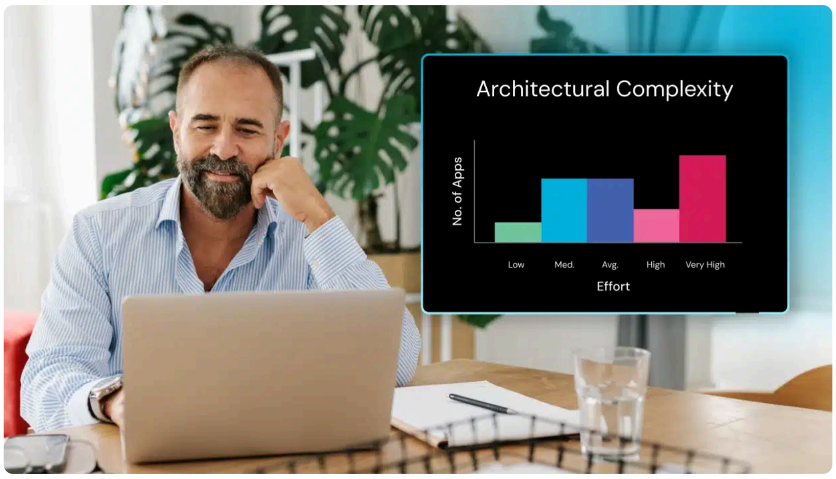
# Microservices architecture and design: A complete overview

Shatanik Bhattacharjee July 21, 2025



Microservices continue to gain traction as the go-to architecture for cloud-based enterprise applications. Their appeal lies in scalability, flexibility, selective deployability, and alignment with cloud-native design. Often viewed as an evolution of service-oriented architecture (SOA), a microservices approach allows each service to be developed, tested, and deployed independently.

But the benefits aren't automatic. To truly realize the promise of microservices, teams must follow sound design principles and architectural practices. This is especially true when breaking a monolith into microservices (https://vfunction.com/blog/monolith-to-microservices/)—a step often taken during cloud migration. Organizations expect the cloud to deliver agility, velocity, elasticity, and cost savings, yet those outcomes rarely materialize without a well-designed microservices architecture.

Let's take a closer look at how microservices architecture works and what makes it effective.

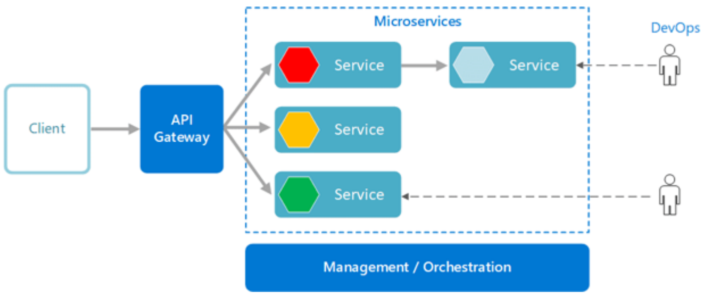
GUIDE

Learn how to end microservices chaos and manage your distributed architecture

[Download Now](https://vfunction.com/resources/guide-ending-microservices-chaos/)

## What is microservices architecture?

Microservices architecture, or simply microservices, comprises a set of focused, independent, autonomous services that make up a larger business application. The architecture provides a framework for independently writing, updating, and deploying services without disrupting the overall functionality of the application. Within this architecture, every single service within the microservices architecture is self-contained and implements a specific business function. For example, building an e-commerce application involves processing orders, updating customer details, and calculating net prices. The app will utilize various microservices, each designed to handle specific functions, working together to achieve the overall business objectives.



*Credit: [Microsoft Azure (https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices)], microservice architecture style.*

To fully understand microservices, it's helpful to contrast them with monolithic architecture.

## Benefits of microservices

Adopting a microservices architecture brings a range of benefits that can transform how organizations build and operate software. One of the primary advantages is the ability to scale individual services independently, which helps optimize resource usage and eliminates bottlenecks that can affect the entire application. This independent scalability also means that development teams can deploy services independently, reducing the risk of system-wide outages and enabling continuous delivery of new features. Deployments can be performed with one service at a time or span multiple services, depending on the specific needs of the deployment.

