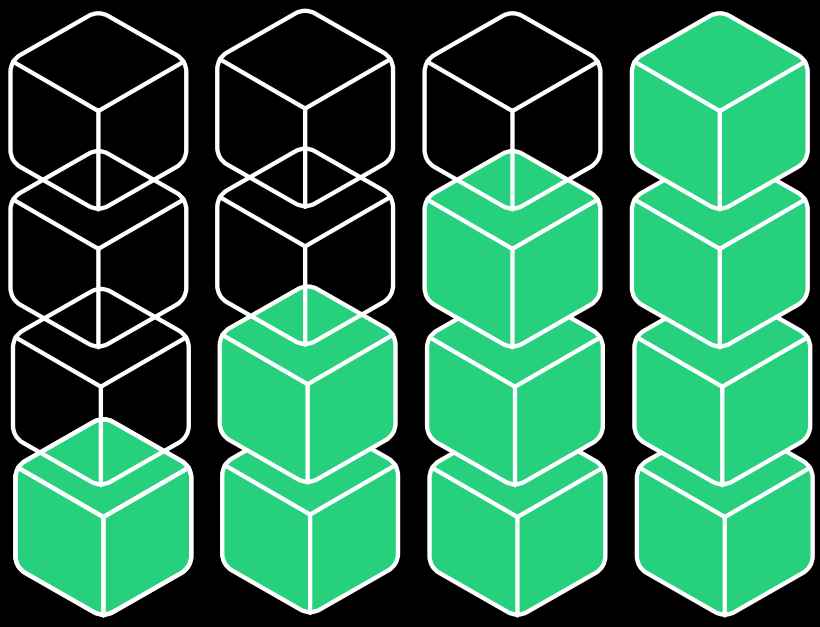
CAMUNDA

Decoding Microservices:

Best Practices Handbook for Developers



March 2022

camunda.com

Contents

Introduction

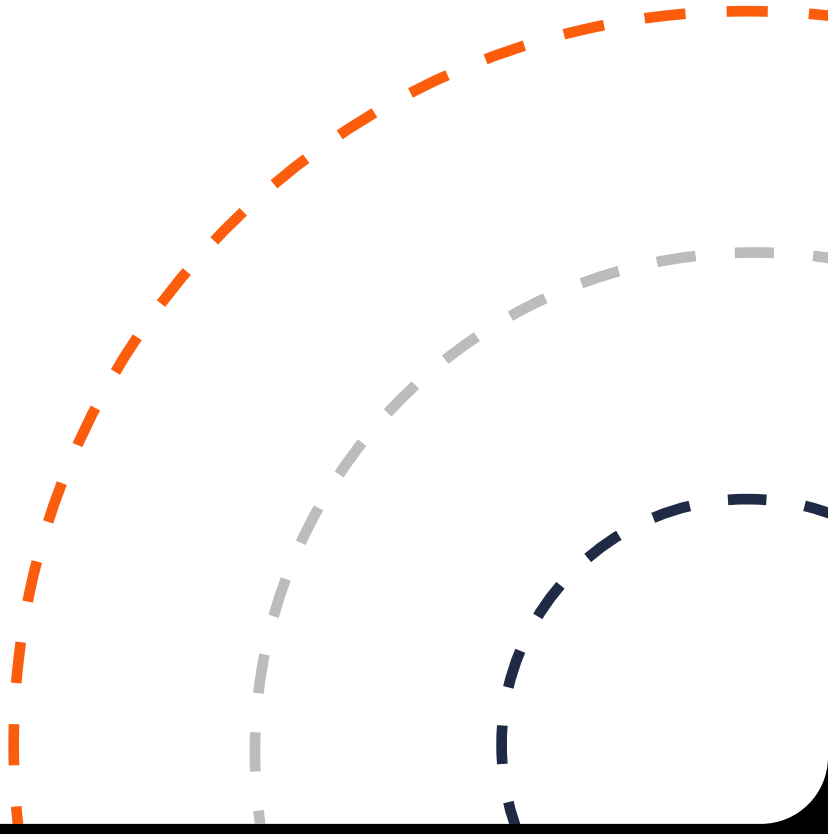
The popularity of building modern applications using a microservices architecture has proven to have a host of benefits, including developer independence, scalability, fault isolation, faster time to market, and more.

