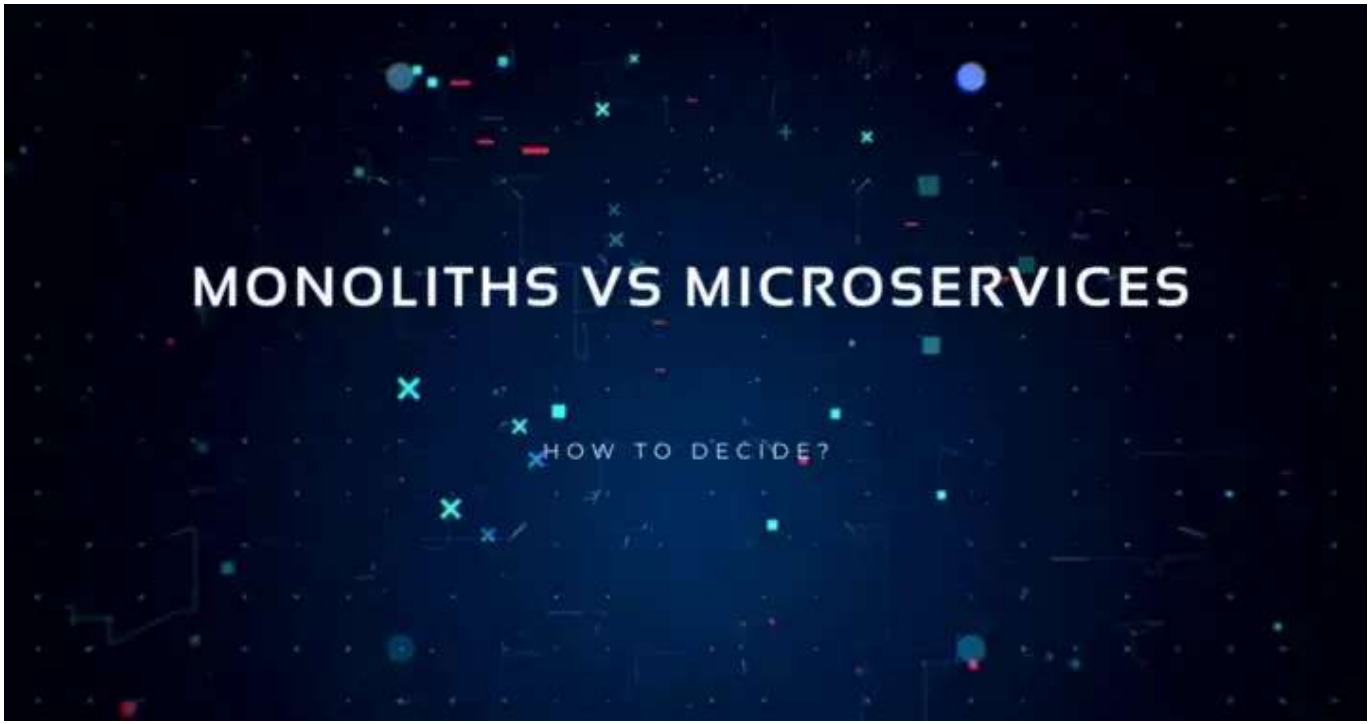
Published in Dev Genius

cloudeasyclub Follow

Jun 29, 2022 · 8 min read · Listen



Do you really need microservices? — How to decide?

Microservices is one of the most searched topics on the internet by developers. Unfortunately, most of the content available tends to lean on implementing microservices in one language/framework. The topic is mainly covered from the lens of a single developer or a single product (or maybe two) experiences.

The problem with that approach- you get to hear one side of the story. You get a single pro and cons list that tries to oversimply a relatively complex decision. The result is that people are often confused about practically using microservices in their system architecture. Most software architects I have met are either still figuring out the sweet spot (a pragmatic mental model) or following some old books or articles (which they strongly believe to be correct). But in most cases, people forget the first question they need to ask themselves — **do we need a microservices-based architecture or not?**

This blog post will share my mental model (decision-making approach) when I face the same question. I have been writing code for production for over a decade. I have done numerous freelance consulting projects and

designed systems that have served millions of users (with peak concurrency crossing 10 million+ Operations per second). Please think of this blog post as a summary of my experiential learning in this field. **The idea behind this post is not to dictate or push "the right approach"!. But to give you a starting point of the thought process.** The decision will always be highly subjective and contextual, and you will always need your mental models.