Microservices architectures promote faster development cycles, as teams can focus on specific business functions without waiting for changes to the entire system. It also means that teams can choose their preferred programming language to build with, allowing different programming languages to be used across the application as a whole. This approach also enhances system resilience, as failures in one service are less likely to impact the entire system. By empowering teams to innovate and experiment with new features and technologies, microservices foster a culture of agility and continuous improvement, enabling organizations to respond quickly to evolving market demands.

## Monolith vs. microservices architecture

Traditionally, software applications were built as monoliths ([3-tier architectural style (/blog/3-tier-application/)](/blog/3-tier-application/)) – single units containing all business logic, which simplified deployment to on-premise infrastructure. As applications grew more complex, these monoliths became difficult to maintain, test, and scale effectively. The advent of cloud computing and containerization enabled a new approach: breaking applications into smaller, independent services. This microservices architecture allowed teams to develop, deploy, and scale components independently, fostering innovation and agility. Success stories from early adopters drove widespread industry adoption of microservices as the preferred architecture for modern cloud applications. Here is a brief overview of the critical differences between the monolithic and microservices architectures.

| | **Monoliths** | **Microservices** |
|---|---|---|
| **Structure** | Monoliths bundle all functionality into a single executable. | Microservices are a collection of independent, lightweight applications that work together. |
| **Development** | Monoliths have a tightly coupled codebase, | Microservices are a set of independent codebases, allowing for |

| | | |
|---|---|---|
| | making changes risky and complex. | easier updates and faster development cycles. |
| **Complexity** | Monoliths can become massive and complex to manage. | Microservices decompose complexity into smaller, more manageable units. |
| **Resilience** | A single point of failure in a monolith can bring down the entire system. | Microservices isolate faults, preventing system-wide outages. |
| **Scalability** | Monoliths scale vertically by adding more resources to a single instance. | Microservices scale both vertically and horizontally, allowing for more efficient resource utilization. |
| **Team Structure** | Monolithic teams are often organized by technology (e.g., frontend team, backend team, database team). | Microservice teams are organized around business capabilities, each owning a specific service. |
| **Development** | Due to their complexity and risk, monoliths are typically deployed infrequently. | Microservices leverage CI/CD (https://www.infoworld.com/article/2269266/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html) for frequent and reliable deployments. |
| | | Microservices |

| | | |
|---|---|---|
| **Technology choice** | Monoliths have a single technology stack as they are deployed as a single runnable unit. | support polyglot development allowing teams to choose the best technology stack for their specific service. |

While monoliths can be suitable for smaller applications, microservices offer the agility, resilience, and scalability required for complex applications in dynamic environments.

## Key microservices architecture concepts

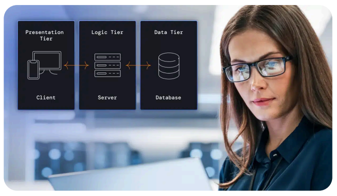
While every microservices architecture is unique, they share common characteristics that enable agility, scalability, and resilience. When you examine the concepts that encapsulate a microservices architecture, it resonates as a more modern approach to building and scaling applications. Let's look at the core concepts in more detail below.

- **Cloud-native:** Microservices are ideally suited for cloud environments. Their independent nature enables the efficient use of cloud resources, allowing for scalability on demand and cost-effectiveness through pay-as-you-go models. This avoids over-provisioning resources and paying for unused capacity, as would be the case with a monolithic application that requires more extensive, dedicated infrastructure.

  Additionally, this implies that microservices are inherently ephemeral, meaning they can be created and terminated easily without affecting the overall system. Therefore, they should be as stateless as possible, meaning each service instance should avoid storing information about user sessions or other temporary data. Instead, state information should typically be stored in caches and datastores that are external to the services themselves for easier independent scaling.

