DZone

# Microservice vs. Monolith: Which One to Choose?

**by Shamik Mitra MVB · Aug. 13, 18 · Microservices Zone · Opinion**

"Microservice" is a buzzword in the industry, right? Apart from microservice design, other designs are marked as "Monoliths." But as an architect, when trying to create new software based on a specific domain, what will you do? Adopt microservices without judging as this is the one everyone is getting behind, or stop for a second and think about your company infrastructure and employee expertise, and based on that, choose microservices.

As architects, it is our duty to choose microservices when it is required, not going with the flow and using microservices. In this tutorial, I will talk about a few aspects I think are most important before you adopt microservices. I am open to hearing any other proposals or aspects to cover which I missed, even counter-logic — these are my realizations while working on microservice, and so-called monolithic, architectures.

Let's begin with the aspects to consider before choosing microservices as a default architecture.

## 1. Strength of Project Associates

When you are starting a project, the first parameter to consider is, how many associates are going to work on this project? What are their experience levels? If your strength is low, like 10-12, don't go for microservices, as microservices mean independent services, and the motto is "You build it, you run it." There should be one team with complete ownership of a microservice. If your associate strength is small, statistically, one or two people have ownership of two or three microservices, and it creates a critical problem; knowledge is confined to them and they will become a pivot employee. Also, if one or two people take ownership of two or three microservices, the motto "Developers should only focus on a small portion but know everything about that service" is going to break. So think about that.

## 2. Microservice and It's Associated Knowledge

Microservices is a new architecture and there are various concepts related to microservices, like distributed architecture rules, high availability, resiliency, service discovery, the CAP theorem, domain-driven design, circuit breaker patterns, distributed cache mechanisms, and route tracing. DevOps culture is tightly coupled with microservices in order to unleash their full power, so you need to know CI/CD tools and their culture (automated deployment). The team should be efficient to drive all of these for microservices. If the microservices are deployed in the cloud or a container, the team should have an understanding of cloud architecture (PCF, Amazon, Bluemix, etc) or container architecture (Docker, Docker Swarm, Mesos, Kubernetes, etc). Think carefully about the team's knowledge — always

Docker Swarm, Messos, Kubernetes, etc). Think carefully about the team's knowledge — always remember that, when you are dealing with microservices, it means multiple services and multiple teams, so each team member should have a clear understanding of all the associated knowledge needed to run their microservices independently. If you don't have such leisure, make sure that for each team, one or two people know all the associated knowledge so they can educate their team.

## Organization Infrastructure

Another important dimension is the organizational infrastructure. Before adopting microservices, always check in what mode your organization works. As per Conway's law, whatever your organization's structure, that is reflected in your code implementation. Check if your organization has adopted Agile technology. Is the team built with cross-technology members like UI, middle layers, and a database? What are the deployment and QA test modes — does your organization follow manual deployment and manual testing, or have they adopted DevOps culture. Where are the artifacts going to deploy? Does the organization have servers where you install application servers and deploy your artifacts, or has the organization moves to the cloud? Based on these parameters, choose whether you are going to adopt microservices or not.

If the organization has its own servers where application servers are installed, all the different microservice artifacts are going to deploy to the same server space, which does not fulfill horizontal scaling, so microservices adoption will be in danger.

In the case of manual testing and manual deployment, adopting microservices is a nightmare for the Ops and QA teams. For developers, it is a nightmare to integrate the changes and test in SIT.

## Domain Criticality

Check the nature of the domain you are working on. Sit with a business analyst to understand how critical this is — is there a need to break the domain into subdomains so that related functions can be encapsulated under a context? Based on that, make a decision if the domain is very complex. If not, it is better to stick with a monolith, so there's no need to create a context map and encapsulate subdomains in a bounded context.

## The Budget of the Project

This is another important aspect to look at. Think about the budget of the project; is the nature of the project fixed bid or TNM? Microservice projects are more expensive than monoliths, as they need more servers or cloud infrastructure, an automated pipeline, count of resources, and the whole team should be a cross-skills team. In terms of infrastructure and resources, it is much more costly than a monolith, so think about how the budget aligns with your project.

Secretly, I am giving you a tip — please do not publish it to anyone: as a good, loyal architect, always think about revenue margin. A modular monolith is better than microservices in the case of a small budget as it serves the purpose of revenue and, if you want to move to microservices in the future, that is also possible.

## Conclusion

Microservices technology hit in such a way that novice or mid-level developers think a monolith means a BBOM (big ball of mud) with all modules tangled with each other, but if you maintain a modular approach, it is good, and in many cases, it is better than microservices when our resources are finite. Choose carefully and don't go with the flow.

