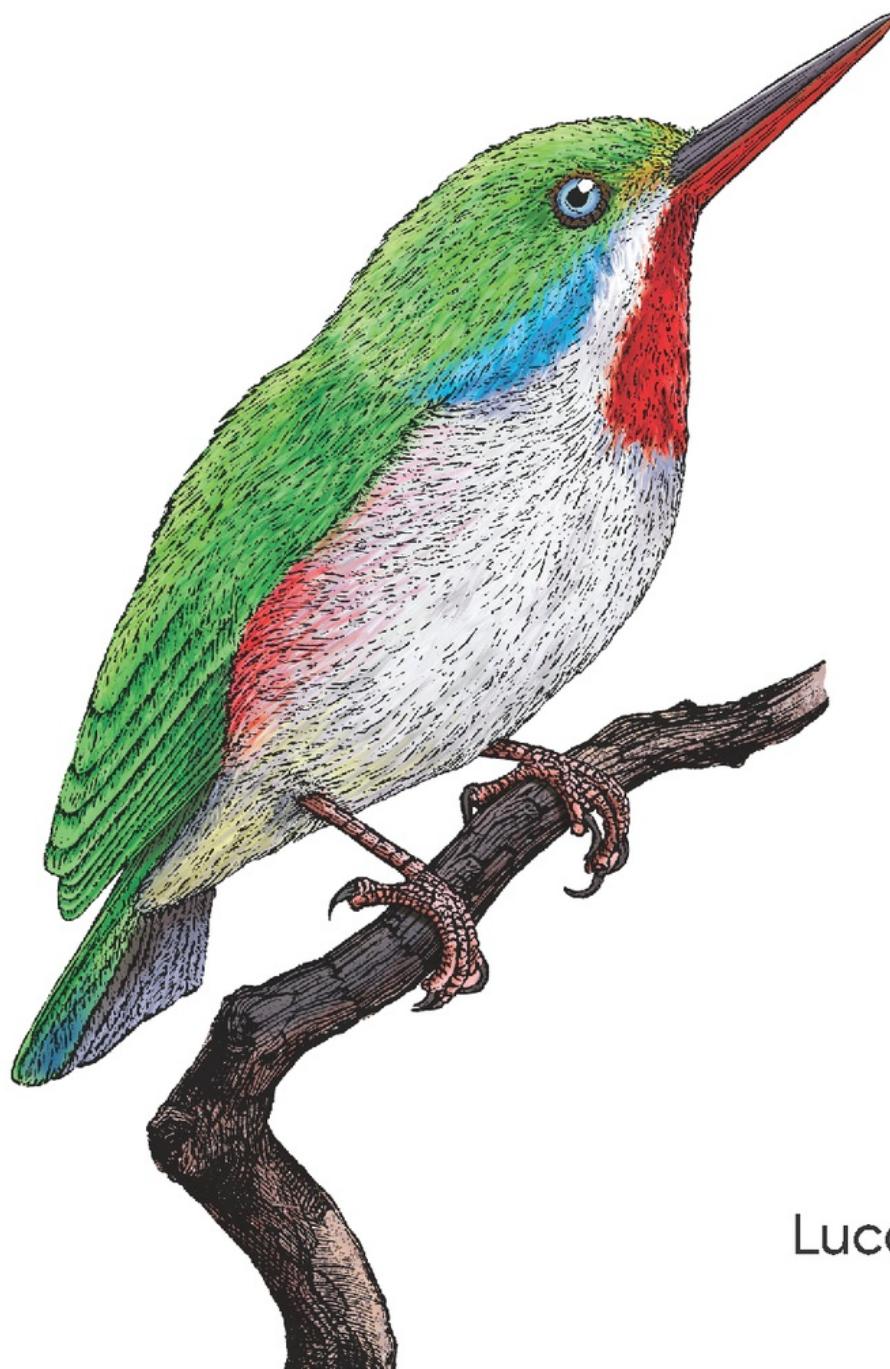


O'REILLY®

2nd Edition

Building Micro-Frontends

Distributed Systems for the Frontend



Luca Mezzalira

Praise for *Building Micro-Frontends*

This book is the intersection of Luca's multidisciplinary excellence—combining the rigor of a seasoned architect, the clarity of a tech video creator, and the insight of a widely read newsletter curator—distilled into the definitive guide to micro-frontends.

—Max Gallo, Distinguished Engineer at DAZN

The first edition covered every critical aspect of adoption. This update keeps pace with today's tech and practices, modern composition, platform needs, and real scenarios, so it remains my go-to field guide.

—Alessandro Cinelli, Engineering Manager at Samsara

If microservices and team topologies had a frontend baby, this book would be it.

—Michael Di Prisco, Tech Lead at Jointly

If you are a technical lead or decision maker looking into introducing and implementing micro-frontends in your organization, then Luca Mezzalira has written the book for you.

—Jens Oliver Meiert, Engineering Manager and Author
(meiert.com)

Building Micro-Frontends

SECOND EDITION

Distributed Systems for the Frontend

Luca Mezzalira

O'REILLY®

Building Micro-Frontends

by Luca Mezzalira

Copyright © 2026 Luca Mezzalira. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Louise Corrigan

Development Editor: Angela Rufino

Production Editor: Christopher Faucher

Copyeditor: Piper Content Partners

Proofreader: Andrea Schein

Indexer: Ellen Troutman-Zaig

Cover Designer: Karen Montgomery

Cover Illustrator: Susan Thompson

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

November 2021: First Edition

November 2025: Second Edition

Revision History for the First Edition

- 2025-10-24: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098170783> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Building Micro-Frontends*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17078-3

[LSI]

Foreword for the 1st Edition

Named architecture styles (such as microservices) are like art periods in history—no one plans for them, no single person is responsible for the ideas, yet they suffuse through a community. For example, no group of artists gathered in France in the late 19th century and decided to create impressionism. Rather, forces in the art world (reacting to the introduction of primitive photography) drove artists toward representation rather than capturing reality.

The same is true for styles of architecture—regardless of what some developers may suspect, there is no ivory tower to which architects retreat to concoct the New Big Thing. Instead, clever architects notice new capabilities appearing within the ecosystem, and they identify ways to combine these emerging capabilities to solve old problems. For microservices, the advent of DevOps, programmatic control of machine configuration (which led to containerization), and the need for faster turnaround spawned this architecture style at several influential organizations.

In the past, the name of the architecture style would lag for several years as people figured out that a trend was underway and identified the distinguishing characteristics that would lead to a name. However, that story slightly differs for microservices. Architects have become clever about noticing the development of new trends and keep a keen eye out for them. In March 2014, Martin Fowler and James Lewis published an [iconic essay on Fowler's website](#) describing a new architecture style going by the name *microservices*. They didn't coin the term, but they certainly contributed to the popularity of the new style. And I suspect the authors did the industry a favor as well—their delineation of the characteristics of microservices quite early in their life cycle helped teams hone in on what *is* and *isn't* a microservice more quickly, avoiding months or years of churn trying to figure out their real uses.

Because they were describing a new phenomenon, Fowler and Lewis necessarily predicted a few things, including the impact microservices would have on user interface design. They observed that one of the defining features of microservices is the decoupling of services, and they predicted that architects would partition user interfaces to match the level of decoupling.

Alas, the real world interfered with their prediction...until now. It turns out that user interfaces are necessarily monolithic in nature: users expect to go to a single place to interact with an application, and they expect certain unified behaviors—all the parts of the user interface work in concert. While it is possible to create truly decoupled user

interfaces, this has proved challenging for developers, who have awaited proper framework and tool support.

Fortunately, that support has finally arrived. You hold in your hand the definitive guide to this important aspect of microservice development. Luca Mezzalira has done an excellent job describing the problem in clear terms, following up with cutting-edge support to solve common roadblocks.

This well-organized book begins by covering the frontend issues that developers currently face, then delves into the various aspects of micro-frontends. Luca provides not only technical details but also critical ecosystem perspectives, including how to untangle a monolith into a more decoupled user interface, and how common engineering practices such as continuous integration can fit into teams' use of this new technology.

Every developer who builds microservices, regardless of whether they build user interfaces, will benefit from this enjoyable guide to a critical subject.

Neal Ford, Director/Software Architect/Meme Wrangler at Thoughtworks, Inc.

Preface

At the beginning of December 2016, I took my first trip to Tokyo. It lasted just a week, but—as I would discover—I would travel to the Japanese capital many more times in the near future. I can clearly remember walking to the airplane at London Heathrow Airport and mentally preparing my to-do list for the impending 12-hour flight.

By then, I’d already been traveling for a few weeks in the United States: attending conferences in the San Francisco Bay Area and Las Vegas.

The project I was working on at that time was an over-the-top platform similar to Netflix but dedicated to sports, with daily live and on-demand content available in multiple countries and on more than 30 different devices (web, mobile, consoles, set-top boxes, and smart TVs). It was near the end of the year, and as a software architect, I had to make a proposal for a new architecture that would enable the company to scale to hundreds of developers distributed across multiple locations, without reducing the current throughput and ideally enhancing it as much as possible.

When I settled in my seat, I became relatively calm. I was still tired from the Vegas trip and admittedly a bit annoyed about the 12 hours I would have to spend on the plane, but I was excited to see Japan for the first time. A few minutes later, I had a glass of champagne. For the first time in my life, I was in business class, with a very comfortable seat and plenty of space to work.

At the end of the first hour, it was time to get my laptop out of my backpack and start working on “the big plan.” I still had over 10 hours of flight time during which I could make progress on this huge project that would serve millions of customers around the world. I didn’t know then that the following hours would deeply change the way I would architect cross-platform applications for the frontend.

In this book, I want to share my journey into the micro-frontend world: the lessons I learned, the tips for getting a solid micro-frontend architecture up and running, and (finally) the benefits and pitfalls of this approach. Hopefully, these lessons will enable you to evaluate whether this architecture fits your current or next project.

Now it’s time for your own journey to begin.

Why I Wrote This Book

I started thinking about micro-frontends in 2015. During the years that followed, I had the opportunity to implement them in a large-scale organization with distributed teams composed of hundreds of developers, and to explain their benefits as well as their pitfalls. During this time, I also shared these experiences at conferences, webinars, and meetups, which enabled me to engage with the community—listening to their stories, answering their questions, and collaborating with other companies that embraced this paradigm in different ways.

More recently, I suggested several of the practices presented in this book to enterprise organizations all over the world, from Australia to North America. I was exposed to multiple challenges during the design and implementation phases, and the lessons I learned are gathered in these pages as well.

This book represents my research, experiences, studies, and insights from the ground up, collected over several years of work with hundreds of teams worldwide. I want to share real examples and topics that I believe are key to succeeding with micro-frontends.

Finally, everyone who is interested in learning how to use this architecture style end to end—despite the inevitable evolution we are going to see in the next few years—will find a pragmatic guide that covers every aspect: from engineering culture and organizational structure, to best approaches for dealing with this architecture in production. What you are going to learn in these pages will act as your North Star for creating successful micro-frontend projects.

Who This Book Is For

This book is for developers, architects, and tech leaders who are looking to scale their organizations and frontend applications. It's a collection of mental models and experiences useful for approaching any micro-frontend architecture, and you will find the principles and solutions applied to every approach implemented so far. By following these practices, you will be able to complete a micro-frontend project with the right mindset and overcome common challenges that your teams are going to face during the journey.

In this book, you'll find technical architectures and implementation as well as end-to-end coverage of micro-frontends—from the design phase through to how to organize your teams for migrating existing or greenfield projects to micro-frontends.

How This Book Is Organized

The chapters in this book cover specific topics so a reader can jump from one to another without too many references across chapters. Although the best way to read this book is sequentially, it is also useful as a reference while working—so if you need to jump to a specific topic, you can pick the chapter and read just the part you are interested in.

The book covers the following:

Chapter 1, “Micro-Frontend Principles”

We begin by examining the foundational ideas behind microservices and how those concepts translate to frontend development. You will learn the core principles that serve as a “North Star” for micro-frontend implementations, principles that will guide every architectural decision you make.

Chapter 2, “Micro-Frontend Architectures and Challenges”

This chapter lays the groundwork for you to understand micro-frontends in depth. I introduce four key pillars for designing successful architectures, supported by a decision-making framework for identifying, composing, orchestrating, and communicating micro-frontend solutions. With these choices in place, you can design the rest of the system—from automation strategies to design systems—with confidence.

Chapter 3, “Discovering Micro-Frontend Architectures”

Micro-frontends can be implemented in many different ways. Here, I categorize and evaluate those approaches, exploring their benefits, trade-offs, and ideal use cases so you can choose the right fit for your context.

Chapter 4, “Client-Side Rendering Micro-Frontends”

This chapter offers a deep dive into best practices for building client-side rendered micro-frontends using common frameworks such as Module Federation.

Chapter 5, “Server-Side Rendering Micro-Frontends”

Server-side rendering (SSR) brings its own opportunities and complexities. We look at how to implement SSR micro-frontends effectively, including the infrastructure considerations that make this one of the most challenging approaches to get right.

Chapter 6, “Micro-Frontend Automation”

Successful micro-frontend architectures rely on solid automation. This chapter shares strategies for repository organization, continuous integration, and other automation fundamentals that keep teams moving quickly without sacrificing quality.

Chapter 7, “Discover and Deploy Micro-Frontends”

In production-grade systems, managing multi-environment deployments is critical. You will learn how to apply discovery patterns—as well as techniques like canary releases and blue-green deployments—to reduce risk and to build deployment confidence.

Chapter 8, “Automation Pipeline for Micro-Frontends: A Case Study”

Building on [Chapter 6, “Micro-Frontend Automation”](#) and [Chapter 7, “Discover and Deploy Micro-Frontends”](#), this case study walks through a real-world automation pipeline for micro-frontends, showing how theory translates into practice. These are insights you can apply immediately to your own continuous integration or continuous delivery (CI/CD) setups.

Chapter 9, “Backend Patterns For Micro-Frontends”

Frontends do not exist in isolation. In this chapter, we explore integration patterns for working with monolithic backends and microservices, including backend for frontend (BFF), API gateways, and service dictionaries, with examples and best practices for each.

Chapter 10, “Common Antipatterns in Micro-Frontend Implementations”

Not every pattern stands the test of time. Here, I share common antipatterns that I have encountered over the past five years, explain why they fail, and offer practical alternatives to avoid costly mistakes.

Chapter 11, “Migrating to Micro-Frontends”

Migrations are complex. Drawing on years of experience helping organizations transition, this chapter focuses on asking the right questions, identifying priorities, and tackling the critical aspects of moving from a monolithic frontend to micro-frontends.

Chapter 12, “From Monolith to Micro-Frontends: A Case Study”

Here, we follow the fictional ACME Inc. through its migration journey, exploring the technical and organizational decisions that shaped its move to a distributed frontend architecture.

Chapter 13, “Introducing Micro-Frontends in Your Organization”

Technology is only part of the story. This chapter addresses the organizational aspects of micro-frontends, including how to align teams, foster collaboration, and set up processes that support long-term success.

Chapter 14, “AI and Micro-Frontends: Augmenting, Not Replacing”

In the final chapter, I share a year of experimentation with AI in the micro-frontend space, highlighting where it shines, where it falls short, and how it can augment rather than replace your work. You will leave with a practical playbook for leveraging AI to accelerate micro-frontend development.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/lucamezzalira>.

If you have a technical question or a problem using the code examples, please send email to support@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Building Micro-Frontends*, 2nd Edition, by Luca Mezzalira (O’Reilly). Copyright 2026 Luca Mezzalira, 978-1-098-17078-3.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

<https://oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *<https://oreil.ly/building-micro-frontends-2e>*.

For news and information about our books and courses, visit *<https://oreilly.com>*.

Find us on LinkedIn: *<https://linkedin.com/company/oreilly-media>*

Watch us on YouTube: *<https://youtube.com/oreillymedia>*

Acknowledgments

First, I'd like to thank my family, Maela, and my daughters for everything they do and the strength I receive from them to move forward every single day. Thanks to all the people who inspire me on a daily basis through every form of communication.

A huge thank-you to DAZN, who allowed me to apply a micro-frontend architecture and explore its benefits end to end, and who trusted my ideas and judgment.

Thanks to O'Reilly for the opportunity to write about micro-frontends. In particular, thanks to Louise Corrigan and Angela Rufino for all the support they gave me during these months of writing and the constant feedback that improved the book. And thanks also to Erin Brenner, my fantastic editor who spent a considerable amount of time unwinding my thoughts and translating them into what you are about to read.

Finally, thanks to all the people who reviewed this manuscript and provided fundamental suggestions to improve the book, and to all the attendees of my talks and workshops who shared their experiences and challenges, which are probably now reflected in these pages.

Chapter 1. Micro-Frontend Principles

I remember working on many software projects at the beginning of my career, where small- or medium-size teams were developing a monolithic application. All the functionalities of a platform were available in a single artifact, and the product was produced during the development of the software and deployed to a web server.

When we have a monolith, we write a lot of code that should harmoniously work together. However, in my experience, we tend to preoptimize or even overengineer our application logic, more often than not. Abstracting reusable parts of our code can create a more complex codebase, and sometimes the effort of maintaining a complex logic doesn't pay off in the long run. Unfortunately, something that looked straightforward at the start could look very unwieldy a few months later.

In the past decades, public cloud providers like Amazon Web Services (AWS) or Google Cloud started to gain traction. Nowadays, they are popular for delegating what is increasingly becoming a commodity, freeing up organizations to focus on what really matters in a business: the services offered to the final users.

While cloud systems offer easier scalability compared to on-premises infrastructure, monolithic architectures require us to scale either horizontally—adding more containers or virtual machines—or vertically, increasing the configuration of the machine where our application is running. Furthermore, working on a monolith codebase with both distributed and co-located teams could be challenging as well. This is particularly the case when reaching medium or large team sizes, because of the communication overhead and centralized decisions where a few people decide for everyone.

In the long run, companies with large monoliths usually slow down all the operations needed to release any new feature, losing the great momentum they had at the beginning of a project where everything was easier and smaller, with few complications and risks.

A further complication with monolithic applications is that we have to test and deploy the entire codebase every single time. This comes with a higher chance of breaking APIs in production, introducing new bugs, and making mistakes, especially when the codebase is not rock-solid or extensively tested.

In seeking to solve these and many other challenges, a company might move from complex monolith codebases to multiple smaller codebases and scoped domains called *microservices*. Nowadays, microservices are a well-known and established

architecture pattern used by many organizations across the world. They divide a single codebase into smaller parts, each responsible for a subset of functionalities. This business logic is embraced by developers because each microservice tackles a smaller, simpler problem instead of requiring them to look at thousands of lines of monolithic code. Moreover, considering the reduced cognitive load, it is easier for a developer to maintain a clear picture of the codebase and its functionality. Another significant advantage is that we can scale individual parts of the application independently, using the most suitable approach for each microservice rather than the one-size-fits-all model of a monolith.

There are also some pitfalls to working with microservices. Significant investments in automation, observability, and monitoring are required to keep a distributed system under control. Another pitfall is poorly defining a microservice's boundary—for instance, creating a microservice that is too small to complete an action on its own and instead relies heavily on other microservices. This causes a strong coupling between services and forces them to be deployed together every time. When this phenomenon spreads to multiple services, we risk ending up with a system so complex that it is difficult to extend—like a big ball of mud!

So, while microservices bring many benefits to the table, they could bring many cons as well. In particular, when we are embracing them in a project, the complexity of maintaining microservice architecture could become more painful than beneficial. Considering the options available in software architecture, we should pick microservices only when needed—avoiding adopting them recklessly just because they're the latest and greatest approach.

On the frontend side, micro-frontends have gained more traction in the frontend community and enterprise organizations thanks to their strong alignment with other distributed architectures like microservices. A micro-frontend is an independently developed, tested, and deployed slice of a user interface representing a business domain. It can be combined with other micro-frontends at runtime to build a complete application, enabling teams to work autonomously and release features independently. Keep in mind, however, that just as microservices aren't a universal answer to all software decomposition, neither are micro-frontends. To understand where they fit in and what they are, let's look at some of the forces pushing us in this direction.

Monolith to Distributed Systems

When we start a new project—or even a new business offering a service online—the first iteration should be used to understand if the idea has the potential to succeed.

Usually, we start by identifying a tech stack—a list of tech services used to build and run a single app—that is familiar to our team. By minimizing all the unnecessary bells and whistles around the system and concentrating on the bare minimum, we’re able to gather information quickly about our business idea directly from our users. This is also called a *minimum viable product* (MVP).

Often, we design our API layer as a single codebase (monolith), which means we need to set up a single continuous integration or continuous delivery (CI/CD) pipeline for the project. Integrating observability into a monolith application is quite easy; we just need to run an agent per virtual machine or container to retrieve the health status of our application servers. The deployment process is simpler, as we only need to handle one automation strategy for the entire API layer, along with one deployment and release strategy. And when the traffic starts to increase, we can scale horizontally by adding as many application servers as needed to fulfill user requests.

That’s also why monolithic architecture is often a good choice for new projects: it lets us focus more on the business logic of an application rather than investing too much effort on complexities like automation.

Where are we going to store our data? We have to decide which database best suits our project needs—a graph, a NoSQL, or a SQL database? Another decision that must be made is whether we want to host our database on a cloud service or on-premises. We should select the database that will fit our business case better.

Finally, we need to choose a technology for representing our data, such as within a desktop or mobile browser, or even a mobile application. We can pick the best-known JavaScript framework available or our favorite programming language or we can decide to use server-side rendering or a single-page application (SPA) architecture. Then, we define our code conventions, linting, and CSS rules.

At the end, we should end up with something like what you can see in [Figure 1-1](#).

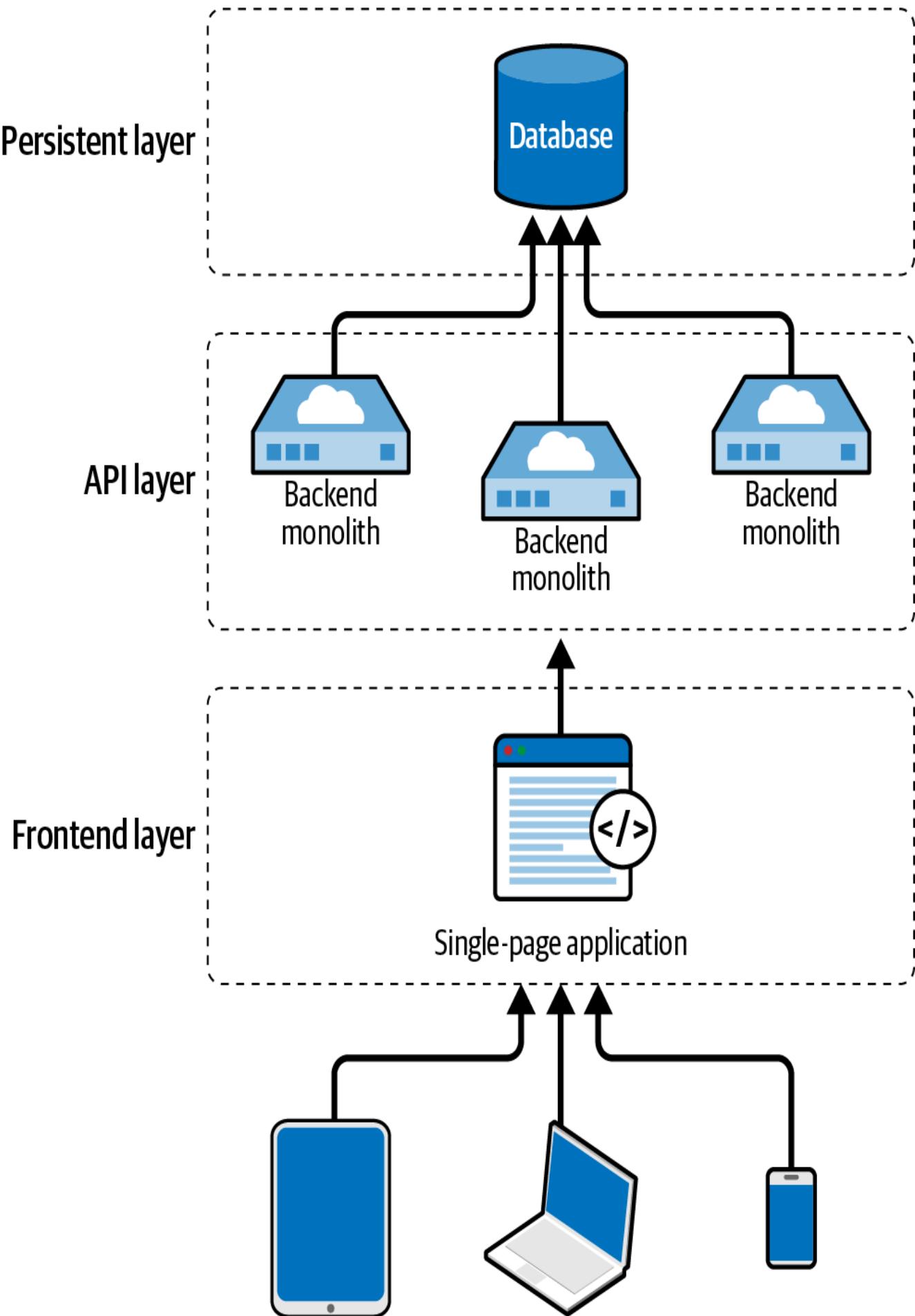


Figure 1-1. Three-tier architecture composed by a presentation layer (frontend), an application layer (APIs layer), and a persistent layer (database)

Hopefully, the business ideas and goals behind our project will be validated as more users subscribe to our online service or buy our products.

Moving to Microservices

Now imagine, thanks to the success of our system, that our business decides to scale up the tech team, hiring more engineers, quality assurance analysts, scrum masters, and so on.

While monitoring our logs and dashboards, we realize that only some of our APIs are scaling automatically. Some of them are highly cacheable, so the content delivery networks (CDNs) are serving the vast majority of clients. Our origin servers are under pressure only when our APIs are not cacheable. Luckily, only a small subset of API responses cannot be cached.

Splitting our monolith starts to make more sense at this point, given the internal growth and our improved understanding of how the system works.

Embracing microservices also means reviewing our database strategy and, therefore, having multiple databases that are not shared across microservices. If needed, our data is partially replicated so that each microservice reduces the latency for returning responses.

Suddenly, we are moving toward a decentralized ecosystem with many moving parts that are providing more agility and less risk than before. Each team is responsible for its set of microservices. Team members can make decisions on the best database to choose, the best way to structure the schemas, how to cache some information to make responses even faster, and which programming language to pick for the job. Basically, we are moving to a world where each team is entitled to make decisions and be responsible for the services they run in production, where a shared standard for the entire system is not needed apart from the key decisions, like logging and monitoring. We can see this in [Figure 1-2](#).

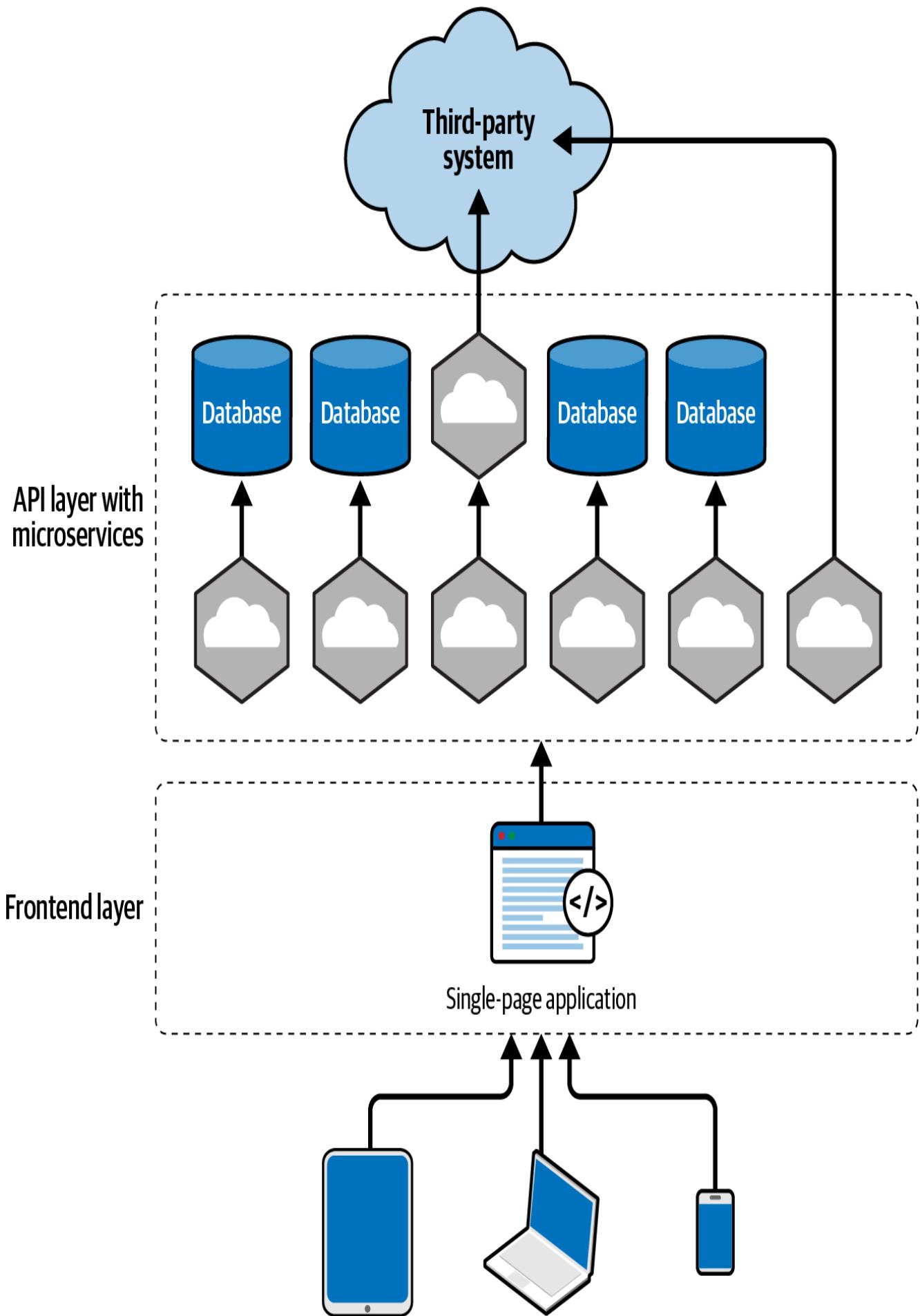


Figure 1-2. Microservices with SPA

However, we are still missing something here. We can scale our APIs layer and our persistent layers with well-defined patterns and best practices, but what happens when our business is growing and we need to scale our frontend teams, too?

Introducing Micro-Frontends

So far, on the frontend, we didn't have many options for scaling our applications. Up until a few years ago, there wasn't a strong need to do so because the standard approach was to have a fat server (where all the business logic runs) and a thin client (for displaying the result of the computation made available by the servers).

This has changed a lot in recent years. Our users now expect better experience when navigating our web platforms, including more interactivity and smoother interactions.

Companies have emerged that provide various subscription-based services, and many people are embracing them. Now, it's normal to watch videos on demand instead of on a linear channel, to listen to our favorite music through an application instead of buying CDs, and to order food from a mobile app instead of calling a restaurant.

This shift in behavior requires us to improve our users' experiences and provide a frictionless path for accomplishing what a user wants without forgetting about quality content and services.

In the past, we would have approached these problems by dividing parts of our application into a shared components library or abstracting some business logic into other libraries for reuse across different parts of the application. In general, we would have tried to reuse as much code as possible.

I'm not advocating against solutions that are still valid and fit perfectly with many projects, but we might encounter quite a few challenges when embracing them. For instance, when multiple development teams are involved, all the rules applied to the codebase are often decided once, and we stick with them for months—or even years—because changing a single decision would require substantial effort across the entire codebase, representing a large investment without providing any immediate value for the customers or the company.

Also, many decisions made during development could involve trade-offs due to lack of time, ideal consistency, or simply human laziness. We must consider that a business, like technology, evolves at a certain pace, and this is unavoidable.

Code abstraction is not a silver bullet either; prematurely abstracting code for reuse often causes more problems than benefits. I have frequently seen abstractions make code thousands of times more complicated than necessary in order to be reused just twice within the same project. Many developers are prone to overengineering solutions, thinking they will reuse them tens of times, but in reality, they are used far fewer times. Using libraries across multiple projects and teams can introduce more complexity than benefits, such as making the codebase harder to maintain, increasing manual testing effort, or adding communication overhead.

We also need to consider the monolith approach on the frontend. Such an approach limits our ability to improve our architecture over the long term, particularly for platforms intended to remain available for many years or for distributed teams working across time zones. Asking any business to extensively revise the tech it uses will cause a large investment upfront before it gets any results.