- **Organized around business capabilities:** Teams are structured around business domains, owning the entire lifecycle of a service—from development and testing to deployment and maintenance. This fosters a sense of ownership and accountability, streamlines development by reducing dependencies between teams, and ultimately improves the quality of the service. This domain-focused approach aligns with Domain-Driven Design (DDD) principles, where the software's structure reflects the business's structure.
- **Automated deployment:** Microservices rely heavily on automated CI/CD pipelines, enabling frequent and reliable deployments with minimal manual intervention. Automation accelerates the delivery of new features and updates, reduces the risk of errors, and enables faster feedback loops. With mature CI/CD pipelines, organizations can deploy changes multiple times daily, increasing agility and responsiveness to customer needs.
- **Intelligence at the endpoints:** Microservices favor "smart endpoints and dumb pipes." Intelligence resides within each service, enabling them to operate independently and communicate through simple protocols or lightweight message buses, such as Kafka. This promotes loose coupling, reduces reliance on centralized components, and allows for greater flexibility in technology choices and data management.
- **Decentralized control:** Teams can select the best technologies and tools for their specific service. This encourages innovation and enables teams to optimize for performance, scalability, or other relevant factors. The freedom to choose the right tool

In this article:
What is microservices architecture?
Benefits of microservices
Monolith vs. microservices architecture
Key microservices architecture concepts
**What is microservices architecture used for?**
Key technologies supporting microservices
When to use microservices
Microservices and Java
Critical steps to decompose a monolithic app into microservices
Best practices in microservices development
How big should a microservice be?
Microservices design patterns
Microservices architecture patterns
Design of communications for microservices
Considerations for microservices API design
Convert your monolithic applications to microservices
Manage existing microservices



for the job, known as polyglot programming, can lead to more efficient and effective solutions than a monolithic architecture, where a single technology stack is mandated.

- **Designed for failure:** Microservices are designed with fault tolerance in mind, recognizing that failures are inevitable in complex systems. Observability is ensured through robust monitoring, logging, and automated recovery mechanisms, which promote resilience. By isolating failures and enabling quick recovery, microservices minimize disruptions and maintain the application's overall health.

By embracing these concepts, organizations can leverage microservices to build highly scalable, resilient, and adaptable applications that thrive in dynamic environments.

## What is microservices architecture used for?

Microservices have become an extremely popular architectural tool for building applications, offering several benefits, including faster development, deployment, and scalability. It's important to note that many times adopting microservices architecture is an evolutionary process and very few large scale applications were born as microservices. Many times applications are lifted and shifted to the cloud as monoliths and only later are rearchitected and transformed into microservices that allow organizations to take advantage of advanced cloud services, like serverless computing.

Let's consider some real-world use cases for a microservices architecture:

- **E-commerce platforms**
  E-commerce platforms benefit from

microservices by breaking down functionalities such as product catalogs, payment processing, order management, and user profiles into independent services. This allows teams to update specific features, like checkout or search, without affecting other parts of the application, resulting in faster deployments and more reliable scaling during high-traffic events, like holiday sales.

- **Streaming services**
  On streaming platforms, microservices can independently handle various functionalities, such as video streaming, user recommendations, search, and user profiles. This enables personalized experiences by allowing the recommendation service to quickly update suggestions based on viewing history while the streaming service handles high data loads. It also improves fault isolation, ensuring that an issue with one component doesn't disrupt the entire service.
- **Banking and financial applications**
  Banks and financial institutions utilize microservices to separate services such as account management, transaction processing, customer support, and fraud detection. Microservices help ensure that critical services like transaction processing remain available and performant, while allowing other system components to evolve independently. This approach provides better security, compliance, and a faster time-to-market for new features.

## Key technologies supporting microservices

Microservices adoption necessitates specific tools for effective management, orchestration, and scaling. The following key technologies, though not comprehensive, are crucial for deploying robust microservices architectures that enhance application agility and efficiency.

- **Containers (e.g., Docker):** Containers package applications and their dependencies into isolated units, ensuring consistent runtime environments across different underlying virtual environments and simplifying deployment and management. This isolation benefits microservices, allowing independent development, testing, and deployment.
- **Orchestration platforms (e.g., Kubernetes):** Kubernetes automates the deployment, scaling, and management of containerized applications. It handles tasks like load balancing, rolling updates, and self-healing, freeing developers to focus on application logic.
- **Service mesh (e.g., Istio):** A service mesh enhances communication between microservices, providing features like traffic management, security, and observability. It acts as a dedicated infrastructure layer for inter-service communication, improving resilience and reducing development overhead.
- **Serverless computing (e.g., AWS Lambda):** Serverless platforms abstract away infrastructure management, allowing developers to focus solely on code. This model can be highly cost-effective for microservices, as resources are consumed only when needed, and scaling infrastructure like this is seamless.

