Event-Driven Microservice Architecture: Patterns for Enterprise Applications Supporting Business Agility

Pavel Hruby (hruby.pavel@gmail.com) and Christian Scheller (scheller2@msn.com)

Acknowledgment

The authors would like to thank our shepherd Ernst Oberortner, for his valuable time on weekly shepherding calls and for his indispensable input. 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 Ernst's feedback.

Summary[1]

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.
- Many developers work on a large codebase simultaneously. Despite careful coordination, teams must wait too often for each other, which slows down progress. With the growing number of developers working on the system, most organizations experience the law of diminishing returns.

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. This paper aims to provide a reader with a single direct path to microservice architecture that has been proven successful.

Introduction

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

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 are often tightly coupled, where every component is dependent 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

[1] The authors would appreciate feedback from writer's workshop participants on the following issues:
- Several patterns such as MICROSERVICES, EVENT SOURCING, and MICRO FRONTEND have more specific meanings in this paper than in the existing literature. For example, if a component does not communicate with others **only** by events, it is not a MICROSERVICE in this paper. Is it a problem? If yes, what to do?
- The main pattern, MICROSERVICES, starts on page 9, which is very late in the paper. Is it a problem?

effect of changes refers to a situation where the impact of a change depends on the size of the change multiplied by the size of the system (Manaert et al. 2016). If the system is large enough, measured by the number of “components” and their relationships, implementing even simple changes could be extremely 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 everything 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” Melvin Conway [ref 3].

In such siloed organizations, each developer is working on multiple projects, and the wait time to get tasks executed 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 [ref 4].

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 applications are often called microservices.

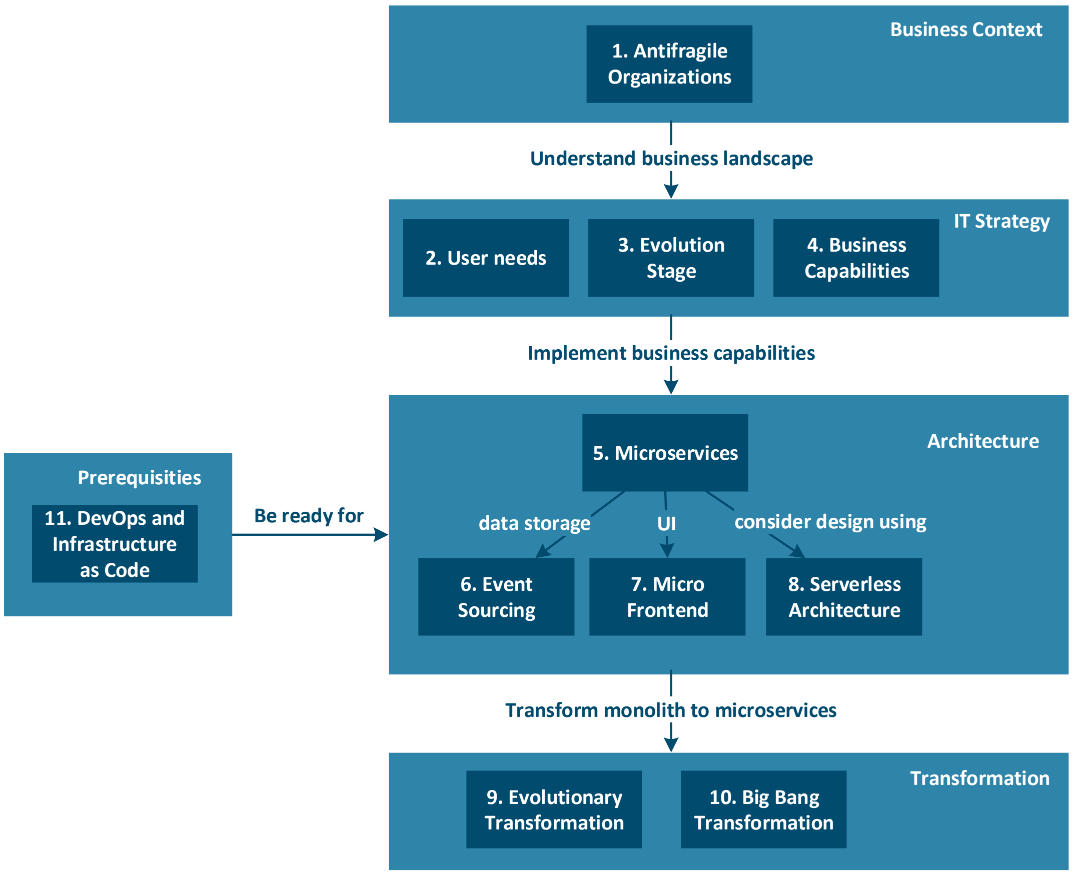
The application modernization is described in eleven patterns, see Figure 1 below.

The first pattern ANTIFRAGILE ORGANIZATIONS summarizes the business context of an organization seeking business agility. The following three patterns, USER NEEDS, 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. 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 DEVOPS and INFRASTRUCTURE AS CODE.