Now the question becomes quite obvious: do we have the opportunity to adopt a well-known pattern or architecture that enables us to add new features quickly, evolve with the business, and deliver parts of the application autonomously—without requiring big-bang releases?. I picture something like [Figure 1-3](#)

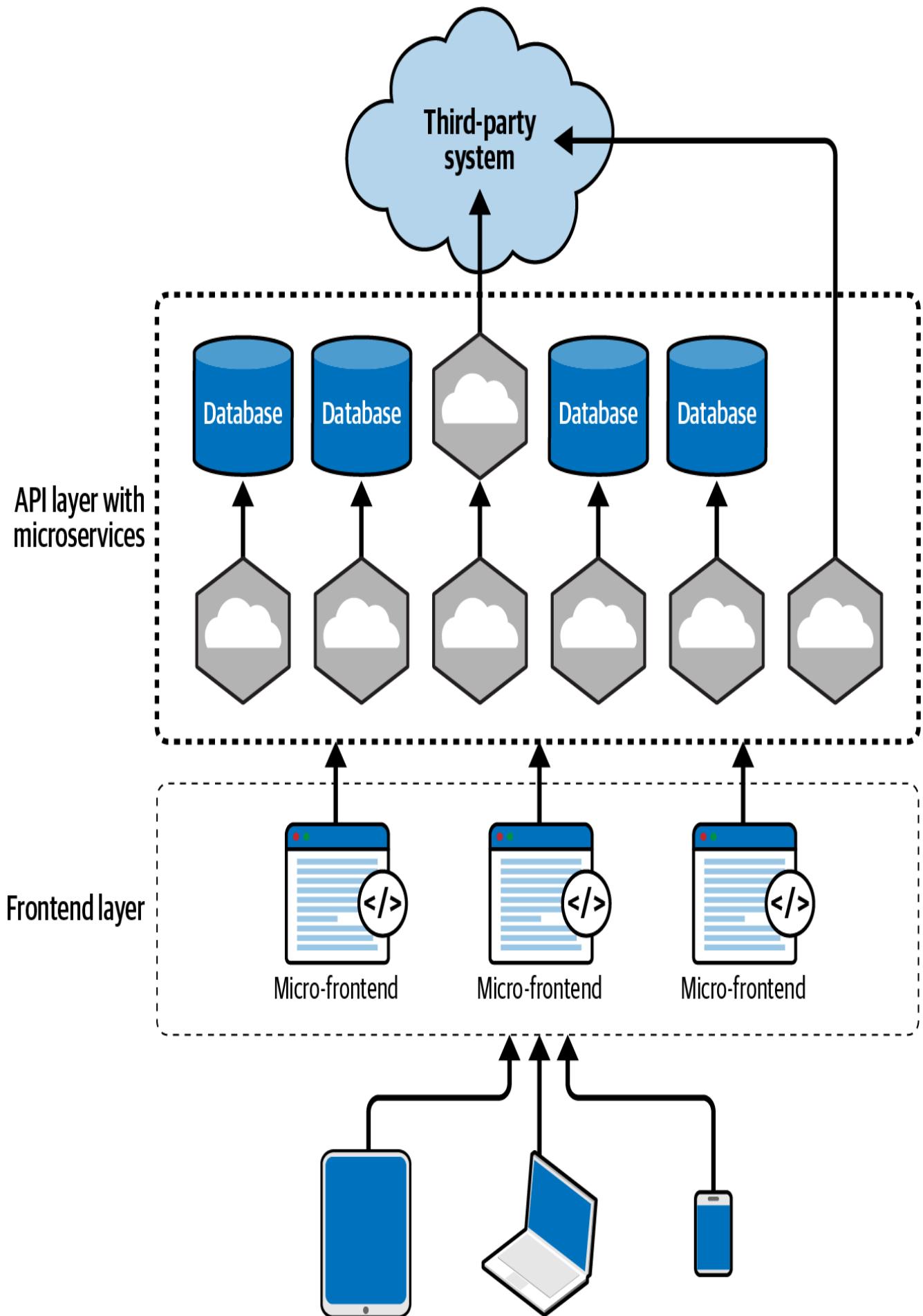


Figure 1-3. Microarchitectures combined: a high-level diagram showing how microservices and micro-frontends can live together

The answer is YES!

We can definitely do it, and it's where micro-frontends come to the rescue.

During the following chapters, we are going to explore how to successfully structure our micro-frontend architectures.

However, we first need to understand what the main principles are behind micro-frontends, so that we can use this as guidance during the development of our projects.

Microservices Principles

At the beginning of my journey into micro-frontends in 2016, there wasn't any guidance on how to structure such architecture. So, I had to take a step back from the technical implementation and look at the principles behind other architectures for scaling a software project. Would those principles be applicable to the frontend too?

The principles of microservices offer a few useful concepts. [Sam Newman has highlighted](#) these ideas in his book *Building Microservices*, and I've summarized the theories in [Figure 1-4](#).

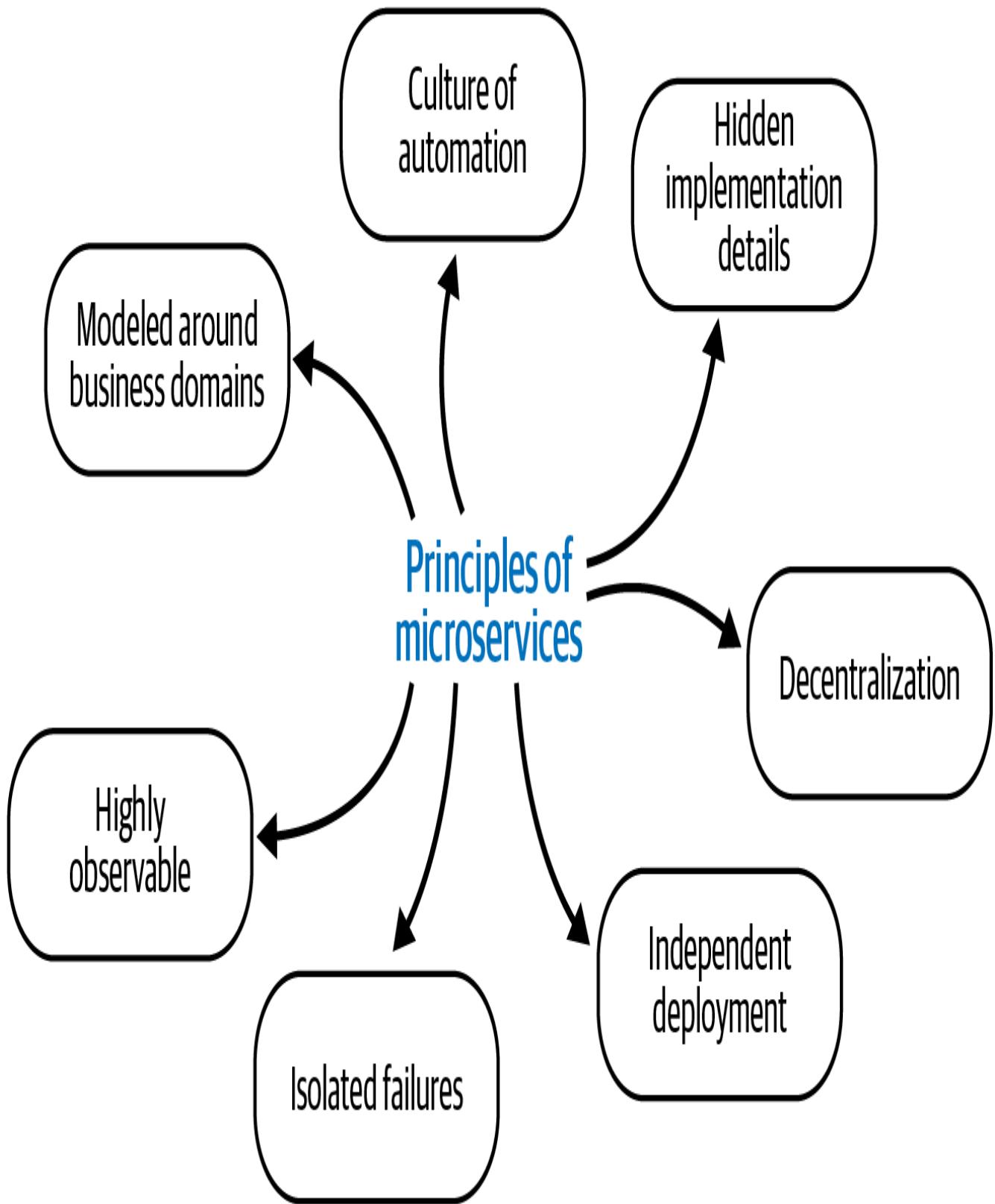


Figure 1-4. Principles of microservices

Let's discuss these principles and see how they apply to the frontend.

Modeled Around Business Domains

Modeling around business domains is a key principle of domain-driven design (DDD). It assumes that each piece of software should reflect what the organization does. We should design our architectures based on domains and subdomains, leveraging a ubiquitous language shared across the business.

When working with a business point of view, this provides several benefits, including a better understanding of the system, an easier definition of how to technically represent a business domain, and clear boundaries for a team to operate within.

As a practical tip, identify frontend domains based on user journey steps (e.g., check-out, search, profile) or clear business capabilities, rather than technical frameworks or component types. This prevents teams from splitting along technology lines instead of aligning with business needs.

Culture of Automation

Considering that microservices consist of a multitude of services that should be autonomous, we need a robust culture of automating the deployment of independent units across different environments. In my experience, this is a key process for leveraging microservices architecture effectively. A strong automation culture enables us to move faster and provide a better feedback loop for developers, while delivering shared capabilities such as security guardrails, compliance checks, and architecture standards that are built into the continuous integration process.

Hidden Implementation Details

Hiding implementation details when releasing autonomously is crucial. If we are sharing a database between microservices, we won't be able to change the database schema without affecting all the services relying on the original schema. DDD teaches us to encapsulate services inside the same business domain, exposing only what is needed via APIs and hiding the rest of the implementation. This enables us to change internal logic at our own pace without impacting the rest of the system. Very often, this approach is called API-first. We begin by defining APIs as the contract binding the producer and consumer(s). This enables them to work in parallel, focusing on either producing or consuming. By focusing on the API early during the development process, teams can enhance collaboration, scalability, and adaptability, making it easier to integrate and extend functionalities as the project evolves.

Decentralization

Decentralizing governance empowers developers to make the right decisions at the right stage to solve a problem. With a monolith, many key decisions are often made by the most experienced people in the organization. These decisions, however, frequently lead to trade-offs alongside the software life cycle. Decentralizing these decisions may have a positive impact across the entire system by allowing a team to take a technical direction based on the problems they are facing, instead of creating compromises for the entire system. Bear in mind that in distributed systems, each team carries less cognitive load, so members quickly become domain experts and can make the best decisions to evolve their portion of the system.

Independent Deployment

Independent deployment is key for microservices. With monoliths, we are used to deploying the entire system every time, which increases the risk of live issues and makes deployments and rollbacks slower. With microservices, however, we can deploy autonomously without raising the possibility of breaking our entire API layer. Furthermore, solid techniques like blue-green deployment and canary releases enable us to release a new version of a microservice with even less risk, which clears the path for new or updated APIs.

Isolated Failures

Because we are splitting a monolith into tens (if not hundreds) of services, even if some microservices become unreachable due to network issues or service failures, the rest of the system should remain available for our users. There are several patterns that support graceful microservices failures, and the autonomy of microservices just reinforces the principle of isolating failure.

Highly Observable

One reason that you might favor monolithic architecture over microservices is that it's easier to observe a single system than a system split into multiple services.

Microservices provide a lot of freedom and flexibility, but this doesn't come without consequence; we need to keep an eye on everything through logs, monitors, and so on. For example, we must be ready to follow a specific client request end to end inside our system. Keeping the system highly observable is one of the main challenges of microservices.

Embracing these principles in a microservices environment will require a shift in mindset not only for your software architecture but also for how your company is organized. It involves moving from a centralized to a decentralized paradigm, enabling cross-functional teams to own their business domains end to end. This can be a particularly significant challenge for medium-to-large organizations.

Applying Principles to Micro-Frontends

Now that we've grasped the principles behind microservices, let's find out how to apply them to a frontend application.

Modeled Around Business Domains

Modeling micro-frontends to follow DDD principles is not only possible but also very valuable. Investing time at the beginning of a project to identify the different business domains and how to divide the application will be very useful when you add new functionalities or deviate from the initial project vision in the future. DDD can provide a clear direction for managing backend projects, but we can also apply some of these techniques to the frontend. Granting teams full ownership of their business domain can be very powerful, especially when product teams are empowered to work with technology teams.

The primary difference between a micro-frontend and a component lies in their modularization approach. A micro-frontend completely owns a business domain, whereas a component focuses on addressing a technical challenge, often characterized by code duplication or the creation of complex, configurable components used across multiple domains. The component approach exposes an API that is frequently coupled with its container. Therefore, any modification made to the component is likely to impact its containers as well, creating an unwanted coupling that prevents it from aligning with the principles behind distributed systems.

With micro-frontends, we streamline the API surface to the essential minimum required for comprehending the user's context. Typically, micro-frontends require little beyond access to a session token and other pertinent information such as a product ID. This approach effectively diminishes the coupling between elements of the frontend application and enhances team autonomy by reducing the need for coordination across teams, owing to the infrequent changes in the minimal API exposed.

Culture of Automation

As for the microservices architecture, we cannot afford to have a poor automation culture inside our organization; if we do, any micro-frontend approach we take will end up being a pure nightmare for all our teams. Considering that every project contains tens or hundreds of different parts, we must ensure that our continuous integration and deployment pipelines are solid and have a fast feedback loop to support this architecture. Investing time in getting our automation right will result in the smooth adoption of micro-frontends and help solve common challenges like aligning shared libraries to a specific version, enforcing budget size per micro-frontend, or requiring updates of every micro-frontend to the latest design system version. Moreover, automation is not just important for generating technical artifacts; more significantly, it provides a fast feedback loop for developers, which will foster the right behaviors within teams and enforce important architecture characteristics across the distributed system.

Hidden Implementation Details

Hiding implementation details and working with contracts are two essential practices, especially when parts of our application need to communicate with each other. It's crucial to define an API contract from early on, and to share that across the teams who need to interact with different micro-frontends. Also, strong encapsulation is required to avoid domain leaks in other parts of the application. In this way, each team can change the implementation details without impacting other teams unless there is an API contract change. These practices enable a team to focus on the internal implementation details without disrupting the work of other teams. Each team can work at its own pace, drastically reducing external dependencies while creating more effective collaboration.

Decentralization

Decentralizing a team's decisions finally moves us away from a one-size-fits-all approach that often ends up as the lowest common denominator. Instead, the team can use the right approach or tool for the job. As with microservices, the team is in the best position to make certain decisions when it becomes an expert in the business domain. This doesn't mean that each team should take its own direction, but rather that the tech leadership (architects, principal engineers, CTOs), in conjunction with the developers and practices applied in the field, should provide guardrails within which teams can operate without needing to wait for a central decision. This leads to an organizational sharing culture, which is essential for introducing successful practices across teams.

Independent Deployment

Micro-frontends enable teams to deploy independent artifacts at their own speed; they don't need to wait for external dependencies to be resolved before deploying.

Achieving independence in micro-frontends means not treating the user interface as just components. We need to reduce the external dependencies for a team, and in this way, we optimize for a fast flow that enables a team to run its operations independently.

When we combine this approach with microservices, a team can own a business domain end to end, with the ability to make technical decisions based on the challenges within their business domain rather than defaulting to a one-size-fits-all approach.

Isolated Failures

Isolating failure in SPAs, for instance, isn't a huge problem due to their architecture, but it is a problem with micro-frontends. In fact, micro-frontends require composing a user interface at runtime, which may result in network failures or 404 errors for one or more parts of the UI. To avoid impacting the user experience, we must provide alternative content or hide a specific part of the application. This might mean gracefully hiding non-essential micro-frontends from the interface if they fail or return a 500 error, if the main micro-frontend of a page is not loaded.

Highly Observable

Frontend observability is becoming more prominent every day, with tools like Sentry, New Relic, or LogRocket providing great visibility for every developer. Using these tools is essential for understanding where our application is failing and why. As Werner Vogels, Amazon's CTO, famously says: "Everything fails, all the time." Therefore, being able to resolve issues quickly is far more important than preventing problems. This moves us toward a paradigm where we can better invest our resources by being ready to address system failures rather than trying to prevent them completely. As with all principles of microservices, this is also applicable to the frontend.

The exciting part of recognizing these principles on the frontend and backend is that, finally, we have a solution that will empower our development teams to own the entire range of a business domain. This offers a simpler way to divide labor across the organization and iterate improvements swiftly into our system.

When we start this journey into the *micro-world*, we need to be conscious of the level of complexity we are adding to a project, which may not be required for any other

projects.

There are plenty of companies that prefer using a monolith over microservices because of the intrinsic complexity that microservices bring to the table. For the same reason, we must understand when and how to use micro-frontends properly, as not all projects are suitable for them.

Challenges Unique to Micro-Frontends

While many principles of microservices map well to frontend architectures, micro-frontends introduce distinct challenges that require careful architectural decisions:

Shared state

Shared state is a common source of complexity. Managing user sessions and data shared across independently deployed micro-frontends—such as authentication tokens or shopping cart contents—demands strong boundaries and disciplined communication patterns. Without this, tight coupling can quickly undermine the benefits of modularity.

Routing

Routing presents another challenge. Coordinating navigation across micro-frontends, especially when combining global and local routing responsibilities, can lead to fragmented logic and brittle integrations if not handled systematically.

Performance monitoring

Performance must be closely monitored, too. Composing multiple micro-frontends at runtime increases the risk of inflated bundle sizes, additional network requests, and degraded user experience. Optimizations like smart caching, lazy-loading, and shared dependency management are critical to mitigating these effects.

UX consistency

UX consistency can also suffer as independent teams evolve their micro-frontends in isolation. Without a shared design system and strong visual governance, subtle inconsistencies in layout, styling,

and behavior may accumulate, leading to a disjointed user experience.

Addressing these challenges requires a combination of well-defined API and UI contracts, shared platform guidelines, and robust automation. Micro-frontends demand not just technical boundaries, but also strong organizational alignment to succeed at scale.

Micro-Frontends Are Not a Silver Bullet

It's very important that we use the right tool for the right job. I cannot stress this point enough. Too often I have seen projects failing or drastically delayed due to poor architectural decisions.

NOTE

We need to remember that *micro-frontends are not appropriate for every application*. This is because of their nature and the potential complexity they add at the technical and organizational levels.

Micro-frontends are a sensible option in the following circumstances:

- When we are working on software that requires an iterative approach and long-term maintenance
- When we have projects that require a large development team
- In multi-tenant applications
- When we want to replace a legacy project in an iterative way

However, they are not suitable for all frontend applications. They are an additional available option of frontend architecture for our projects. Micro-frontend architecture has plenty of benefits but also plenty of drawbacks and challenges. If the latter exceed the former, micro-frontends are not the right approach for a project.

As [Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani](#) have described in their book *Software Architecture: The Hard Parts*, “Don’t try to find the best design in software architecture; instead, strive for the least worst combination of trade-offs.” This should be your motto from now on!

Summary

In this chapter, we introduced what micro-frontends are, what their principles are, and how those principles are linked to an architecture like microservices that was created for solving similar challenges.

Next, we will explore how to structure a micro-frontend project from an architectural point of view and the key technical challenges to understand when we design our frontend applications using them.

Chapter 2. Micro-Frontend Architectures and Challenges

A micro-frontend represents a business subdomain that is autonomous, independently deliverable, built with the same or different technologies, has a low degree of coupling, and is owned by a single team.

Various types of micro-frontends offer many opportunities, and selecting the right type depends on the project requirements, organizational structure, and developers' experience.

In these architectures, we face specific challenges framed around questions like how we communicate between micro-frontends, how we route the user from one view to another, and (most importantly) how we identify the size of a micro-frontend.

In this chapter, we will cover the key decisions in starting a project with micro-frontend architecture. We'll then discuss some of the companies that are using micro-frontends in production and their approaches.

Micro-Frontend Decision Framework

There are different approaches for architecting a micro-frontend application. To choose the best approach for our project, we first need to understand the context in which we'll be operating.

Some architectural decisions must be made upfront because they will guide future decisions—such as how to define a micro-frontend, orchestrate the different views, and compose the final view for the user, as well as how micro-frontends will communicate and share data.

These types of decisions exist within the *micro-frontend decision framework*, which is composed of four key areas:

- Defining what a micro-frontend is in your architecture
- Composing micro-frontends
- Routing micro-frontends

- Communicating between micro-frontends

Defining Micro-Frontends

Let's start with the first key decision, which will have a significant impact on the rest. We need to define what constitutes a micro-frontend from a technical point of view.

We can decide to have multiple micro-frontends in the same view or only one micro-frontend per view. [Figure 2-1](#) is an example of how you can split your user interface in micro-frontends inside a system.

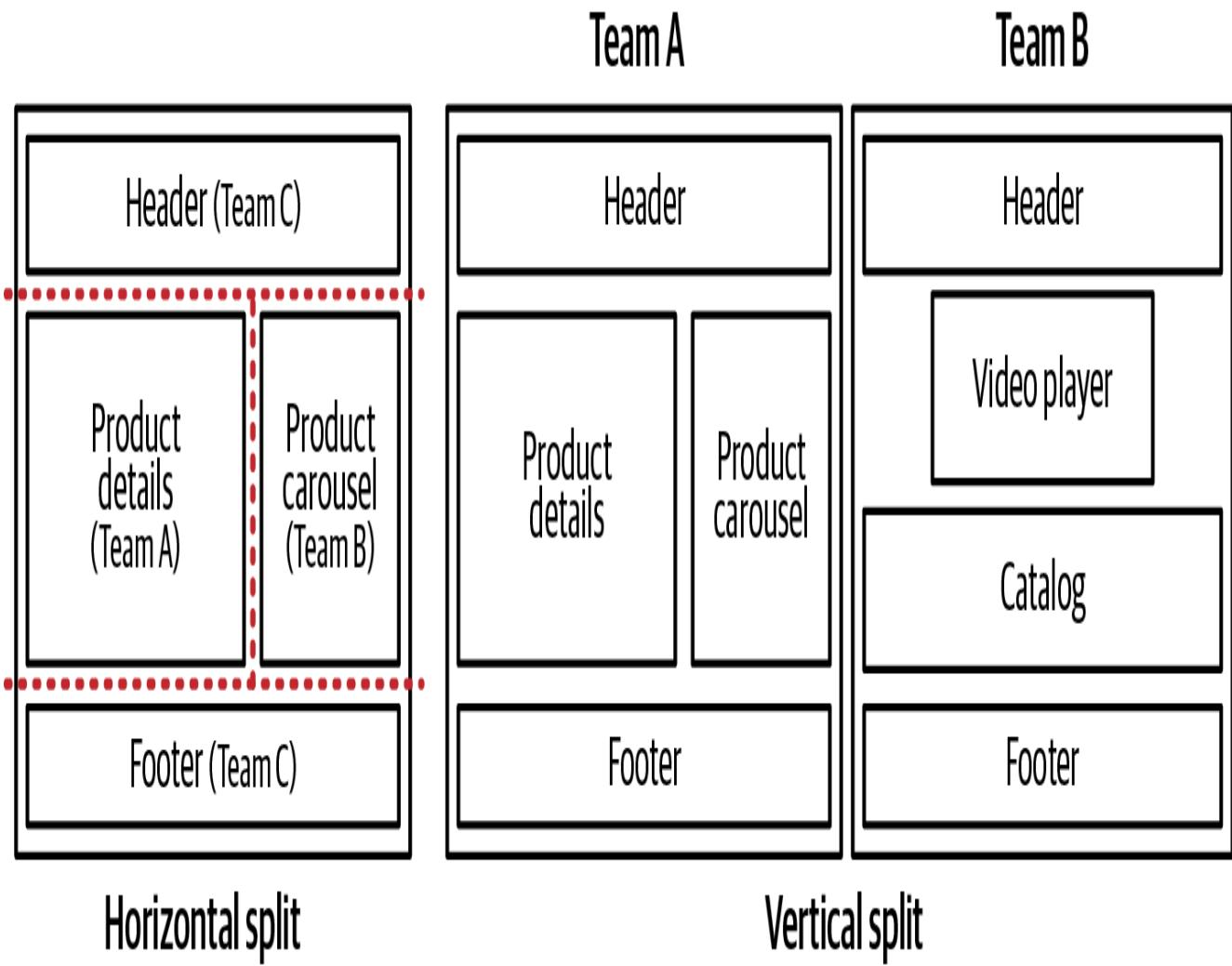


Figure 2-1. Horizontal versus vertical split

These approaches are not mutually exclusive within an application. Some domains can be split into multiple micro-frontends in certain views, while others can be represented by a single micro-frontend.

With the horizontal split, multiple micro-frontends will be on the same view. Multiple teams will be responsible for different parts of the view and will need to coordinate their efforts. This approach provides greater flexibility, as we can even reuse some micro-frontends in different views. However, it also requires more discipline and governance to avoid having hundreds of micro-frontends in the same project. Very often, higher granularity leads to higher coupling and increases the risk of creating a distributed monolith.

DISTRIBUTED MONOLITH

A distributed monolith in software architecture refers to a system that, despite being distributed across multiple servers or nodes, exhibits characteristics commonly associated with a monolithic architecture. In this context, the term “monolith” refers to a single, tightly coupled unit with interconnected components that lack clear separation of concerns. The distributed nature of the system may introduce complexities in terms of communication between components spread across different locations, but the overall structure remains monolithic in its design and interdependencies. This can hinder the independent nature of micro-frontends, risking the accumulation of several external dependencies that will nullify the effort of building such architecture.

In the vertical-split scenario, each team is responsible for a business domain, like authentication or catalog experience. In this case, domain-driven design (DDD) comes to the rescue. It’s not often that we apply DDD principles to frontend architectures, but in this case, we have a good reason to explore it.

DDD is an approach to software development that centers on programming a domain model with a rich understanding of the processes and rules within a specific domain.

Applying DDD to the frontend is slightly different from the approach taken on the backend. Certain concepts are not applicable, while others are fundamental for designing a successful micro-frontend architecture.

When examining the system holistically, you might wonder how to identify different areas that are independent. Various techniques exist, but one of my favorites by far is event storming, as shown in [Figure 2-2](#).



Figure 2-2. An example of an event storming outcome for the onboarding experience of a subscription service

Event storming is a workshop-based method that brings together individuals from across the same company with different backgrounds, such as product managers, testers, and developers. The workshop focuses on the business perspective, rather than the technical details.

By bringing people from different roles into the same room, you can create a timeline that maps the system end to end. This collaborative approach helps to identify potential independent areas of the system by examining the vocabulary and interactions captured during the session.

Thanks to this workshop—applicable to the entire system, not just the frontend—you can visualize your system architecture. This makes it easier to understand and, more importantly, to recognize the distinct components—or, as DDD would call them: subdomains.

In the context of DDD, subdomains refer to distinct and isolated components within a larger business domain. Each subdomain represents a specialized area with its own unique set of responsibilities, business logic, and models. The purpose of identifying subdomains is to facilitate a modular and organized approach to software development,

enabling teams to focus on specific aspects of the overall business functionality. Subdomains are delineated based on clear and cohesive boundaries, enabling more effective management, development, and maintenance of complex systems by addressing individual business concerns in a targeted manner.

EVENT STORMING

It is beyond the scope of this book to engage further with event storming. However, I highly encourage you to read the chapter on this subject in *Learning Domain-Driven Design* by Vlad Khononov.

DDD refers to three subdomain types. The following are concrete examples for you to understand better what they refer to:

Core subdomains

These are the primary reasons an application exists. Core subdomains should be treated as first-class citizens in our organizations because they deliver the most value. For example, the video catalog is a core subdomain for Netflix.

Supporting subdomains

These subdomains are related to the core ones but are not key differentiators. They support the core subdomains but aren't essential for delivering real value to users. For instance, Netflix's voting system for videos is a supporting subdomain.

Generic subdomains

These subdomains are used to complete the platform. Often, companies decide to go with off-the-shelf software because it's not strictly related to their domain. With Netflix, for instance, payment management is not related to the core subdomain (the catalog), but it is a key part of the platform because it has access to the authenticated section.

Table 2-1 breaks down this Netflix example into these categories.

Table 2-1. Subdomain examples

Subdomain type	Example
Core subdomain	Catalog
Supportive subdomain	Voting system
Generic subdomain	Sign-in or sign-up

Why categorize subdomains, you may wonder? The answer is straightforward: categorizing subdomains enables you to apply different characteristics and strategies to each one.

For instance, a core domain is the essence behind your system's functionality. Therefore, investing in developer seniority, code quality, and a fast feedback loop will likely yield the best outcomes.

A generic domain, on the other hand, lacks a competitive advantage. However, in such cases, opting for an off-the-shelf solution integrated into your system may suffice for achieving your objectives. Changes in this part of the system won't be as frequent as in other areas, and the complexity of writing the code is likely lower. Therefore, you can adopt a different strategy for assembling a development team compared to other subdomains.

In essence, DDD offers more than just a rich vocabulary for system description. It introduces heuristics and techniques that guide organizations in the right direction, helping to align both technology and organizational structure from the start.

DDD with Micro-Frontends

After identifying subdomains, DDD introduces another concept: the *bounded context*. This is a logical boundary that hides the implementation details, exposing an API contract to consume data from the model.

Usually, the bounded context translates the business areas defined by domains and subdomains into logical areas where we determine the model, our code structure, and potentially our teams. A bounded context also defines how different contexts

communicate with each other by creating a contract between them—often represented by APIs. This enables teams to work simultaneously on different subdomains while respecting the contract defined upfront.

In a new project, subdomains often overlap with bounded contexts because we have the freedom to design our system in the best way possible. Therefore, we can assign a specific subdomain to a team to deliver a certain business value while defining the contract that connects it to other parts of the system. However, in legacy software, these lines may be more blurred due to insufficient analysis during the project life cycle.

Too often, we identify a technical solution early on without first gathering the architecture characteristics we need to optimize for. Think about this scenario: three teams, distributed across three different locations, working on the same codebase. These teams may choose a horizontal split using iframes or web components for their micro-frontends. After a while, they realize that micro-frontends that are in the same view must communicate somehow. One of those teams will then be responsible for aggregating the different parts inside the view. The team will spend more time aggregating different micro-frontends in the same view and debugging to ensure everything works properly.

Obviously, this is an oversimplification. It could be worse when taking into consideration the different time zones, cross-dependencies between teams, knowledge sharing, or distributed team structure, for example.

All those challenges could escalate very easily, leading to low morale and frustration on top of delivery delays. Therefore, we need to be sure that the path we take won't let our teams down.

Approaching the project from a business point of view, however, enables you to create an independent micro-frontend with less need for communication across multiple subdomains.

Let's reimagine our scenario. Instead of working with web components or iframes, we are working with single-page applications (SPAs).

This approach enables a full team to design all the APIs needed to compose a view and to create the infrastructure required to scale services according to traffic. The combination of microarchitectures, microservices, and micro-frontends provides independent delivery without the high risk of compromising the entire system when releasing to production.

The bounded context helps design our systems, but we need to have a good understanding of how the business works to identify the right boundaries inside our project.

Developers, tech leads, or architects need to work more closely with the product teams—investing time with them and understanding customer needs—so they can identify the different domains and subdomains while working collaboratively. Once again, event storming could be a natural fit in these cases.

After defining all the bounded contexts, we will have a map of our system representing its different areas. In [Figure 2-3](#), we can see a representation of a bounded context. In this example, the bounded context contains the catalog micro-frontends that consume APIs from a microservices architecture via a unique entry point: a backend for frontend (BFF). We will explore API integration in more detail in [Chapter 9](#).

In DDD, the frontend is not explicitly considered, but when we work with micro-frontends using a vertical split, we can easily map the frontend and the backend together inside the same bounded context.

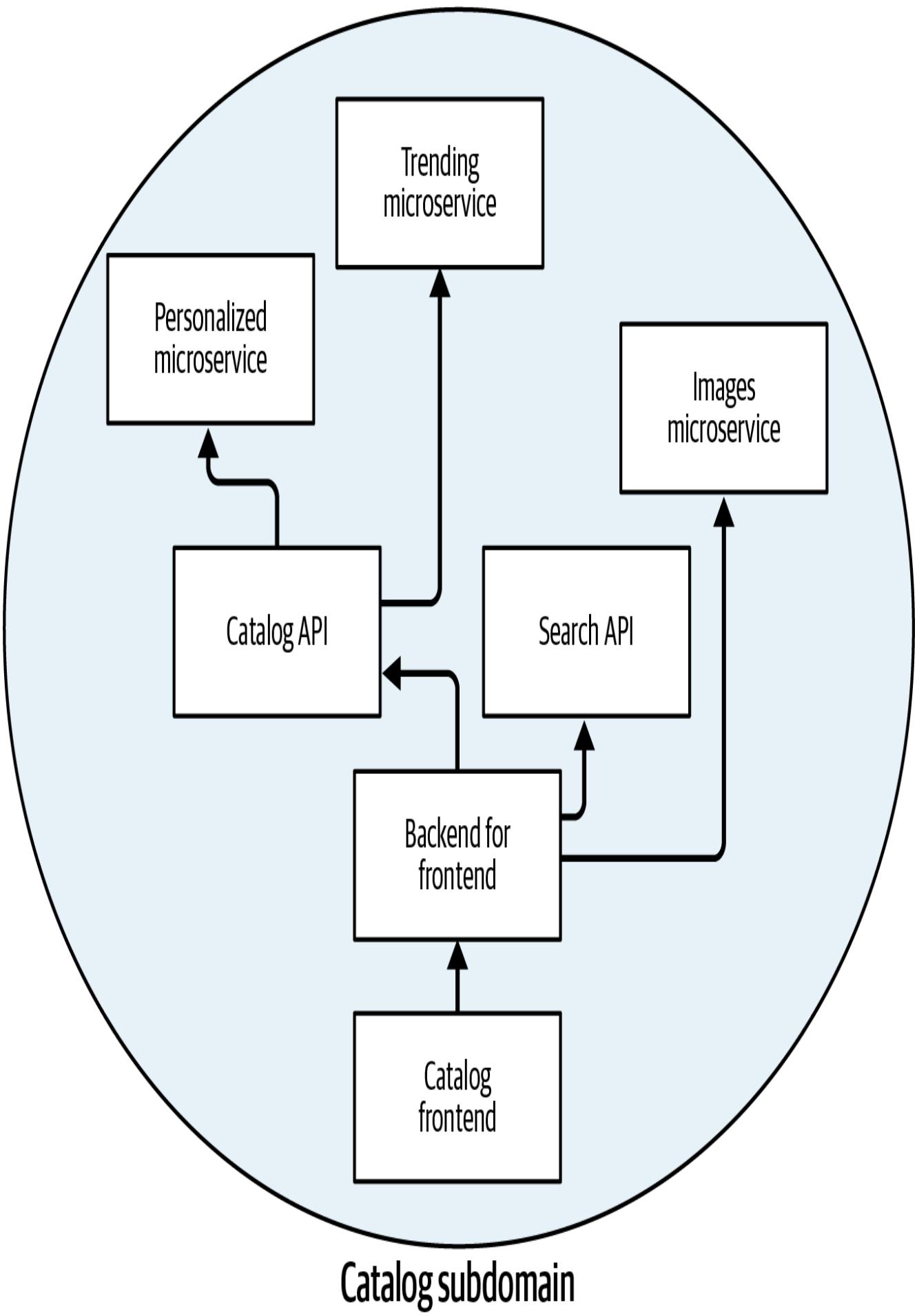


Figure 2-3. Representation of a bounded context

I've often seen companies design systems based on their team structure.¹ Instead, team structures should be flexible enough to adapt to the best possible solution for the organization in order to reduce friction and move faster toward the final goal: delivering a great product that satisfies customers.²

Both approaches to structuring your teams and designing system architecture are acceptable, as long as the coupling is recognized between the organizational structure and the software architecture. Very often, a change in one of these two areas will affect the other indirectly.

How to Define a Bounded Context

Premature optimization is always around the corner, and can tempt us to split bounded contexts too early to accommodate future integrations. Instead, we need to wait until we have enough information to make an educated decision.

Because our business evolves over time, we need to review our decisions about bounded contexts and subdomain types as well.

Sometimes we start with a larger bounded context. Over time, the business evolves and, eventually, the bounded context becomes unmanageable or too complex. At that point, we may decide to split it. This decision could result in a large code refactor, but it could also simplify the codebase drastically, speeding up new functionalities in the future.

To avoid premature decomposition, we should make the decision at the last possible moment. This way, we have more information and clarity on which direction we need to follow. We should engage upfront with the product team or the domain experts inside our organization as we define the subdomains. They can provide the context of where the system operates.

