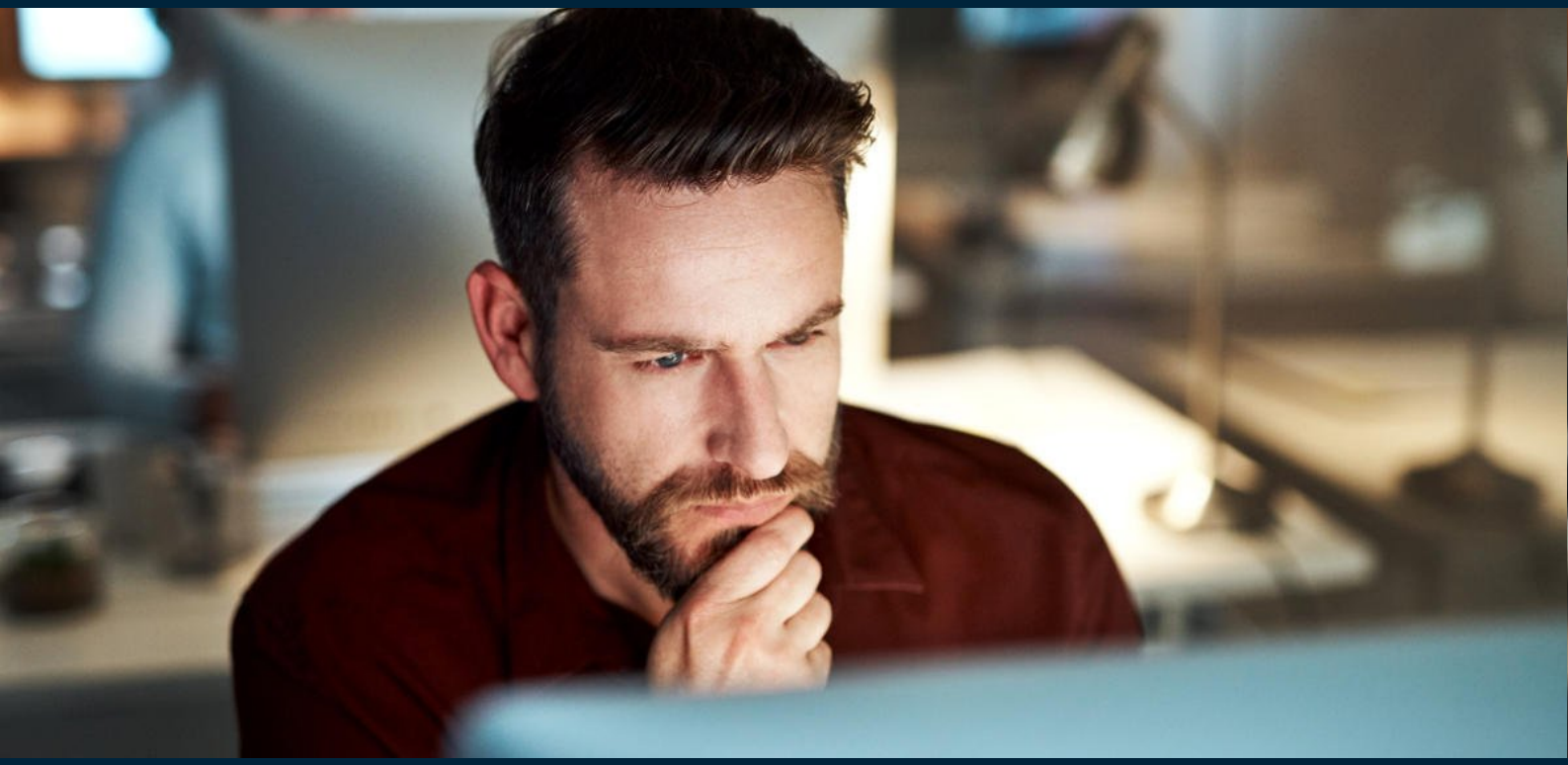
E-NGINEERS



FEAR OF MISSING OUT AND MICROSERVICES: WHY AND WHEN TO SWITCH

Microservice architecture is an immense popular topic nowadays. It offers a powerful approach with inherent complexity. Many IT experts fear of missing out (FOMO) and would rather start today than tomorrow. But don’t underestimate the complexity; investigate it first in the context of your project and processes. This white paper provides practical advice of the right time to switch to microservices and what needs to be in place before that happens.

WHEN AND HOW MOVE TO MICROSERVICES

MICROSERVICES INCLUDE EFFICIENCY, TEAM INDEPENDENCE, BETTER CONTROL OVER YOUR CODE BASE AND EASY TESTING.