Business Context
1. Antifragile Organizations
Understand business landscape
IT Strategy
2. User needs
3. Evolution Stage
4. Business Capabilities
Implement business capabilities
Prerequisites
11. DevOps and Infrastructure as Code
Be ready for
Architecture
5. Microservices
data storage
UI
consider design using
6. Event Sourcing
7. Micro Frontend
8. Serverless Architecture
Transform monolith to microservices
Transformation
9. Evolutionary Transformation
10. Big Bang Transformation

.

Figure 1: Pattern Map

Pattern 1: Antifragile Organizations

Certain organizations grow after external shocks.

During the COVID-19 pandemic in 2000 – 2002, many organizations were under pressure and their performance decreased. However, after adapting to the initial shock, a portion of companies returned to near-normal performance, but there were some companies, which performance significantly increased. Several research and consulting firms started to analyze what makes these companies different and called them “antifragile organizations”. Antifragility[2] is a capability of an organization to emerge stronger from external shocks [ref 31]. What characterizes antifragile organizations?

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 [ref 32]:

- 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

[2] The term antifragility was first used by the American economist Nassim Nicholas Taleb in his book Antifragile: Things That Gain from Disorder [ref *30*].

- Antifragile – organizations that grow after a shock. They do not only respond to shocks but seek and embrace them.

Analysts of Leading Edge Forum [ref 32 and 34] 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” [ref 32]. “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” [ref 34].

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 a 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 USER NEEDS and the EVOLUTION STAGE of components in the organization’s application portfolio, see patterns 2 and 3. It has been observed that at the beginning of the COVID-19 pandemic in March 2020, organizations with a Wardley map, see the pattern EVOLUTION STAGE, adapted better than organizations without it [ref 35].

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 [ref 33].

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.” [ref 36].

Pattern 2: User Needs

Application modernization projects have little chance of success if the needs of all of their users 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. In other words, it is imperative to fully understand the needs of all stakeholders.

User needs 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 that would also identify business capabilities satisfying these needs.

A possible solution is a method called the user journey. User journeys describe the steps for completing a specific task within a system. Figure 2 illustrates a buyer’s user journey in an e-shop: search for a product, add the selected product to a basket, specify delivery details, pay, and receive confirmation. These steps can be matched by the BUSINESS CAPABILITIES of an existing or future software system.

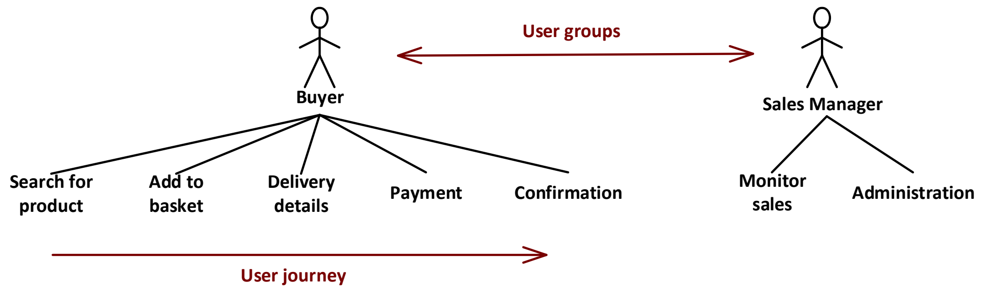


Figure 2: User Journey

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 provides vertical partitioning of user needs, and focuses attention on things that are important for the stakeholders [ref 10]. It helps to determine the scope of MICROSERVICES, see 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.

Pattern 3: Evolution Stage

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

Which systems to focus on in software modernization?

The evolution stage of software components together with user needs can be visualized using a technique called Wardley mapping [ref 7]. The Wardley mapping is an IT strategy method developed by Simon Wardley [ref 8], [ref 9]. 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 depending on the connected components lower on the map.