Always begin with data and metrics. For instance, we can easily find out how our users are interacting with our application and what the user journey looks like when a user is authenticated and when they're not. Data provides powerful clarity when identifying a subdomain and can help create an initial baseline from which we can see if we are improving the system or not.

If there isn't much observability within our system, we should invest time in creating it. Doing so will pay off the moment we start identifying our micro-frontends.

Without dashboards and metrics, we are blind to how our users operate within our applications.

Let's assume we see a huge amount of traffic on the landing page, with 70% of those users moving into the authentication journey (login, sign-up, payment, etc.). From here, only 40% of those users subscribe to a service or use their credentials to access it.

These are good indications of our users' behaviors on our platform. Following DDD, we would start with our application's domain model, identifying the subdomains and their related bounded contexts, and using behavioral data to guide us in how to slice the frontend applications.

Enabling users to download only the code related to the landing page will give them a faster experience because they won't have to download the entire application immediately, and the 40% of users who don't move forward to the authentication area will have just enough code downloaded to understand our service.

Obviously, mobile devices with slow connections benefit most from this approach for multiple reasons: less data is downloaded, less memory is used, and less JavaScript is parsed and executed, resulting in a faster first interaction with the page.

It's important to remember that not all user sessions contain all the URLs exposed by our platform. Therefore, a bit of research upfront will help us to provide a better user experience.

Usually, the decision to pick a horizontal split instead of a vertical split depends on the type of project being built. In the next chapter, we will dive deeper into this topic. As noted earlier in this chapter, horizontal and vertical splits are not mutually exclusive. You might have parts of the application where a vertical split is more appropriate than a horizontal one, and vice versa.

Another thing to consider is the skill set of our teams. Typically, a vertical split is better suited to teams that are new to micro-frontends. The horizontal split requires an upfront investment to create a solid and fast development experience, enabling teams to test their parts individually as well as within the overall view.

Testing Your Micro-Frontend Boundaries

Often, I've conducted meetings with teams that have implemented micro-frontend architecture but treated a micro-frontend as if it were a component loaded at runtime. I

have developed a mental model that can assist you in determining whether your boundaries are well-established:

- To enhance the robustness of your architecture, consider reducing the API surface exposed to the containers. When you expose too many properties of a micro-frontend, the risk of coupling increases significantly. This is because you allow the container of the micro-frontend to take ownership of the context instead of the micro-frontend itself. This leads to accidental complexity that becomes evident when deploying a micro-frontend and constant coordination efforts across teams.
- Micro-frontends are inherently context aware. Typically, a micro-frontend requires a minimal amount of information to function properly. They are designed with awareness of the context. For example, passing a product ID or enabling the retrieval of a session token to consume an API are common characteristics of the horizontal-split approach. Sharing more common properties from the micro-frontend container should lead you to question the implementation and revisit the API contract or the micro-frontend boundaries.
- In contrast to components, micro-frontends are less extensible. When designing a component, the focus is on high reusability and code abstraction. Micro-frontends, however, are designed for independence. Due to their context-aware nature, they are less likely to be extensible or integrated with each other. A sign of incorrect boundaries is a proliferation of micro-frontends per view or deep nesting between micro-frontends.
- Furthermore, micro-frontends are more coarse-grained than components. While a classic component, such as a button, is small and highly flexible for composition with other components, micro-frontends are highly specialized in their functionality. This specialization limits their reusability and makes it difficult to combine them into larger micro-frontends. It is recommended to avoid fine-grained micro-frontends, as they tend to increase coupling, create external dependencies, and can lead to context leakage toward their containers.

Having gained a comprehensive understanding of micro-frontends and their identification, let's now delve into the robust mental models widely embraced within the frontend community: the micro-frontend decision framework.

Micro-Frontend Composition

There are different approaches for composing a micro-frontend application, as outlined in [Figure 2-4](#).

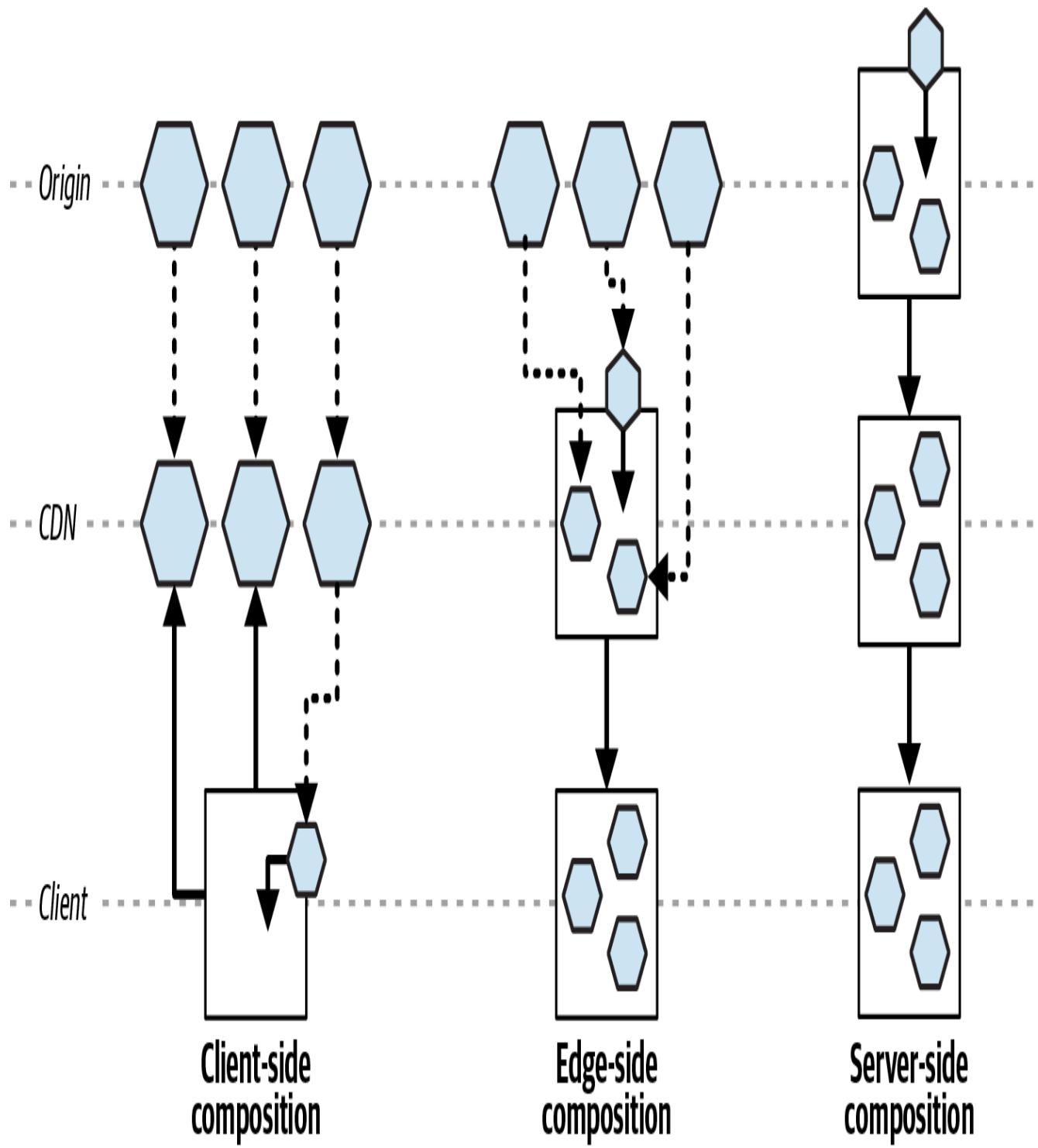


Figure 2-4. Micro-frontend composition diagram

In this diagram, we can see three different approaches to compose a micro-frontend architecture:

1. Client-side composition

2. Edge-side composition
3. Server-side composition

Starting from the left of our diagram, we have a client-side composition, where an application shell loads multiple micro-frontends directly from a content delivery network (CDN)—or from the origin if the micro-frontend is not yet cached at the CDN level. This composition is beneficial for both horizontal- and vertical-split micro-frontends. In the middle of the diagram, we compose the final view at the CDN level, retrieving our micro-frontends from the origin and delivering the final result to the client. The right side of the diagram shows a micro-frontend composition at the origin level where our micro-frontends are composed inside a view, cached at the CDN level, and finally served to the client. For edge-side and server-side composition, we mainly use a horizontal-split approach.

Let's now observe how we can technically implement this architecture.

Client-Side Composition

In the client-side composition case—where an application shell loads micro-frontends inside itself—the micro-frontends should have a JavaScript or HTML file as an entry point. This is necessary for the application shell to dynamically append the Document Object Model (DOM) nodes or to initialize the JavaScript application with a JavaScript file or an ECMAScript module.

In the beginning, we also used a combination of iframes to load different micro-frontends, or a transclusion mechanism on the client side via a technique called client-side include. This technique works by lazy-loading components and substituting empty placeholder tags with complex components. For example, a library called *h-include* uses placeholder tags that create an AJAX request to a URL and replace the element's inner HTML with the response of the request.

This approach gives us many options, but using client-side includes has a different effect than using iframes. From 2019 onward, more micro-frontend solutions began gaining traction for building successful client-side composition, such as Module Federation or single-spa. In the next chapters, we will explore this topic in detail.

NOTE

According to [Wikipedia](#), in computer science, *transclusion* is the inclusion of part or all of an electronic document into one or more other documents by hypertext reference. Transclusion is usually performed when the referencing document is displayed, and it's normally automatic and transparent to the end user. The result of transclusion is a single integrated document made of parts assembled dynamically from separate sources, possibly stored on different computers in disparate places.

An example of transclusion is the placement of images in HTML. The server asks the client to load a resource at a particular location and insert it into a particular part of the DOM.

Edge-Side Composition

With edge-side composition, we assemble the view at the CDN level. Many CDN providers give us the option of using an XML-based markup language called *Edge Side Includes* (ESI). [ESI](#) is not a new language; it was proposed as a standard by Akamai and Oracle, among others, in 2001. ESI enables a web infrastructure to be scaled in order to exploit the large number of points of presence around the world provided by a CDN network, compared to the limited amount of data center capacity on which most software is normally hosted. One drawback to ESI is that it's not implemented in the same way by each CDN provider; therefore, a multi-CDN strategy—as well as porting our code from one provider to another—could result in a lot of refactors and potentially new logic to implement. It's important to highlight that this practice is not embraced massively by organizations worldwide. The recommendation is to use mainly client-side or server-side composition.

Server-Side Composition

The last possibility we have is the server-side composition. In this case, the origin server is composing the view by retrieving all the different micro-frontends and assembling the final page. If the page is highly cacheable, the CDN will then serve it with a long time-to-live policy. However, if the page is personalized per user, serious consideration will be required regarding the scalability of the eventual solution, particularly when many requests are coming from different clients. When we decide to use server-side composition, we must thoroughly analyze the use cases within our application. If we decide to implement a runtime composition, we must have a clear scalability strategy for our servers in order to avoid downtime for our users.

From these possibilities, we need to choose the technique that is most suitable for our project and the team's knowledge. As we will learn later, we also have the opportunity

to deploy an architecture that exploits both client-side and server-side composition—and that's absolutely fine, as long as we understand how to structure our project.

Routing Micro-Frontends

The next important decision is how to route the application views. This choice is closely tied to the micro-frontend composition mechanism that we intend to use for the project.

We can route the page requests on the server side, on the edge side, or on the client side (see [Figure 2-5](#)).

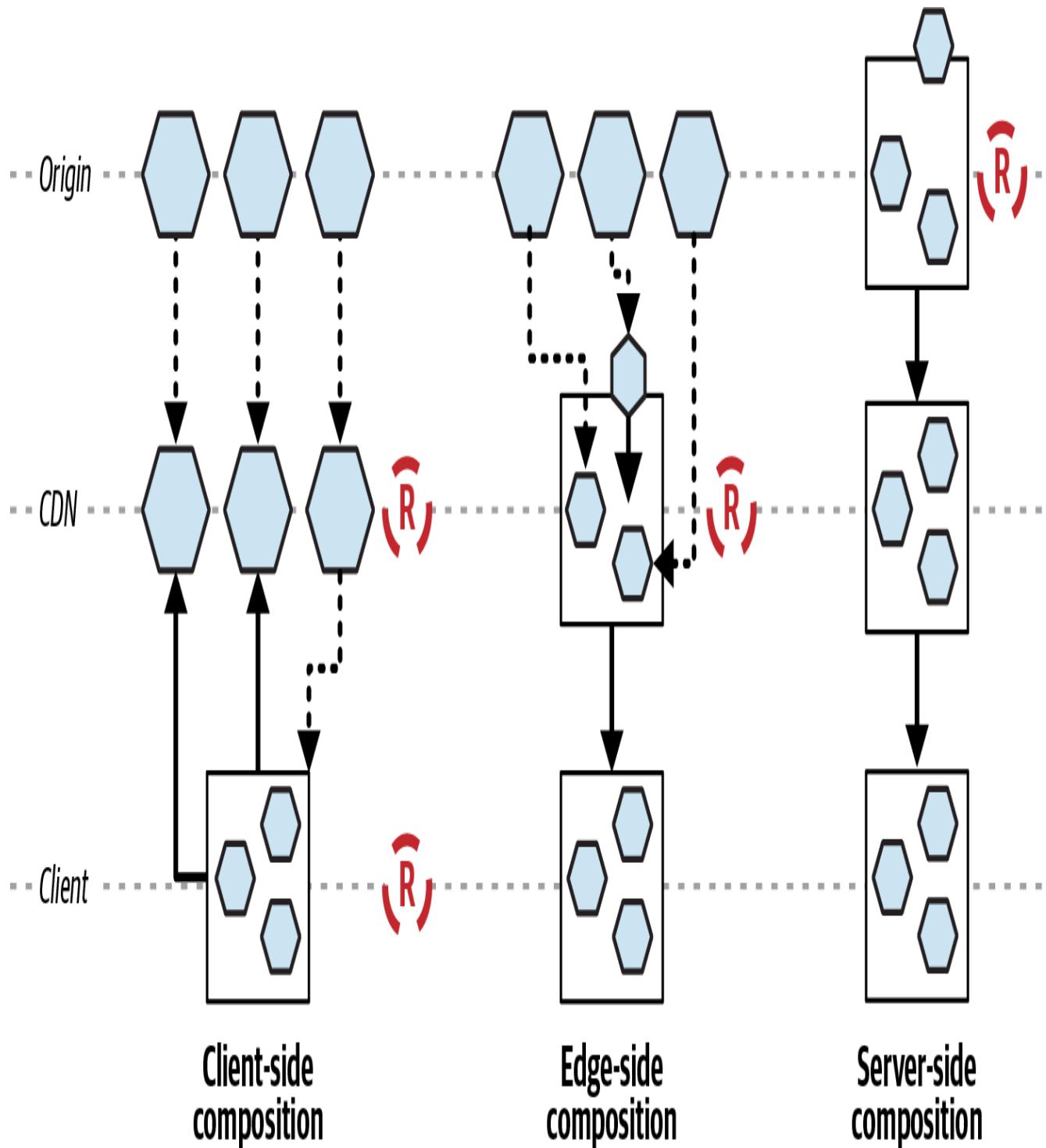


Figure 2-5. Micro-frontend routing diagram

When we use the server-side composition (as on the right of Figure 2-5), routing must happen at the origin because all application logic runs in the application servers.

However, we need to consider that scaling this infrastructure could be challenging, especially when we have to manage burst traffic with many requests per second (RPS). Our servers need to keep up with all the requests and scale horizontally very rapidly.

Each application server must also retrieve the micro-frontends for the composing page to be served.

We can mitigate this problem with the help of a CDN. The main downside is that when we have dynamic or personalized data, we won't be able to rely extensively on the CDN to serve pages because the data may be outdated or not personalized.

When we use edge-side composition in our architecture, the routing is based on the page URL, and the CDN serves the page by assembling the micro-frontends via transclusion at edge level.

In this case, we won't have much room for advanced routing—an important limitation to keep in mind when we pick this architecture.

The final option is client-side routing. In this instance, we load our micro-frontends according to the user state, such as loading the authenticated area of the application when the user is already authenticated or is loading just a landing page (e.g., if the user is accessing our application for the first time).

If we use an application shell that loads a micro-frontend, the application shell is responsible for owning the routing logic, meaning the application shell retrieves the routing configuration first and then decides which micro-frontend to load.

This approach works well when we have complex routing, such as when our micro-frontends are based on authentication, geolocation, or any other sophisticated logic. When we use a multipage website, micro-frontends may be loaded via client-side transclusion. There is almost no routing logic that applies to this kind of architecture because the client relies completely on the URL typed by the user in the browser or the hyperlink chosen on another page—similar to the edge-side include approach. We won't have any scalability issues in either case.

Those routing approaches are not mutually exclusive either. As we will see later in this book, we can combine them—for example, using CDN and origin together, or client side and CDN together.

The important thing is determining how we want to route our application. This fundamental decision will affect how we develop our micro-frontend application.

Micro-Frontend Communication

Communication between micro-frontends typically falls into two categories:

UI events

Where micro-frontends need to notify each other while staying decoupled

Shared data

Can be either short-lived (ephemeral) or long-lived, such as authentication tokens

When we have multiple micro-frontends on the same page, managing a consistent, coherent user interface for our users can become difficult. This is also true when we want to enable communication between micro-frontends owned by different teams. Bear in mind that each micro-frontend should be decoupled from the others on the same page—if not, we break the principle of independent deployment.

New teams approaching this paradigm might be tempted to use a global state manager to share state across micro-frontends. However, this is considered an *antipattern* in distributed systems—a topic that we will dive deeper into later in the book. While it’s a common approach for monolith codebases, using it on micro-frontends would create coupling and friction between teams—the exact opposite of what we aim for in a distributed frontend system.

In this case, we have a few options for notifying other micro-frontends that an event has occurred. In general, we must maintain the micro-frontends as decoupled from each other as possible, avoiding the use of shared global state. We can inject an event bus—a mechanism that allows decoupled components to communicate with each other via events—in each micro-frontend and notify every micro-frontend. If some of them are interested in the event dispatched, they can listen for it and react to it. This is shown in [Figure 2-6](#).

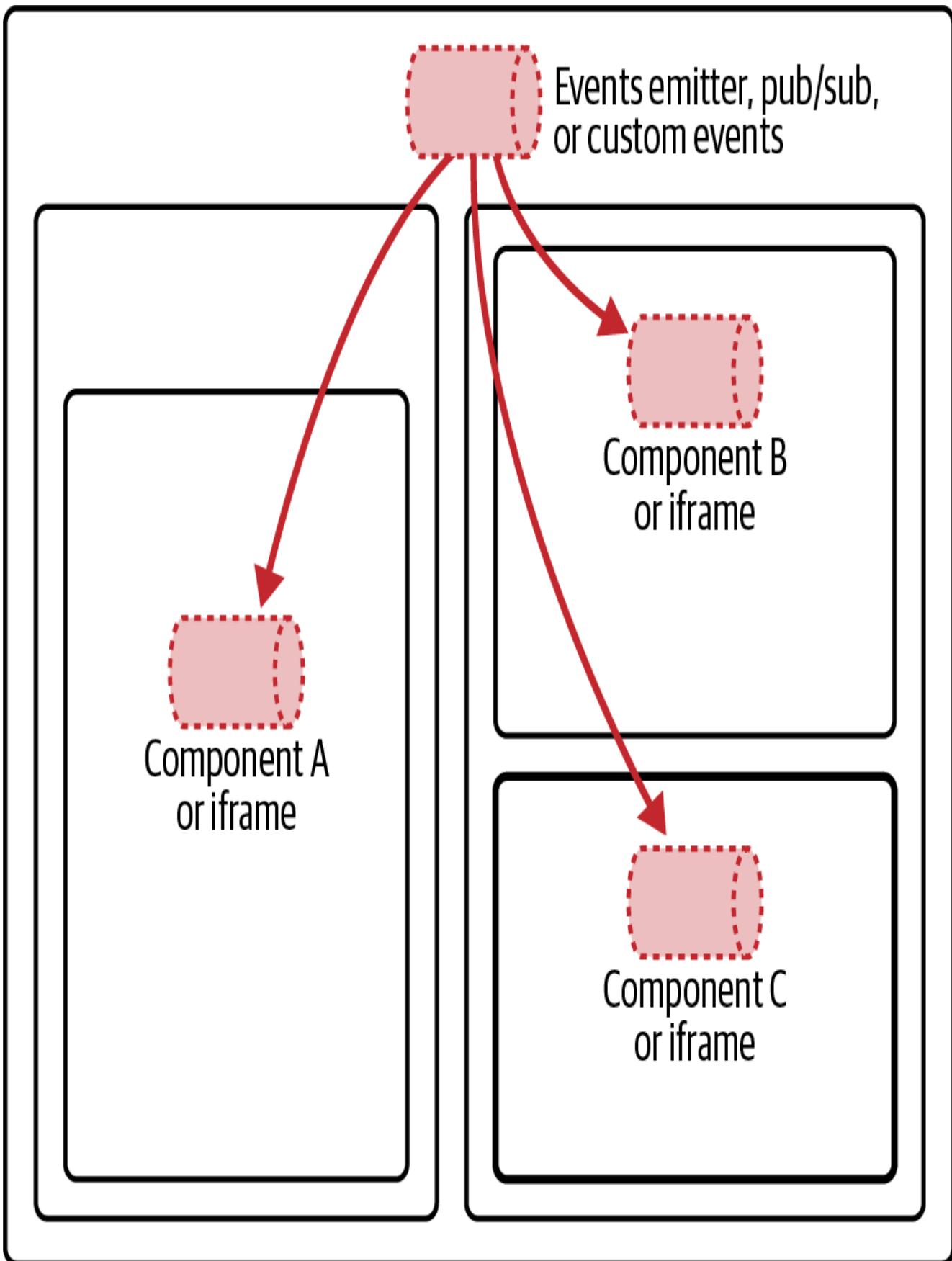


Figure 2-6. Event emitter and custom events diagram

To inject the event bus, we need a micro-frontend container that can instantiate the event bus and inject it into all of the page's micro-frontends. An alternative is for the application shell to apply this logic and either inject or expose the event bus to every micro-frontend.

Another solution is to use *custom events*. These are normal events, but with a custom body. In this way, we can define the string that identifies the event and an optional custom object for the event. Here's an example:

```
new CustomEvent('myCustomEvent', {  
  detail: {  
    someData: 12345  
  }  
});
```

The custom events should be dispatched via an object available to all the micro-frontends—such as the `window` object, the representation of a window in a browser. If you decide to implement your micro-frontends with iframes, using an event bus would enable you to avoid challenges like determining which `window` object to use from inside the iframe, because each iframe has its own `window` object. No matter whether we have a horizontal or a vertical split of our micro-frontends, we need to decide how to pass data between views. Moreover, a custom event propagates to the `window` object by traversing the elements tree expressed in the DOM. Imagine if a team accidentally stops the propagation of the custom event before reaching the DOM. This might cause more than just a headache, so the recommendation is to use an event emitter as the first choice.

Now, imagine we have one micro-frontend for signing in a user and another for authenticating the user on our platform. After being successfully authenticated, the sign-in micro-frontend has to pass a token to the authenticated area of our platform. How can we pass the token from one micro-frontend to another? We have several options.

We can use a web-storage-like session, local storage, or cookies (see [Figure 2-7](#)). In this situation, we might use the local storage for storing and retrieving the token independently. The micro-frontend is loaded because web storage is always available and accessible, as long as the micro-frontends live in the same subdomain.

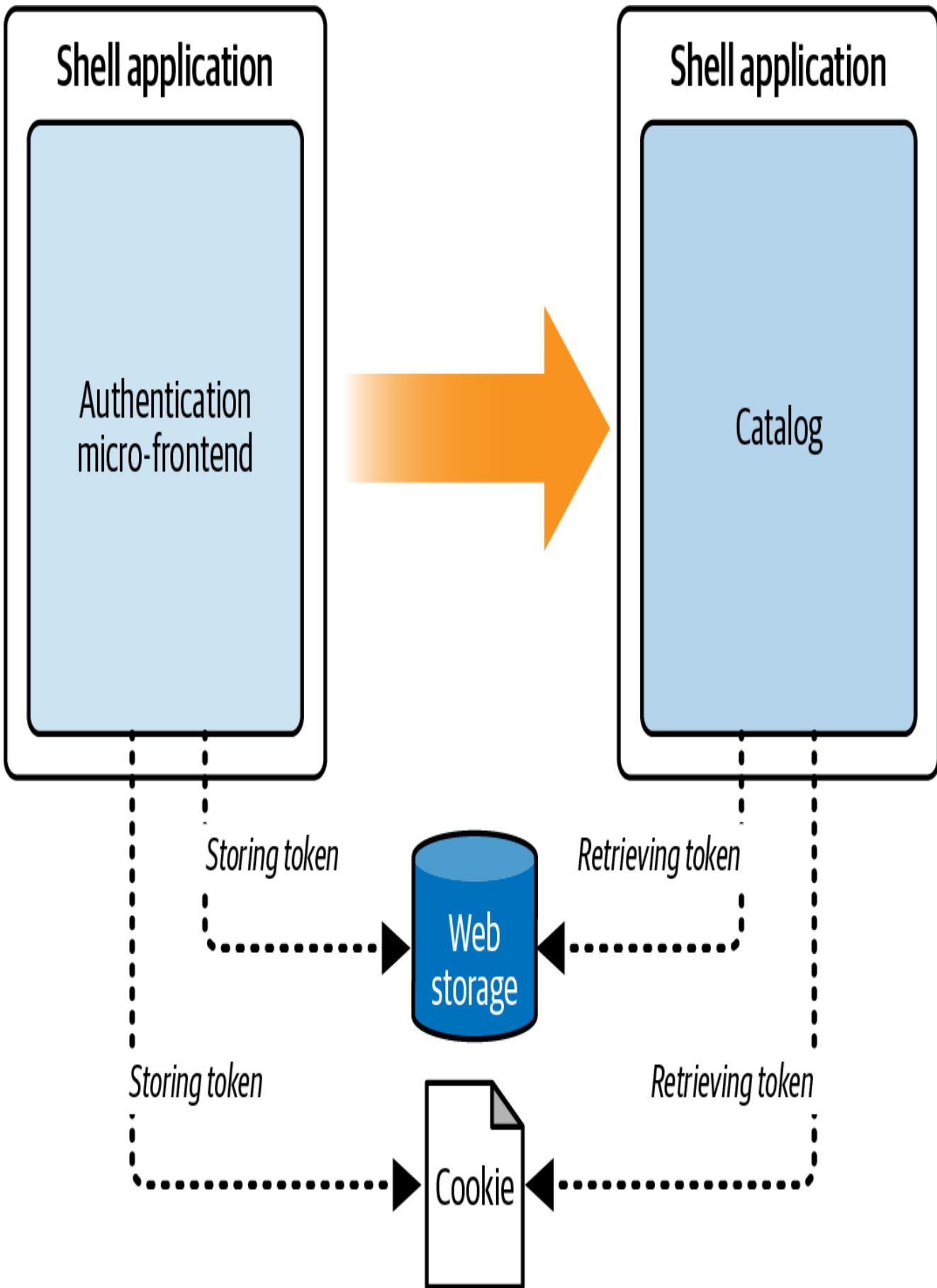


Figure 2-7. Sharing data between micro-frontends in different views

Alternatively, for ephemeral data, you could pass some of it via query strings. For example, in the link `www.acme.com/products/details?id=123`, the text after the question mark represents the query string—in this case, the ID (123) of a specific product selected by the user. It retrieves the full details to display via an API (as in [Figure 2-8](#)). However, keep in mind that using query strings is not the most secure way to pass sensitive data, such as passwords and user IDs.

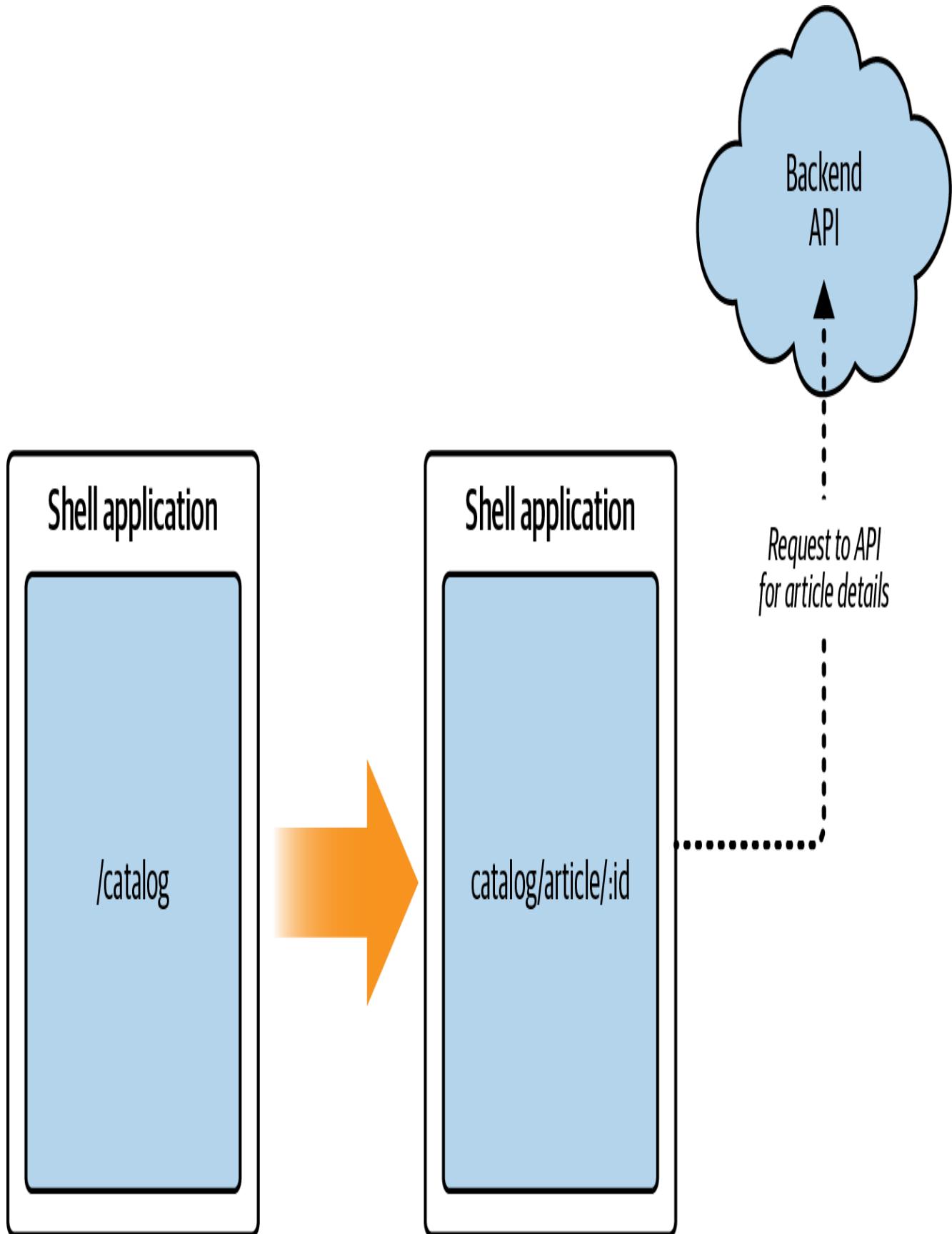


Figure 2-8. Micro-frontend communication via query strings or URL

To summarize, the micro-frontend decision framework is composed of four key decisions: defining, composing, routing, and communicating.

In [Table 2-2](#) you can find all the combinations available based on how you define a micro-frontend.

Table 2-2. Micro-frontend decision framework summary

Micro-frontends definition	Composition	Routing	Communication
Horizontal	Client side	Client side	Event emitter
	Server side	Server side	Custom events
	Edge side	Edge side	Web storage Query strings
Vertical	Client side	Client side	Web storage
	Server side	Server side	Query strings
		Edge side	

Micro-Frontends in Practice

Although micro-frontends are a fairly new approach in the frontend architecture ecosystem, they have been used for 10 years in medium and large organizations. Many well-known companies have made micro-frontends their main system for scaling their business to the next level. In this section, we review some of the organizations that embraced micro-frontends at scale.

Zalando

The first company worth mentioning is Zalando, a European fashion and ecommerce company. I attended a conference presentation made by their technical leads, and I have to admit I was very impressed by what they created as open source software (OSS). This is particularly admirable considering it was still early days and not many companies were talking about microapps.

More recently, Zalando has replaced the well-known OSS project Tailor.js with [Interface Framework](#). Interface Framework is based on concepts similar to Tailor.js but is more focused on components and GraphQL than on fragments.

Formula 1

Formula 1's digital technology team, responsible for the F1 website and apps, aimed to increase web traffic, content consumption, and subscriptions. As explained by Ivan Rigoni—the digital solutions architect in the F1 digital technology team—at [AWS re:Invent 2024](#), they were faced with user expectations for fast loading times, so they turned to micro-frontends to improve website performance and scalability. By migrating from a monolithic architecture to a micro-frontend-based system, F1 achieved a 34% increase in subscriptions and sign-ups, a 26% reduction in platform costs, and significant improvements in Lighthouse performance scores (30% on web, 56% on mobile web). This transformation allowed for independent testing and deployment, faster delivery of changes, full integration with their design system, and more granular caching policies—all contributing to a faster and more engaging user experience. They adopted a test-and-learn iterative approach using the strangler fig pattern, gradually migrating functionality to the new micro-frontend architecture while extracting reusable components into separate systems for use by other domains like F1 TV and F1 Fantasy.

Dunelm

Dunelm is a well-known ecommerce company in the United Kingdom. They have embraced micro-frontends to enable multiple teams to work together in a server-side rendering composition. They started with Next.js, but are moving toward a simpler implementation with React.js due to the realization that Next.js was used mainly for routing and not much for all the features offered by the framework.

They worked on their implementation using a serverless approach, fully in AWS. I highly encourage you to hear more about their story from their principal engineer, Warren Fitzpatrick, in my [“Micro-frontends in the Trenches” series](#).

Netflix

In the revenue and growth department of Netflix, the engineers decided to embark on a micro-frontend approach, creating an internal framework called [Lattice](#).

They discovered prevalent design patterns and architectures dispersed among different tools, with potential for duplicating efforts among teams. Their goal was to streamline

these tools in a manner that aligns with the scalability of the supported teams. The solution needed to embody the flexibility of a micro-frontend and the adaptability of a framework, enabling the stakeholders to enhance these tools effectively. They used Module Federation as well, which helped them solve many challenges, such as dependency management and runtime loading of micro-frontends.