These benefits are not without challenges, however.

By their very nature, microservices architectures require services to be as loosely coupled as possible. At the same time, these services must be able to communicate effectively to execute the business processes at the proper time and in the correct way. Failure to communicate properly could result in the whole process falling apart.

This issue and many other challenges can be mitigated by following some best practices.

These best practices can be consumed in any order, but like microservices themselves, they all interact with one another and can help you as you design, build, and scale your distributed system.



Avoid building a “distributed monolith”

If you look at any given business process from end to end, starting from the original user’s need and ending with some meaningful end result for them, it will often stretch across business functions, multiple microservices, and possibly even some monolithic applications.

Therein lies one of the major paradoxes of microservices architecture.

As we touched on in the introduction, microservices are supposed to operate independently yet, at the same time, need to communicate with each other. Too many dependencies between your services mean you’ve just built a distributed monolith. As a result, you’re now saddled with the downsides of both architectural styles but with the benefits of neither.

The way to avoid this issue is by properly modeling the business processes that the microservices will be handling. By mapping out the different business processes end to end, you can organize them into appropriate “pieces” that fit into the different services, helping you avoid unnecessary dependencies as you build and scale your platform.

Align each microservice to the business domain

As you map out your business processes, each microservice should be well aligned to your business requirements and goals. You should avoid aligning services based on organizational structure, technology choices, or the presence of technical debt.

Handling this will vary from organization to organization, and there is, unfortunately, no one-size-fits-all answer. One of the ways to think about this is to consider who would be held responsible if the microservice goes down. You can usually recognize this issue if you cannot point to the person owning a process, and therefore, the service is responsible for executing it.

Definitions

ACID transactions stands for:

Atomicity – All the steps of a transaction succeed or fail together, no partial state, all or nothing.

Consistency – All data in the database is consistent at the end of the transaction.

Isolation – Only one transaction can touch the data at the same time; other transactions wait until the completion of the working transaction.

Durability – Data is persisted in the database at the end of the transaction.

Business Process Modeling – A graphical representation of an organization’s business processes and workflows. It is a vital part of effective process management as it generates critical insights into how smoothly a process is running.

Business Process Model and Notation (BPMN) – A visual language used to model and automate processes. It has become a widely adopted standard in the business process modeling community.

Microservices – A modular architecture style in which a complex application is made up of small, autonomous, and loosely coupled services. These services are fine-grained, implemented using various programming languages, messaging enabled, independently deployable, and decentralized.

Distributed Monolith – A distributed monolith is an application that’s deployed like a microservice but is built like a monolith.

Choreography – Event-driven communication.

Orchestration – Command-drive communication.

You absolutely want to avoid having dozens of people responsible for the same single process. If everybody owns the process, then nobody owns the process. This makes it impossible to update or improve without significant downtime or resource investment, and you're stuck with a distributed monolith.

Balance choreography and orchestration

Orchestration and choreography are two approaches to how microservices communicate with one another and work together.

These approaches are often viewed as an either/or scenario— you choose either event-driven or command-driven communication. One approach is not necessarily better than the other, and each has its own pros and cons.

For instance, orchestration can simplify communication as you can chart out how each service interacts with one another. However, this is a path to building a distributed monolith. Meanwhile, choreography helps keep microservices loosely coupled, but you can lose sight of the larger-scale flow of information from one service to another. Being able to understand the flow, change it, or operate it can get very complicated very quickly.

The ideal solution is usually a mixture of choreography and orchestration. As you design and implement microservices, consider whether events or commands are the right approaches for every single communication link between services.

Build fault-tolerant microservices and handle degradation gracefully

There are many ways to succeed with microservices, but there are equally as many ways to fail.

