023 from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html.

Anirban Roy, Solution Architect at DXC. 2022. *Personal communication*

Christian Scheller, Pavel Hruby. 2016. Business Process and Value Delivery Modeling Using Possession, Ownership and Availability (POA) in Enterprises and Business Networks, Journal of Information Systems, Journal of Information Systems 30(2) :141231074316001

Eran Stiller. 2019. 6 Lessons I Learned on My Journey from Monolith to Microservices. *NET Fest 2019*. https://www.youtube.com/watch?v=Isz16TZtWRM

Nassim Nicholas Taleb. 2012. *Antifragile: Things That Gain from Disorder*. Random House 2012

TPXimpact. 2020. *Wardley Mapping during a pandemic*. Retrieved 3 December 2020 from https://difrent.co.uk/blog/wardley-mapping-during-a-pandemic/

Joe Yoder, Paulo Merson. 2020. Strangler Patterns. Proc. of 27th Pattern Lang. of Prog. 2020.

Simon Wardley. 2017. *Wardley maps, Topographical intelligence in business*. Retrieved 8 February 2022 from

https://medium.com/wardleymaps
VDMBee, 2022. Continuous Business Model Planning. Retrieved 28. January 2023 from https://vdmbee.com/community/continuous-business-model-planning-cbmp/
Wikipedia 2022a. *Commutative property*. Retrieved 8 February 2022 from https://en.wikipedia.org/wiki/Commutative_property
Wikipedia. 2022b. *Idempotence*. Retrieved 8 February 2022 from https://en.wikipedia.org/wiki/Idempotence
Wikipedia. 2022c. *Eventual consistency*. Retrieved 10 February 2022 from https://en.wikipedia.org/wiki/Eventual_consistency
Wikipedia. 2022d. *CAP Theorem*. Retrieved 25 July 2022 from https://en.wikipedia.org/wiki/CAP_theorem