I always recommend going with a monolith first, but make it modular. If you use Java 9, use modules; otherwise, you can adopt SOA, but start with a modular architecture and when there's an actual need, like the module boundaries are going to leak or you can't maintain Acyclic Graph (DAG) among modules, then think about breaking up the monolith on the basis of DDD and leaning towards microservices.

What I am trying to say is, do not use microservices just because others are adopting it. Adopt it when you need it. In many cases, I saw a project try to adopt microservices, but just because of lack of knowledge, they created a project that was neither a microservice nor a monolith. They created many services without proper encapsulation and each service was chained with others in such a way that one agile team can't make an independent decision on deploying one microservice. Multiple teams and multiple artifacts depended on each other. Teams were blaming each other as functionality spanned over many services — a horrible situation. I will discuss that in my future articles. For now, think twice before adopting microservices — as Uncle Ben said, "With great power comes great responsibility."

---

**Learn why microservices are breaking traditional APM tools that were built for monoliths.**

---

Topics: MICROSERVICES , MONOLITH , SOFTWARE ARCHITECTURE , DISTRIBUTED ARCHITECTURE , AUTOMATED DEPLOYMENT

Published at DZone with permission of Shamik Mitra , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.

ghtStep

# The Micro-Hexagon Architectural Pattern

**by Safouen Bezzine · Feb 13, 19 · Microservices Zone · Analysis**

**Deploy commerce faster and keep pace with the demands of your customers and executives. Read this blueprint to learn how to create your own microservices-based commerce foundation so you can quickly move onto building innovative and unique shopping experiences for your customers.**

Although microservices solve many technical problems, it is time perhaps to conclude, after a few years from the launch date of this concept, that this architecture needs to be further improved, to provide a way to have a clearer vision of the functional components. That's why many architects now use DDD decomposition.

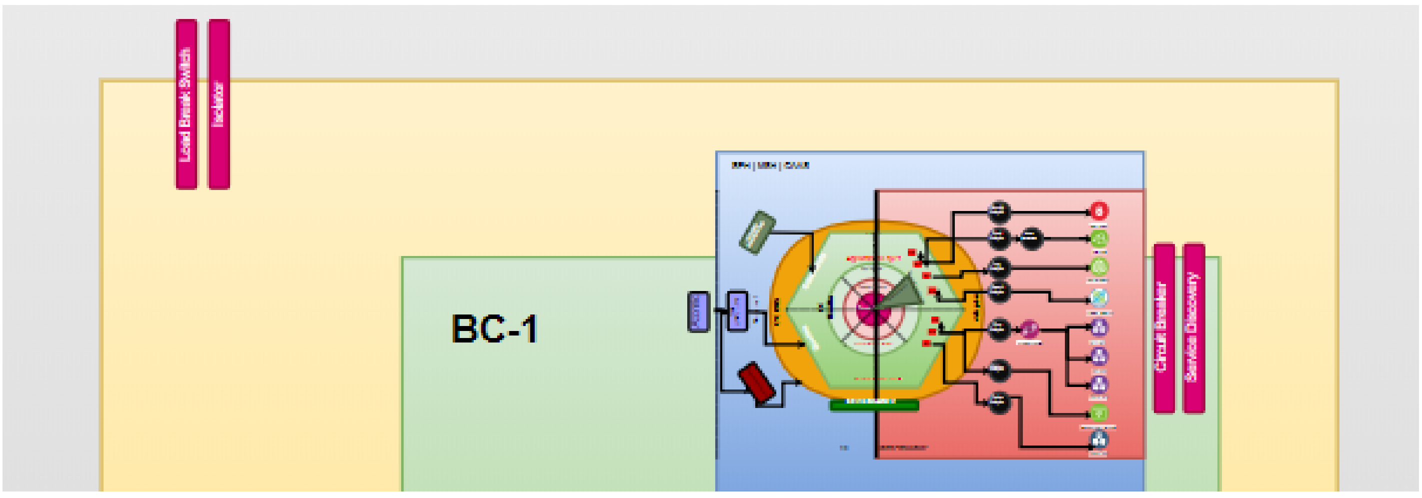
Another concept that has been put in place with DDD is event sourcing and CQRS. In fact, we do not deny the benefits of these implementations but it could also have other technical drawbacks which impact our architecture with more complex debts.