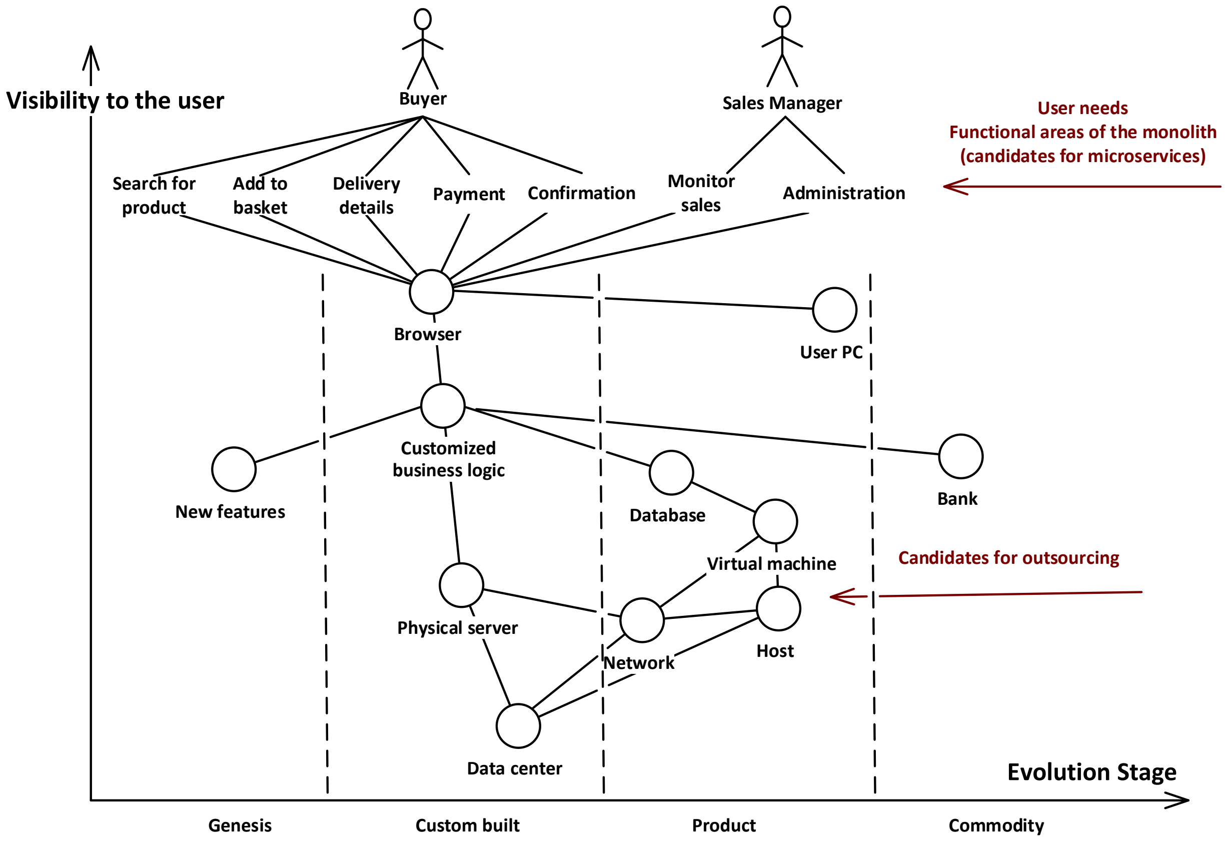


Figure 3: A Wardley map of a client-server application

Wardley map can be created practically for any software system and Figure 3 is an example of a Wardley map illustrating a web-based legacy application. The UI 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 Salesman and Sales Manager, such as Search for a product, Add to basket, Delivery details, Payment and Confirmation of sale, and Monitoring of sales. 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 Figure 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.

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

- The components in the genesis phase can be developed as microservices directly.
- The custom build components, such as client-server monoliths, have to be transformed into microservices. It is the focus of this paper.
- 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 evolution stage of each component determines many aspects of IT development: the required skills and mindset of the team (people that like experimenting, usually best fit the tasks on the components in the genesis stage, while people that understand the economies of scale better fit to the tasks related to commodities), the 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 [ref 9].

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.

Pattern 4: Business Capabilities

Microservices represent business capabilities. Thus, business capabilities determine the structure of the modernized application.

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 [ref 4].

To mitigate these challenges, a new way of structuring the components in the modernized architecture, so that components – microservices - represent business capabilities. Each microservice has a full-stack team responsible for the UI, business logic, persistence, deployment, etc.

Understanding USER NEEDS, and using user journeys are good initial candidates for 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.