RELATED

The case for migrating legacy java applications to the cloud → Read More (/blog/the-case-for-migrating-legacy-java-applications-to-the-cloud/)

## When to use microservices

Despite their benefits, microservices aren't always the universal solution, especially if a current monolith fulfills business requirements. Experts like Martin Fowler (https://martinfowler.com/) and Sam Newman (https://samnewman.io/) recommend adopting microservices only when they address specific, unmet needs.

Consider transitioning to a microservices architecture if you:

- Aim for scalability, swift deployments, faster time-to-market, or enhanced resiliency.
- Require the ability to deploy updates with minimal downtime, crucial for SaaS businesses, by updating only the affected services.
- Handle sensitive information necessitating strict compliance with data protection standards (GDPR, SOC2, PCI), achievable through localized data handling within specific microservices.
- Seek improved team dynamics, as microservices support the "two-pizza team (https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/two-pizza-teams.html)" model, meaning teams no larger than those that can be fed by two pizzas, promoting better communication, coordination, and

ownership.

## Microservices and Java

Cloud-native applications, with their well-known benefits, have rapidly shifted the software development lifecycle to the cloud. Java, in particular, is well-suited for cloud-native development due to its extensive ecosystem of tools and frameworks, such as Spring Boot (https://spring.io/projects/spring-boot), Quarkus (https://quarkus.io/), Micronaut (https://micronaut.io/), Helidon (https://helidon.io/), Eclipse Vert.x (https://vertx.io/), GraalVM (https://www.graalvm.org/) and OpenJDK (https://openjdk.java.net/). This section will delve into cloud-native Java applications.

### A typical cloud-native Java application stack

Here is a simplified view of a Java cloud-native applications stack: Spring, Maven or Gradle, JUnit, Docker, and others. Although only one option is mentioned at each step, several alternatives exist:

- Use Spring Initializr (https://start.spring.io/) to create a Spring Boot project
- Use Spring Boot (https://spring.io/projects/spring-boot) as the microservice development framework
- Write the facade layer using a combination of REST endpoints (Spring Web (https://docs.spring.io/spring-framework/reference/web/webmvc.html)) and message listeners (https://docs.spring.io/spring-boot/reference/messaging/index.html) as needed
- Write the business logic using Spring beans (https://docs.spring.io/spring-

boot/reference/using/spring-beans-and-dependency-injection.html)

- Write the data persistence (https://spring.io/projects/spring-data) layer and integration (https://spring.io/projects/spring-integration) layers
- Test with Spring Boot Test (https://docs.spring.io/spring-boot/reference/testing/index.html)
- Keep every microservice in a separate Maven (https://maven.apache.org/) or Gradle (https://gradle.org/) project
- Build either OpenJDK (https://openjdk.java.net/) or GraalVM (https://www.graalvm.org/) images with Docker (https://www.docker.com/)
- Deploy the application with Kubernetes (https://kubernetes.io/)
- Use Apache Kafka (https://kafka.apache.org/), ActiveMQ (https://activemq.apache.org/) or RabbitMQ (https://www.rabbitmq.com/) as a message broker
- Use Spring Cloud Gateway (https://spring.io/projects/spring-cloud-gateway) as the API gateway

## Critical steps to decompose a monolithic app into microservices

Converting an existing monolithic application to a microservices architecture is called app modernization (/use-cases/application-modernization/). While there are many ways of doing this, broadly, the process followed would be:

1. Identify functional domains in the

application. Group these domains into the minimum number of modules that need to be independently scalable.

2. Choose a module and refactor it to disentangle it from other modules such that it can be extracted out into an independently deployable unit. This refactoring should include the removal of method calls to classes and database table access in other modules. Additionally, refactor unnecessary dependencies and dead code out to have focused business logic necessary in it.
3. Extract this module out with its corresponding library dependencies into a new codebase from the monolith
4. Develop synchronous and asynchronous APIs for client interactions and create corresponding clients (e.g., in the User Interface, other modules, other applications, etc.)
5. Optionally, upgrade its technology stack to the latest libraries and frameworks as appropriate.
6. Compile, deploy, and test this module as a service in the target environment of choice.
7. Repeat the last five steps until the monolith has been decomposed into a set of services.
8. Split the monolithic database into databases/schemas per service
9. Plan the transition to be iterative and incremental.