PayPal

If you are a PayPal user and log into the web application, you are interacting with a micro-frontend architecture. Thanks to this approach, their teams have changed how they build the web application. Moreover, they have started sharing their approaches at scale. To build the web interface, multiple teams work together to generate one of the best payment experiences out there. PayPal team members have also started contributing to the community by [sharing what they have learned](#) while building their application.

BMW

BMW implemented a business-to-business (B2B) portal that collects several applications under the same umbrella. The main rationale was to reduce the cognitive load for users who need to perform actions across multiple portals.

Their approach emphasizes maximum flexibility with only a few constraints. In fact, any framework or JavaScript library can be loaded inside their Angular shell, which uses Module Federation to manage the runtime loading of micro-frontends and external dependencies. One of their engineers demonstrates this in [a great demo](#).

OpenTable

Another interesting approach is OpenTable's [OpenComponents project](#), embraced by Skyscanner and other large organizations and released as open source.

OpenComponents takes a really interesting approach to micro-frontends: a registry, similar to the Docker registry, gathers all available components—including their data and UI—and exposes them as HTML fragments that can then be embedded in any HTML template.

Using this technique provides many benefits: team independence, rapid composition of multiple pages by reusing components built by other teams, and the option of rendering components on the server or on the client.

When I spoke with team members at OpenTable, they told me that this project enabled them to scale their teams around the world without creating a large communication overhead. For example, micro-frontends made it possible to repurpose parts developed in the United States for use in Australia—a significant competitive advantage.

DAZN

Last but not least is DAZN, a live and video-on-demand sports platform that uses a combination of SPAs and components orchestrated by a client-side agent called bootstrap.

DAZN's approach focuses on targeting not only the web but also multiple smart TVs, set-top boxes, and consoles. Its approach is fully client side, with an orchestrator always available during the navigation of the video platform to load different SPAs at runtime whenever there is a change of business domain. Max Gallo, a distinguished engineer at DAZN, who followed the creation of the platform from day one, shared his insights and the reasons for embracing this approach in an episode of [“Micro-Frontends in the Trenches”](#).

These are just some of the possibilities micro-frontends offer for scaling co-located and/or distributed teams. More and more companies are embracing this paradigm, including New Relic, Starbucks, Amazon, and Microsoft.

Summary

In this chapter, we discovered the different high-level architectures for designing micro-frontend applications. We dove deep into the key decisions to make: defining, composing, orchestrating, and communicating.

We also introduced a heuristic to test micro-frontend boundaries after defining them.

Finally, we saw that many organizations are already embracing this architecture in production, with successful software not only available inside browsers but also in other end uses, such as desktop applications, consoles, and smart TVs.

It's fascinating how quickly this architecture has spread across the globe. In the next chapter, I will discuss how to develop micro-frontends technically, providing real examples you can use within your own projects.

-
- ¹ Conway's Law states that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."
 - ² The Inverse Conway Maneuver recommends evolving your team and organizational structure to promote your desired architecture.

Chapter 3. Discovering Micro-Frontend Architectures

In the previous chapter, we learned about the decision framework—the foundation of any micro-frontend architecture. In this chapter, we will review the different architecture choices, applying what we have learned so far.

Application of the Micro-Frontend Decision Framework

The decision framework helps you to choose the right approach for your micro-frontend project based on its characteristics, as shown in [Figure 3-1](#).

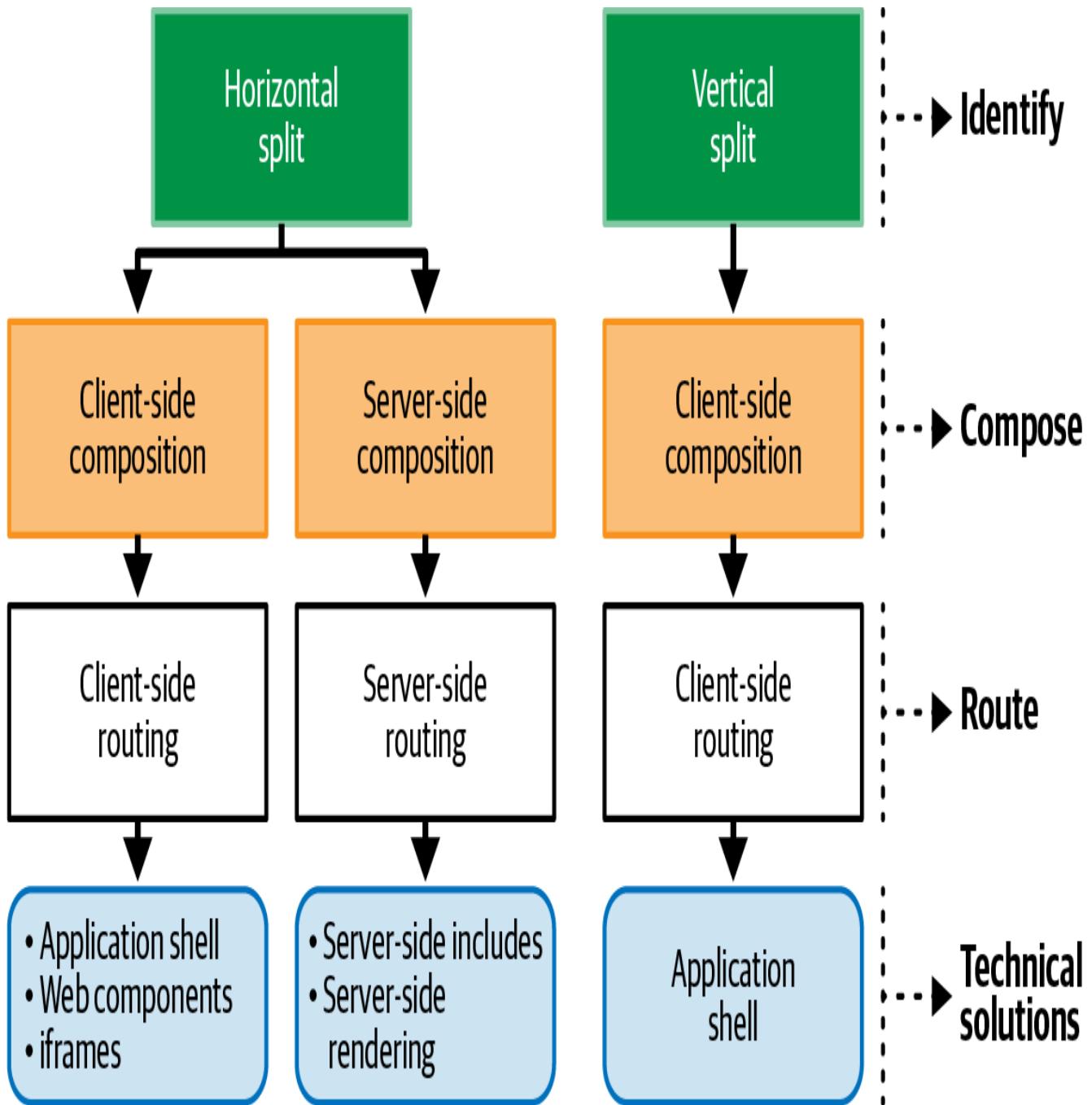


Figure 3-1. Micro-frontend decision framework determining the best architecture for a project

Your first decision will be between a horizontal and vertical split.

Vertical Split

A vertical split offers fewer choices and less complexity because it is likely well known to frontend developers who are used to writing single-page applications (SPAs).

You'll find a vertical split helpful when your project requires a consistent user interface evolution and a fluid user experience across multiple views. That's because a vertical

split provides the closest developer experience to an SPA, enabling developers to apply familiar tools, best practices, and patterns when building a micro-frontend.

In a vertical-split approach, the relation between a micro-frontend and the application shell is always one-to-one; therefore, the application shell loads only one micro-frontend at a time.

You'll also want to use client-side routing with a vertical split. The routing is usually split into two parts: global and local routing. Global routing is used for loading different micro-frontends, and is handled by the application shell, as shown in [Figure 3-2](#).

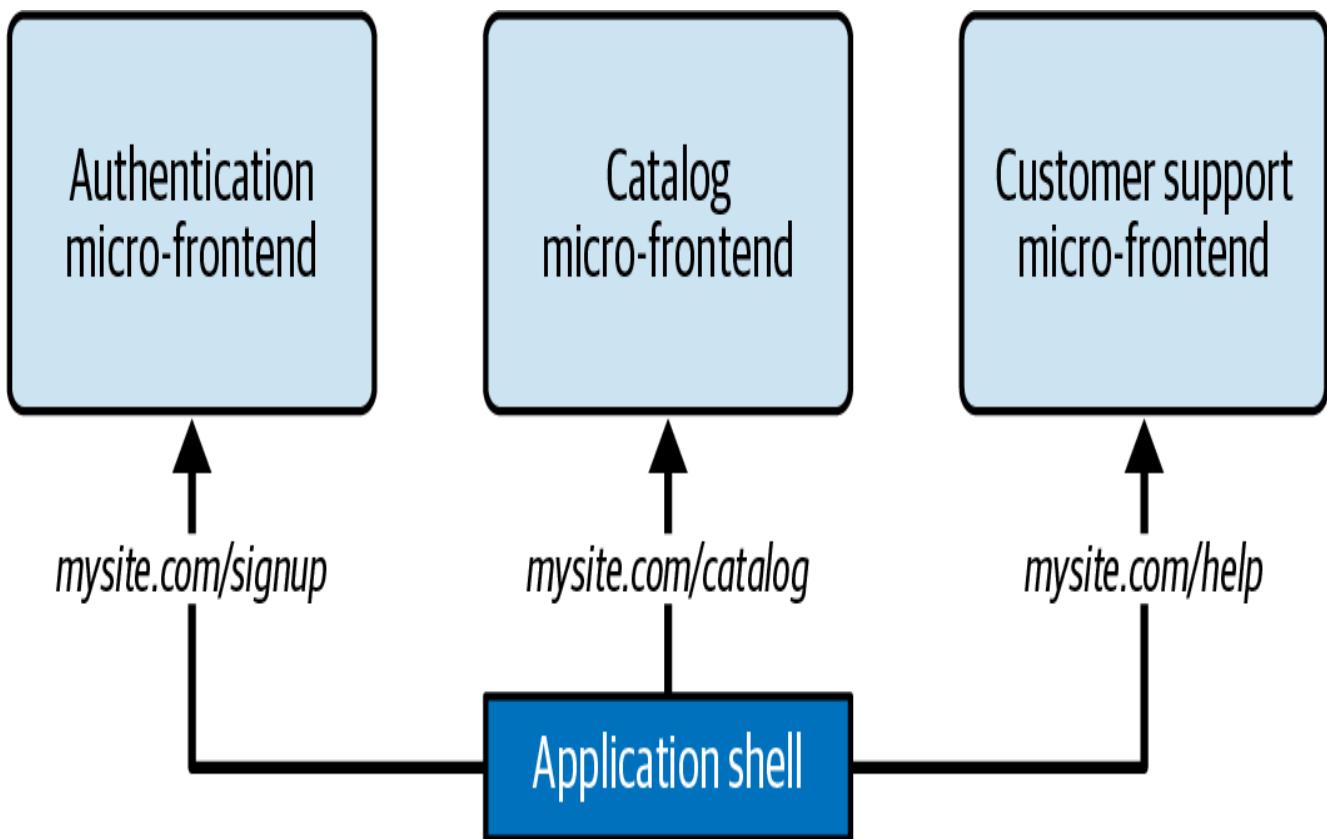


Figure 3-2. Application shell responsible for global routing between micro-frontends

Local routing between views inside the same micro-frontend is managed by the micro-frontend itself. Therefore, you'll have full control over the implementation and evolution of the views it contains, because the team responsible for a micro-frontend is also the subject-matter expert for that business domain of the application. This is shown in [Figure 3-3](#).

A good rule of thumb is that the first-level URLs (e.g., `www.mysite.com/catalog`) are defined in the application shell, while second-level URLs and beyond are defined by micro-frontends (e.g., `www.mysite.com/catalog/books`). Nowadays, the majority of router libraries allow you to implement this concept easily.

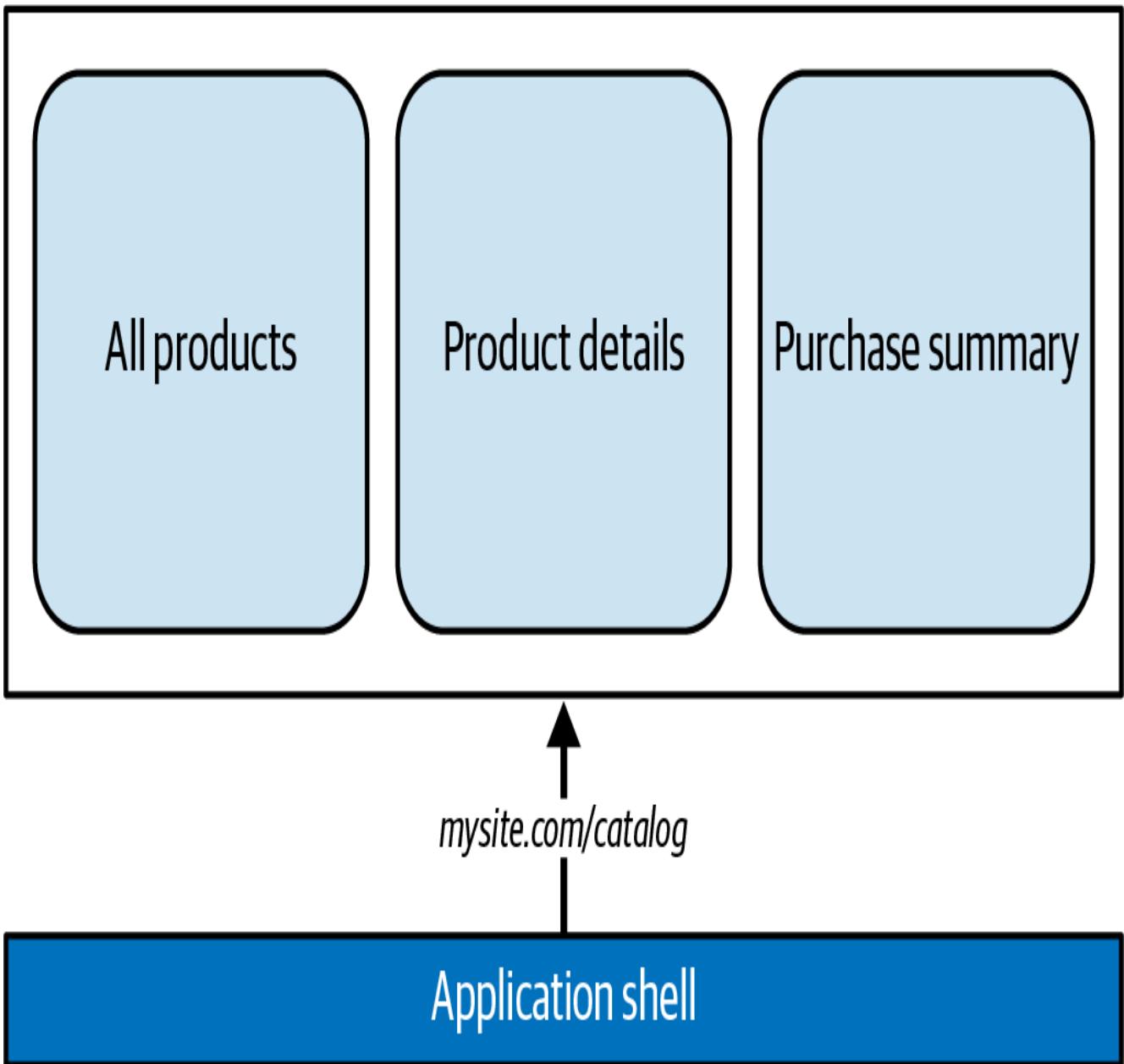


Figure 3-3. Micro-frontend responsible for routing between views available inside the micro-frontend itself

For implementing an architecture with a vertical-split micro-frontend, the application shell loads HTML or JavaScript as an entry point. Therefore, the entry point of each micro-frontend will be one of these file formats. The application shell shouldn't share any business-domain logic with the other micro-frontends and should remain technology agnostic to allow for future system evolution. Therefore, you don't want to rely on any specific UI framework for building the application shell.

This approach helps keep the application shell codebase very stable and prevents unintentional complexity from spreading across multiple teams.

The application shell is always present during user sessions because it's responsible for orchestrating the web application and exposing some life cycle APIs for micro-

frontends to react when they are fully mounted or unmounted.

When vertical-split micro-frontends need to share information with other micro-frontends, such as tokens or user preferences, they can use query string for volatile data, or cookies, web storage, or web workers for more persistent data (like tokens or preferences). This is similar to how horizontal-split micro-frontends share data across views.

Horizontal Split

A horizontal split works well when a business subdomain needs to be presented across several views. Therefore, the reusability of the subdomain becomes key, especially in the following circumstances:

- SEO is a key requirement.
- Your frontend application requires tens, if not hundreds, of developers working together.
- You have a multi-tenant project with customer customizations in specific parts of your software.

The next decision you'll make is choosing between client-side, edge-side, and server-side compositions. Client-side composition is a good choice when your teams are more familiar with the frontend ecosystem, or when your project experiences high traffic with significant spikes. You'll avoid scalability challenges on the frontend layer because you can easily cache your micro-frontends by leveraging a content delivery network (CDN).

Edge-side composition is suitable for a project with static content and high traffic, enabling you to delegate scalability challenges to the CDN provider instead of having to manage them in your infrastructure. As we discussed in [Chapter 2](#), embracing this architecture style has some challenges—such as a more complicated developer experience and the fact that not all CDNs support it—but projects like online catalogs without personalized content may be good candidates for this approach.

Server-side composition gives us the most control over our output, which is great for highly indexed websites, such as news sites or ecommerce platforms. It's also a good choice for websites that require excellent performance metrics—like PayPal and American Express, both of which use server-side composition.

Next is your routing strategy. While you can technically apply any routing to any composition, it's common to use the routing strategy associated with your chosen

composition pattern. For example, if you choose a client-side composition, most of the time, routing will happen at the client-side level. You might use computation logic at the edge to avoid polluting the application shell's code with canary releases or to provide an optimized version of your web application to search engine crawlers, leveraging server-side rendering to improve indexing and page speed rankings.

On the other hand, an edge-side composition will have an HTML page associated with each view. Each time a user loads a new page, the CDN composes a new page by retrieving multiple micro-frontends to create that final view.

Finally, with server-side routing, the application server will know which HTML template is associated with a specific route, so routing and composition happen entirely on the server side.

Your composition choice will also help narrow your technical solutions for building a micro-frontend project. When you use client-side composition and routing, the approach is an application shell loading multiple micro-frontends in the same view, using solutions like the Webpack Module Federation plug-in, iframes, or web components.

For edge-side composition, the only solution currently available is using Edge Side Includes—though we are seeing hints that this may change in the future, as cloud providers extend their edge services to provide more computational and storage resources.

When you decide to use server-side composition, you can rely on server-side includes or one of the many server-side rendering (SSR) frameworks for your micro-frontend applications. SSRs will give you greater flexibility and control over your implementation.

In recent years, I've witnessed an increasing usage of horizontal splits with SSR, where the header and footer are loaded separately from the page content. This is common in workloads with frameworks like Next.js or Astro, where one team is responsible for a group of pages while another team manages the application shell, such as the header and footer.

Missing from the decision framework is the final pillar: how the micro-frontends will communicate when they are in the same or different views. This is mainly because with a horizontal split, you must avoid sharing state across micro-frontends—this is considered an antipattern. Instead, you'll use the techniques mentioned in [Chapter 2](#), such as an event emitter, custom events, or reactive streams using the publish/subscribe (pub/sub) pattern to decouple the micro-frontends and maintain their independence. When you need to communicate between different views, use query string parameters to

share volatile data (such as product identifiers), and use web storage or cookies for persistent data (such as user tokens or local settings).

OBSERVER PATTERN

The observer pattern (also known as pub/sub pattern) is a behavioral design pattern that defines a one-to-many relationship between objects. When one object changes state, all dependent objects are notified and updated automatically.

An object that maintains this one-to-many relationship with other objects that are interested in its state is called the subject or publisher. Its dependent objects are known as observers or subscribers. The observers are notified whenever the state of the subject changes and then react accordingly. The subject can have any number of dependent observers.

Avoid sharing state across horizontal-split micro-frontends whenever possible. This is considered an anti-pattern—something that can cause severe long-term issues, which I've had to help several companies recover from. The problem doesn't lie in the act of sharing state itself, but in the long-term challenges of ownership and maintainability. Testing and coordinating deployments become increasingly difficult because any change to the shared state requires all micro-frontends—and, consequently, the teams responsible for them—to be updated, tested, and deployed simultaneously. This creates unwanted coupling between parts of the application. In general, state should remain within each micro-frontend, maintained by a single team. Communication with the rest of the application should occur through a pub/sub pattern of your choice.

Architecture Analysis

To help you select the right architecture for your project, we'll now analyze the technical implementations for each option, looking at both challenges and benefits. We'll review the different implementations in detail and then assess their characteristics.

The characteristics we'll analyze are as follows:

Deployability

The reliability and ease of deploying a micro-frontend in an environment

Modularity

The ease of adding or removing micro-frontends, and the ease of integrating with shared components hosted by micro-frontends

Simplicity

The ease of understanding or doing, with software considered simple being typically easy to reason about and work with

Testability

The degree to which a software artifact supports testing in a given test context, with higher testability making it easier to find faults in the system by means of testing

Performance

An indicator of how well a micro-frontend would meet the quality of user experience described by **web vitals**, essential metrics for a healthy site

Developer experience

The experience that developers are exposed to when using your product—be it client libraries, software development kits (SDKs), frameworks, open source code, tools, API, technologies, or services

Scalability

The ability of a process, network, software, or organization to grow and manage increased demand

Coordination

Unification, integration, or synchronization of group members' efforts to provide unity of action in the pursuit of common goals

Characteristics are rated on a five-point scale—with one point indicating that the specific architecture characteristic isn't well supported, and five points indicating that the architecture characteristic is one of the strongest features in the architectural pattern.

The score indicates which characteristics shine in every approach. It's almost impossible to have all the characteristics performing perfectly in an architecture, due to the tension they exercise with each other. So, our role is to find the most suitable trade-

off for the application we need to build—hence the decision to use a scoring mechanism to evaluate all of these architectural approaches.

Architecture and Trade-Offs

As I've pointed out elsewhere in this book, I firmly believe that the perfect architecture doesn't exist; there is always a trade-off. The trade-offs are not only technical but also influenced by business requirements and organizational structure. Modern architecture recognizes that multiple forces contribute to the final outcome—not just technical aspects. We must account for sociotechnical aspects and optimize for the context we operate in, rather than searching for the (nonexistent) "perfect architecture" or borrowing architecture from another context without first assessing whether it would be appropriate for our context.

Before settling on a final architecture, take the time to understand the context that you operate in, along with your teams' structures and the communication flows between them. When we ignore these aspects, we risk creating a great technical proposition that is completely unsuitable for our company.

The same precaution must be taken when we read case studies from other companies embracing specific architectures. We need to understand how those companies work and how that compares to our own company's context. Often, case studies focus on how a company solved a specific problem, which may or may not overlap with your challenges and goals. It's up to you to find out if the case study's challenges match your own.

Read widely and talk with different people in the community to understand the forces behind certain decisions. Taking the time to research will help you avoid making incorrect assumptions and will help you become more aware of the environment you are working in.

Every architecture is optimized for solving specific technical and organizational challenges, which is why we see so many approaches to micro-frontends. Remember: there is no right or wrong in architecture—just the best trade-off for your own context.

Vertical-Split Architectures

For a vertical-split architecture, using a client-side composition, client-side routing, and an application shell (as described earlier) is a fantastic strategy for teams with a

solid background of building SPAs and serves as a good first foray into micro-frontends, because the development experience will be mostly familiar. This is probably also the easiest way to enter the micro-frontend world for developers with a frontend background. Client-side rendering architectures often work well for B2B web applications or B2C applications where most interactions occur behind authentication, minimizing concerns about organic SEO.

Application Shell

A persistent part of a micro-frontend application, the application shell is the first thing downloaded when an application is requested. It will shepherd a user session from the beginning to the end, loading and unloading micro-frontends based on the endpoint the user requests.

The main reasons to load micro-frontends inside an application shell are as follows:

Handling the initial user state (if any)

If a user tries to access an authenticated route via a deep link but the user token is invalid, the application shell will redirect the user to the sign-in view or to a landing page. This process is needed only for the first load; after that, every micro-frontend in an authenticated area of a web application should manage the logic for keeping the user authenticated or redirecting them to an unauthenticated page.

Retrieving global configurations

When needed, the application shell should first fetch a configuration file that contains any information used across the entire user session, such as the user's country if the application provides different experiences based on the country.

Fetching the available routes and associated micro-frontends to load

To avoid needlessly deploying the application shell, the route configurations should be loaded at runtime with the associated micro-frontends. This will guarantee control over the routing system without deploying the application shell multiple times.

Setting logging, observability, or marketing libraries

Because these libraries are usually applied to the entire application, it's best to instantiate them within the application shell.

Handling errors if a micro-frontend cannot be loaded

Sometimes, micro-frontends are unreachable due to a network issue or a system bug. It's wise to add an error message (e.g., a 404 page) to the application shell or to load a highly available micro-frontend that displays errors and suggests possible solutions to the user, such as recommending similar products or asking them to come back later.

Loading shared libraries

When using import maps (or other similar approaches), the application shell is responsible for loading common resources, such as UI libraries, router libraries, or other shared dependencies. This ensures that micro-frontends do not need to include these shared resources in their final package unless they require a different version of the same library.

You could achieve similar results by including libraries in every micro-frontend rather than using an orchestrator like the application shell. However, ideally, you want just one place to manage these things from. Having multiple libraries means ensuring they are always in sync between micro-frontends, which requires more coordination and adds complexity to the overall process. It also increases the risk during deployment, where breaking changes may occur, compared to centralizing libraries inside the application shell.

Never use the application shell as a layer to interact constantly with micro-frontends during a user session. The application shell should only be used for edge cases or initialization. Using it as a shared layer for micro-frontends creates a logical coupling between micro-frontends and the application shell, forcing testing or redeployment of all micro-frontends in the application. This situation is also called a distributed monolith—and it is a developer's worst nightmare.

In this pattern, the application shell loads only one micro-frontend at a time. That means you don't need to create a mechanism for encapsulating conflicting dependencies between micro-frontends because there won't be any clash between libraries or CSS styles (as shown in [Figure 3-4](#)), as long as both are removed from the `window` object when a micro-frontend is unloaded.

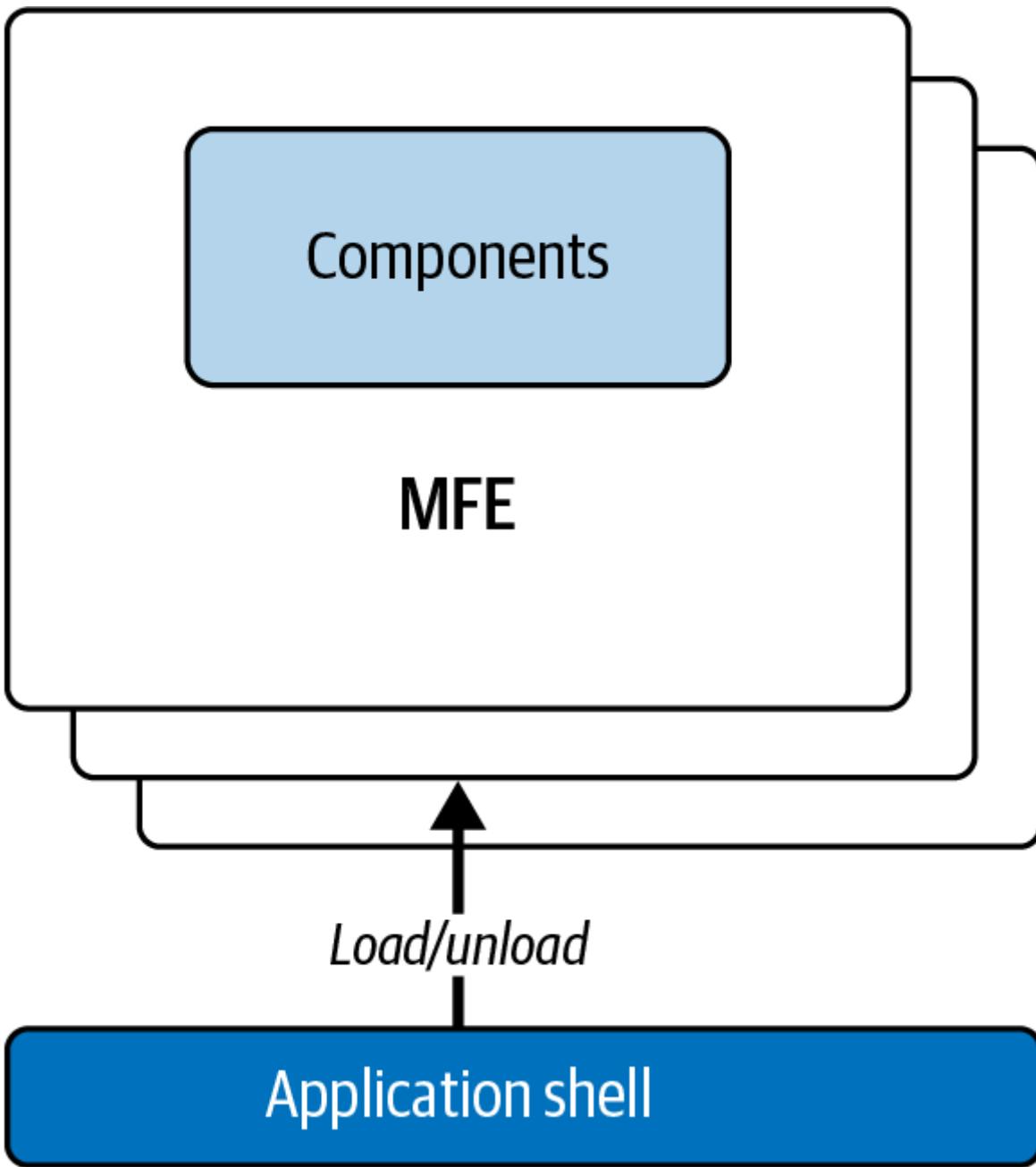


Figure 3-4. Vertical-split architecture with client-side composition and routing using the application shell

The application shell is nothing more than a simple HTML page with logic wrapped in a JavaScript file. Some CSS styles may or may not be included in the application shell to improve the initial loading experience—e.g., displaying a loading animation like a spinner.

Every micro-frontend entry point is represented by a single HTML page containing the logic and style of a single view, or by a small SPA containing several routes that include all the logic needed to enable a user to consume an entire subdomain of the application without requiring a new micro-frontend to load. A JavaScript file can also be loaded instead as a micro-frontend entry point. But, in this case, the initial user

experience is limited because we have to wait until the JavaScript file is interpreted before it can add new elements to the Document Object Model (DOM).

The vertical split works well when we want to create a consistent user experience while providing full control to a single team. A clear sign that this may be the right approach for your application is when there are few repetitions of business subdomains across multiple views, but every part of the application can be represented by an independent action.

Identifying micro-frontends becomes easy when we have a clear understanding of how users interact with the application. If you use an analytics tool like Google Analytics, you'll have access to this information.

If you don't have this information, you'll need to collect it before you can determine how to structure the architecture, business domains, and your organization as a whole.

As explained in the previous chapter, micro-frontends are not designed for high reusability. While components are intended to be reusable, they are fundamentally different from micro-frontends. Therefore, your goal should not be to create reusable micro-frontends at all costs.

However, within every micro-frontend, we can reuse components (think about a design system), generating a modularity that helps avoid too much duplication.

It's more likely, though, that micro-frontends will be reused in different applications maintained by the same company. Imagine that in a multi-tenant environment, you have to develop multiple platforms and want a similar user interface with some customizations for part of every platform. You will be able to reuse vertical-split micro-frontends, reducing code fragmentation and enabling the system to evolve independently based on the business requirements.

Challenges

Of course, there will be some challenges during the implementation phase, as with any architecture pattern. Apart from domain-specific challenges, we'll have common ones —some of which have an immediate answer, while others depend more on context. Let's look at four major challenges: sharing state, micro-frontend composition, a multi-framework approach, and the evolution of your architecture.

Sharing state

As mentioned before, a common challenge we face when working with micro-frontends in general is how to share state between micro-frontends. While we don't need to share information as much with vertical-split architecture, the need still exists.

Some of the information that we may need to share across multiple micro-frontends can be safely stored via Web Storage, such as the audio volume level for media that the user played or the fonts recently used to edit a document.

When information is more sensitive, such as personal user data or an authentication token, we need a way to retrieve this information from a public API and then share it across all the micro-frontends interested in this information.

There are multiple approaches we can implement. We can centralize the token management in the application shell and decorate every API request by appending the token in the header using a middleware function, like [Dream.mf](#) does, or we can create a shared library to handle this concern in every micro-frontend. With vertical split, this problem is easily manageable without causing too many headaches.

Composing micro-frontends

You have several options for composing vertical-split micro-frontends inside an application shell. Remember, however, that vertical-split micro-frontends are composed and routed only on the client side, so we are limited to what the browser's standards allow.

There are five techniques for composing micro-frontends on the client side:

ES modules

JavaScript modules can be used to split our applications into smaller files that can be loaded at compile time or at runtime, fully implemented in modern browsers.

This provides a solid mechanism for composing micro-frontends at runtime using standards.

To implement an ES module, we simply define the `type="module"` attribute in our script tag, and the browser will interpret it as a module:

```
<script type="module" src="catalogMFE.js"></script>
```

This module will always be deferred and can implement cross-origin resource sharing (CORS) authentication.

ES modules can also be defined for the entire application inside an import map, allowing us to use the syntax to import a module inside the application. Import maps are definitely an emerging standard gaining traction for micro-frontend architectures.

SystemJS

This module loader supports import-map specifications, which are not natively supported in all browsers. This allows them to be used inside the SystemJS implementation, where the module loader library makes the implementation compatible with all browsers.

This is a handy solution when we want our micro-frontends to load at runtime, because it uses a syntax similar to import maps and allows SystemJS to take care of the browser's API fragmentation.

Module Federation

This library, introduced in Webpack 5, is used for loading external modules, libraries, or even entire applications inside another one. The library handles the complex work needed for composing micro-frontends, managing the micro-frontend scope, sharing dependencies between different micro-frontends, and handling different versions of the same library without runtime errors.

The developer experience and the implementation are so slick that it feels like writing a normal SPA. Every micro-frontend is imported as a module and then implemented in the same way as a UI framework component. The abstraction made by this library makes the entire composition challenge almost completely painless. With version 2.0, Module Federation is completely bundle-agnostic and can be used with most popular bundlers like Webpack, Vite, Rollup, or Rspack.