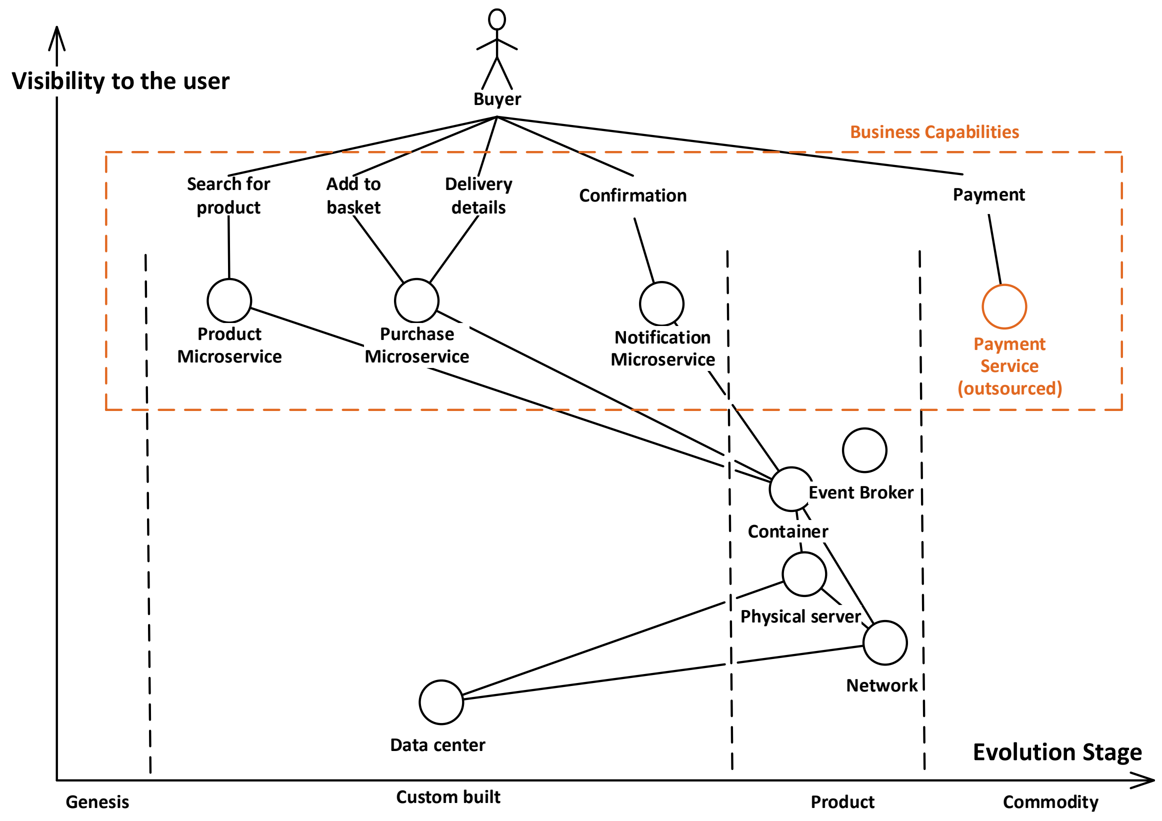


Figure 4: Business Capabilities

For example, in the user journey described in Figure 4, we can decide to implement the Product, Basket, and Notification microservices and outsource the Payment. The needs of the Sales manager are omitted for simplicity.

The task of determining the size and scope of 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.

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

Challenges: Strangling the monolith, i.e. 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.

Pattern 5: Microservices

Microservices[3] are loosely coupled applications, with their user interface[4], business logic, and storage, communicating with other microservices using events. The event broker[5] transmits events between microservices.

In the monolithic architecture, 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.

The solution is to rewrite the custom-built monolithic components in the application portfolio (see EVOLUTION STAGE) into smaller and simpler applications, each with its own user interface, business logic, and storage. They 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 [ref 5]. 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 by AWS [ref 6], 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 Figure 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. 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 EVOLUTIONARY TRANSFORMATION.

[3] There are many definitions and interpretations of microservices. The reasons why we define microservices in such, a rather narrow meaning, are explained later in this pattern.
[4] With the exception of autonomous microservices, see later in this pattern.
[5] The event broker is an essential element of this paper’s microservice architecture.

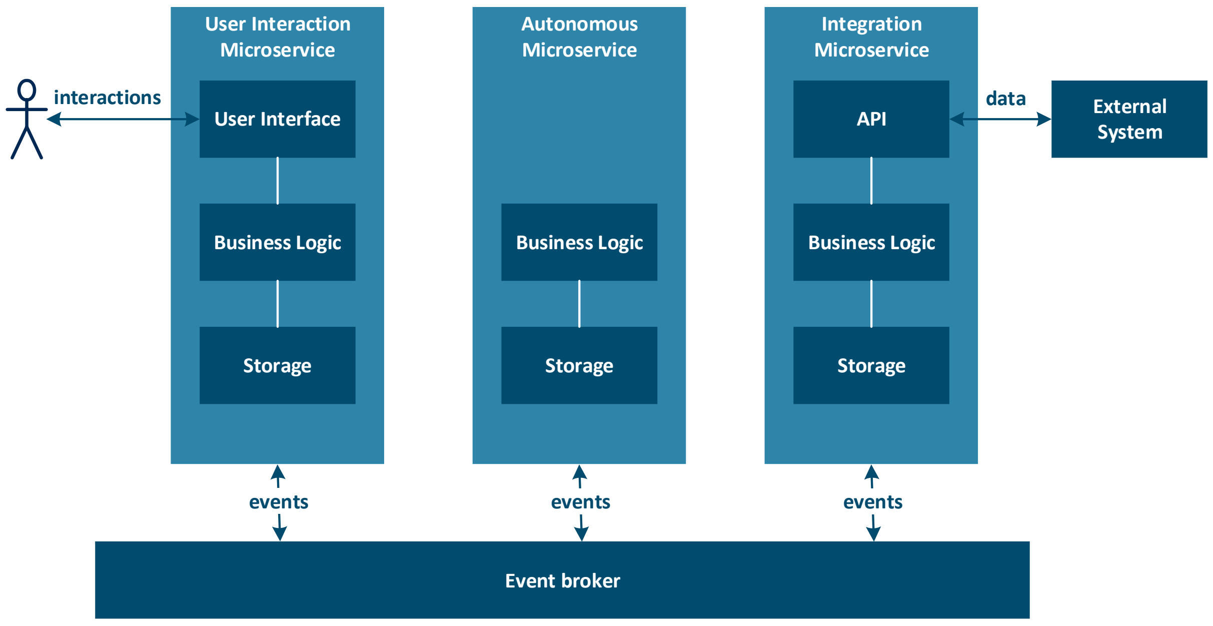


Figure 5 Microservices are small applications

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 combining 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 be designed to accept that they might receive the events in the incorrect order. “Changing the order of the events does not change the result” [ref 17]. 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 be designed to accept that they may receive a single event more than once. “Receiving an event multiple times does not change the result beyond the initial application” [ref 18]. 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 denote a situation when the system's behavior is dependent on the sequence of uncontrollable 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.