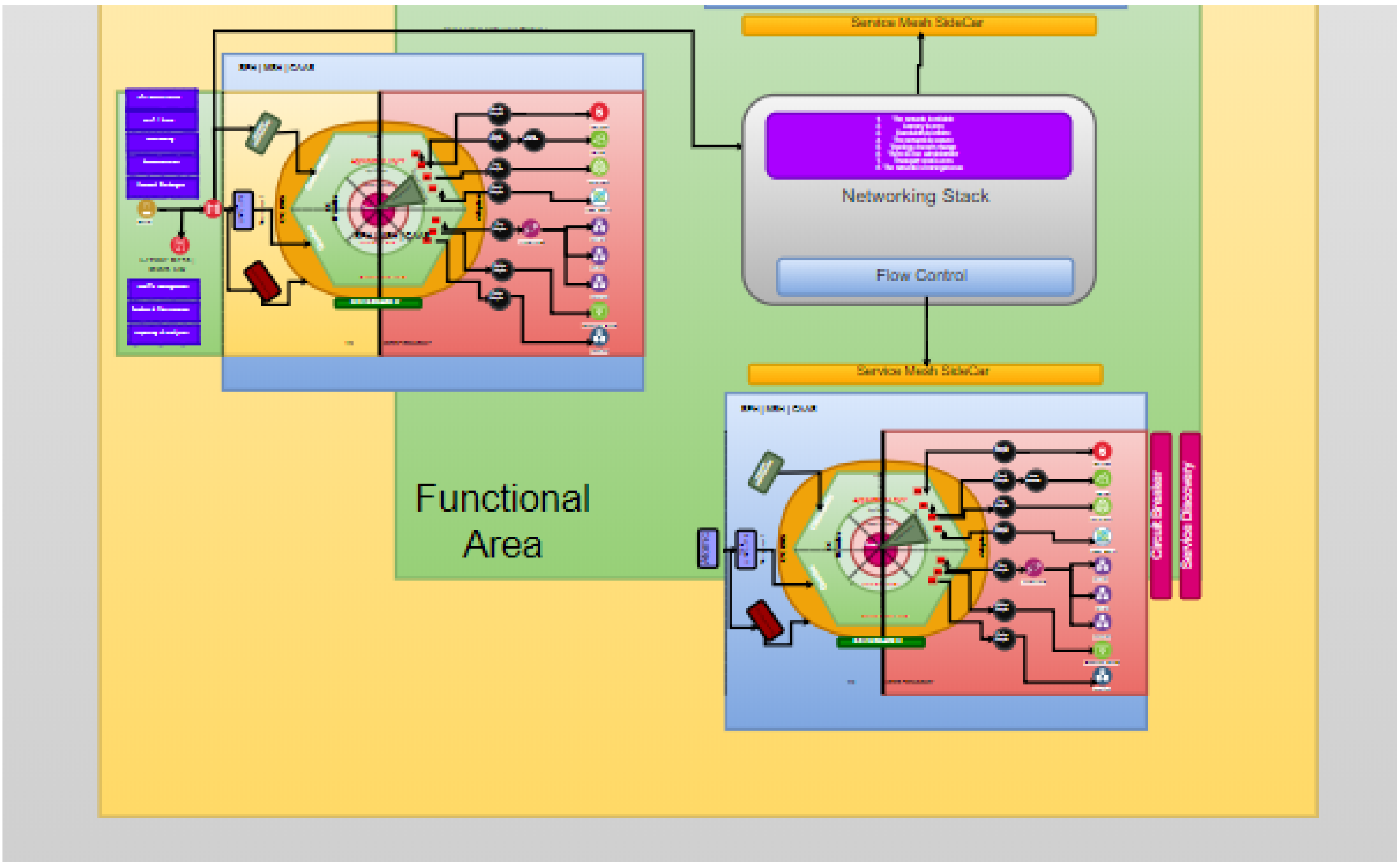
In this situation, I have restored the basic concepts of the hexagonal architecture which have been neglected since 2005 by many developers and I will show you how it will help us to solve the above issues.

In fact, Alistair Cockburn defined hexagonal architecture as the architecture of the ports adapters. The name can give you an idea about the concept. But this architecture also has disadvantages, such as inflated functions.

Every design has its own advantages, but what about combining MSA, CQRS, and HEXA? I have noticed that many developers try to implement CQRS and event-driven architectures in every software component, but they would have bad moments in the production environment. We will discuss this in further posts.

I have searched for a while for a compromise between complexity and power, and here I am suggesting this architectural design based on Alistair's Architecture.