Native Federation

Native Federation is a widely adopted library in the Angular community, though not tightly coupled to the framework. It draws inspiration from Module Federation but leverages web standards to address similar challenges. Native Federation utilizes import maps to

load micro-frontends and shared dependencies. The Angular team recommends this library for implementing Angular micro-frontends.

HTML parsing

When a micro-frontend has an entry point represented by an HTML page, we can use JavaScript to parse the DOM elements and append the nodes needed inside the application shell's DOM.

At its simplest, an HTML document is really just an XML document with its own defined schema. Given that, we can treat the micro-frontend as an XML document and append the relevant nodes inside the shell's DOM using the **DOMParser object**.

After parsing the micro-frontend DOM, we then append the DOM nodes using **adoptNode** or **cloneNode** methods. However, these methods don't work with the script element, because the browser doesn't evaluate it. So, in this case, we create a new script element, using the source file found in the micro-frontend's HTML page. Creating a new script element will trigger the browser to fully evaluate the JavaScript file associated with this element.

In this way, you can even simplify the micro-frontend developer experience, because your team will know how the initial DOM will look.

This technique is used by some frameworks, such as Qiankun, which allows HTML documents to be micro-frontend entry points.

All the major frameworks composed on the client side implement these techniques, and sometimes you even have multiple options to pick from. For example, with single-spa, you can use ES modules, SystemJS with import maps, or Module Federation.

Multi-framework approach

Using micro-frontends for a multi-framework approach is a controversial decision, because many people think that this forces them to use multiple UI frameworks, like React, Angular, Vue, or Svelte. But what is true for frontend applications written in a monolithic way is also true for micro-frontends.

Although technically you can implement multiple UI frameworks in an SPA, it creates performance issues and potential dependency clashes. This applies to micro-frontends

as well, so using a multi-framework implementation for this architecture style isn't recommended.

Instead, follow best practices like reducing external dependencies as much as you can, importing only what you use rather than entire packages, which may increase the final JavaScript bundle. Many JavaScript tools implement tree-shaking mechanisms to help achieve smaller bundle sizes.

There are some use cases in which the benefits of having a multi-framework approach with micro-frontends outweigh the challenges, such as when it creates a healthy developer flywheel, reducing the time to market for their business logic without affecting production traffic.

Imagine you start porting a frontend application from an SPA to micro-frontends. Working on a micro-frontend and deploying the SPA codebase alongside it would help you to provide value for your business and users.

First, we would have a team identifying best practices for the porting process (such as identifying libraries to reuse across micro-frontends), setting up the automation pipeline, sharing code between micro-frontends, and so on. Second, after creating the minimum viable product (MVP), the micro-frontend can be shipped to the end user, retrieving metrics and comparing it with the older version.

In a situation like this, asking a user to download multiple UI frameworks is less problematic than developing the new architecture for several months without knowing if the direction leads to better results. Validating your assumptions is crucial for generating best practices shared by different teams inside your organization. Improving the feedback loop and deploying code to production as fast as possible demonstrates the best approach for overcoming future challenges with microarchitectures in general.

You can apply the same reasoning to other libraries in the same application but with different versions, such as when you have a project with an old version of Angular and want to upgrade to the latest version.

Remember, the goal is to create the muscles for moving quickly with confidence, reducing potential mistakes, automating what is possible, and fostering the right mindset across the teams.

Architecture Evolution

Perfectly defining the subdomains on the first try isn't always feasible. In particular, using a vertical-split approach may result in coarse-grained micro-frontends that

become complicated after several months of work because of broadening project scope as the team's capabilities grow. Additionally, we can gain new insights into assumptions made at the beginning of the process.

Fear not! This architecture's modular nature helps you address these challenges and provides a clear path for evolving it alongside the business. When your team's cognitive load starts to become unsustainable, it may be time to split your micro-frontend. One of the many best practices for splitting a micro-frontend is code encapsulation, which is based on a specific user flow. Let's explore how this works.

CODE ENCAPSULATION

The concept of encapsulation comes from object-oriented programming (OOP) and is associated with classes and how to handle data. Encapsulation binds together the attributes (data) and the methods (functions and procedures) that manipulate the data to protect it. The general rule, enforced by many languages, is that attributes should only be accessed—that is, retrieved or modified—using methods that are contained (encapsulated) within the class definition.

Imagine your micro-frontend is composed of several views, such as a payment form, sign-up form, sign-in form, and email-and-password-retrieval form—as exemplified in [Figure 3-5](#).

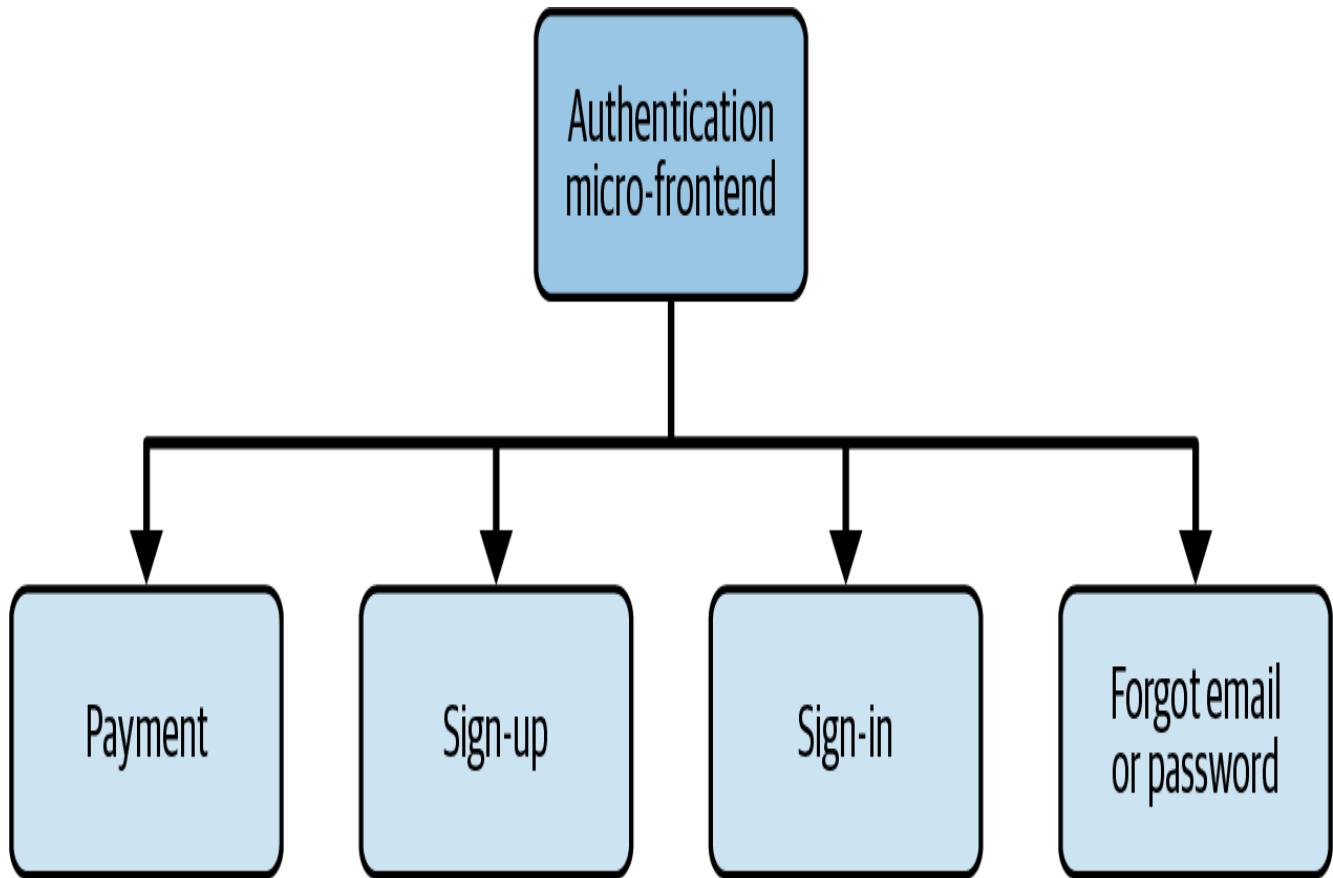


Figure 3-5. An authentication micro-frontend composed of several views that may create a high cognitive load for the team responsible for this micro-frontend

An existing user accessing this micro-frontend is more likely to sign in to the authenticated area or to want to retrieve their account email or password, while a new user is likely to sign up or make a payment. A natural split for this micro-frontend, then, could be one micro-frontend for authentication and another for subscription. In this way, you'll separate the two according to business logic without having to ask the users to download more code than the flow would require. This split is shown in [Figure 3-6](#).

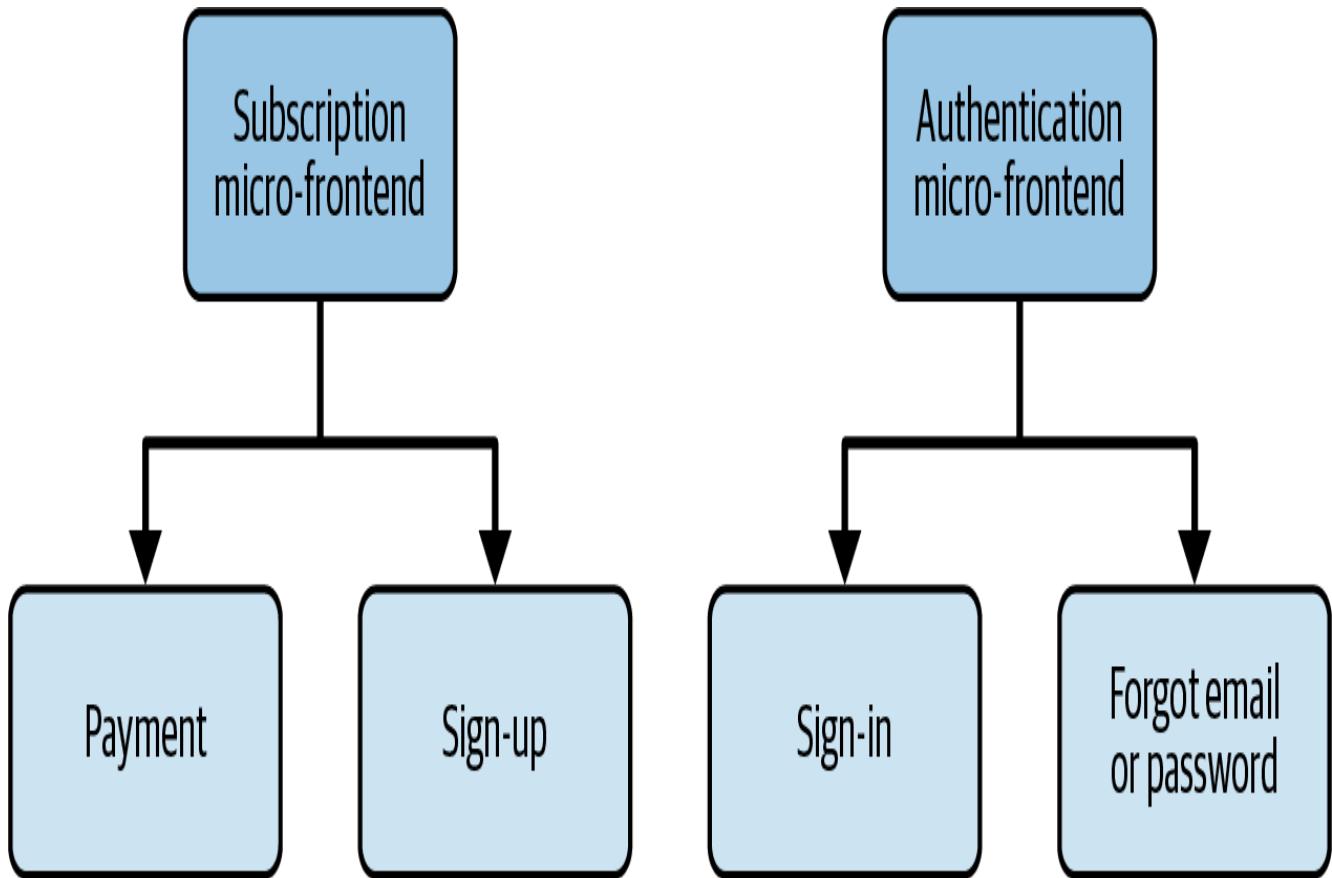


Figure 3-6. Splitting the authentication micro-frontend to reduce the cognitive load following user experience

This isn't the only way to split this micro-frontend, but regardless of how you split it, make sure you're prioritizing business outcomes rather than technical ones. Prioritizing the customer experience is the best way to provide a final output that your users will enjoy.

Encapsulation helps with these situations. Invest time in evaluating how to implement the application state, and you may save time later while also reducing your cognitive load. It's always easier to think clearly when the code is well organized within defined boundaries than spread across multiple parts of the application.

When libraries or even logic are used in multiple domains, such as in a form validation library, you have a few options:

Duplicate the code.

Code duplication isn't always a bad practice; it depends on what you are optimizing for and the overall impact of the duplicated code.

Let's say you have a component that has different states based on user status and the view where it's hosted, and that this component is subject to new requirements more often in one domain than in others. You may want to centralize it. Keep in mind, though, that

every time you have a centralized library or component, you must build a solid governance to make sure that when this shared code is updated, it is also updated in every micro-frontend that uses the code. When this happens, you must ensure that the new version doesn't break anything inside each micro-frontend, and you need to coordinate the activity across multiple teams.

In this case, the component isn't difficult to implement, and it will become easier to build for every team that uses it because there are fewer states to take care of. That enables every implementation to evolve independently at its own speed. Here, we're optimizing for speed of delivery and reducing the external dependencies for every team.

This approach works best when you have a limited amount of duplication. When you have tens of similar components, this reasoning doesn't scale anymore; you'll want to abstract into a library instead.

Abstract your code into a shared library.

In some situations, you really want to centralize the business logic to ensure that every micro-frontend is using the same implementation, as with integrating payment methods. Imagine implementing multiple payment methods with their validation logic, handling errors, and so on in your check-out form. Duplicating such a complex and delicate part of the system isn't wise. Creating a shared library instead will help maintain consistency and simplify the integration across the entire platform.

Within the automation pipelines, you'll want to add a version check on every micro-frontend to review the latest library version. Unfortunately, while distributed systems help you scale the organization and deliver with speed, sometimes you need to enforce certain practices to ensure the health and stability of the system.

Delegate to a backend API to provide some configuration and implement the business logic in both micro-frontends.

The third option is to delegate the common part to the backend, so it can be served to all your vertical-split micro-frontends.

Imagine you have multiple micro-frontends implementing an input field with specific validation that is simple enough to represent using a regular expression. You might be tempted to centralize the logic in a common library, but this would require updating this dependency every time something changes.

Considering the logic is easy enough to represent and the common part would use the same regular expression, you can provide this information as a configuration field when the application loads and make it available to all the micro-frontends via web storage. That way, if you want to change the regular expression, you won't need to redeploy every micro-frontend that implements it. You'll just update the regular expression in the configuration, and all the micro-frontends will automatically use the latest version.

CODE DUPLICATION OVER INCORRECT ABSTRACTIONS

Many industry experts have started to realize that abstracting code is not always beneficial, especially in the long run. In certain cases, code duplication can provide more advantages than a premature or hasty abstraction. Moreover, duplicated code can be easily abstracted if and when needed; it's more challenging to try to move away from abstractions once they're present in the code.

If you are interested in this topic, read “[The Wrong Abstraction](#)”, a 2016 blog post by Sandi Metz. Kent C. Dodds’ [AHA programming](#)—his “avoid hasty abstractions” concept—is heavily inspired by the work that Metz describes in her blog and talk.

Additionally, the well-known DRY principle (Don’t Repeat Yourself) appears to be misapplied by many developers, who just looked in the code for duplicated lines of code and abstracted them. In the second edition of [The Pragmatic Programmer](#), where the DRY principle was first introduced, the authors provide a great explanation about this point:

In the first edition of this book we did a poor job of explaining just what we meant by Don’t Repeat Yourself. Many people took it to refer to code only: they thought that DRY means “don’t copy-and-paste lines of source.”

That is part of DRY, but it’s a tiny and fairly trivial part.

DRY is about the duplication of knowledge, of intent. It’s about expressing the same thing in two different places, possibly in two totally different ways.

Another scenario is to start with code duplication and then abstracting later, once you gain more clarity. This approach enables teams to move quickly at the beginning of a project or feature development, without getting bogged down in premature abstractions. By duplicating code early on, developers can explore different implementations and gather real-world usage data. As patterns emerge and requirements stabilize, it becomes clearer which parts of the code are truly common and which parts need to remain specific to each use case. This clarity makes it much easier to create meaningful abstractions that solve actual—not hypothetical—problems. Moreover, when abstracting later, developers can avoid overengineering and concentrate on creating simpler, more focused shared components or utilities that precisely meet the needs of multiple teams. This method also reduces the risk of creating abstractions that become

obsolete or require frequent changes, because the abstraction is based on proven, battle-tested code rather than speculative requirements.

It's important to understand that no solution fits all situations. Consider the context your implementation represents and choose the best trade-off within the guardrails you are working under.

Could you have designed the micro-frontends in this way from the beginning? You potentially could have. But the whole point of this architecture is to avoid premature abstractions, optimize for fast delivery, and evolve the architecture when complexity increases or the direction changes.

Implementing a Design System

In a distributed architecture like micro-frontends, design systems may seem difficult to implement. But in reality, the technical implementation doesn't differ too much from that of a design system in an SPA.

When thinking about a design system applied to micro-frontends, imagine a layered system composed of design tokens, basic components, a user interface library, and the micro-frontends that host all these parts together—as depicted in [Figure 3-7](#).

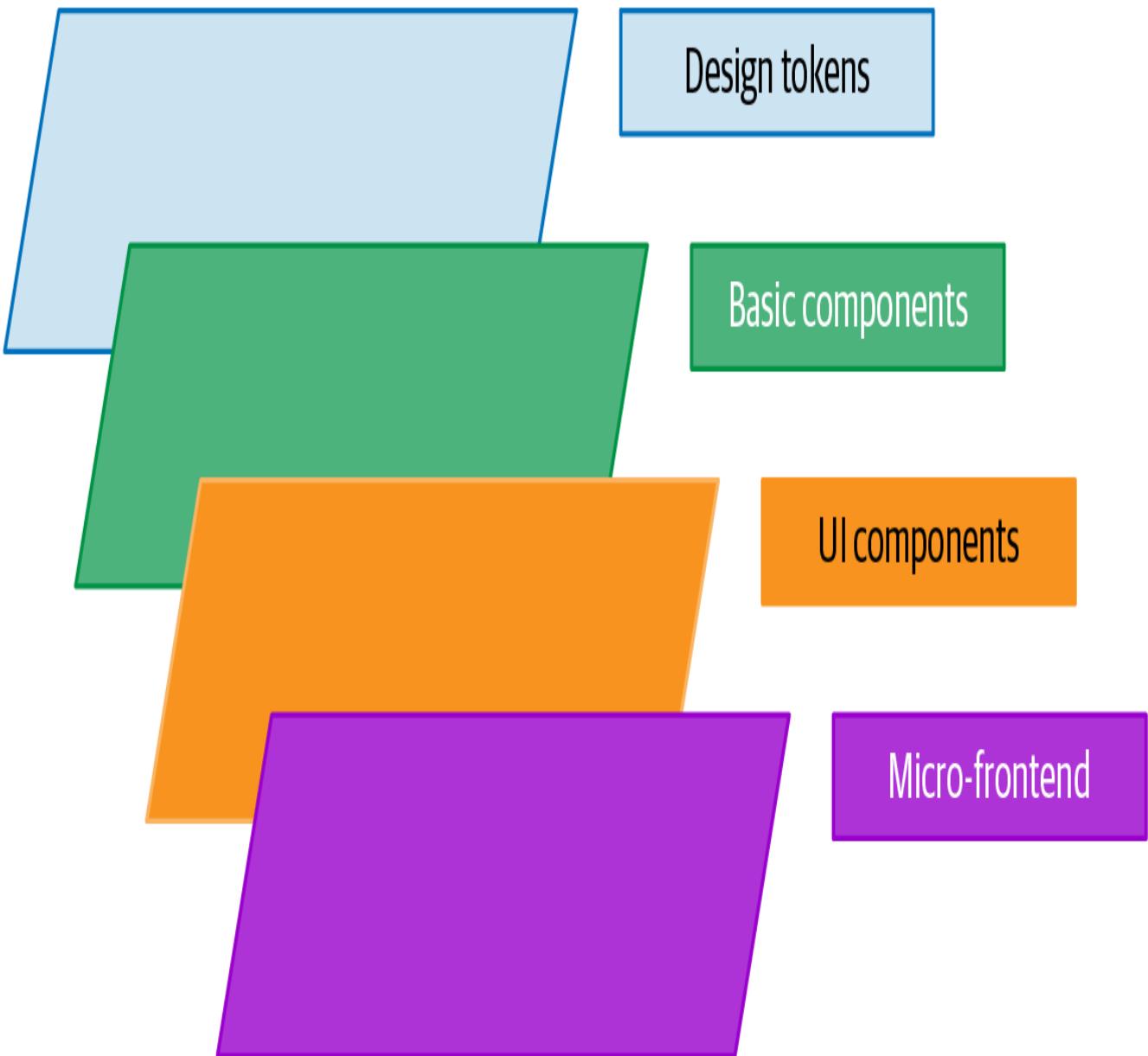


Figure 3-7. How a design system fits inside a micro-frontend architecture

The first layer (design tokens) enables you to capture low-level values and then to create the styles for your product, such as font families, text colors, text size, and many other characteristics within our final user interface. Generally, design tokens are listed in JSON or YAML files, expressing every detail of our design system.

We don't usually distribute design tokens across different micro-frontends because each team will implement them in their own way, which risks introducing bugs in some parts of the application but not in others, increasing code duplication across the system and, more generally, slowing down the maintenance of a design system.

However, there are situations when design tokens can serve as an initial step for creating a level of consistency that can be iterated on later, with basic components shared across all the micro-frontends.

Often, teams do not have enough capacity for implementing the final design system components inside every micro-frontend. Therefore, it is important to make sure that if you go down this path, you will have sufficient time and space for iterating on the design system.

The next layer is the basic components. Usually, these components don't hold the application business logic and are completely unaware of where they will be used. As a result, they should be as generic as possible—such as a label or button, which provides the consistency we need while remaining flexible enough to be used anywhere in the application.

This is the perfect stage for centralizing the code that will be used across multiple micro-frontends. In this way, we create the UI consistency necessary to enable every team to use components at the level they need.

The third layer is a UI components library, usually a composition of basic components that contain some reusable business logic within a given domain.

We may be tempted to share these components as well; however, exercise caution. The governance to maintain and the organizational structure may create many external dependencies across teams, creating more frustrations than efficiencies.

One exception is when there are complex UI components that require many iterations, and there is a centralized team responsible for them. Imagine, for instance, building a complex component such as a video player with multiple functionalities, such as closed captions, a volume bar, and trick play. Duplicating these components is a waste of time and effort; centralizing and abstracting your code is by far more efficient.

Note, though, that shared components are often not reused as much as we expect, resulting in wasted effort. Therefore, think carefully before centralizing a component. When in doubt, start duplicating the component and, after a few iterations, review whether these components need to be abstracted. The wrong abstraction is far more expensive than duplicated code.

The final layer is the micro-frontend that hosts the UI components library. Keep in mind the importance of a micro-frontend's independence. To ensure we are finding the right trade-offs between development speed, independent teams, and UI consistency, we should consider validating the dependencies regularly throughout the project life cycle. In the past, I've worked at companies where this exercise was done every two weeks at the end of every sprint, and it helped many teams postpone tasks that may not have been achievable during a sprint due to blocks from external dependencies. In this way, you can reduce your team's frustration and increase their performance.

On the technical side, the best investment you can make when creating a design system is in web components. Because you can use web components with any UI framework, if you decide to change frameworks at a later stage, the design system will remain intact, which will save you time and effort.

There are some situations where using web components is not viable, such as projects that need to target old browsers. However, in most cases, you won't have such strong requirements and will be able to target modern browsers, enabling you to leverage web components with your micro-frontend architecture.

While getting the design system ready to be implemented is half the work, to accomplish the delivery within your micro-frontend architecture, you'll need solid governance to maintain that initial investment. Remember, dealing with a distributed architecture is not as straightforward as it may seem.

Usually, the first implementation happens quite smoothly because there is ample time allotted; the problems come with subsequent updates.

Especially when working with distributed teams, the best approach is to automate the design system version validation as early as possible in the development process, following a shift-left approach. For example, you could use a tool like Dependabot to automate the management of shared dependencies across micro-frontends. In this way, you automate the update of common libraries, and every team will always deliver the latest versions.

Some companies have custom dashboards for dealing with this problem—not only for design systems but also for other libraries, such as logging or authentication. Every team can then check in real time whether their micro-frontend implements the latest versions.

Finally, let's consider the team's structure. Traditionally, in enterprise companies, the design team is centralized, taking care of all the aspects of the design system—from ideation to delivery—and the developers just implement the library that the design team provides. However, some companies implement a distributed model wherein the design team is a central authority that provides the core components and direction for the entire design system. Other teams populate the design system with new components or new functionalities of existing ones. In this second approach, we reduce potential bottlenecks by allowing the development teams to contribute to the global design system, while keeping guardrails in place to ensure every component respects the overall plan—for example, regular meetings between design and development, office hours during which the design team can guide development teams, or even collaborative

sessions where the design team sets the direction but the developers actually implement the code inside the design system.

Developer's Experience

For vertical-split micro-frontends, the developer's experience is very similar to SPAs. However, there are a few suggestions that you may find useful to think about upfront.

I've seen many companies implement a command-line tool to scaffold micro-frontends with a basic setup and shared libraries (e.g., a logging library) that are used across all micro-frontends. While not an essential tool to have from day one, it's definitely helpful in the long term, especially for new team members. In addition, consider creating a dashboard that summarizes the micro-frontend version you have in different environments.

In general, all the tools you are using for developing an SPA are still relevant for a vertical-split micro-frontend architecture.

Performance and Micro-Frontends

Is good performance achievable in a micro-frontend architecture? Definitely!

Performance of a micro-frontend architecture, as in any other frontend architecture, is key to the success of a web application. And a vertical-split architecture can achieve good performance thanks to the split of domains and, consequently, the code that needs to be shared with the client.

Think for a moment about an SPA. Typically, the user has to download all the code specifically related to the application, the business logic, and the libraries used in the entire application. For simplicity, let's imagine that an entire application code is 500 KB. The unauthenticated area—composed of the login, sign-up, landing page, customer support, and a few other views—requires 100 KB of business logic, while the authenticated area requires 150 KB of business logic. Both areas share the same bundled dependencies, which total 250 KB. This is visualized in [Figure 3-8](#).

A new user has to download all 500 KB, even if the action they want to perform requires only part of the SPA. For example, a user may just want to understand the business proposition by visiting just the landing page, another may want to see the payment methods available, and an authenticated user may be interested mainly in the authenticated area where services or products are offered.

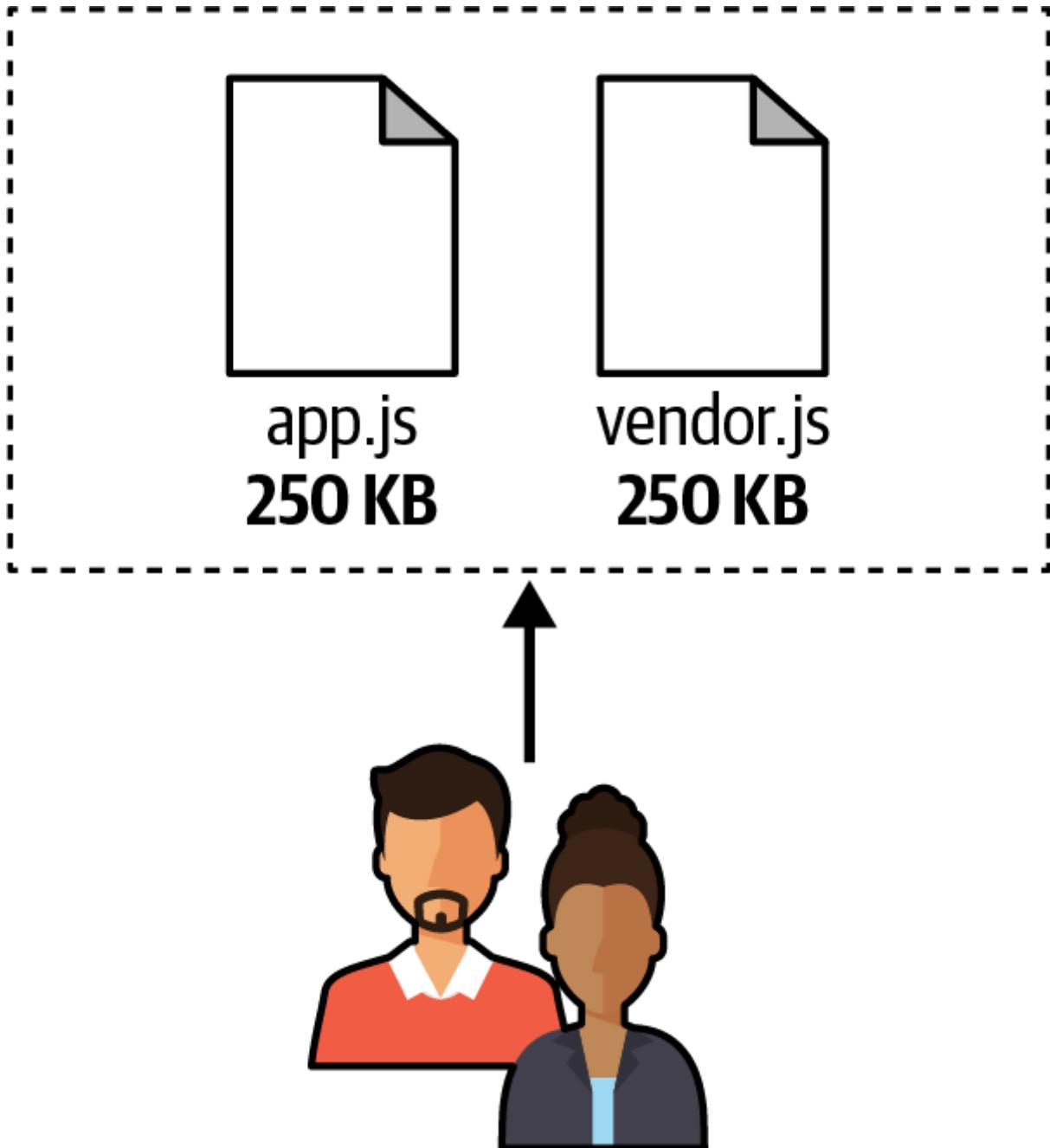


Figure 3-8. Any user of an SPA has to download the entire application regardless of the action they intend to perform in the application

No matter what the user is trying to achieve, they are forced to download the entire application. In a vertical-split architecture, however, our unauthenticated user who wants to see the business proposition on the landing page will download only the code for that specific micro-frontend, while an authenticated user will download only the codebase for the authenticated area.

We often don't realize that our users' behaviors are different from the way we interpret the application, because we often optimize the application's performance as a whole

rather than by how users interact with the site. Optimizing our site according to user experiences results in a better outcome.

Applying the previous example to a vertical-split architecture, a user interested only in the unauthenticated area will download less than 100 KB of business logic plus the shared dependencies, while an authenticated user will download 250 KB plus the shared dependencies.

Clearly, a new user who moves beyond the landing page will download almost 500 KB, but this approach still saves some kilobytes if we have properly defined the application boundaries, because it's unlikely a new user will visit every single view. In the worst-case scenario, the user will download 500 KB, just as they would in a standard SPA, but this time not everything is loaded upfront.

There is, of course, some additional logic to download due to the application shell, but it is usually only in double-digits size, making it negligible for this example. See [Figure 3-9](#).

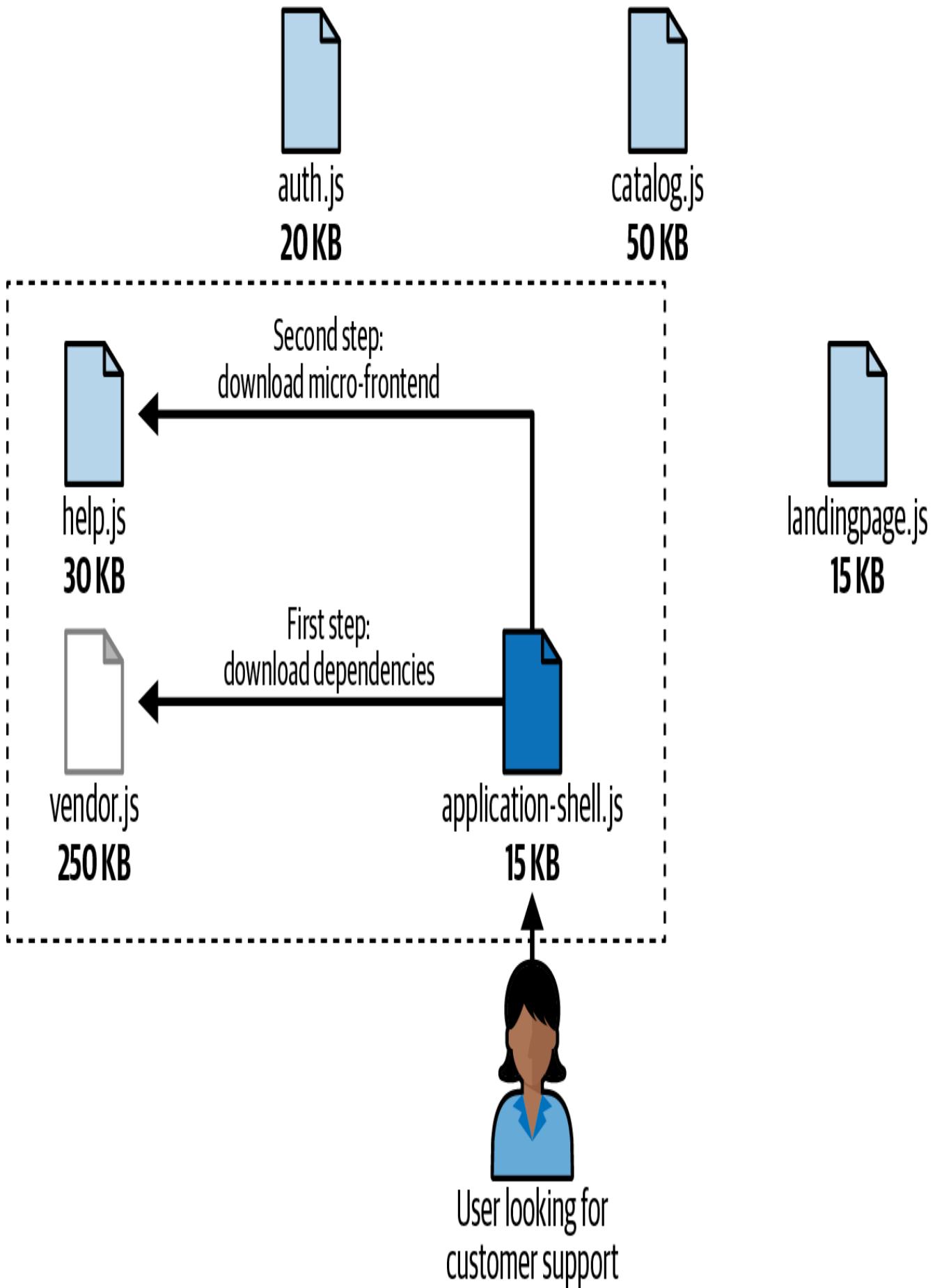


Figure 3-9. A vertical-split micro-frontend, enabling the user to download only the application code needed to accomplish the action the user is looking for

A good practice for managing performance on a vertical-split architecture is introducing a performance budget, which is a limit for micro-frontends that a team is not allowed to exceed.