RELATED

The easy way to transition from monolith to microservices

(/blog/the-easy-way-to-Readtransition-Morefrom-monolithic-to-microservices/)

## Best practices in microservices development

We have seen that microservices architecture can provide several benefits. However, those benefits will only accrue if you follow good design and coding principles. Let’s take a look at some of these practices.

- You should model your services on business features and not technology. Every service should only have a single responsibility.
- Decentralize. Give teams the autonomy to design and build services.
- Don’t share data. Data ownership is the responsibility of each microservice, and should not be shared across multiple services to avoid high latency.
- Don’t share code. Avoid tight coupling between services to avoid inefficiencies in the cloud.
- Services should have loose logical coupling but high functional cohesion. Functions likely to change together should be part of the same service.
- Use a distributed message bus. There should be no chatty calls between microservices.
- Use asynchronous communication to handle errors, isolate failures within a service, and prevent them from cascading into broader issues.
- Determine the correct level of abstraction for each service. If too coarse, then you will not reap the benefits of microservices. If too fine, the resulting overabundance of services will lead to an operational nightmare. Practically, it is best to start with a coarse set of services and make them finer-grained based on scalability needs.

## How big should a microservice be?

The size of a microservice, measured in lines of code, isn't the main concern; instead, each microservice should manage a single business feature and be sized appropriately to fulfill this responsibility effectively.

This approach raises the question of defining what a business feature includes, which entails establishing service boundaries. Utilizing Domain-Driven Design (DDD), we define the 'bounded context' for each domain, a key concept in DDD that sets clear limits for business features and scopes individual services. With a well-defined bounded context, a microservice can be updated independently of others without interference.

## Microservices design patterns

Microservices architecture is difficult to implement, even for experienced programmers. Using the following design patterns can reduce the complexity.

### Ambassador

Developers use the Ambassador design pattern (https://docs.microsoft.com/en-us/azure/architecture/patterns/ambassador) to handle common supporting tasks like logging, monitoring, and security.

### Anti-corruption

This is an interface between legacy and modern applications. It ensures that the limitations of a legacy system do not hinder the optimum design of a new system.

## Backends for front-ends

A microservices application can serve different front-ends (clients), such as mobile and web. This design pattern concerns itself with designing different backends to handle the conflicting requests coming from different clients.

## Bulkhead

The bulkhead design pattern describes allocating critical system resources such as processor, memory, and thread pools to each service. Further, it isolates the assigned resources so that no entities monopolize them and starve other services.

## Sidecar

A microservice may include some helper components that are not core to its business logic but help in coding, such as a specialized calendar class. The sidecar pattern (https://dzone.com/articles/sidecar-design-pattern-in-your-microservices-ecosy-1) specifies deploying these components in a separate container to enforce encapsulation.

## The Strangler pattern

To transition from a monolith to microservices, follow these steps: First, develop a new service for the desired function. Next, configure the monolith to bypass the old code and call the new service. Then, ensure the new service operates correctly. Finally, eliminate the old code. The Strangler design pattern (/resources/ebook-strangler-fig-pattern-for-application-modernization/)–based on the lifecycle of the strangler fig plant described by Martin Fowler in this 2004 blog post (https://martinfowler.com/bliki/StranglerFigApplication.html)–helps implement this approach.

## Microservices architecture patterns

We have seen some patterns that help in microservices design. Now let us look at some of the architectural best practices.

### Dedicated datastore per service

It's best not to use the same data store across microservices because this will result in a situation where different teams share database elements and data. Each service team should use its own database that is the best fit for it rather than sharing it to ensure performance at scale.

### Don't touch stable and mature code

If you need to change a microservice that is working well, it is preferable to create a new microservice, leaving the old one untouched. After testing the new service and making it bug-free, you can merge it into the existing service or replace it.

### Version each microservice independently

Build each microservice separately by pulling in dependencies at the appropriate revision level. This makes it easy to add new features without breaking existing functionality.