- 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 UI. Although the customer journey in the USER NEEDS and EVOLUTION STAGE patterns provide initial candidates, there is not a simple solution for partitioning microservices; it is a task for a software architect.

- 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” [ref 15].

- Conflicts: different microservices can update the duplicated data while in an inconsistent state. This problem cannot be eliminated completely due to the CAP theorem [ref 13], 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. Large 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.

- 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 independent development, deployment, and scaling.

Challenges: The distributed computing model: 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.

Pattern 6: Event Sourcing

The sequence of changes is the source of truth, instead of the current state.

The microservice architecture is a distributed computing model, which imposes certain challenges to microservice design. They are idempotency and commutativity, see the MICROSERVICES pattern.

There is a solution to these challenges, called event sourcing. 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). When a microservice detects an event received in an incorrect order, it can reconstruct its application state by replaying the events in the correct chronological order [ref 20].

- Idempotency – event sourcing allows the microservice to detect duplicated events in the event store, so that the microservice can ignore them.
- Commutativity – event sourcing allows the microservice to sort the events in the event store, so the microservice can process them in the correct order. The order can be chronological or based on a version of the updated resource.

The list of events 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.

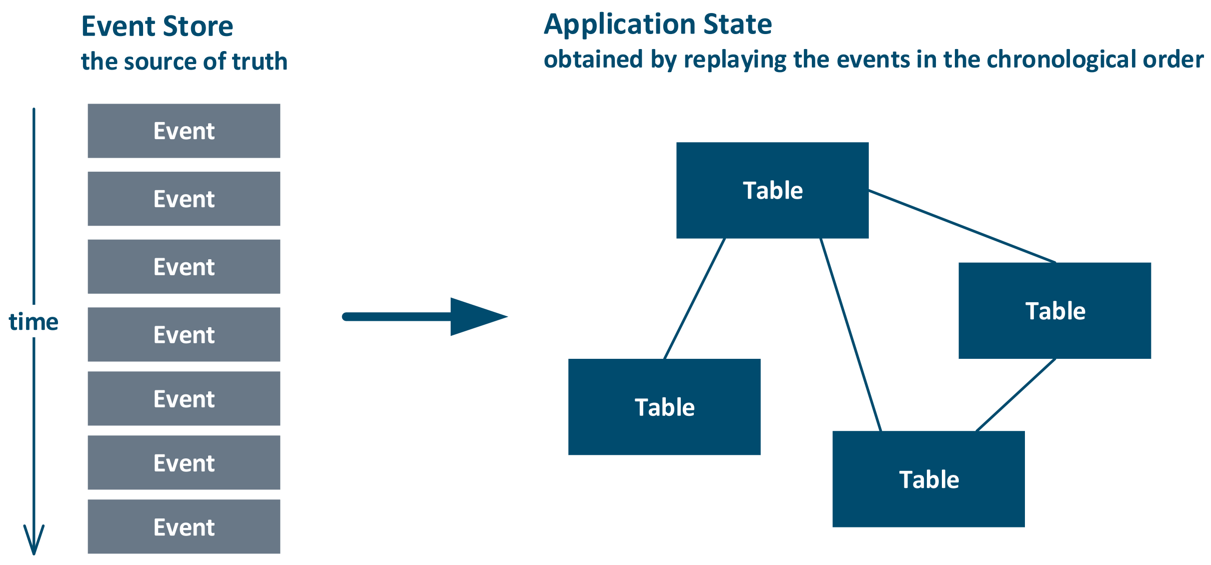


Figure 6 Event sourcing

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 [ref 19]: “CRUD, only when you can afford it”. 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. The unfamiliar programming model for mainstream developers, because traditional practices such as normalized relational databases no longer apply, and data duplication is no longer a problem that needs to be fixed. On the contrary, data duplication is a natural characteristic of microservice architecture.

Pattern 7: Micro Frontend

Microservice’s user interface occasionally needs to invoke a UI 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. We would like to 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 Figure 7.

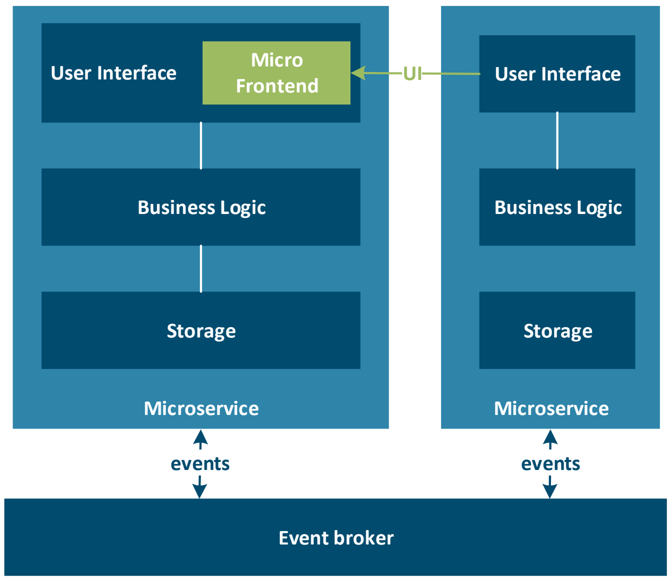


Figure 7 Micro frontend

For example, a Sales Order microservice encapsulates the business capabilities of a business transaction. While working on an order, users would also like to see details of a product, which is handled by another microservice, without switching to the product UI. A solution is to create a Product Micro Frontend, embedded in the user interface of the Order microservice.