The performance budget includes the final bundle size, multimedia content to load, and even CSS files.

Setting a performance budget is an important part of ensuring that every team optimizes its own micro-frontend effectively, and it can even be enforced during the continuous integration (CI) process.

You won't set a performance budget until later in the project, but it should be updated whenever there is significant refactoring or additional features are introduced to the micro-frontend codebase.

The time to display the final result to the user is a key performance indicator. Therefore, metrics to track include time-to-interactive, first contentful paint, final artifact size, font size, and JavaScript bundle size, as well as metrics like accessibility and SEO. A tool like [Lighthouse](#) is useful for analyzing these metrics and is available in a [command-line version](#) for use in the CI process.

Although these metrics have been discussed extensively for SPA optimization, bundle size can be trickier when it comes to micro-frontends. With vertical-split architectures, you can decide to bundle either all shared libraries together or the libraries for each micro-frontend.

The former option can provide greater performance because the user downloads the bundle only once, but you'll need to coordinate updates across all the micro-frontends whenever a library changes. While this may sound like an easy task, it can be more complicated than you might expect when it happens regularly. For example, if there is a breaking change in a shared UI framework, you can't update to the new version until all the micro-frontends have done extensive tests on the new framework version. So, while we gain in performance in this scenario, we have to first overcome some organizational challenges.

The latter approach—maintaining every micro-frontend independently—reduces the communication overhead for coordinating the shared dependencies but may increase the amount of content the user must download. As noted earlier, however, a user may spend

the entire session within the same micro-frontend, resulting in the same number of kilobytes being downloaded.

Once again, there isn't a right or wrong approach in any of these strategies.

Make a decision on the requirements you need to fulfill and the context in which you are operating. Don't be afraid to make a call and monitor how users interact with your application. You may discover that, overall, the solution you picked (despite some pitfalls) is the best fit for the project.

Remember, you can always reverse this decision. Spend the appropriate amount of time evaluating which path your project requires, but be aware that you can change direction if new requirements arise or if the decision causes more harm than benefit.

Available Frameworks

There are some frameworks available for embracing this architecture, but building an application shell on your own won't require too much effort, as long as you keep it decoupled from any micro-frontend business logic. Polluting the application shell codebase with domain logic is not only a bad practice but can also undermine all the effort and long-term benefits of using micro-frontends due to code and logic coupling.

The main framework that is fully embracing this architecture is [single-spa](#). The concept behind single-spa is very simple; it's a lightweight library that provides undifferentiated heavy lifting for the following:

Registration of micro-frontends

The library provides a root configuration to associate a micro-frontend with a specific path of your system.

Life cycle methods

Every micro-frontend is exposed to many stages when mounted. Single-spa allows a micro-frontend to perform the right task for the life cycle method. For instance, when a micro-frontend is mounted, we can apply logic for fetching an API. When unmounted, we should remove all the listeners and clean up all the DOM elements.

Single-spa is a very mature library, with years of refinement and many integrations in production. It's open source and actively maintained, and has a great community behind it.

In the latest version of the library, you can develop horizontal-split micro-frontends too, including server-side rendering ones. Qiankun is built on top of single-spa, adding some functionality from the latest single-spa releases.

Module Federation and **Native Federation**, introduced earlier, are also good alternatives for implementing a vertical-split architecture. These approaches are becoming increasingly popular among companies adopting micro-frontends. Both approaches enable seamless composition of independently developed applications. They excel at dynamically loading shared libraries and micro-frontends at runtime and enabling efficient resource sharing. These tools are particularly effective in vertical splits, where each micro-frontend represents a fully-fledged application tied to a specific business domain, enabling teams to work autonomously while maintaining a cohesive user experience.

Architecture Characteristics

The following architecture characteristics should be taken into account:

Deployability (5/5)

Because every micro-frontend is a single HTML page or an SPA, we can easily deploy our artifacts on cloud storage or an application server and stick a CDN in front of it. It's a well-known approach, used for several years by many frontend developers for delivering their web applications. Even better, when we apply a multi-CDN strategy, our content will always be served to our user, no matter what faults a CDN provider may have.

Modularity (3/5)

This architecture is modular but not as composable as the horizontal-split approach. While we have a certain degree of modularization and reusability, it's more at the code level, such as for components or libraries.

Moreover, when we have to divide a vertical-split micro-frontend into two or more parts because of new features, a bigger effort will be required for decoupling all the shared dependencies implemented because it was designed as a unique logical unit.

Simplicity (4/5)

Taking into account that the main aim of this approach is to reduce the team's cognitive load and create domain experts using well-known practices for frontend developers, the simplicity is intrinsic. There aren't too many mindset shifts or new techniques to learn to embrace this architecture. The overhead for starting with single-spa or Module Federation should be minimal for a frontend developer.

Testability (4/5)

Compared to SPAs, this approach shows some weakness in the application shell end-to-end testing. Apart from that edge case, however, testing vertical-split micro-frontends doesn't represent a challenge with existing knowledge of unit, integration, or end-to-end testing.

Performance (4/5)

You can share the common libraries for a vertical-split architecture, though it requires at least some coordination across teams. As it's very unlikely that you'll have hundreds of micro-frontends with this approach, you can easily create a deployment strategy that decouples the common libraries from the micro-frontend business logic while keeping commonalities in sync across multiple micro-frontends.

Developer experience (4/5)

A team familiar with SPA tools won't need to shift their mindset to embrace the vertical split. There may be some challenges during end-to-end testing, but all the other engineering practices, as well as tools, remain the same.

Not all the tools available for SPA projects are suitable for this architecture, so your developers may need to build some internal tools to fill the gaps. However, the out-of-the-box tools available should be enough to start development, enabling your team to defer the decisions to build new tools.

Scalability (5/5)

The scalability aspect of this architecture is so strong that we can almost forget about it when we serve our static content via a CDN. We

can also configure the time-to-live according to the assets we are serving, setting a higher time for assets that don't change often—like fonts or vendor libraries—and a lower time for assets that do change often, like the business logic of our micro-frontends.

This architecture can scale almost indefinitely based on CDN capacity, which is usually great enough to serve billions of users simultaneously. In certain cases, when you absolutely must avoid a single point of failure, you can even create a multiple-CDN strategy where your micro-frontends are served by multiple CDN providers. Despite being more complicated, it solves the problem elegantly without investing too much time in creating custom solutions.

Coordination (4/5)

This architecture, compared to others, enables strong decentralization of decision making as well as team autonomy.

Usually, the touchpoints between micro-frontends are minimal when the domain boundaries are well defined. Therefore, there isn't too much coordination needed, apart from an initial investment for defining the application shell APIs and keeping them as domain-agnostic as possible.

Table 3-1 gathers the architecture characteristics and their associated scores for this micro-frontend architecture.

Table 3-1. Architecture characteristics summary for developing a micro-frontend architecture using vertical split and application shell as composition and orchestrator

Architecture characteristics	Score out of 5
Deployability	5/5
Modularity	3/5
Simplicity	4/5
Testability	4/5
Performance	4/5
Developer experience	4/5
Scalability	5/5
Coordination	4/5

Horizontal-Split Architecture

Horizontal-split architecture offers great flexibility for micro-frontend applications, enabling a granular level of modularization by enabling multiple teams to work on different parts of the same view. This approach makes it possible to compose views by reusing micro-frontends built by various teams across the organization. A good starting point is to keep the split coarse-grained—dividing responsibilities at a higher level and then refining it over time as the application evolves. Prematurely splitting views into too many micro-frontends can lead to unnecessary complexity and increased coordination overhead between teams.

Horizontal-split architectures are recommended not only for companies that already have a sizable engineering department, but also for projects that have a high level of

code reusability. For example, this approach could be utilized in a multi-tenant B2B project where one customer requests a customization or an ecommerce platform with multiple categories that have small differences in behavior and user interface. Your team can easily build a personalized micro-frontend just for that customer and for that domain only. This reduces the risk of introducing bugs in different parts of the applications, thanks to the isolation and independence that every micro-frontend should maintain.

At the same time, due to this high modularization, horizontal-split architecture is one of the most challenging implementations because it requires solid governance and regular reviews to get the micro-frontend boundaries right. Moreover, these architectures challenge the organization's structure unless they are well thought out upfront. It's very important with these architectures that we review the communication flows and the team structure to enable the developers to do their jobs and avoid too many external dependencies across teams.

In addition, we need to share best practices and define guidelines to maintain a good level of freedom while providing a unique, consolidated experience for the user.

One of the recommended practices when we use horizontal-split architectures is to reduce the number of micro-frontends in the same view, especially when multiple teams have to merge their work. If you find yourself using more than six or seven micro-frontends in a single view, it may indicate that you're actually implementing remote components rather than true micro-frontends. This may sound obvious, but there is a real risk of overengineering the solution, resulting in several tiny micro-frontends living together in the same view, which creates an anti-pattern. This is because you are blurring the line between a micro-frontend and a component (as discussed in [Chapter 2](#)), where the former is a business representation of a subdomain and the latter is a technical solution used for reusability purposes.

Moreover, managing the output of multiple teams in the same view requires additional coordination at several stages of the software development life cycle.

Another sign of overengineering a page is having multiple micro-frontends fetching from the same API. In that case, there is a good chance that you have pushed the division of a view too far and need to refactor.

Remember that embracing these architectures provides great power and, therefore, we have great responsibility for making the right choices for the project.

In the next sections, we will review the different implementations of horizontal-split architectures: client side, edge side, and server side.

Client Side

A client-side implementation of the horizontal-split architecture is similar to the vertical-split one in that there is an application shell used for composing the final view. The key difference is that, here, a view is composed of multiple micro-frontends, which can be developed by the same or different teams.

Due to the horizontal split's modular nature, it's important not to fall into the trap of thinking too much about components. Instead, stick to the business perspective. Let's take a look at an example.

Imagine you are building a video-streaming website, and you decide to use a horizontal-split architecture with a client-side composition. There are several teams involved in this project but, for simplicity, we will only consider two views: the landing page and the catalog. The bulk of the work for these two experiences involves the following teams:

Foundation team

This team is responsible for the application shell and the design system, working alongside the UX team but from a more technical perspective.

Landing page team

This team supports the marketing team to promote the streaming service and creating all the different landing pages needed.

Catalog team

This team is responsible for the authenticated area where a user can consume a video on demand, working in collaboration with other teams to provide a compelling experience to the service subscribers.

Playback experience team

Considering the complexity of building a great video player available on multiple platforms, the company decides to establish a team dedicated to the playback experience. The team is responsible for the video player, video analytics, implementation of digital rights management (DRM), and additional security concerns related to the video consumption from unauthorized users.

When it comes to implementing one of the many landing pages, we have three teams responsible for the final view to present to every user. The foundation team provides the application shell, the footer, and the header, and composes the other micro-frontends present in the landing page. The landing page team provides the streaming service offering, with additional details of the video platform. Finally, the playback experience team provides the video player for delivering the advertising needed to attract new users to the service. This is visualized in [Figure 3-10](#).

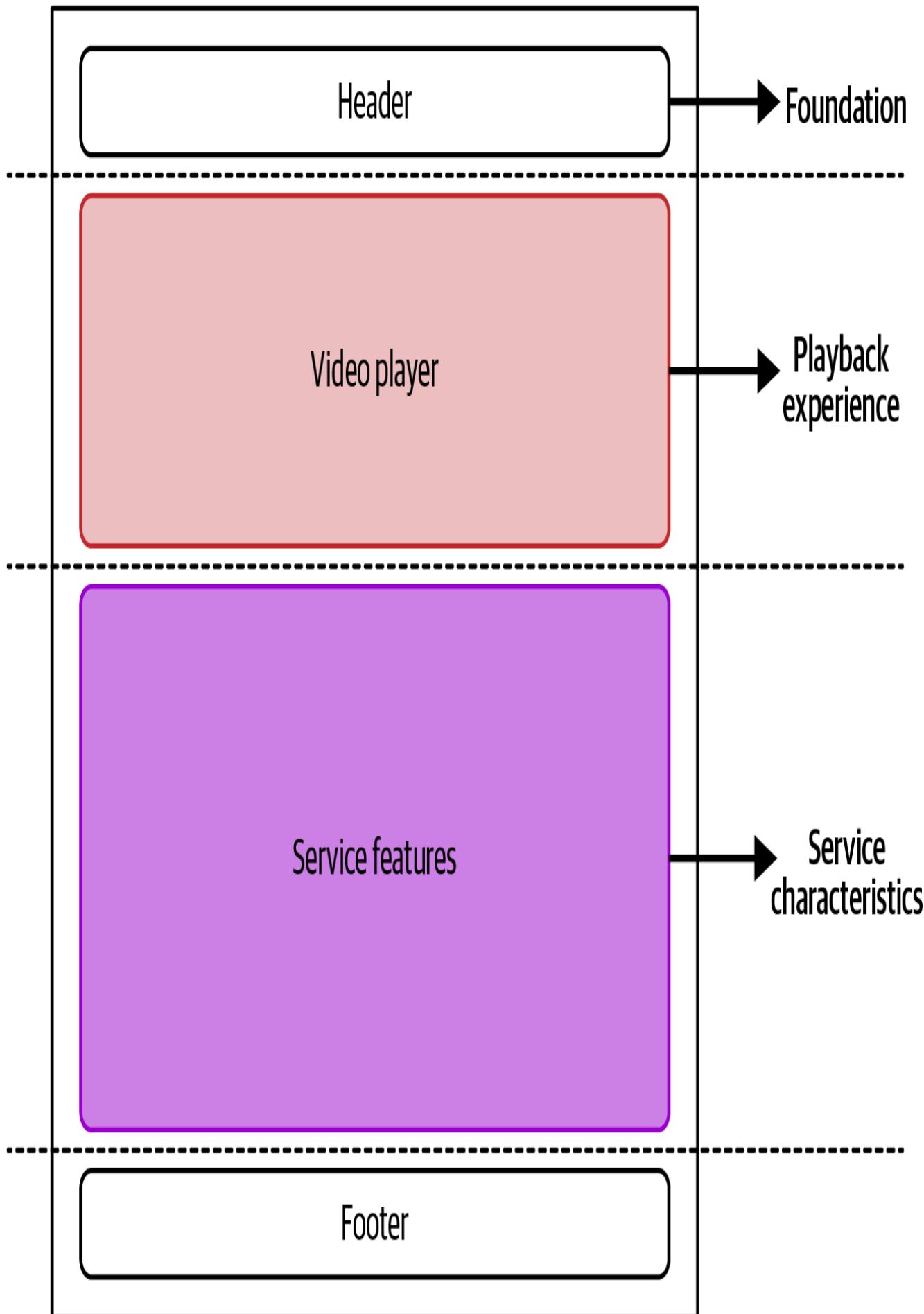


Figure 3-10. The landing page view, composed by the application shell, which loads two micro-frontends: the service characteristics and the playback experience

This view doesn't require particular communication between micro-frontends, so once the application shell is loaded, it retrieves the other two micro-frontends and provides the composed view to the user.

When a subscriber wants to watch any video content, after being authenticated, they will be presented with the catalog that includes the video player, as shown in [Figure 3-11](#).

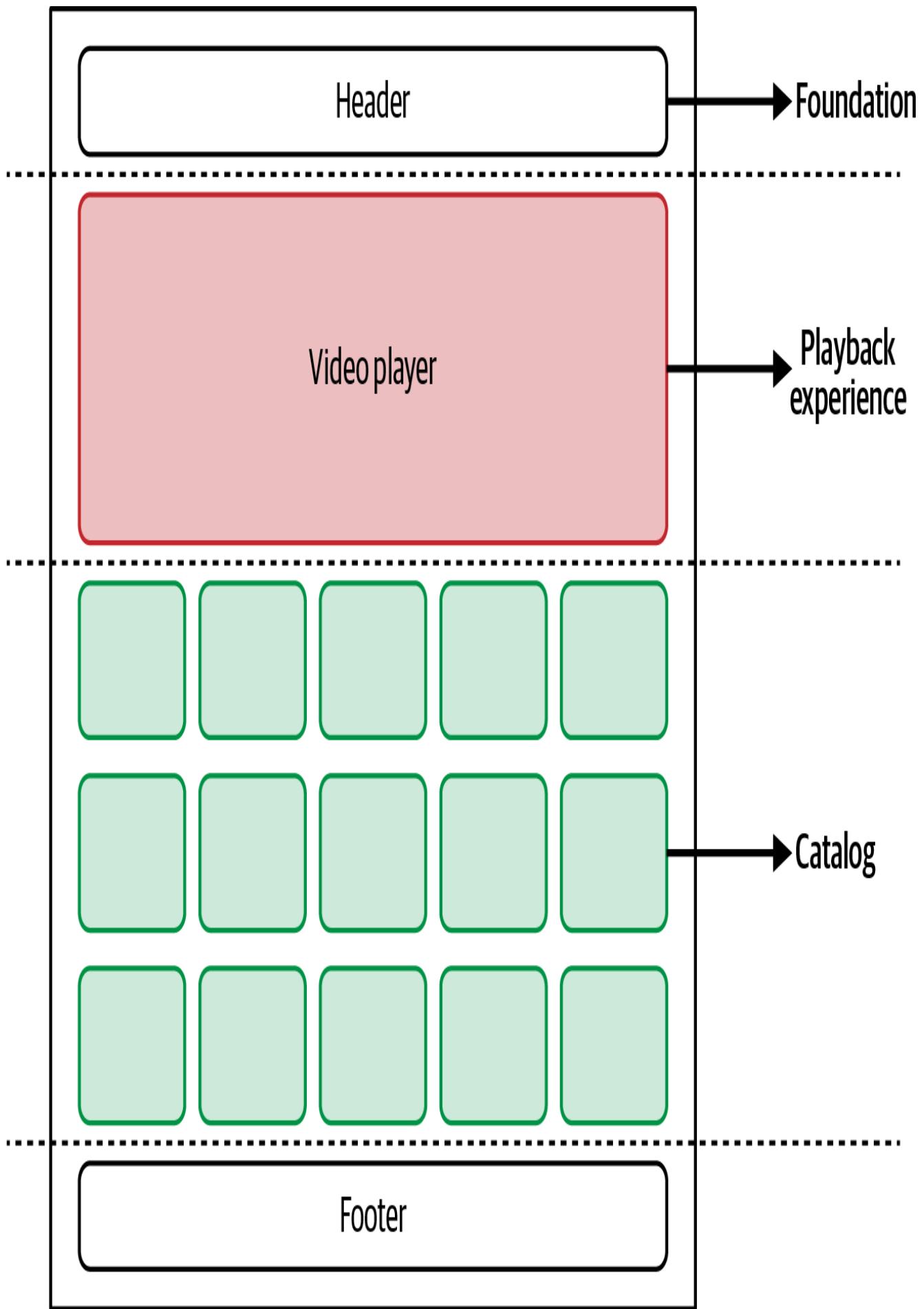


Figure 3-11. The catalog view composed by the application shell, which loads two micro-frontends: the playback experience and the catalog itself

In this case, every time a user interacts with a tile to watch the content, the catalog micro-frontend has to communicate with the playback micro-frontend to provide the selected video's ID. When an error has to be displayed, the catalog team is responsible for triggering a modal with the error message for the user. And when the playback has to trigger an error, the error will need to be communicated to the catalog micro-frontend, which will display it in the view. This means we need a strategy that keeps the two micro-frontends independent but allows communication between them when there is a user interaction or if an error occurs.

There are many strategies available to solve this problem, like using custom events or an event emitter, but we will discuss the different approaches later on in this chapter.

So, why wasn't there a specific composition strategy for this example? That is mainly because every client-side architecture has its own approach to composing a view. In addition, in this case, we will see the best practice for doing so, architecture by architecture.

Do you want to discover where the horizontal-split architecture really shines?

Let's fast-forward a few months to after the release of the video-streaming platform. The product team requests an unauthenticated version of the catalog to improve the discoverability of platform assets and to provide a preview of the best shows to potential customers. This boils down to offering a similar catalog experience without the playback features. The product team also wants to add more information to the landing page so users can make an informed decision about subscribing to the service.

In this case, the foundation team, catalog team, and landing page team will be needed to fulfill this request. This new view is shown in [Figure 3-12](#).

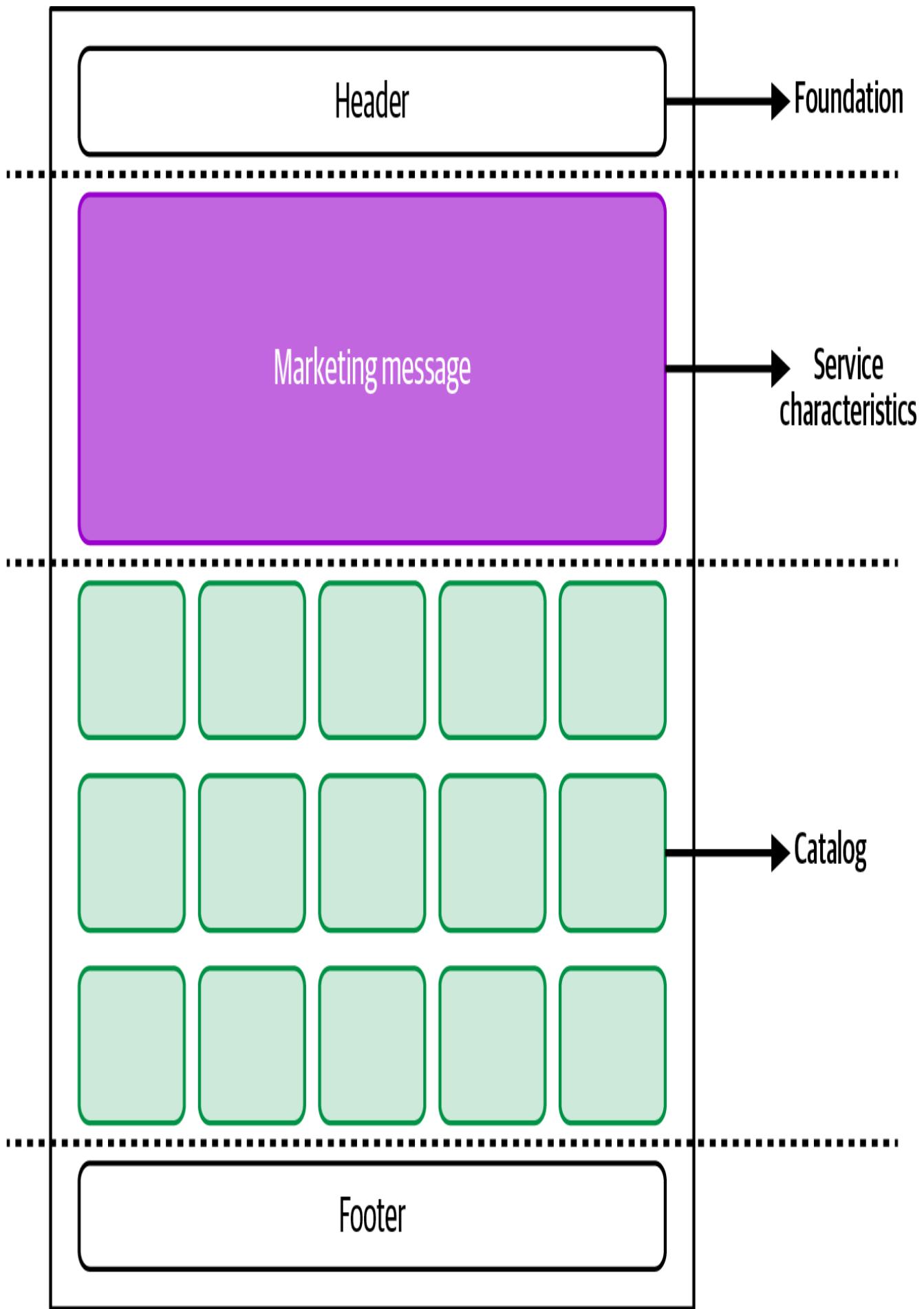


Figure 3-12. New view composed of the catalog (owned by the catalog team) and the marketing message (owned by the landing page team)

Evolving a web application is never easy, for both technical and collaboration reasons. Having a way to compose micro-frontends simultaneously and then stitching them together in the same view—with multiple teams collaborating without stepping on each other’s toes—makes life easier for everyone and enables the business to evolve at speed and in any direction.

Challenges

As with every architecture, horizontal splits have both benefits and challenges that are important to understand to ensure they are a good fit for your organization and projects. Evaluating the trade-offs before embarking on a development puts you one step closer to delivering a successful project.

Micro-frontend communication

Embracing a horizontal-split architecture requires understanding how micro-frontends developed by different teams share information, or state, during the user session.

Inevitably, micro-frontends will need to communicate with each other. For some projects, this may be minimal, while in others it will be more frequent. Either way, you need a clear strategy up front to address this challenge.

Many developers may be tempted to share state between micro-frontends, but this creates a sociotechnical antipattern.

On the technical side, working with a distributed system that has shared code owned by different teams means that the shared state must be designed, developed, and maintained collaboratively by multiple teams. See [Figure 3-13](#) for an example.

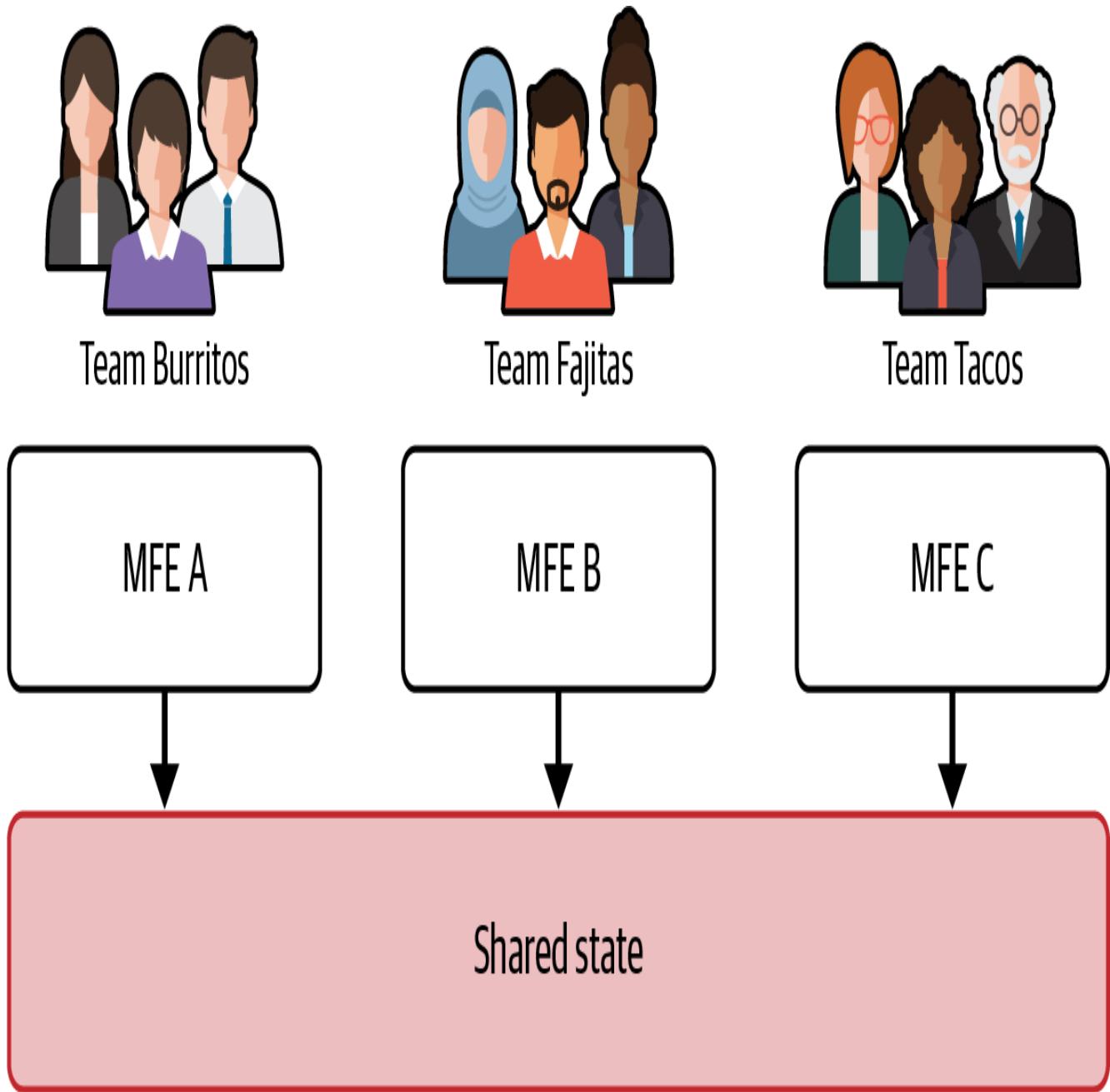


Figure 3-13. Shared state between multiple micro-frontends, representing an antipattern

Every time a team makes a change to the shared state, all other teams must validate the change to ensure it doesn't impact their micro-frontends. Such a structure breaks the encapsulation that micro-frontends provide, creating an underlying coupling between teams with frequent, if not constant, external dependencies to manage.

Moreover, we risk jeopardizing the agility and evolution of the system because a key part of a micro-frontend is now shared across multiple micro-frontends. Even worse, when a micro-frontend is reused across multiple views, a team may need to be responsible for maintaining multiple shared states with other micro-frontends.

On the organizational side, this approach introduces team coupling, resulting in the need for significant coordination that could be avoided while maintaining the boundaries of each micro-frontend.

The coordination between teams doesn't stop in the design phase, either—it becomes even more exasperating during testing and release because all the micro-frontends in the same view depend on the same state, which cannot be released independently.

Having constant coordination to handle shared state instead of maintaining a micro-frontend's independent nature can be a team's worst nightmare. In the microservices world, this is called a distributed monolith: an application deployed like a microservice but built like a monolith.

One of the main benefits of using micro-frontends is the strong boundaries that enable each team to move at their own speed—loosely coupling the organization, reducing coordination time, and enabling developers to take control.

In the microservices world, to achieve loose coupling between microservices—and therefore between teams—we use the choreography pattern, which relies on an asynchronous communication, or an event broker, to notify all the consumers interested in a specific event. With this approach, we achieve the following:

- Independent microservices that can choose whether or not to react to external events triggered by one or more producers
- A solid, bounded context that doesn't leak into multiple services
- Reduced communication overhead for coordinating across teams
- Agility for every team, enabling them to evolve their microservice based on their customers' needs

With micro-frontends, we should approach things in the same way to achieve similar benefits. Instead of using a shared state, we maintain each micro-frontend's boundaries and communicate any events that need to be shared within the view using asynchronous messages—something we're used to dealing with on the frontend.

Other options include implementing either an event emitter or a reactive stream (if you are in favor of the reactive paradigm) and sharing it across all the micro-frontends in a view, as shown in [Figure 3-14](#).

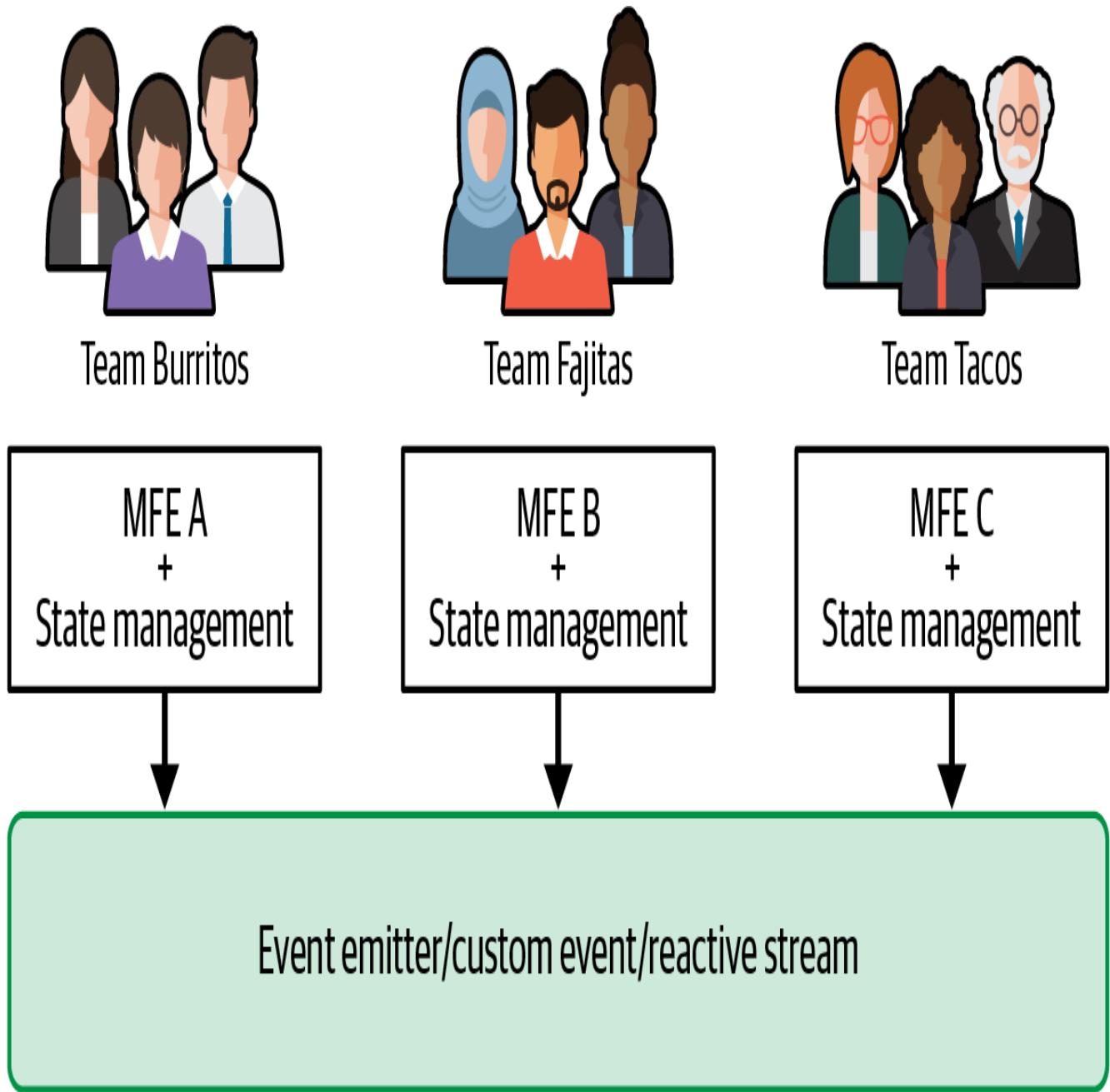


Figure 3-14. Every micro-frontend in the same view, owning its own state and communicating changes via asynchronous communication using an event emitter, custom event, or reactive streams

In [Figure 3-14](#), Team Fajitas is working on a micro-frontend (MFE B) that needs to react when a user interacts with an element in another micro-frontend (MFE A), which is run by Team Burritos. Using an event emitter, Team Fajitas and Team Burritos can define how the event name and the associated payload will look and then implement them, working in parallel—as shown in [Figure 3-15](#).