### Use containers to deploy

When you package microservices in containers, all you need is a single tool for deployment. It will know how to deploy the container. Additionally, containers provide a consistent runtime environment to microservices irrespective of the underlying hardware infrastructure they are deployed on.

## Remember that services are ephemeral

Services are ephemeral, meaning they can be scaled up and down. Therefore, do not maintain stateful sessions or write to the local filesystem within a service. Instead, use caches and persistent datastores outside the container to hydrate a service with the required state.

## Other patterns

We have covered the simplest and most widely used patterns here. Other patterns are available, including Auto Scaling, Horizontal Scaling Compute, Queue-Centric Workflow, MapReduce, Database Sharding, Co-locate, Multisite Deployment, and many more.

## Gateway aggregation

This design pattern merges multiple requests to different microservices into a single request. This reduces traffic between clients and services.

## Gateway offloading

The gateway offloading pattern deals with microservices offloading common tasks (such as authentication) to an API gateway. Clients call the API gateway instead of the service. This decouples the client from the service.

### Gateway routing

Gateway routing enables several microservices to share the same endpoint, freeing the operations team from managing many unique endpoints.

### Adapter pattern

The adapter pattern acts as a bridge between incompatible interfaces in different services. Developers implement an adapter class that joins two otherwise incompatible interfaces. For example, an adapter can ensure that all services provide the same monitoring interface. So, you need to use only one monitoring program. Another example is ensuring that all log files are written in the same format so that one logging application can read them.

## Design of communications for microservices

To deliver a single business functionality, multiple microservices might collaborate by exchanging data through messages, preferably asynchronously, to enhance reliability in a distributed system. Communication should be quick, efficient, and fault-tolerant. We will explore issues related to microservices communication.

### Synchronous vs. asynchronous messaging

Microservices can use two fundamental communication paradigms for exchanging messages: synchronous and asynchronous.

In synchronous communication, one service calls another service by invoking an API that the latter exposes. The API call uses a protocol such as HTTP or [gRPC (https://grpc.io/)](https://grpc.io/) (Google Remote Procedure Call). The caller waits until a response is received. In programming terms, the API call blocks the calling thread.

In asynchronous communication, one service sends a message to another but does not wait for a response and is free to continue operations. Here, the calling thread is not blocked on the API call.

Both communication types have their pros and cons. Asynchronous messaging offers reduced coupling, isolation of a failing part, increased responsiveness, and better workflow management; however, if not set up with the understanding that system design will be different, you may experience disadvantages like increased latency, reduced throughput, and tighter coupling on a distributed message bus.

## Distributed transactions

Distributed transactions with several operations are common in a microservices application. This kind of transaction involves several microservices, each executing some steps. In some cases, transactions are successful only if all the microservices correctly execute the steps they are responsible for; here, if even one microservice fails, the transaction fails. In other cases, such as in asynchronous systems, sequence is of lower consequence.

A failure could be transient. An example is a timeout failure due to resource starvation, which might result in long retry loops. A non-transient failure is more serious. In this case, an incomplete transaction results, and it may be necessary to roll back, or undo, the steps that have been executed so far. One way to do this is by using a Compensating Transaction.

*Compensation logic used in booking travel itinerary. Credit: Microsoft Azure (https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction).*

## Other challenges

An enterprise application may consist of multiple microservices, each potentially running hundreds of instances, which can fail for various reasons. To build resilience, developers should retry API calls.

For load balancing (https://en.wikipedia.org/wiki/Load_balancing_(computing)), Kubernetes uses a basic random algorithm. A service mesh can provide sophisticated load balancing based on metrics for more advanced needs.

When a transaction spans multiple microservices, each maintains its own logs and metrics. Correlating these in the event of a failure is achieved through distributed tracing.

## Considerations for microservices API design

Many microservices "talk" directly to each other. All data exchanged between services happens via APIs or messages. So, well-designed APIs are necessary for the system to work efficiently.

Microservices apps support two types of APIs.

- Microservices expose public APIs called from client applications. An interface called the API gateway handles this communication. The API gateway is responsible for load balancing, monitoring, routing, caching, and API metering.
- Inter-service communication uses private (or backend) APIs.