But there are drawbacks, too. You'll need to consider system resilience in the event of service failure. And you need logging off coordinating features for release and rolling back affected services during release.

Microservices is not what you need at the very beginning of your project. At this stage, development processes are not well organized and requirements change frequently. Should Team A deploy a slightly modified code, the microservice from Team B could malfunction during production because they were not properly informed.

With monolithic architecture, the code changes might be caught at the compilation stage. With microservices, end-to-end testing would be needed - and you would need to maintain this testing, causing additional costs due to changing requirements.

Using microservices at an early stage might be right within isolated parts of a system, such as authorization or creating an entity in a system. It should not be implemented separately in each application of a large system. Instead, you have in place a central module with necessary and sufficient validation and an intelligible set of returned results, with errors and reports.

COMMUNICATION AND COMPREHENSIVE TESTING

WHEN SWITCHING TO MICROSERVICES, IT'S IMPORTANT THAT YOUR TEAM OR COMPANY HAS A TROUBLE-FREE DEVELOPMENT PROCESS, FROM DEVELOPMENT AND TESTING TO FULL RELEASE CONTROL.

The quality of communication within the team should be very high, and testing should be comprehensive.

I agree with Martin Fowler's recommendation of a monolith-first strategy,
'where you should build a new application as a monolith initially, even if you think it's likely that it will benefit from microservice architecture later on'.

With a small-scale project, it's always logical to use a monolith for design – or several monolith services. In the future, you can evaluate which modules are critical for the application and which require processing, and gradually allocate them to microservices as necessary.

Developing microservices from the very beginning can be considered for large-scale projects and very carefully in some specific cases.
Especially when it's clear that the project is going to be developed by different teams, or that using various programming languages is really necessary.

MICROSERVICE MYTHS

THERE ARE OFTEN MYTHS THAT,
WITH MICROSERVICES, EXISTING PROBLEMS
WILL MELT AWAY LIKE SNOW IN THE SUN

In No Silver Bullet, Frederick Brooks points to two kinds of complexity: **accidental** and **essential**. My view is that developers should aim to spend most of our time solving essential tasks.

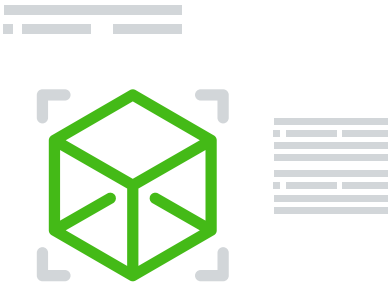
- If your code becomes difficult to maintain, that doesn't necessarily mean that it's time to get into microservices. Most likely, it means it's time to divide it into modules. These modules will be easy to delegate to other developers, and isolating the build and quality control of these modules is simple. And unlike microservices, build errors will be found out immediately, rather than in production.

- The argument that different technologies can be used in microservices is attractive, but in practice no one strives to have a variety of technologies. If you need to do integration with third parties, it's better to implement the communication layer in json or xml format with the necessary encryption and authorization. The communication layer can be validated via, for instance, XSD. In this way, we can avoid spending extra time on cross-team communication.

- Microservices' fault tolerance and scaling are also not reasons in themselves to switch. Consider instead a number of load-balancing, failover solutions for monolith architecture.

- Beware of false economies, such as grouping code by responsibility or reusing code. In reality, this approach leads to lots of code duplication. For example, checks for zero appear in thousands of places, which necessitates further, more complex checks. Later on, corrected in one place, errors and duplication can persist, unnoticed, in others. This is sure to lead to a substantial and uncontrolled increase in the code base.

WHEN IT'S WORTH TAKING THE MICROSERVICES ROUTE

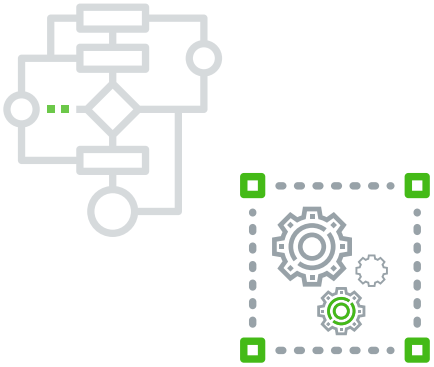


When you need to implement processing, calculations or other operations on a different technology or paradigm. For example, everything is implemented as an object model, and for a specific task (part of development) a functional paradigm is needed (a lot of mathematics). In this case, the use of various technologies is justified.

When some of the tasks require large hardware resources that will consume a lot of processing time. As such, they should be moved to a separate server, and microservices architecture comes into its own.

Where code access restrictions are required, such as when accessing sensitive information or secure algorithms.