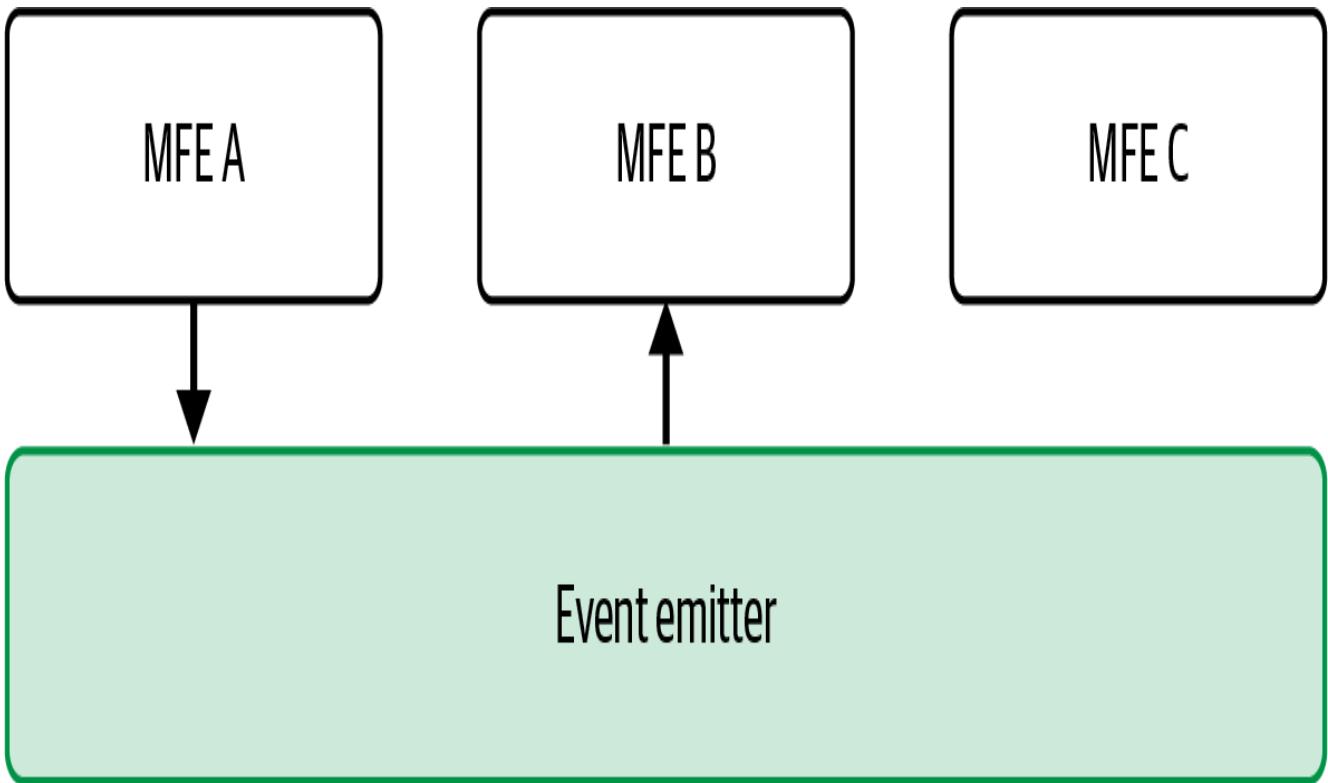


Figure 3-15. MFE A emitting an event using the event emitter as a communication bus, with other micro-frontends interested in the event listening and reacting accordingly

When the payload changes for additional features implemented in the platform, Team Fajitas will need to make a small change to its logic and can then start integrating these features without waiting for other teams to make any changes, maintaining its independence.

The third micro-frontend in our example (MFE C, run by Team Tacos) doesn't care about any event shared in that view because its content is static and doesn't need to react to any user interactions. Team Tacos can continue to do its job knowing its part won't be affected by any state change associated with a view.

A few months later, a new team (Team Nachos) is created to build an additional feature in the application. **Figure 3-16** shows how Team Nachos' micro-frontend (MFE D) lives alongside MFE A and MFE B.

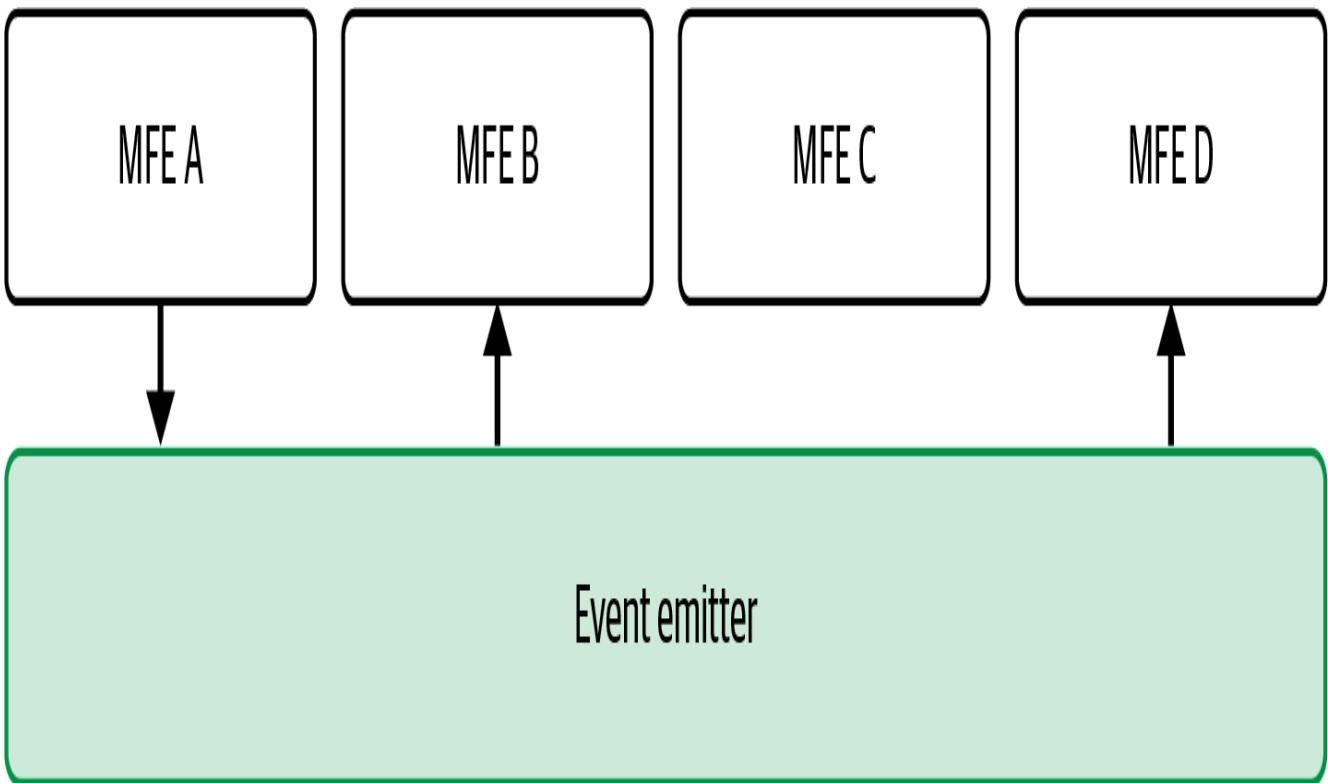


Figure 3-16. A new micro-frontend added in the same view that has to integrate with the rest of the application, reacting to the event emitted by MFE A.

Because of the loose-coupling nature of this approach, MFE D just listens to the events emitted by MFE A. In this way, all micro-frontends and teams maintain their independence. Every micro-frontend is well encapsulated, and the only communication protocol is a pub/sub system like the event emitter, so the new team can easily listen to all the events it needs to for plugging the new feature alongside the existing micro-frontends.

This approach not only enhances the technical architecture but also creates loose coupling between teams while enabling them to continue working independently. Once again, we notice how important our technology choices are for maintaining independent teams and reducing external dependencies that could cause frustration.

In addition, having the team document all input and output events for every horizontal-split micro-frontend will help facilitate the asynchronous communication between teams. Providing an up-to-date, self-explanatory list of contracts for communicating in and out of a micro-frontend will result in clear communication and better governance of the entire system.

What these processes help achieve is faster delivery, independent teams, agility, and a high degree of evolution for every micro-frontend without affecting others.

Clashes with CSS class names and how to avoid them

One potential issue in horizontal-split architecture during implementation is CSS class name clashes. When multiple teams work on the same application, there is a strong possibility of having duplicate class names, which would break the final application layout.

To avoid this risk, we can prefix each class name for every micro-frontend, creating a strong rule that prevents duplicate names and, therefore, avoids undesired outcomes for our users.

Block Element Modifier, or **BEM**, is a well-known naming convention for generating unique names for CSS classes. As the name suggests, we assign three elements to a component in a micro-frontend:

Block

This is an element in a view. For example, an avatar component is composed of an image, the avatar name, and so on.

Element

This is a specific element of a block. In the previous example, the avatar image is an element.

Modifier

This is a state to display. For instance, the avatar image can be active or inactive.

Based on the example described, we can derive the following class names:

```
.avatar {}
.avatar_image {}
.avatar_image--active {}
.avatar_image--inactive {}
```

While following BEM can be extremely beneficial for architecting your CSS strategy, it may not be enough for projects with multiple micro-frontends. So, we build on the BEM structure by prefixing the micro-frontend name to the class.

For our avatar example, when it's used in the “my account” micro-frontend, the names become:

```
.myaccount_avatar {}
.myaccount_avatar_image {}
```

```
.myaccount_avatar_image--active {}
.myaccount_avatar_image--inactive {}
```

Although this results in long names, it guarantees the isolation needed and makes clear what every class refers to.

Multi-framework approach

Using multiple frameworks isn't ideal for vertical-split architectures due to performance issues. On horizontal-split architectures, it's even more dangerous.

When this issue is not addressed in the design phase, it can cause runtime errors in the final view.

Imagine having multiple versions of React in the same view. This does not provide a good experience for the user. When the browser downloads two versions for a rendering view, performance issues can crop up. Consider, too, the potential variable clashes when loading libraries or appending new components in the view.

There are several approaches to address this problem. Iframes, for instance, create a sandbox so that what loads inside one iframe doesn't clash with another. Module Federation allows you to share libraries and avoid clashing dependencies between different versions of the same library. Import maps let us define **scopes for each dependency**, enabling different versions of the same library for different scopes. And web components can "hide" the frameworks needed for a micro-frontend behind the "shadow" DOM.

While these solutions exist, using a multi-framework approach is still strongly discouraged due to performance concerns. Having more kilobytes to download to render a page reduces the user experience, and our job as developers and architects is to provide the best possible experience. Using multiple frameworks isn't acceptable in other frontend architectures, such as SPAs, and micro-frontends should be no exception.

There are only two valid scenarios for adopting a multi-framework strategy. One is migrating a legacy application to a new one, where micro-frontends are released iteratively rather than all at once. In this case, a multi-framework approach delivers customer value while reducing deployment risks. The other scenario is when a company acquires another and needs to capitalize on the investment quickly by integrating the acquired software into its micro-frontend architecture.

Authentication

Horizontal-split architecture presents an interesting challenge when it comes to system authentication, because (more often than not) multiple teams are working on the same view, and they need to maintain a unique experience for the customer. When a user enters an authenticated area of a web application, all the micro-frontends composing the page have to communicate with the respective APIs providing tokens.

Let's say we have three different teams creating a micro-frontend, each composing a view for the customer. As shown in [Figure 3-17](#), these micro-frontends must fetch data from the backend, which is a distributed system composed of multiple microservices.

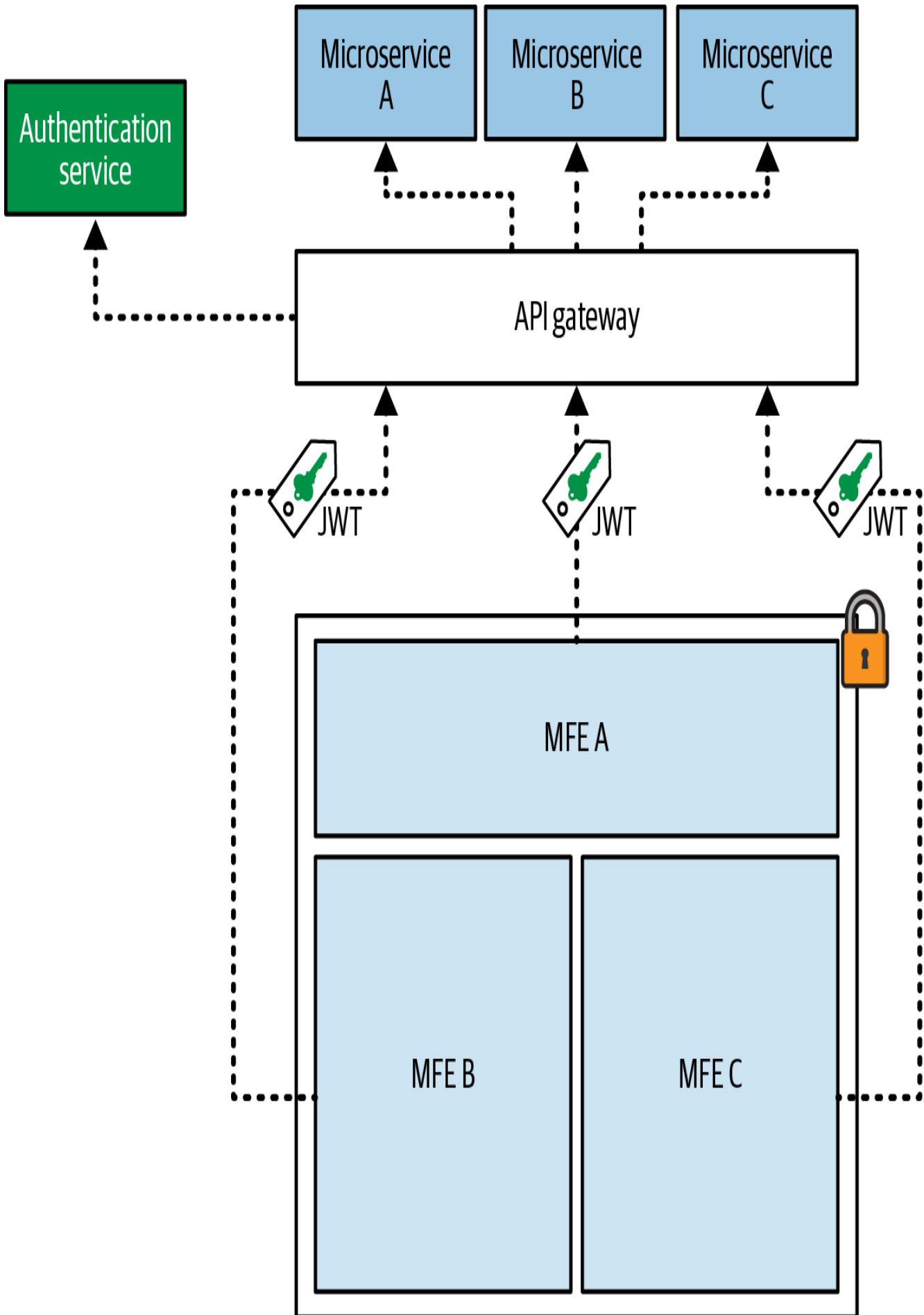


Figure 3-17. Every micro-frontend in a horizontal-split architecture fetching data from an API, passing a JSON Web Token (JWT) to the backend to validate the user

How can different micro-frontends retrieve and store a token safely without multiple round trips to the backend? The best options are storing the token in `localStorage`, `sessionStorage`, or a cookie. In this case, all micro-frontends retrieve the same token from the chosen web storage solution by convention.

Different security restrictions apply depending on the web storage selected for hosting the token. For instance, if we use `localStorage` or `sessionStorage`, all micro-frontends must be hosted on the same subdomain; otherwise, the storage containing the token won't be accessible. In the case of cookies, we can configure the domain attribute to make them accessible across multiple subdomains (e.g., `beta.mysite.com` or `preview.mysite.com`). By setting the domain to `.mysite.com`, the browser allows the cookie to be shared and fetched by all subdomains under that domain.

We also have to consider that, when multiple micro-frontends consume the same API with the same request body, it's very likely that they can be merged into a unique micro-frontend.

Don't be afraid to review the domain boundaries of micro-frontends, because they will evolve alongside the business. Additionally, because there isn't a scientific way to define boundaries, taking a step back and reassessing the chosen direction is more beneficial than ignoring the problem. The longer we ignore the issue, the more disruption teams will experience. It's far better to invest time early in the project to refactor a bunch of micro-frontends.

Another option is to have the application shell handle the authentication mechanism by creating a middleware that intercepts every HTTP request to an API and appends a new header containing the user's token. While there will be some edge cases where APIs don't require a token, these can be easily resolved in code.

Micro-Frontend Refactoring

A benefit of the horizontal-split architecture is the ability to refactor specific micro-frontends when the code becomes too complicated for a single team to manage, or when a new team takes ownership of a micro-frontend that they didn't originally develop. While this is also possible with a vertical split, horizontal-split micro-frontends contain far less logic to maintain, making them especially beneficial for enterprise organizations that work on the same platform for many years.

Because every micro-frontend is independent, refactoring code to make it more understandable for the team is beneficial, as it won't impact anyone else in the company.

While you need to keep tech leadership's guidelines in mind, refactoring a well-designed micro-frontend requires far less time than refactoring a large monolithic codebase. This characteristic makes micro-frontends more maintainable in the long run. Additionally, when a complete rewrite is needed, having the domain experts—the team—in charge of rewriting something they know thoroughly requires significantly less work than rewriting an unfamiliar application from scratch. And because of the micro-frontends' nature, you can rewrite them iteratively and ship them in production to gain immediate benefits instead of working for several months before releasing everything all at once.

I'm not encouraging refactoring or rewriting just because it's easier. But sometimes the team gains additional business knowledge or must implement a tactical solution due to a tight delivery deadline; in such cases, refactoring or rewriting from scratch can make life easier in the long run, speeding up new-feature development and reducing the likelihood of production bugs.

Search Engine Optimization

Dynamic rendering is another valid technique for this architecture, especially when using iframes to encapsulate our micro-frontends. In that case, redirecting crawlers to an optimized version of static HTML pages helps improve the search engine's ranking. Overall, what has been discussed so far about dynamic rendering also applies to client-side horizontal-split architectures.

Developer Experience

The developer experience (DX) of the horizontal-split architecture with a client-side composition is very similar to the vertical split when a team is developing its own micro-frontend. However, it becomes more complex when the team needs to test micro-frontends together within a view. The main challenge is keeping track of versions and having a quick turnaround for assembling a view on the developer's device.

One option is using the Webpack DevServer proxy (or similar tools) for testing locally, with micro-frontends available in testing, staging, or production environments.

Companies that embrace this architecture often create tools to improve their teams' feedback loop, frequently in the form of command-line tools—like Rollup, Webpack, or Snowpack—that can enhance the standard tools available to the frontend developer community.

It's important to note that this architecture will likely require internal investments to establish a solid DX. Currently, frameworks and tools (e.g., Module Federation) take an opinionated approach; while this isn't necessarily a bad thing, in large companies additional effort is usually needed to maintain the guidelines and standards designed by tech leadership based on the industry the company operates in.

Team Communication and Best Practices for Maintaining Control of the Final Outcome

Although horizontal-split architectures are the most versatile, they also present intrinsic implementation challenges from an organizational point of view—with the main issue being the difficulty in coordinating a final output.

When we have multiple micro-frontends owned by different teams composed in the same view, we have to create a social mechanism for avoiding runtime issues in production due to dependency clashes or CSS classes overriding each other. In addition, observability tools must be added to quickly identify which micro-frontends are failing in production so the team can be provided with clear information for diagnosing the issue in their individual micro-frontends.

The best way to avoid issues is to keep the communication channels open and maintain a fast feedback loop that keeps all teams in sync, such as a weekly or bi-weekly meeting with a member from every micro-frontend team responsible for a view.

Syncing the work between teams has to happen either in a live meeting or via asynchronous communication, such as emails or instant messaging.

We must also reduce the number of teams working on the same page and make one team responsible for the final output presented to the users. This doesn't mean that this team should do all the work; but because shared responsibilities often lead to misunderstandings, having one team leading the effort will result in a better experience for end users.

As we saw in our client-side video player example, three teams are involved in delivering the catalog page. It's very likely that the catalog team would perform any additional checks on the playback experience because, after a user clicks on a movie or

a show, it should play in the video player. In this case, then, the catalog team should be responsible for the final outcome and should coordinate the effort with the playback experience team to provide the best output for their users.

When possible, reducing external dependencies should be a periodic task for an engineering manager or a team lead. Don't automatically accept the status quo. Instead, embrace a continuous improvement mindset and challenge the work done so far to find better ways to serve your customers.

To keep the teams in sync and enable them to discuss potential breaking changes, strongly encourage your teams to document their micro-frontend inputs and outputs, the events a micro-frontend expects to receive, and the events it will trigger. Keeping track of breaking changes using requests for comments (RFCs) or similar documents is strongly recommended for several reasons:

- It creates asynchronous communication between teams, which is especially useful when teams are distributed across time zones.
- It maintains a record of decisions along with the context the company was operating in when the decision was made.
- Not everyone performs well during meetings; sometimes one person monopolizes the discussion, preventing others from sharing their opinion. Moving from verbal to written communications helps ensure everyone's voice is heard.

Use Cases

One reason to embrace the horizontal-split architecture is the reusability of micro-frontends across a single application or multiple applications. Imagine a team responsible for the payment micro-frontend of an ecommerce website, with the micro-frontend containing different states based on the type of view and payments available. The payment micro-frontend appears in every view where a user can perform a payment action, including the landing page, product detail view, or even a subscription page for another product.

This situation is applicable at a larger scale on a B2B application, where similar UX constructs are replicated across multiple system views.

Another use case for this architecture is enterprise applications, which often include dashboards with various types of data that must be collected into different views for different purposes, such as financial or monitoring dashboards. Tech company New

Relic uses this approach to provide monitoring tools for cloud services, as well as a frontend **that implements micro-frontends** for scaling the organization, allowing multiple teams to contribute different data representations, all collected into a unique dashboard.

In [Figure 3-18](#), you can see how New Relic divided its application so that a small number of teams work in the same view, reducing the amount of communication needed for composing the final view, but allowing the team to be well encapsulated inside its business domain.

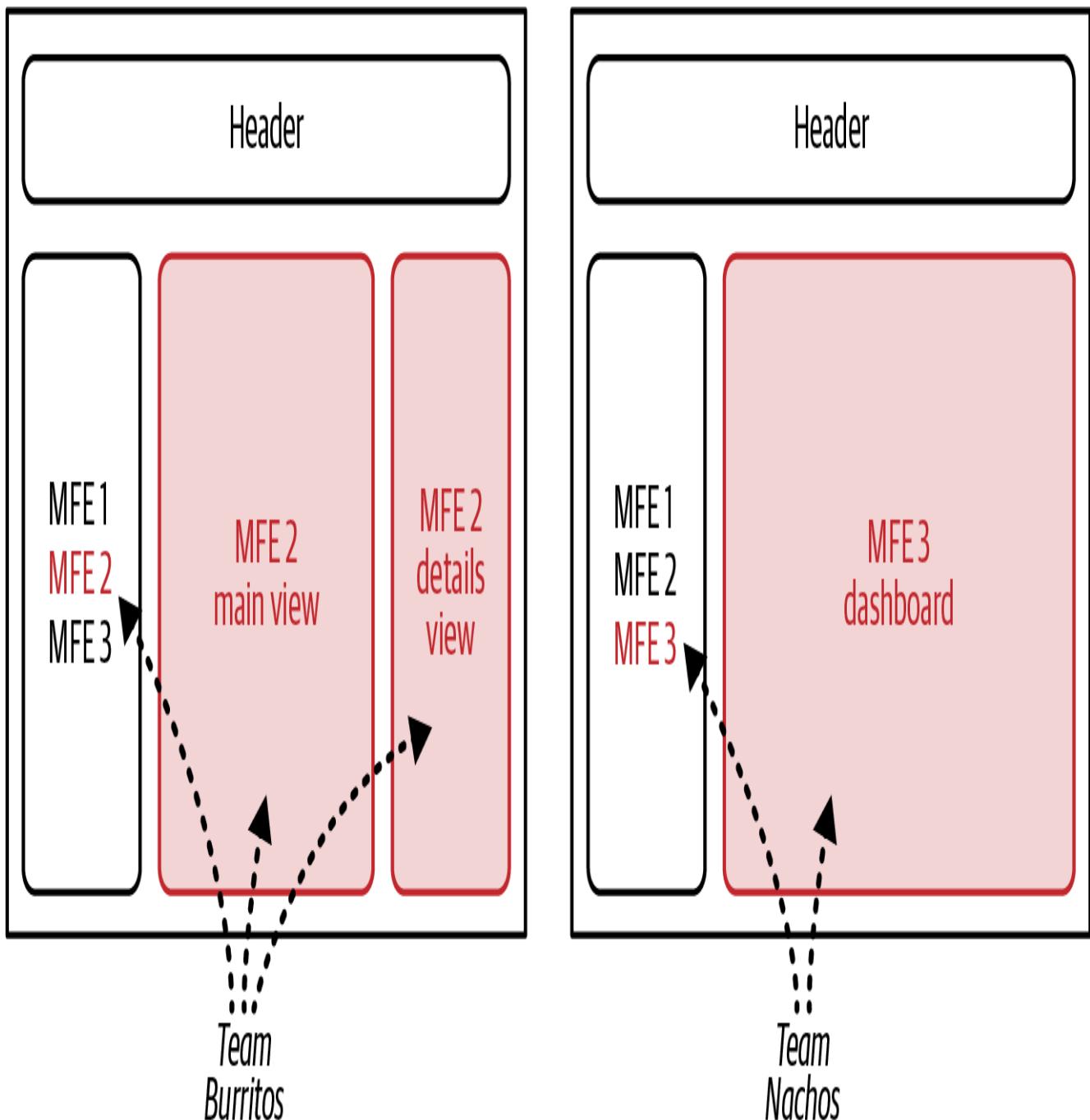


Figure 3-18. New Relic micro-frontends implementation, where each team is responsible for their own domain and, when a user selects a dashboard, the related micro-frontend is lazy-loaded inside the application shell

This approach enables New Relic teams to work on their own micro-frontends and, by following deployment contracts, they can see the final results directly in their web application.

The final use case for this architecture is when we are developing a multi-tenant application where most of the interface is the same, but customers can build specific features to make the software suitable for their organization's needs.

For example, let's say we are developing a digital till system for restaurants, and we want to configure the tables on the floor on a customer-by-customer basis. The application will have the same functionality for every single customer, but a restaurant chain can request specific features in the digital till system. The micro-frontend team responsible for the application can implement these features without forking the code for every customer; instead, they will create a new micro-frontend for handling the specific customer's needs and deploy it in their tenant.

Available Frameworks

With horizontal splitting, we have multiple options because rendering can happen on either the client or server side. Therefore, we have a wider range of frameworks available out of the box. Let's explore what our options are.

Module Federation

Micro-frontend architectures received a great gift with the release of Webpack 5: a new native plug-in called Module Federation.

With version 2.0, Module Federation changed heavily. In fact, it's no longer bound to a specific bundler and is now available for a variety of them, including Rspack, Rolldown, Webpack, Vite, and many others in the future.

Thanks to these enhancements, you can now combine Module Federation modules exported by different bundlers within the same application. Therefore, in a client-side rendering scenario where you want to apply a horizontal split of your UI, it becomes a really solid solution. Moreover, the main maintainer of this library moved to ByteDance —the company behind TikTok and CapCut. Considering the heavy usage of Module Federation at ByteDance, a full-fledged team of engineers now works on this project. This is another important indicator to highlight when pitching the idea of micro-frontends to your tech leadership.

Embedding an open source project heavily sponsored by a strong company and by the wider community is definitely a major advantage in favor of Module Federation.

Module Federation allows chunks of JavaScript code to load synchronously or asynchronously, meaning multiple developers or even teams can work in isolation and take care of the application composition, lazy-loading different JavaScript chunks behind the scenes at runtime. This is shown in [Figure 3-19](#).

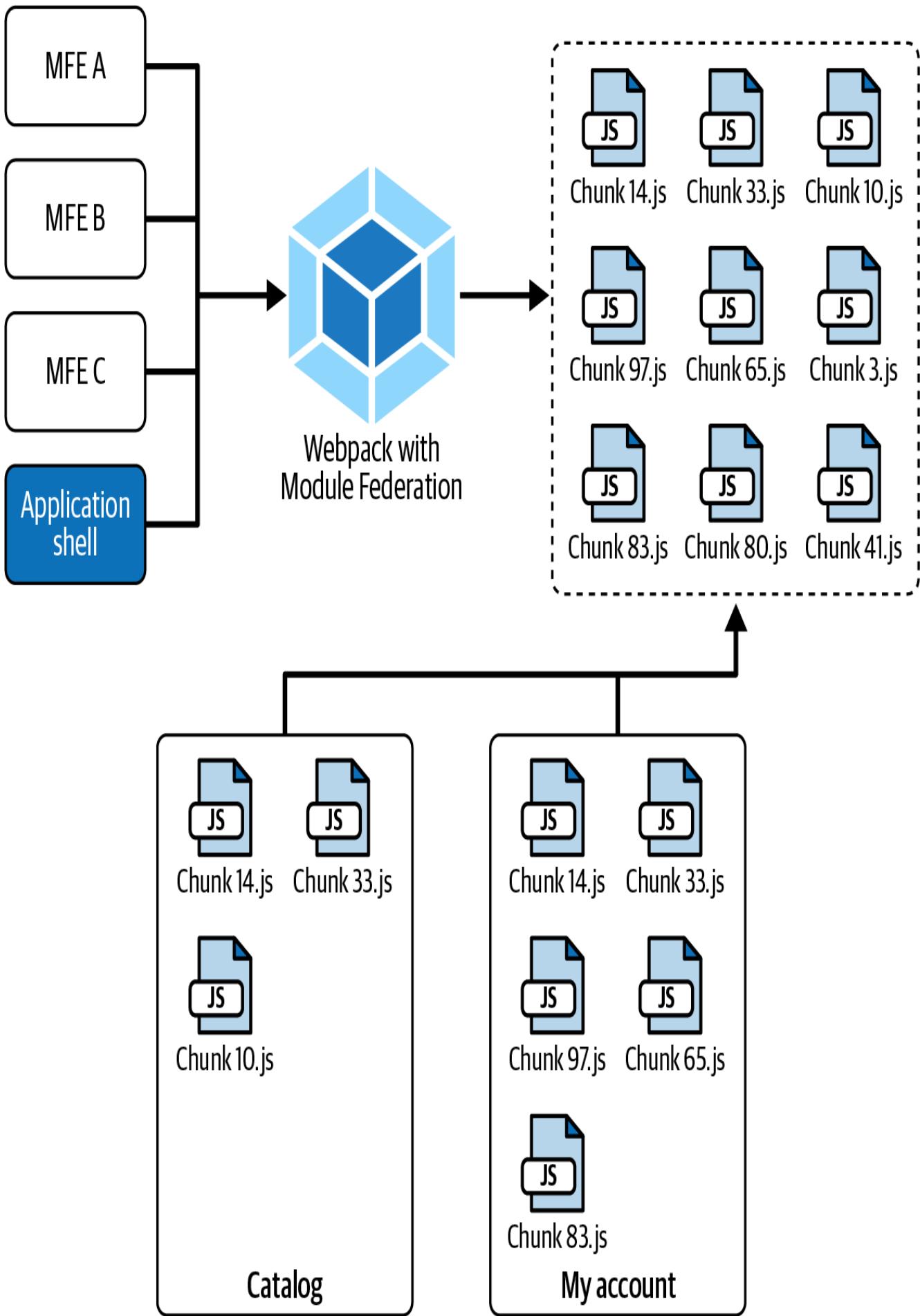


Figure 3-19. Module Federation enabling multiple micro-frontends to be loaded asynchronously, providing the user with a seamless experience

A Module Federation application is composed of two parts:

- The *host* represents the container of one or more micro-frontends or libraries loaded.
- The *remote* represents the micro-frontend or library that will be loaded inside a host at runtime. A remote exposes one or more objects that can be used by the host when the remote is lazy-loaded into an application.

Module Federation truly shines in its simplicity—exposing different micro-frontends or even shared libraries, such as a design system, for easy asynchronous integration.

The developer experience is incredibly smooth, almost transparent within an application. As with a monolithic codebase, you can import remote micro-frontends and compose a view as needed.

Testing locally or pointing to a specific online endpoint makes no difference, because we can work similarly to handling multiple environments, with every bundler having a shared configuration augmented by environment-specific settings (test, stage, or production).

We will discuss potential solutions to tackle the multi-environment challenge in [Chapter 9](#).

Another important feature of Webpack with Module Federation is its ability to share external libraries across multiple micro-frontends without risking runtime clashes. We can specify which libraries are shared, and Module Federation will load just one version for all the micro-frontends using the library.