The more microservices you have, the more points of failure that exist. If one microservice fails, it shouldn't bring down the whole system or the whole business process. That is one of the situations that microservices are being implemented to avoid.

Building fault tolerance starts with identifying the most common issues that might go wrong in a given process and implementing a fallback plan. This should solve a majority of issues.

Keep in mind, that once you have reached a critical mass of microservices, the level of complexity can become overwhelming, and problems that you did not plan for can arise.

Likewise, implementing a "default flow" is another solution. This means having a flow or process to fall back on, so users aren't just dumped out of the process. That way, there's a fallback plan, even for scenarios you may not have considered.

Using a workflow engine to accomplish building a default path has been shown to be effective. Workflow engines can enable persistent states of a process instance, manage timeouts and alerts, compensate when business transactions fail, and provide rich reporting on the efficiency of business processes.

Build-in security from the beginning

Microservices architecture is not microwave or toaster. You can't just set up your microservices, ensure they're running properly, and then go onto a new project.

As your microservices environment grows, so do your security risks. While it's common practice to protect your pre-prod or final application, more recent attacks—like Solar Winds—have been able to inject malicious code into dev environments. And due to their distributed nature, it's harder to monitor each individual service for compliance since they are isolated by design.

It's crucial to the security and success of your application that you build security protocols from the beginning, and continually monitor microservice performance and resource

availability. This way, you can quickly identify compromised resources, enhancing security.

With the complexity of a microservices environment, you shouldn't only rely on standalone or isolated security stacks that can't do the job. You need to have a centralized system for security that will help you identify and close security gaps so that you can secure your entire application.

Each microservice should have its own data storage

Data storage is a key issue with microservices that can cause many headaches, typically resulting in a performance bottleneck.

In a monolithic architecture, relational databases form a "single source of truth" for accessing data.

Each microservice must have its own database. Moreover, you shouldn't have multiple microservices touching the same database or data persistence layer.

There are several reasons for this. First, you avoid creating a dependency that will lead you back toward making a distributed monolith. Second, changes to one microservice's database will not impact other services should they need to be upgraded or changed. Finally, each microservice can use the best type of database for its needs.

Another best practice is to have your data or reporting versus execution stored in a different database.

As for managing persistent state, using a lightweight workflow engine like Camunda can manage state for long-running processes across multiple services.

Make monitoring easier

Monitoring the performance of microservices is not as simple as tracking the CPU or RAM utilization of a monolithic application.

You have gone from one big application to many smaller processes. Not only do you need to monitor all of them to ensure that nothing has crashed or is frozen, but you also have to monitor the communication between them.

You need to make sure that the services are properly authenticated with the correct credentials, which can add a layer of complexity. There are tools available for monitoring the technical health of your microservice environment and the health of communication between them, but this isn't the same as the process working successfully.

This requires understanding what these microservices are actually trying to accomplish in the bigger picture. Using a workflow engine can assist with this. By sitting atop the microservice layer, you can not only monitor for health and process success, but also get overarching features like graphical visualization, status monitoring, and reporting.

Success with Microservices

When your monolithic architectural environment starts impacting your organization's ability to thrive due to insufficient agility and scalability, it's time to migrate to a microservices-based architecture.

Microservices make it easier for individual teams to design, build, upgrade, and scale autonomously and in parallel, without depending on anything outside of the services' confines.