Crucial things to keep in mind

Before jumping into the decision-making approach for choosing between monoliths and microservices, I would like to highlight some significant truths that people often forget. There are three pillars involved whenever I have to make the above decision. Those three pillars are -

1. Product
2. People
3. Technology

I realize that the above points make me sound more like a product manager than a developer. Still, when you are an engineering leader, you should balance product thinking and engineering thinking because your decisions

can have long-term repercussions that can make or break an entire organization. Let me explain what these pillars are and what role they play.

Pillar number one — Product

- **Define the product well** — Defining what you are trying to build is essential to choosing between microservices and monoliths. As a developer/software architect, please understand that you are trying to solve a problem that isn't defined fully. I have seen numerous startups start writing code even when the product's scope is unclear. Never do that. Writing code is expensive!
- **Software is written to solve a problem. It shouldn't become a problem itself** — This happens most of the time because of the people in the engineering team who are part of the decision-making process. Their biases toward a language/technology/architectural pattern tend to creep in and mess up what otherwise could have been a perfectly designed system. Use this X database, not Y. Use this A framework, not B, etc. I hate language X because of Y. Trying to unnecessary constraint yourself because of your biases will often result in chaos when your application needs to scale. So, try and rise above it and make decisions with only one thing in mind — The most optimal solution for the given problem.

Pillar number two — People

- **Engineering team** — People are the biggest asset when building any successful software. Always include every Developer/SRE in the decision-making process. If you are taking specific calls related to the technology used in the project, discuss them with your team and try to come to a common conclusion that brings everyone on the same page.
- **End Users** — Software applications are written to solve problems for end-users. So, User behavior should be at the core of your decision. Predicting user behavior based on your product's user journeys will give you sufficient insights into the access patterns of your system and help you design the system well. We will discuss this in more detail later in this post.

Pillar number three — Technology

- **Understand the technology well** — I have seen services written in PHP and Ruby perform far better than those written in Go or Rust. And the reason is not that Ruby/PHP or similar languages are better than Go or Rust. The reason is that the people who wrote those programs understand their technology well (how the compiler works, the memory usage, limitations, etc.), and they know how to get the maximum output from those. And people who wrote the corresponding Go and Rust programs didn't fully understand the language features (specifically the runtime and concurrency part). The same logic applies to

frameworks/database/queues/file storage etc. Technology isn't magic. Using a new technology will not automatically solve the problems of your poorly written logic or your lousy system design. So, always try to understand the technology you are working with as much as possible!

- **Benchmark, monitor, and optimize** — 80% of the application code written is never monitored or optimized later. If the functionality works, then people think it's good to go. Then what was the point of those hundreds of competitive programming problems you solved where time and space complexity was everything? That is how you will get that "Experiential learning." By measuring and optimizing to get the most optimal performance possible and saving a ton of money.

I wish more people in product, technology, and executive positions understood the above pillars better.

Shameless plug

Hi there. Before presenting the solution, I decided to do a shameless plug. If you are someone looking to learn how to create kick-ass technology products, I have created a dedicated learning community called cloudeasy.club just for you. By joining this club, you can learn about system design, software architecture, and distributed systems. You can find us on

Discord or Whatsapp group. Also, feel free to ping me on LinkedIn or Twitter if you have any questions.

High-level decision flow — For deciding whether you should go for microservices or monoliths

Before I get into the details of this decision-making flow, please understand that this is a very audacious attempt to simplify something complex. Again, the idea is to give you a starting point of the thought process. I learned the below-mentioned flow the hard way — By making mistakes, By breaking things. So, your context may be different, but if you apply the basic model given below, you can preempt many mistakes you can make while making this decision. It's a structured way that uses product thinking and back-of-the-envelope calculations to make the right decision.