If part of the executable logic consumes a lot of memory, such as a dictionary, it's worth allocating and moving it to a separate server. Or if it is necessary for one or another module to work more efficiently.



DIVIDE AND CONQUER

THERE ARE A NUMBER OF UNDISPUTED BENEFITS IN DIVIDING MONOLITHS INTO MICROSERVICES – PROVIDED IT'S DONE AT THE RIGHT TIME:

- The potential to make independent changes in different parts of a system enables you to deliver more functionality with less delay
- It's easier to introduce/add auto-tests
- Clear separation between business logic and data presentation
- Faster release cycles and much easier deployment

WHEN TO SWITCH FROM MONOLITH TO MICROSERVICES

IT'S LOGICAL TO TRANSFER RESPONSIBILITY TO DIFFERENT CODE AND SEPARATE THE RELEASES INTO INDEPENDENT ONES IN THE FOLLOWING CIRCUMSTANCES:

- When due to the growth of the project, or for security, business structure or other reasons, several teams are working without overlapping.
- When your own software becomes a functionality broker, especially when you provide functionality as bricks.
- When you need to release new functionality quite intensively. The degree of intensity here is quite subjective. It all depends on the severity of the release. If this is a release of small functionality, we're talking about a daily release. For heavy releases, even a weekly release can be painful. Everything is determined by the project and specific situation.

WHAT NEEDS ATTENTION WHEN DEVELOPING MICROSERVICES?

Breaking up the monolith. It's important to allocate self-sufficient modules and try to avoid duplication of code in new services. For example, you don't need to duplicate user authorization in each module. Separate authorization in a separate module. Each service that needs a check for user rights will communicate with the authorization service. Or, for example, each request will be processed by the authorization services and they will generate a token in which all user rights are registered. The choice of the best interaction of services, and their logical separation, will depend on the load.

Handling service errors. When designing microservices, don't neglect negative scenarios. What would happen if one of the microservices does not work? Consider load balancing, and filtering requests to service replicas in case of failure. Consider the resilience of the whole system so that it does not stop working following denial of a certain service.

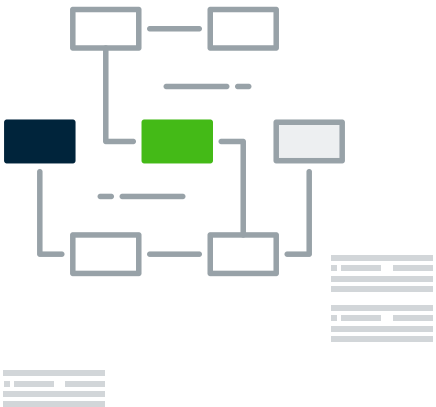
Updating services. It's necessary to provide testing of interfaces of interaction between microservices, methods for updating interfaces, and partial migration to new interfaces. As release times for microservices may vary, you need to provide ways to change and test new logic.

Tracking the status of microservices. Unlike a monolith, microservices can be located on many machines, written in different languages and with different methods and control capabilities, so controlling microservice architecture becomes a difficult task. Do not underestimate this task.

WHAT NEEDS ATTENTION WHEN DEVELOPING MICROSERVICES?

- Pay attention to governance and management, not just technical issues. It’s hard to be aware of all endpoints and flows, and time spent composing the knowledge base, learning and competences may contribute to drag.
- Take the time to provide a good test/development environment, where each and every element can simulate, easily revert or restore an earlier version, returning the system to a consistent state.
- Work out a strategy of identifying and rolling back the chain of dependent services in the event of a bad release.

MICROSERVICE ARCHITECTURE OFFERS POWER WITH COMPLEXITY – BUT POWER IS NOTHING WITHOUT CONTROL. MAKING SURE YOUR PREPARATION AND TIMING ARE ON POINT WILL HELP YOU MAKE A SUCCESS OF THE SWITCH.



THE AUTHOR

VENIAMIN MEDVEDEV

Has a broad experience in IT at the level of architecture design and implementation. He always stands for effective communications in the teams. In this white pager, Veniamin offers his vision on the topic. As a person open to dialogue, he is always glad to share experience and discuss specific details with you.

E-NGINEERS

DO YOU HAVE ANY QUESTIONS ON THE WHITE PAPER OR
JUST WANT TO DISCUSS OTHER ASPECTS OF MICROSERVICES,
PLEASE CONTACT VENIAMIN:

vem@e-ngineers.com

WHEN E-NGINEERS CAN BE OF ANY HELP,
YOU ARE WELCOME TO CONTACT HANS PEETERS

+31 (0)6 51245088
hanspeeters@e-ngineers.com