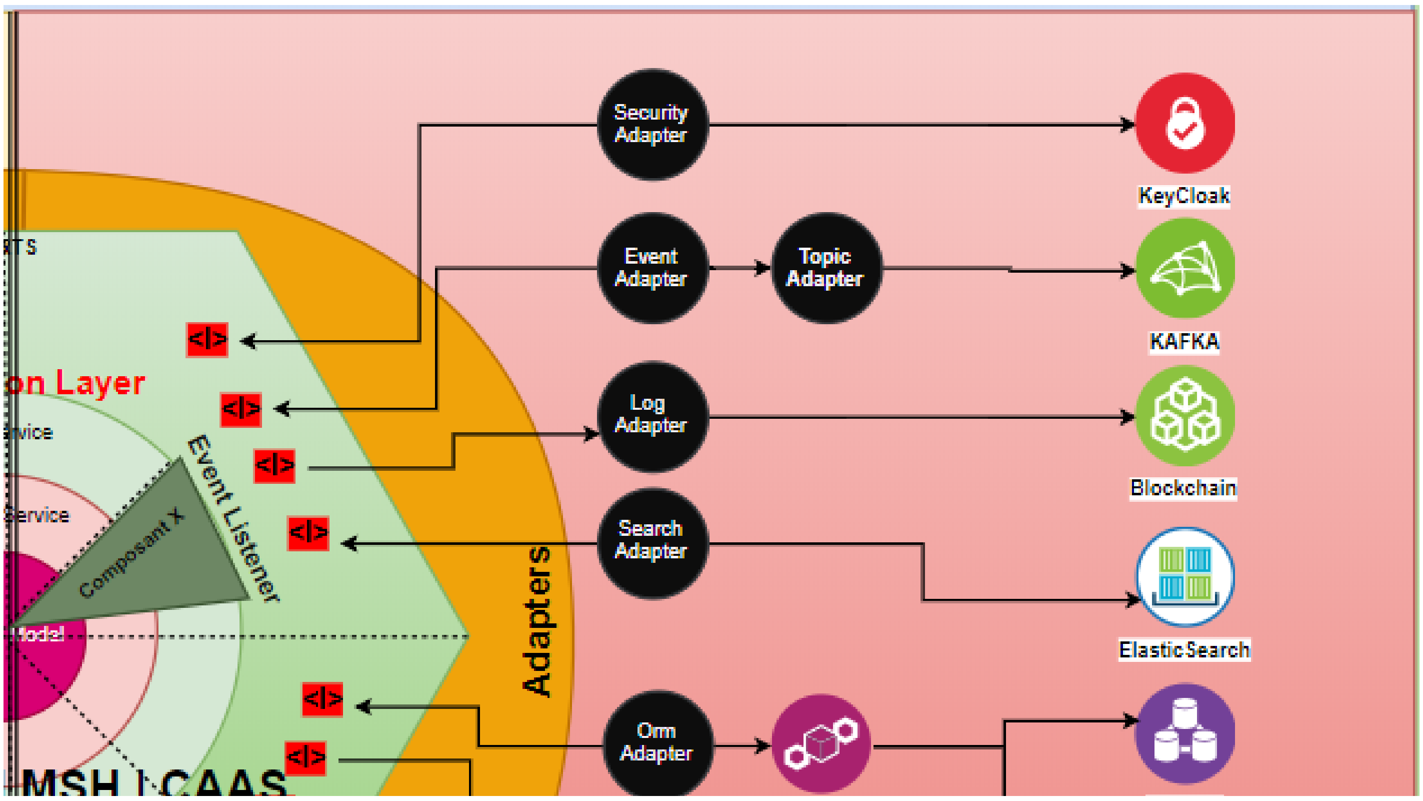


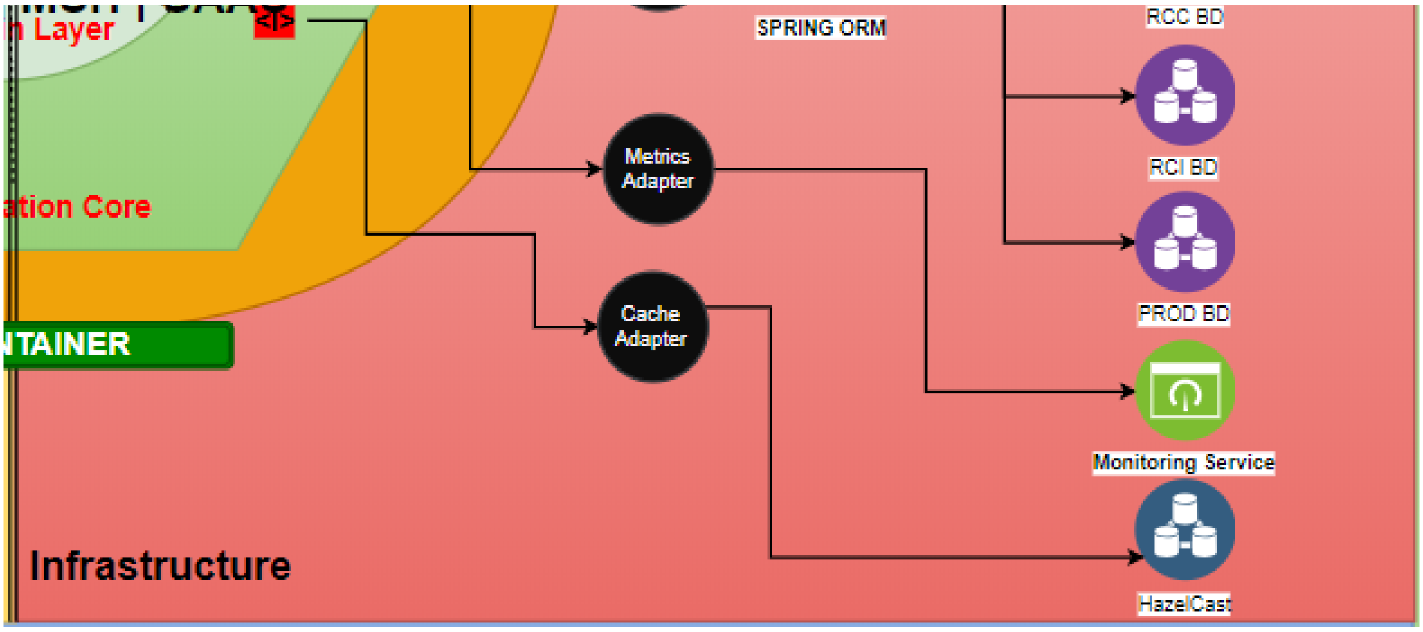


In this area, we will exchange only functional needs, and we will use business language (please do not be confused by DDD UL).

This will allow us to be more aligned with the principle of digitization and to transform our vision into a true digital strategy based on customer insights and the market. First of all, we should think 100 percent business, then we should adopt our functional components to the technology trends (be aware of over-engineering).

Let's focus more on our atomic component: the micro-hexagon.





In fact, according to hexagonal architecture, we will divide our microservice into three levels:

1. Application Side
2. Domain Side
3. Infrastructure Side

The top layer of this architecture will be a microservice layer. We will isolate boundaries using ports and adapters.

Dependencies range from the application to the domain and from the infrastructure to the domain. It is a good way to manage integration projects and tests. The package will be organized by business cases. Think 100 percent business. If we also talk digital, we need to go after technological trends and sometimes, we will exert a lot of effort and cost to be aligned with the new technologies. This micro-hexagon is a flexible way to manage adapters, to mitigate the risk of evolution, and to have a safe business domain.

Sometimes business models are more important than technology projection, but by using this architecture style it's easy to find and identify the business rules. It would also be a great advantage to the new development team members who start in an early stage of the development phase to understand the business cases treated from one part and to master the technological part of the infrastructure.