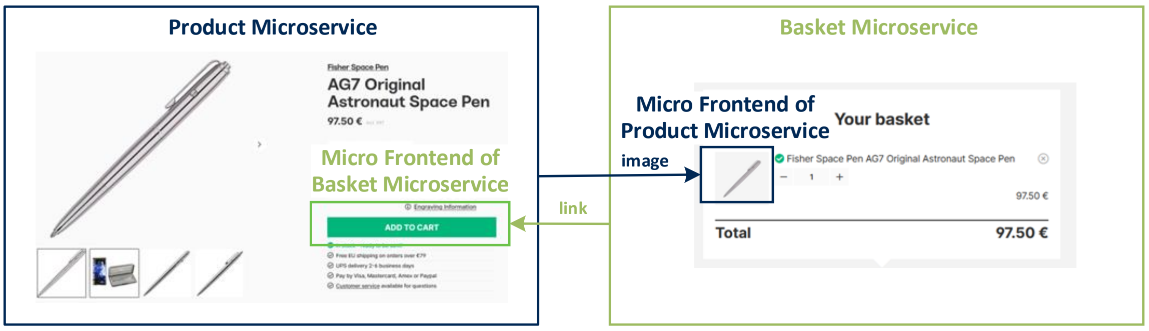

Product Microservice
Fisher Space Pen
AG7 Original Astronaut Space Pen
97.50 €
Micro Frontend of Basket Microservice
ADD TO CART
link
Basket Microservice
Micro Frontend of Product Microservice
image
Your basket
Fisher Space Pen AG7 Original Astronaut Space Pen
97.50 €
Total
97.50 €

Figure 8 Examples of micro frontends

Figure 8 illustrates an example of two microservices, Product and Basket. The UI of the Product microservice contains a UI element “Add to basket”, which is a micro frontend of the Basket microservice. Clicking it opens the main UI of the Basket microservice. The UI 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 UI of the Product microservice.

Benefits: 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 [24].

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

Pattern 8: Serverless Architecture

Use native managed services of a cloud provider, instead of deploying applications in virtual machines.

Traditionally, microservices are deployed in containers, can run everywhere, and be 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 are not used.

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 [ref 28]. 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” [ref *25*]. This architecture also appeals to the developers [ref 26] and [ref 27], because they can often implement a solution only by configurations, which makes them more productive.

There are drawbacks, real or perceived, of the serverless architecture. It 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.

Patter 9: Evolutionary Transformation

Monolith is integrated into 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, or it is necessary to wait until the whole project is finished?

In the evolutionary transformation [ref 16], 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 UI 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 Figure 9.