Step number one — Gather requirements

Make a list of all the product functionalities you are supposed to build. Visualize them on a board. I use Miro boards for the same (https://miro.com/). You can use any visual board tool. Write down only the bullet point and high-level requirements (not the intricacies).

Step number two — Make a high-level bullet point list of all the engineering problems to solve

- Break down the list of all the features in engineering problems.
- Then breakdown the problems into sub-problem based on a straightforward rule — Separation of concerns.
- Estimate the complexity of each subproblem to understand two things — programming effort required and resources required (CPU/Memory/file systems, etc.). This should be reasonably easy to estimate if you are a developer.
- The solution to each sub-problem will either be functionality in your product or some behind-the-scenes system that supports other functionalities.

Step number three — Make the user personas and write down their expected behavior with the individual functionalities in terms of a measurable metric

- Let me give you a simple example — 95% of the users scrolling the Instagram feed will not scroll past 30 posts. Or 90% of users scrolling through Instagram reels won't scroll for more than 50 reels. (I don't know the real numbers. It's just an example assumption). Now that you know the expected behavior of two user personas, you can draw a measurable metric like — HTTP Requests per second, Media consumed per second,

etc. This will give you a rough estimate of how much traffic (or load) each of these functionalities can expect. In most contexts, you can do this for all your sub-problems (or functionalities).

- In this step, also categorize the functionalities into two parts — Least used functionalities and most used functionalities.
- Estimate the load on each of these services using the way described in point 1 above.
- Please note that I am using the word “load” here. The reason is that this load can vary according to the nature of the application you are writing. For example — If you are writing a REST API (or REST Backend), the load will be “number requests per second.” If it’s an event-based (pub-sub type) system, it will be the number of events (or messages) per second If it’s a data crunching system, data transfer (in GB, TB, etc.) per second

Step number four — Estimate further based on platform traffic expected

We estimated the load per user persona (or user journey) in step three. In this step, we multiply that load with the primary metric (Daily Active users, number of concurrent users at any given time, etc.) For example — In the previous step, we took an example persona where 90% of users who come to Instagram feed won’t scroll for more than 50 reels. Now let’s assume that

Instagram makes four API calls per reel, and at any given time, 10000 concurrent users are scrolling through reels. With these assumptions, you can multiply and estimate your service's average load. If you want, you can drill it down to the level of the number of database calls (or IO operations, etc.) you will have to make every second. And that's how you arrive at a moment of truth. After looking at all the data, ask yourself two questions -

- If I group all these functionalities into a single monolith, will it be able to take this load without affecting a particular functionality (or overall user experience)?
- Will simple horizontal scaling help handle any extra uncalculated (more than estimated)load that may come up in the future?

If the answer to both the questions is Yes, go ahead with a Monolithic service. Also, remember that if you are not confident in your estimations, you can always write a proof of concept code (or application), do a simple benchmarking, or simulate the load using a load testing tool to improve the accuracy of your estimate.

Step number five — Take the project budget into consideration

If you are working with an organization where the budget is constrained, you must consider that while making the final call. Explain the advantages and disadvantages of both approaches to the appropriate stakeholders and let them weigh in. In my experience, a well-designed system is always cost-effective if you correctly choose your tools (languages, databases, deployment architecture, etc.).

That is it. By step number five, you should ideally have your answer on whether you should choose a microservices architecture for your next project.

Introduction to Microservices — A complete course

If you are someone who wants to learn all about moving from monoliths to microservices architecture, you can go through my course — "**Introduction to Microservices.**" It's a beginner-level course that gives you a pragmatic, language (tool) agnostic introduction to the world of microservices. It's available on youtube for free. Check out the youtube playlist. It covers a lot of engaging topics like

- How do we convert existing monoliths to microservices?
- Various communication methods in microservices.

- Monitoring and observability in details
- Different types of databases used in microservices
- Deployment methods and options for your microservices

Feel free to ping me on Twitter or LinkedIn if you have any questions. Join my community on Discord or Whatsapp group for more content.

Microservices Cloud Course System Design Interview System Architecture