Development frameworks and purely technical services are being used more and more and they have reached a good level of maturity. We use several types of IOC containers every day and we carry out third-party dependencies which facilitate implementation.

But suppose if your Postgres database is changed by Hadoop or Mongo or the inverse (just an example) according to the strategic view of the enterprise. Now imagine the technical debt you will have.

How will this sudden change impact your development and your solutions that are already in production?

By using a micro-hexagon architecture, we could be aligned with the agnostic technology concept present in microservices, and we can give flexibility to our code base as well. You should also consider also changes applied to the outer layers, intra-service interaction, and inter-service interactions that enable you to refactor the outer layer without impacting the inner layer.

QA managers will also be happy to handle test cases driven by hexagonal-based architectures. With this in place, it becomes easier to test our most valuable assets without mocks. That means writing faster and more concise cases for a specific area.

So to end this preview about this architectural landscape, I should say that if you are looking for an architecture that's flexible, maintainable, easy to test and implement, and allows you to focus more on business logic, go straight to micro-hexagon. We will see an important concept, Uncle Bob's style, in the next post. So stay tuned!

Topics: ARCHITECTURE AND DESIGN, HEXAGONAL ARCHITECTURE, MICROSERVICES, MICROSERVICE ARCHITECTURE, ARCHITECTURAL PATTERNS