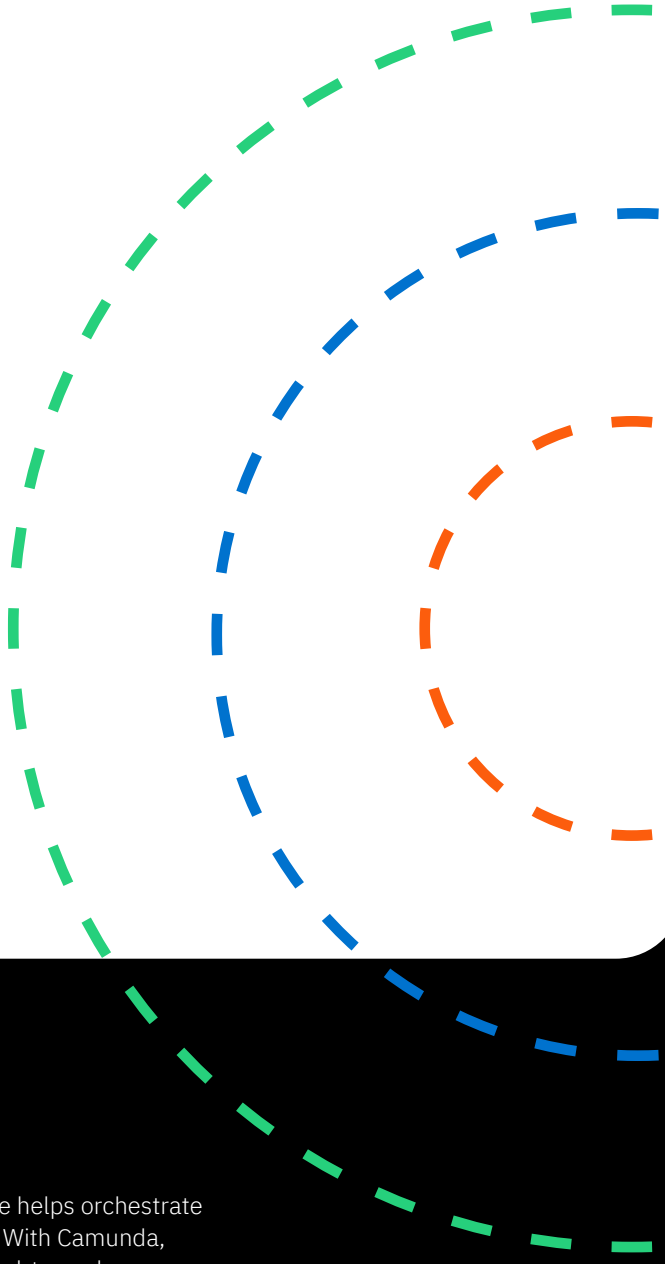
Success with this architectural style relies on carefully mapping out your journey as you continue to build upon your newly established architecture with microservices.

Without careful planning, orchestrating and monitoring microservices across long-running business processes can turn into a logistical nightmare; severely limiting scalability and eliminating any benefits the use of microservices may have had.

Implementing a workflow engine like Camunda can boost your ability to effectively

communicate, monitor, identify, and resolve problems across microservices.

- State Handling – Persists in the state of each instance of a business process (e.g., each order placed on an e-commerce website).
- Explicit Processes – Makes business processes explicit instead of burying them in code, making it easier for teams to understand and modify them.
- Message Correlation and Coordination – Merges messages belonging to a single process instance and decides next steps — BPMN process modeling language automatically implements message patterns such as sequences, synchronization, mutual exclusion, and timeouts.
- Compensation for Problems – Compensates if a business transaction or process encounters a problem that requires previously completed steps to be undone.
- Timeout Handling – Tracks the passage of time and automatically takes action or switches to another path in the process flow if an event does not take place as expected.
- Error Handling – Allows you to specify the behavior that should happen when an error occurs (e.g., retrying an action, taking another path).
- Transparency of Status – Enables operations teams to monitor the status of process instances in real-time.
- Collaboration – Provides graphical models of business processes that facilitate discussions between business stakeholders, developers, and operations teams.



About Camunda

Camunda is the leader in process orchestration software. Our software helps orchestrate complex business processes that span people, systems, and devices. With Camunda, business users collaborate with developers to model and automate end-to-end processes using BPMN-powered flowcharts that run with the speed, scale, and resiliency required to compete in today's digital-first world. Hundreds of enterprises such as Atlassian, ING, and Vodafone design, automate, and improve mission-critical business processes with Camunda to drive digital transformation. To learn more visit **camunda.com**.