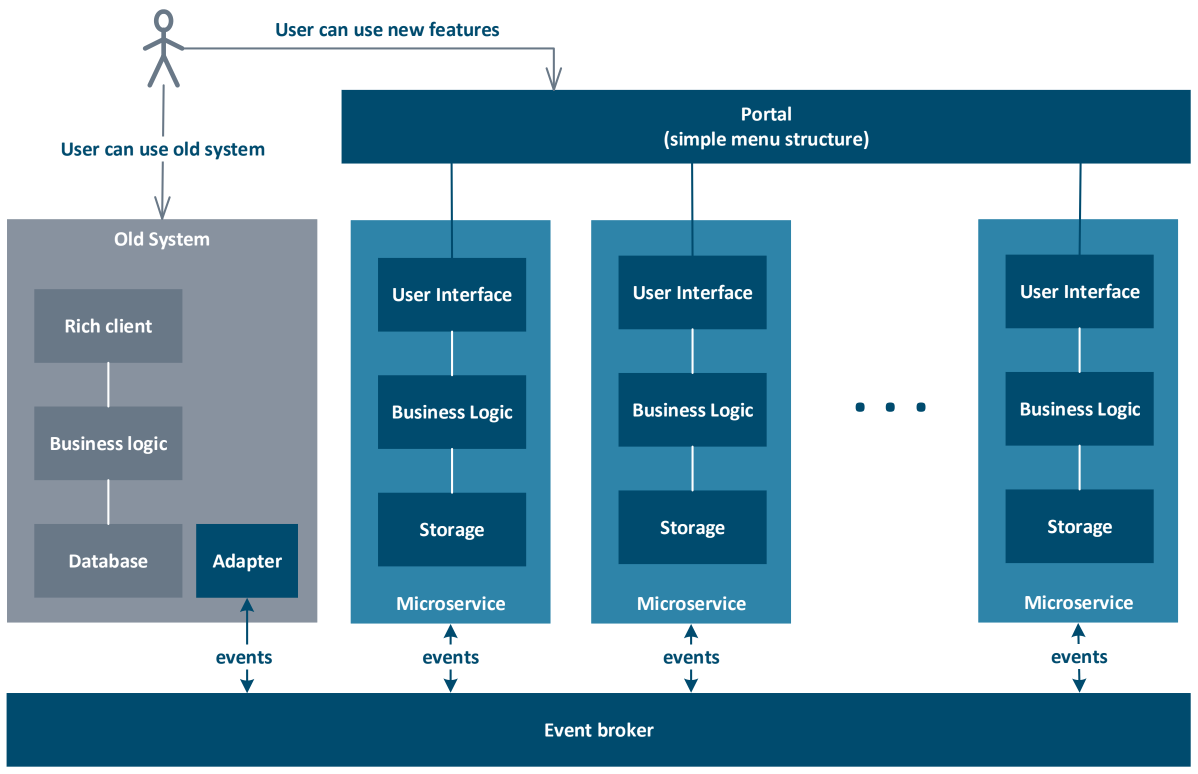


Figure 9: Architecture supporting evolutionary transformation

The benefit of using microservices early comes at a cost: the old system must be modified to consume and raise events; see the Adapter component in Figure 9.

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, we can use them in 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.

Pattern 10: Big Bang Transformation

The new system will be used after 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 finished. 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.

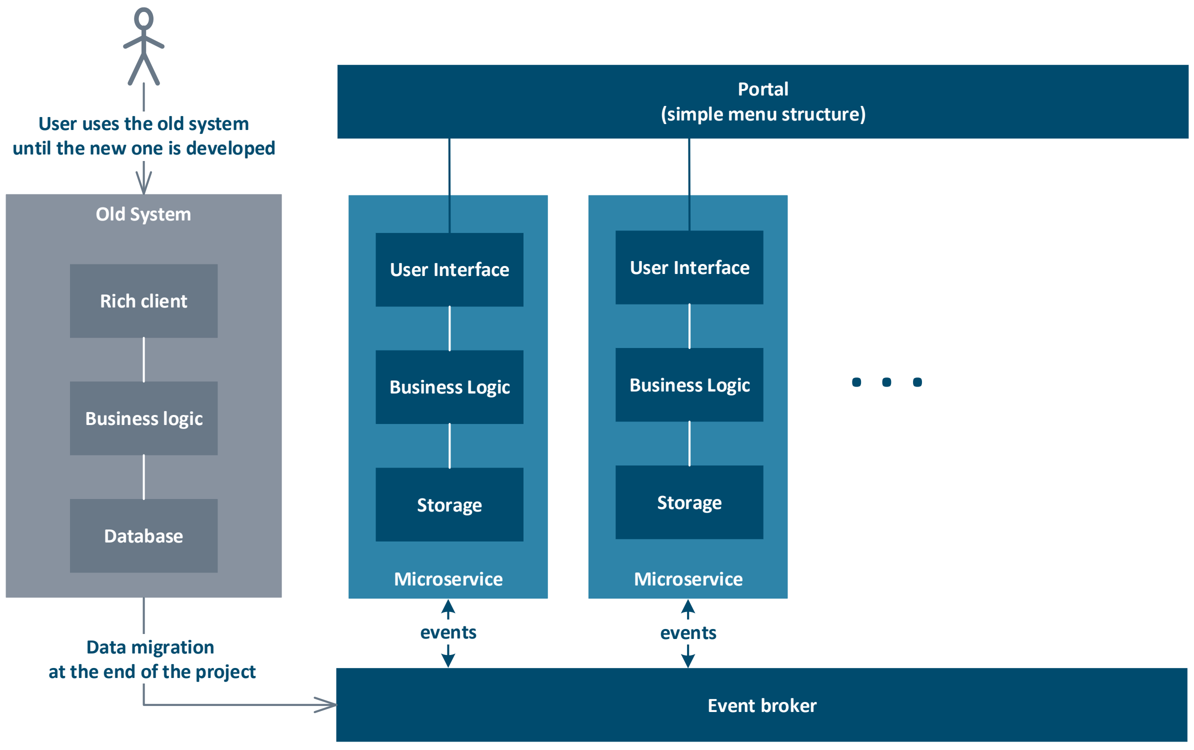


Figure 10: Architecture supporting big bang transformation

The big bang approach 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.

Benefits: Often cheaper than the evolutionary approach

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

Pattern 11: DevOps and Infrastructure as Code

Infrastructure as code and DevOps practices are two essential prerequisites for the successful implementation of the microservice architecture.

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 [ref 11]. However, changes represent risks in service delivery, and minimizing these risks is the fundamental goal of operation management. If the developers following agile development must constantly fight the resistance against changes, natural for the operations, organizations are unable to respond quickly to market changes and opportunities.

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 like the decision 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 them 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 [ref 14].