Public APIs must be compatible with the client, so there may not be many options here. In this discussion, we focus on private APIs.

Based on the number of microservices in the application, inter-service communication can generate a lot of traffic. This will slow the system down. Hence, developers must consider factors such as serialization speed, payload size, and chattiness in API design.

Here are some of the backend API recommendations and design options with their advantages and disadvantages:

## REST vs. RPC/gRPC:

REST is based on HTTP verbs and is well-defined semantically. It is stateless and, hence, freely scalable but does not always support the data-intensive needs of microservices. RPC/gRPC might lead to chatty API calls unless you design them correctly, yet this interface is potentially faster in many use cases than REST over HTTP.

## Message formats

You can use a text-based message format, such as XML or JSON, or a binary format. Text-based formats are human-readable but verbose.

## Response handling

Return appropriate HTTP Status Codes and helpful responses. Provide descriptive error messages.

### Handle large data intelligently

Some requests may result in a large amount of data returned from the database query. Engineering teams may not need all the data; hence, processing power and bandwidth are wasted. Teams can solve this by passing a filter in the API query string, using pagination, compressing the response data, or streaming the data in chunks.

### API versioning

APIs evolve. A well-thought-out versioning strategy helps prevent client services from breaking because of API changes.

## Convert your monolithic applications to microservices

The benefits of a microservices architecture are substantial. If your aging monolithic application hinders your business, consider transitioning to microservices and taking advantage of the microservices infrastructure that public clouds offer.

However, adopting microservices involves effort. It requires careful consideration of design, architecture, technology, and communication. Tackling complex technical challenges manually is risky and generally advised against.

vFunction understands the constraints of costly, time-consuming, and risky manual app modernization. To counter this, vFunction's [AI-driven architectural modernization (https://vfunction.com/platform/)](https://vfunction.com/platform/) platform automates cloud-native modernization through a scalable factory model that leverages code assistants. It is the only platform that feeds architectural context based on runtime analysis into code assistants.

*Once your team decomposes a monolith with vFunction, it's easy to automate extraction to a modern platform.*

Leveraging automation, GenAI, and data science, the platform enables smart transformation of complex Java monoliths into microservices. It stands as the unique and pioneering solution in the market.

## Manage existing microservices

If you already have microservices (/platform/microservices/), vFunction can help you manage complexity, prevent architectural drift (https://vfunction.com/use-cases/architectural-drift/), and enhance performance. With AI-driven architectural observability, vFunction provides real-time visibility into service interactions, revealing anti-patterns and bottlenecks that impact scalability. Its governance features set architectural guardrails, keeping microservices aligned with your goals. This enables faster development, improved reliability, and a streamlined approach to scaling microservices with confidence.

*vFunction supports governance for distributed architectures, such as microservices, to help teams move fast while staying within the desired architecture framework.*

To see how top companies use vFunction to manage their microservices, contact us. We'll show you how easy it is to transform your legacy apps or complex microservices into streamlined, high-performing applications.

Learn more about using vFunction to manage your microservices

Explore Platform (/platform/microservices/)

**Shatanik Bhattacharjee**

Principal Architect

Shatanik is a Principal Architect at vFunction. Prior to that, he was a Software and Systems Architect at a Smart Grid startup. He has vast practical experience in good and not-so-good software design decisions.

(https://www.linkedin.com/in/shatanikbhattacharjee/)

## Other Related Resources

View All(/resource-center/)

BLOG

**What is Software Architecture? A Comprehensive Guide – vFunction**

Read More

(https://vfunction.com/blog/what-is-software-architecture/)

BLOG

**What are microservices in Java? Best practices, and more.**

Read More

(https://vfunction.com/blog/microservices-in-java/)

BLOG

**System architecture diagram basics & best practices**

Read More

(https://vfunction.com/blog/architecture-diagram-guide/)

## Get started with vFunction

Accelerate engineering velocity, boost application scalability, and fuel innovation with architectural modernization. See how.

Request a Demo

(/partners/become-a-partner/)

center/datasheets/)

FAQ (/resource-center/faq/)

Guides (/resource-center/guides/)

Refactor This Podcast (/resource-center/refactor-this-podcast/)

Videos (/resource-center/videos/)