Benefits: Increased deployment frequency decreases the lead time for changes and the time to restore service, and significantly decreases the number of failures [ref 12].

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.

Conclusion

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

The same set of patterns is also applicable to new organizations that design their IT systems from scratch to support business agility.

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 them, and what IT modernization steps an organization must take to meet those needs and grow after a shock.

References

1. Joseph Franz, Solution Architect at DXC, *personal communication* (February 2022)
2. 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
3. Melvin Conway, 1968, *How do committees invent*? (Datamation: 28-31), http://www.melconway.com/Home/pdf/committees.pdf
4. 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,
5. Microsoft, *What is Cloud Native*? (retrieved 8 February 2022 https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition

6 *Microservices architecture on AWS* (retrieved 8 February 2022), https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/simple-microservices-architecture-on-aws.html.

7 Ben Mosier, *Wardley Mapping: Strategy for the Self-Taught* (retrieved 13 February 2022) https://learnwardleymapping.com/

8. Simon Wardley, *Wardley maps, Topographical intelligence in business* (retrieved 8 February 2022) https://medium.com/wardleymaps
9. Leading Edge Forum, *Wardley Mapping, Learn How to Stimulate Future Ideas and Strategies*, https://learn.leadingedgeforum.com/p/wardley-mapping/?product_id=1606147
10. *Wardley Mapping during a pandemic* (TPXimpact: 3 December 2020) https://difrent.co.uk/blog/wardley-mapping-during-a-pandemic/
11. Kent Beck, Cynthia Andres: *Extreme Programming Explained: Embrace Change* (Addison-Wesley 1999).
12. Google, *State of DevOps 2021* https://cloud.google.com/devops/state-of-devops
13. *CAP Theorem*, https://en.wikipedia.org/wiki/CAP_theorem, retrieved 25 July 2022
14. Anders Holte Nielsen, *Cyberattack at Demant, Contingency that works in practice* (morning briefing at DevoTeam, Denmark, 14. October 2021)
15. *Eventual consistency* (Wikipedia, retrieved 10 February 2022), https://en.wikipedia.org/wiki/Eventual_consistency
16. Eran Stiller, *6 Lessons I Learned on My Journey from Monolith to Microservices*. (NET Fest 2019) https://www.youtube.com/watch?v=Isz16TZtWRM
17. Wikipedia, *Commutative property* (retrieved 8 February 2022) https://en.wikipedia.org/wiki/Commutative_property
18. Wikipedia, *Idempotence* (retrieved 8 February 2022) https://en.wikipedia.org/wiki/Idempotence
19. Microsoft blog ”*CRUD, only when you can afford it* (retrieved 8 February 2022) https://docs.microsoft.com/en-us/archive/blogs/maarten_mullender/crud-only-when-you-can-afford-it-revisited.
20. Martin Fowler, *Event Sourcing*, (WOW! Nights March 2016), https://www.youtube.com/watch?v=aweV9FLTZkU&t=145s

21. Erich Gama et al: Design Patterns, *Elements of Reusable Object-Oriented Software*, (Prentice Hall 1997) https://www.amazon.de/-/en/Erich-Gamma/dp/0201633612/
22. Dr. Thomas Porter, DXC Distinguished architect, *personal communication*, (April 2020).
23. Jordi Beentjes of Betty Blocks, *Presentation for Low-Code & No-Code guild of DXC*, (September 2020)
24. Cam Jackson: Micro Frontends, https://martinfowler.com/articles/micro-frontends.html
25. Tom Grey: *5 principles for cloud-native architecture—what it is and how to master it* (retrieved 13 February 2022), https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it
26. Anirban Roy, solution architect at DXC, *personal communication* (9 February 2022)

27- Czeslaw Kazimierczak. Ph.D., system architect at DXC, *personal communication*, (9 February 2022)

28. *Amazon Serverless microservices*, (retrieved 8 February 2022) https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html
29. Susanne Kaiser, *Preparing For a Future Microservices Journey Using Wardley Maps* (DDD Europe 2020) https://www.youtube.com/watch?v=csjaxNGF5LQ
30. Nassim Nicholas Taleb, *Antifragile: Things That Gain from Disorder* (Random House 2012)
31. Bill Murray, *Building an Antifragile Organisation* (CEO Today, October 21, 2020) https://www.ceotodaymagazine.com/2020/10/building-an-antifragile-organisation/.
32. Murray, B. Aron D, *Rethink Risk Through The Lens of Antifragility* (Leading Edge Forum, April 2017) https://leadingedgeforum.com/media/1983/rethink-risk-through-the-lens-of-antifragility.pdf
33. Bill Murray, Senior Researcher and Advisor at Leading Edge Forum, *personal communication* (April 2021).
34. Krzysztof Daniel, Carl Kinson C, Caitlin McDonald C, Bill Murray, *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
35. Krzysztof Daniel of Leading Edge Forum, *Presentation for the DXC Architecture Guild* (2020).
36. Jez Humble, *On Antifragility in Systems and Organizational Architecture*, (Continuous Delivery) https://continuousdelivery.com/2013/01/on-antifragility-in-systems-and-organizational-architecture/