Imagine all your micro-frontends are using Vue.js 3.0.0. With Module Federation, you only need to specify that Vue version 3 is a shared singleton library. Then, at runtime, Module Federation will load Vue.js just once, as long as all other remotes are using the same library version.

What if you want to intentionally work with different versions of Vue in the same project?

Module Federation will wrap each version in a different scope, called a container, to avoid runtime clashes. You can also manually specify a scope for different versions of the same library using Module Federation APIs.

Module Federation is available not only for fully client-side applications but also for server-side rendering. We can asynchronously load different components without needing to redeploy the application server that composes the page and serves the final result to a client request.

Unfortunately, the great simplicity of code sharing across projects is also this plug-in's weakest point. When working with an undisciplined team, sharing libraries, code snippets, and micro-frontends across multiple views can result in a very complicated architecture to maintain, due to the frictionless integration. Therefore, it's critical to create guidelines that follow the micro-frontend decision framework to avoid regretting the freedom that Module Federation provides.

Performance

Module Federation is very transparent. Therefore, there is no particular overhead applied to your systems. One of the main challenges we face when working with micro-frontends is how to share dependencies across this distributed architecture—and Module Federation can help there, too. Let's say you have multiple teams working on the same application. Each team owns a single micro-frontend, and the teams have agreed to use the same UI library for the entire application. You can share these libraries automatically with Module Federation from the plug-in configuration, and they'll be loaded only once at the beginning of the project.

You can also load micro-frontends dynamically inside JavaScript logic instead of defining all of them in the Webpack configuration file.

Composition

Using Module Federation for a micro-frontend architecture is as simple as importing an external JavaScript chunk lazy-loaded inside a project. Composition takes place at runtime, either on the client side (when we use an application shell for loading different micro-frontends) or on the server side (when we use server-side rendering). When we load a micro-frontend on an application shell at runtime, we can fetch the micro-frontend directly from a CDN or from an application server. And the same is true when we are working with a server-side rendering architecture. In this case, composition takes place at the origin, and we can load micro-frontends at runtime before serving them to a client request.

In the next chapter, we will dive deeply into Module Federation composition, providing more insights into how to achieve horizontal- and vertical-split composition with code examples.

Shared code

Module Federation makes sharing code very simple, providing a frictionless developer experience. However, we should carefully consider why we are embracing micro-frontends in the first place.

It allows bidirectional sharing across micro-frontends, therefore flattening the hierarchical nature of an application—where a host micro-frontend can share code with a remote micro-frontend, and vice versa.

However, I tend to discourage this bidirectional practice, because a unidirectional implementation brings several advantages:

- Code is easier to debug, as we always know what code is coming from where.
- It's less prone to errors, as we have more control over our code.
- It's more efficient, because the micro-frontend understands the boundaries of each part of the system.

In the past, we have seen a similar shift in frontend architecture when it moved from bidirectional data flow to a unidirectional flow with the release of Facebook's [Flux](#), which made developers' lives easier and applications more stable. The same reasoning applies to React and its use of props objects injected from parent components into one or more child components. Likewise, reactive architectures have fully embraced this pattern, with interesting implementations like model–view–intent (MVI) applied in [Elm](#) or [Cycle.js](#).

UNIDIRECTIONAL DATA FLOW

One of the more recent and largest revolutions in frontend architecture is the introduction of unidirectional data flow. Compared to previous architectures—such as model–view–viewmodel (MVVM), model–view–presenter (MVP), or even the popular model–view–controller (MVC)—it completely changed the evolution of many state management systems.

[André Stoltz's website](#) is a great resource for seeing how unidirectional data flow was applied in several recent frontend architectures. Stoltz did an amazing job researching the topic and creating MVI—a fully reactive, unidirectional architecture based on RxJS Observables.

Developer Experience

Module Federation makes developers' lives easier. The team behind this project did an incredible job abstracting the complexity needed to create a smooth DX, and now

developers can load shared code in the form of libraries or micro-frontends. Even better, Module Federation fits perfectly inside the bundler ecosystem and can be used alongside other plug-ins or configurations available in a Webpack setup.

By default, this plug-in produces small JavaScript chunks for every micro-frontend, enabling dependencies to be shared across micro-frontends when specified in the plug-in's configuration. However, when we use the optimization capability that a bundler offers out of the box, we can instruct the output to use fewer but larger chunks—maybe dividing our output into vendor and business logic files.

Use Cases

Because this library provides such extensive flexibility, we can apply any horizontal- or vertical-split micro-frontend use case to it. We can compose an application on the client or server side and then easily route using any available routing libraries for our favorite UI framework. Finally, we can use an event emitter library or custom events for communication across micro-frontends. Webpack with Module Federation covers almost all micro-frontend use cases, providing a great DX for every team or developer used to working with Webpack.

Architecture Characteristics

The following architecture characteristics should be taken into account:

Deployability (4/5)

Webpack divides a micro-frontend into JavaScript chunks, making them easy to deploy in any cloud service from any automation pipeline. And because they are all static files, they are highly cacheable. While we must handle the scalability of the application servers responding to any client requests in an SSR approach, the ease of integration and rapid feedback are definitely big pluses for this approach.

Modularity (4/5)

This plug-in's level of modularity is very high, but so is its risk. If we're not careful, we can create many external dependencies across teams; therefore, we have to use Module Federation wisely to avoid creating organizational friction.

Simplicity (5/5)

Module Federation solves many problems behind the scenes, and the abstraction created makes the integration of micro-frontends very similar to other, more familiar frontend architectures, like SPA or SSR.

Testability (4/5)

Although Module Federation offers an initial version of a federated test for integration testing, we can still apply unit and end-to-end testing, similar to how we're used to working with other frontend architectures.

Performance (4/5)

With Module Federation, we gain a set of capabilities—such as sharing common libraries or UI frameworks—that won't compromise the final artifact's performance. Bear in mind the mapping between a micro-frontend and its output files could be one too many, so a micro-frontend may be represented by several small JavaScript files, which may increase the initial chattiness between a client and a CDN performing multiple round trips for loading all the files needed to render a micro-frontend.

Developer experience (5/5)

This is probably one of the best DXs currently available for working with micro-frontends. Module Federation integrates very nicely with the Webpack ecosystem, hiding the complexity of composing micro-frontends, enabling the implementation of more traditional features, and handling tedious tasks (e.g., code sharing and asynchronous import of static artifacts or libraries).

Scalability (5/5)

Module Federation's approach makes scaling easy, especially when the application is fully client side. The static JavaScript chunks easily served via a CDN make this approach extremely scalable for a vertical-split architecture.

Coordination (3/5)

When we follow the decision framework shared in the first chapters of this book, in conjunction with Module Federation, we can greatly facilitate the life of our enterprise organization. However, the accessible approach provided by this plug-in can lead to overuse of modularity, resulting in increased coordination and potential refactors in the long term.

Table 3-2 gathers the architecture characteristics and their associated scores for this micro-frontend architecture.

Table 3-2. Architecture characteristics summary for developing a micro-frontend architecture using Webpack with Module Federation

Architecture characteristics	Score out of 5
Deployability	4/5
Modularity	4/5
Simplicity	5/5
Testability	4/5
Performance	4/5
Developer experience	5/5
Scalability	5/5
Coordination	3/5

iframes

Iframes are probably not the first things that come to mind in relation to micro-frontends, but they provide isolation between micro-frontends that none of the other solutions can offer.

An iframe is an inline frame used inside a webpage to load another HTML document inside it. When we want to represent a micro-frontend as an independent artifact that is completely isolated from the rest of the application, iframes are one of the strongest forms of isolation we can have inside a browser. An iframe gives us granular control over what can run inside it. The less-privileged implementation using the **sandbox attribute** prevents any JavaScript logic from executing or any forms from being submitted:

```
<iframe sandbox src="https://mfe.mywebsite.com/catalog"/>
```

An iframe gives us access to specific functionalities by combining **sandbox** with other **sandbox** attribute values—such as **allow-forms** or **allow-scripts**—to ease the **sandbox** restrictions and allow form submission or JavaScript file execution, respectively:

```
<iframe sandbox="allow-scripts allow-forms"
src="https://mfe.mywebsite.com/catalog"/>
```

Additionally, the iframe can communicate with the host page when we use the **postMessage method**. In this way, the micro-frontend can notify the broader application when there is a user interaction inside its context, and the application can trigger other activities, such as sharing the event with other iframes or changing part of the UI interface present in the host application.

Iframes aren't new, but they are still in use for specific reasons and have found a place within the micro-frontend ecosystem. So far, the main use cases for implementing micro-frontends with iframes come from B2B applications, especially in highly regulated environments such as financial institutions or when it's necessary to integrate an existing application into a micro-frontend architecture due to an acquisition, for example. Note, though, that this approach is strongly discouraged for consumer websites because iframes are really bad for performance. They are CPU-intensive, especially when multiple iframes are used in the same view.

Best Practices and Drawbacks

There are some best practices to follow when we want to compose micro-frontends in a horizontal split with iframes.

First, we must define a list of templates where the iframes will be placed; having a few layouts can help simplify managing an application with iframes. See, for example, [Figure 3-20](#).

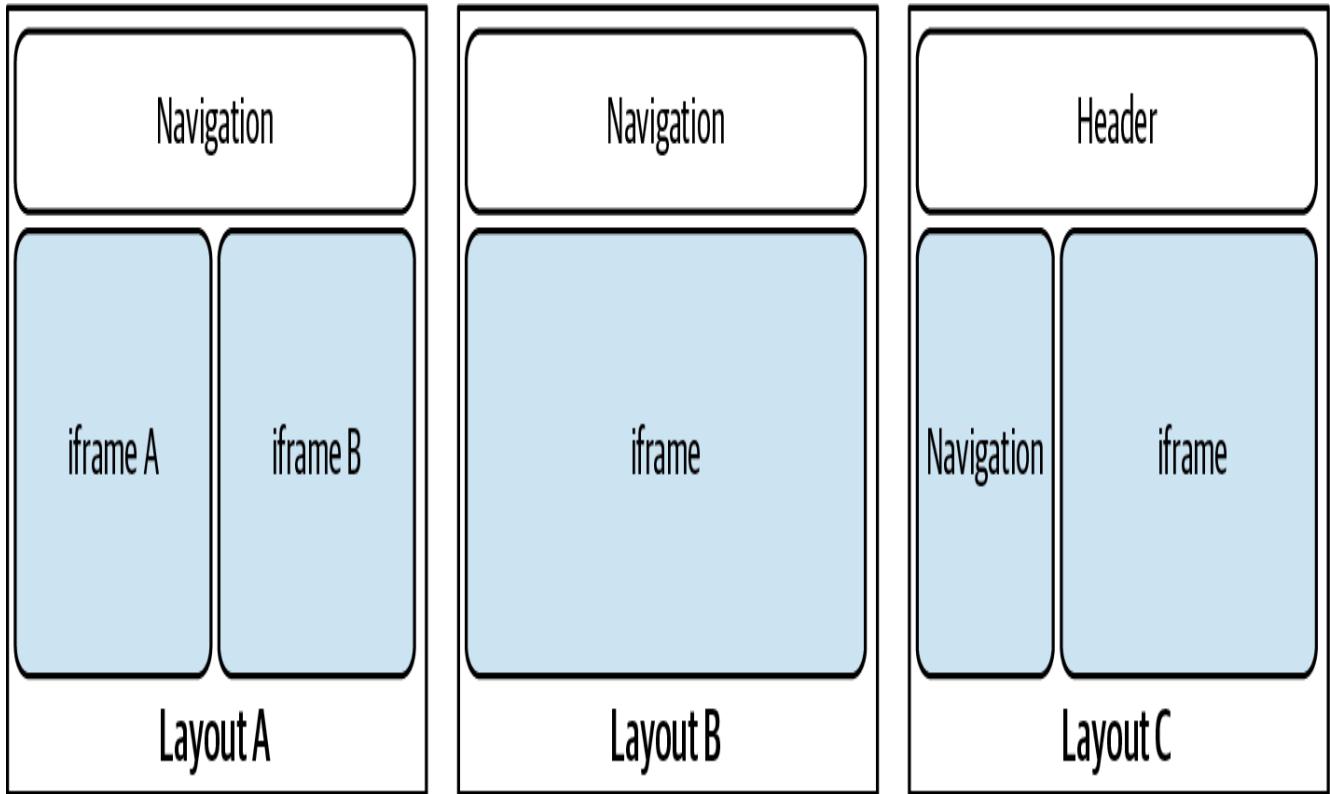


Figure 3-20. Different layouts for composing micro-frontends with iframes

Minimizing the number of iframes on a page would improve performance, despite the intrinsic overhead of integrating one or more iframes into a view. Using templates enables your teams to understand how to implement their micro-frontend UI and helps reduce edge cases by providing clear guardrails to follow.

Try to avoid too many interactions across micro-frontends; excessive interactions increase the complexity of the code that must be maintained. If you need to share a lot of information across micro-frontends, iframes may not be the right approach for the project.

This architecture allows teams to build their micro-frontends in isolation without any potential conflicts between libraries. However, to create a consistent UI, you will need to share the design system at build time.

Using iframes for responsive websites can be challenging, as handling a fluid layout across iframes and their content can be fairly complicated. Try to stick with fixed

dimensions as much as possible. If fixed dimensions aren't possible, one of the other architectures in this chapter may work better for you.

When storing data in web storage or cookies, use the web storage or cookie in the application shell to avoid issues with retrieving data across multiple iframes. In this situation, communication between the host page and every micro-frontend living inside an iframe must be carefully implemented and thoroughly tested.

When using a pub/sub pattern between iframes and the host page, you need to share an event emitter instance among the main actors on the page. Create an event emitter and append it to the iframe `contentWindow` object so that you can communicate via the `emit` or `dispatch` method across all the micro-frontends listening to it.

Developer Experience

Dealing with iframes make developers' lives easier, considering the sandboxed environment they use. One of the main challenges using this approach is with end-to-end testing, when retrieving objects programmatically across multiple iframes can result in a huge effort due to object nesting.

Overall, a micro-frontend will be represented by an HTML entry point, with additional resources loaded, such as JavaScript or CSS files—very similar to what we are used to in other frontend architectures, like SPAs.

Use Cases

Iframes are definitely not the solution for every project, but they can be handy in certain situations.

Iframes shine when there isn't much communication needed between micro-frontends, and when we must enforce the encapsulation of our system using a sandbox for every micro-frontend. The sandboxes help release memory and prevent dependency clashes between micro-frontends, removing some of the complexities of other implementations.

Drawbacks include accessibility, performance, and lack of indexability by crawlers, so the best use cases for iframes are desktop, B2B, or intranet applications.

Many large organizations use iframes in intranet applications as strong security boundaries between teams. For instance, when a company has tens of teams working on the same project and wants to enforce the teams' independence, iframes could be a valid solution to avoid code or dependency clashes without creating too many additional tools.

Imagine you have to build a dashboard where multiple teams contribute their micro-frontends, composing a final view with a snapshot of different metrics and data points to consult. Iframes can help isolate the different domains without the risk of clashes between codebases from different teams. They can even prevent specific features inside an iframe using the sandbox attribute.

The final use case is when we have to maintain a legacy application that isn't actively developed but is in support mode, and it has to live alongside a new application that is under development—both of which need to be presented to users. In this case, the legacy application can be easily isolated inside an iframe, allowing it to live alongside a micro-frontend architecture without the risk of polluting the new application.

If your web application doesn't fit these use cases, I recommend you look elsewhere and consider a more modern approach to make your application more performant.

Architecture Characteristics

The following architecture characteristics should be taken into account:

Deployability (5/5)

The deployability of this architecture is nearly identical to the vertical-split one, with the main difference being that we will have more micro-frontends in the horizontal split because we will be dealing with multiple micro-frontends per view.

Modularity (3/5)

Iframes provide an acceptable level of modularity, thanks to the ability to organize a view in multiple micro-frontends. At the same time, we will need to find the right balance to avoid misusing this characteristic.

Simplicity (3/5)

For a team working on a micro-frontend, iframes are not difficult. The challenge is in communicating across iframes, such as orchestrating iframe sizes when the page is resized without breaking the layout. More generally, dealing with the big picture in the absence of frameworks may require a bit of work.

Testability (3/5)

Testing in iframes doesn't bring any particular challenges apart from the one described for horizontal-split architectures. However, end-to-end testing may become verbose and challenging due to the DOM tree structure of iframes inside a view.

Performance (2/5)

Performance is probably the worst characteristic of this architecture. If not managed correctly, performance with iframes may be far from great. Although iframes solve a huge memory challenge and prevent dependency clashing, these features don't come free. In fact, iframes aren't a solution for accessible websites because they aren't screen-reader friendly. Moreover, iframes don't allow search engines to index the content. If either of these is a key requirement for your project, it's better to use another approach.

Developer experience (3/5)

The iframes DX experience is similar to the SPA one—automation pipelines are set up in a similar manner, and final outputs are static files. The main challenge is creating a solid client-side composition that allows every team working with micro-frontends to test their artifacts in conjunction with other micro-frontends. Some custom tools for speeding up your teams' DX may be needed. The most challenging part, though, is creating end-to-end testing due to the DOM replication across multiple iframes and the verbosity for selecting an object inside them.

Scalability (5/5)

The content served inside an iframe is highly cacheable at the CDN level, so we won't suffer from scalability challenges at all. In the end, we are serving static content, like CSS, HTML, and JavaScript files.

Coordination (3/5)

As with all horizontal-split architectures, it's important to avoid too many teams collaborating in the same view. Thanks to the sandbox nature of iframes, code clashes aren't a concern, but we can't have interactions spanning across the screen when we have multiple

iframes because coordinating these kinds of experiences is definitely not suitable for this architecture.

Table 3-3 gathers the architecture characteristics and their associated scores for this micro-frontend architecture.

Table 3-3. Architecture characteristics summary for developing a micro-frontend architecture using horizontal split and iframes

Architecture characteristics	Score out of 5
Deployability	5/5
Modularity	3/5
Simplicity	3/5
Testability	3/5
Performance	2/5
Developer experience	3/5
Scalability	5/5
Coordination	3/5

Web Components

Web components are a set of web platform APIs that enable you to create custom, reusable, and encapsulated HTML tags for use in web pages and web apps.

Web components may not be the most obvious choice when considering micro-frontends. However, they have distinctive characteristics that make them a suitable solution for building micro-frontend architecture.

For instance, we can encapsulate our styles inside web components without fear of leaking into the main application. In addition, all the major UI frameworks—including React, Angular, and Vue—are capable of generating web components. Furthermore, the number of open source libraries that simplify creating web components is increasing—particularly with projects like Svelte, which can compile to web components, and LitElement from Google.

Web components are excellent tools for creating shared libraries in micro-frontend projects, whether teams are using the same or different UI frameworks. They play a pivotal role in micro-frontend architecture, both for sharing components across micro-frontends or for encapsulating micro-frontends themselves. For instance, [Picard.js](#) leverages web components for micro-frontends, working on the backend with Node.js or Deno as well as inside the browser. It's compatible with Piral, Module Federation, Native Federation, and other similar technologies. Picard.js aids significantly in micro-frontend orchestration, making it a valuable library to include in your toolset. Another open source framework based on web standards is [OpenMFE](#). This framework comes directly from TUI—the well-known travel company, which leveraged this approach to build its horizontal-split website, where multiple teams were dedicated to specific features of the web application.

Web Components Technologies

Web components consist of three main technologies that can be used together to create custom elements with encapsulated functionality, reusable without fear of code collisions:

Custom elements

An extension of HTML components, serving as containers for micro-frontends. They allow interaction with the external world via callbacks or events and can be configured through exposed properties.

Shadow DOM

A set of JavaScript APIs for attaching an encapsulated shadow DOM tree to an element, rendered separately from the main document DOM. This technology is crucial for micro-frontends as it provides a powerful encapsulation mechanism.

HTML templates

The `<template>` and `<slot>` elements enable writing markup templates that are not displayed in the rendered page and can be reused as the basis of a custom element's structure.

Considering these three technologies, what makes web components particularly useful for micro-frontend architectures are custom elements and the shadow DOM.

Custom elements serve as wrappers for micro-frontends, while the shadow DOM allows for style isolation and DOM encapsulation. This encapsulation ensures that each micro-frontend can include its own web components without worrying about style clashes or unexpected behavior caused by global CSS or JavaScript.

An important aspect to consider when working with web components as wrappers for our micro-frontends is avoiding domain logic leaks.

The moment we allow the container of our micro-frontends—wrapped inside a web component—to customize its behavior, we are exposing domain logic to the external world, causing the container to know how to interact with a specific API contract via attributes.

It's essential to make sure the communication between a micro-frontend wrapped by a web component and the rest of the view happens in a decoupled and unified way.

Otherwise, we may risk blurring the line between components and micro-frontends: components should be open to extension, while micro-frontends should be closed to extension but open to communication.

Dependency Management

Web components—while excellent for encapsulating logic and creating strong boundaries between micro-frontends—present a challenge when it comes to shared dependencies. Due to their isolated nature, internal libraries cannot be shared easily across different web components. This limitation shifts the focus from providing shared libraries to coordinating which libraries should be used across a project. Instead of offering a centralized set of dependencies, teams must collaborate to establish loose guidelines on library selection and usage.

One approach to sharing dependencies is to add shared libraries to the global `window` object. However, this method introduces its own set of challenges, particularly when dealing with different versions of the same dependency. While this approach allows micro-frontends to access shared libraries, it becomes problematic when different

micro-frontends require different versions of the same library. This can lead to conflicts and unexpected behavior, as the global object can only hold one version of each library. Consider an ecommerce platform using micro-frontends implemented as web components. The product listing, shopping cart, and check-out process might be separate micro-frontends. Each team responsible for these components needs to decide independently on libraries for state management, HTTP requests, or UI utilities. Rather than sharing a common Redux store or Axios instance, teams might agree on using similar libraries (e.g., MobX for state management and Fetch API for HTTP requests) without directly sharing the implementations. This approach maintains the independence of each micro-frontend while ensuring a level of consistency across the platform. The challenge lies in balancing this independence with the need for a cohesive user experience and efficient development practices.

Use Cases

Embracing web components for your micro-frontend architecture is a great choice when you need to support multi-tenant environments. Given their broad compatibility with all the major frameworks, they are an ideal candidate for use in multiple projects, whether the frontend stacks are the same or different, as is often the case with multi-tenant projects. In such projects, our micro-frontends should be integrated into multiple versions of the same application or even across different applications, making web components a simple and effective solution.

For example, imagine your organization is selling a customer-support solution in which the chat micro-frontend needs to work alongside any frontend technology your customers use.

Web components can also play an important role in shared libraries. In a design system, for instance, using a web standard enables you to evolve applications without having to start from scratch each time, because part of the work is reusable regardless of the direction your tech teams or business take. It's a smart investment to make.

Architecture Characteristics

The following architecture characteristics should be taken into account:

Deployability (4/5)

Loading web components at runtime is easily doable. We just need a CDN to serve them, and they can then be integrated anywhere. They

are also easy to integrate at compile time by adding them as we import libraries in JavaScript. Although technically you can render them server side, the DX is not as smooth as with other solutions proposed by UI frameworks like React.

Modularity (4/5)

Web components' high degree of modularity allows you to decompose an application into well-encapsulated subdomains. Moreover, because they are a web standard, we can use them in several situations without too many problems, as long as we operate in browsers that support them.

The risk of using them as a micro-frontend wrapper is that it can confuse new developers joining a project, blurring the line between components and micro-frontends. This often results in a proliferation of “micro-frontends” in a view—but perhaps we should call them “nano-frontends.”

Simplicity (4/5)

Using web components should be a straightforward task for anyone who is familiar with frontend technologies. The main challenge is avoiding splitting micro-frontends too granularly. Because web components can also be used to build component libraries, the line between micro-frontends and components can become blurred. However, focusing on the business aspects of the application should lead us to correctly identify micro-frontends from components in our applications.

Testability (4/5)

Leveraging different testing strategies using web components doesn't present too many challenges, but we must be familiar with their APIs. Web components' APIs differ from UI frameworks, making it challenging to do what we are used to doing with our favorite framework.

Performance (4/5)

One of the main benefits of web components is that we are extending HTML components, meaning that we aren't making them extremely dense with external code from libraries. As a result, they should be one of the best solutions for rendering your micro-frontends on the client side.

Developer experience (4/5)

The DX of your projects shouldn't be too different from your favorite framework. You may need to learn another framework to simplify your workflow, but there aren't too many differences in the development life cycle—especially in syntax. That's why web component frameworks exist: to simplify developers' lives.

Scalability (5/5)

Whether we implement our web components at compile or runtime, we will be delivering static files. A simple infrastructure can easily serve millions of customers without the bother of maintaining complex infrastructure solutions for handling traffic.

Coordination (3/5)

The main challenge is getting the granularity of the micro-frontends right, because this will impact application delivery speed and avoid external dependencies that could frustrate developers. We need a strong sense of discipline when identifying what should be represented by a component versus a micro-frontend.

Table 3-4 gathers the architecture characteristics and their associated scores for this micro-frontend architecture.

Table 3-4. Architecture characteristics summary for developing a micro-frontend architecture using web components

Architecture characteristics	Score out of 5
Deployability	4/5
Modularity	4/5
Simplicity	4/5
Testability	4/5
Performance	4/5
Developer experience	4/5
Scalability	5/5
Coordination	3/5

Web Fragments

A brand-new approach proposed by CloudFlare in 2025 is called Web Fragments. I haven't had the chance to work with Web Fragments myself yet, but I believe it's worth mentioning in this book that brings together all the major micro-frontend projects in the world. **Web Fragments** was conceived by Igor Minar in 2021, and it was developed in collaboration with Natalia Venditto as a vendor-neutral open source project sponsored and used by CloudFlare since 2024, with contributions from Microsoft, Netlify, and others. It represents a new micro-frontend composition technique focused on strong isolation and incremental modernization.

Unlike many approaches that rely on runtime dependency sharing, Web Fragments is founded on the idea that isolation and encapsulation can eliminate **fate-sharing** and the coordination overhead associated with it. This means that multiple versions of

frameworks like React can coexist on the same page without conflicts. The trade-off is straightforward: while duplication increases payload size, the organizational benefits of independent life cycles often outweigh the cost of a few extra kilobytes. Web Fragments also supports dependency sharing at the network level, which reduces the overhead without sacrificing the isolation guarantees that make the model appealing.

Technically, each fragment is rendered inside the same DOM tree, owned by the host application, but each fragment is wrapped with two layers of isolation. CSS is scoped through the shadow DOM, while JavaScript runs inside a hidden iframe that provides a private execution context. This gives fragments the equivalent of Docker-like containers in the browser: they cannot accidentally leak styles or global variables into each other and simplify debugging, while remaining part of a unified application with shared navigation and history.

CloudFlare is using Web Fragments to incrementally modernize its own large, mission-critical dashboard. Instead of a disruptive rewrite, teams can migrate one section at a time while preserving a cohesive user experience. The same approach has also enabled unusual integrations, such as seamlessly composing two entirely separate SPAs into a single interface, mixing SPAs and full stack applications regardless of their tech stack, and performing incremental re-platforming of tech stack or hosting provider.

For enterprises facing the complexity of scaling frontends, Web Fragments offer both horizontal-split and vertical-split strategies: teams can modularize at the page or section level, reducing the risk of migrations, and move faster without tightly coupling release cycles.

Server Side

Horizontal-split architectures with a server-side composition are the most flexible and powerful solutions available in the micro-frontends ecosystem, thanks to the cloud, which is the perfect environment for developers wanting to focus on the value stream more than infrastructure operationalization. In the cloud, we have the agility to spin up the infrastructure as requests increase and to reduce it again when traffic goes back to normal. We can also set up our baseline without too many headaches, focusing on what really matters: the value created for our users.

Server-side composition is usually chosen when our applications have a strong SEO requirement, because this technique speeds up the page load times and renders the page fully without requiring any JavaScript logic—both of which help improve your application’s position in search engine results pages.

In a server-side composition, the final view is created on the server side, where we can control the speed of the final output using techniques like caching at different layers (e.g., in a service, in-memory, or CDN), reducing hops between services to retrieve all the micro-frontends, and optimizing the type of compute used to compose our micro-frontends. Nowadays, we have all the tools and resources needed to impact how fast a view is composed and served to users.

Many of the challenges described in horizontal-split client-side composition also exist on the server side. Therefore, rather than repeating them here, let's focus on a few additional challenges unique to this approach.

Scalability and Response Time

Despite the infrastructure flexibility that the cloud provides, we have to set up the infrastructure correctly in the first place based on our application's traffic patterns. While a cloud provider's autoscaling functionalities can help you to achieve this goal, the type of compute layer you choose will affect how fast you can ramp up your application.

Of course, not all web applications behave in the same way, so there's a risk that our chosen autoscaling solution will not be fast enough to cope with our specific traffic surges. A classic example would be the beginning of Black Friday sales or a global live event available only on one platform. In these cases, we would need to ensure that we meet the predictive load by manually increasing our solution's baseline infrastructure before the users join our platform.

Another challenge we face with this architecture is understanding how to speed up the response time of services—and possibly microservices—and deciding whether we need to consume them every time, or if we can cache the response for specific micro-frontends while embracing eventual consistency. In-memory cache solutions like Redis can be a great ally in this situation, allowing us to store the microservice responses for a short period, thereby increasing the throughput of micro-frontend composition. We can also store the entire micro-frontend DOM inside an in-memory cache and fetch it from there instead of composing it every time.

Alternatively, we can use a CDN, which can increase a web page's delivery speed, reducing the latency between the client and the content requested.

Latency, response time, cache eviction, and similar metrics become our measures of success in these situations. However, creating the right infrastructure is not a trivial process, especially when there is a lack of knowledge or experience.

Infrastructure Ownership

Composition-layer ownership is another challenge with this architecture. In the best implementations, a cross-functional team of frontend and backend developers works together to manage the micro-frontend composition layer end to end. In this way, they can collaborate on the best entry points for the composition layer outputs, improving how data flows through it.

Some companies may decide to separate the composition layer from micro-frontend development. The risk here is that a frontend developer could end up working in a silo, requiring additional mechanisms to keep the teams in sync and consolidate the integration of the two layers. Frontend developers must clearly understand what's going on in the composition layer and be able to help enhance it across code, infrastructure, monitoring, and logging. By doing so, they can optimize the micro-frontend code based on the implementation made in the composition layer.

Composing Micro-Frontends

Composing micro-frontends in a server-side architecture may differ from what we have seen so far, but the differences are in the details, not the substance. As we can see from [Figure 3-21](#), the typical architecture is composed of three layers.

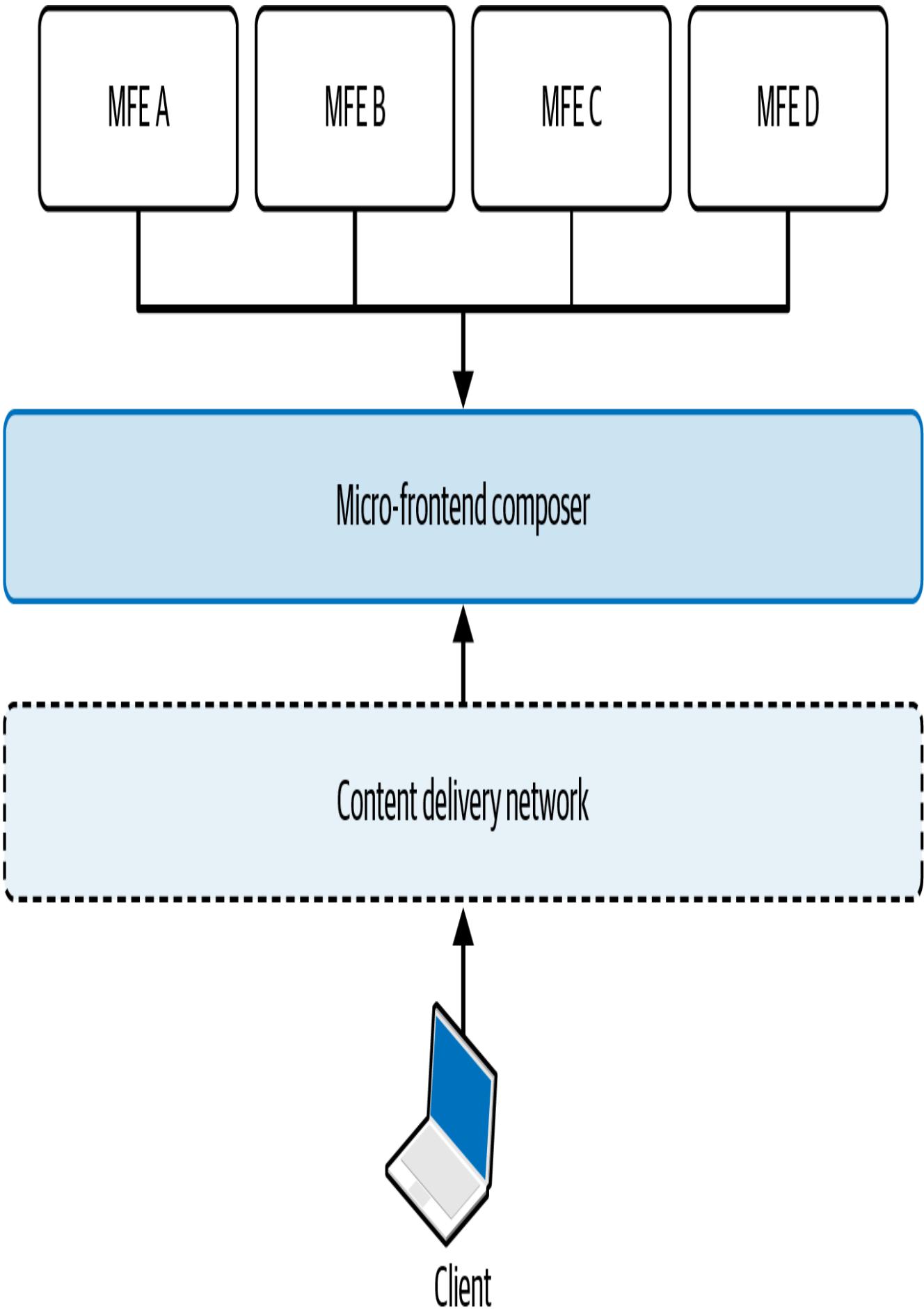


Figure 3-21. A typical high-level architecture for a server-side micro-frontend architecture, where a composer is responsible for stitching together the different micro-frontends at runtime and where a CDN can be used for offloading traffic to the origin

The three levels include:

Micro-frontends

These can be deployed as static assets—possibly prepared at compile time during the automation pipeline—or as dynamic assets, in which an application server prepares a template and its associated data for every user's requests.

Composer

This layer is used to assemble all the micro-frontends before returning the final view to a user. In this case, we can use an NGINX or HTTPd instance for leveraging server-side includes directives, or implement a more complex solution using Kubernetes and custom application logic to stitch everything together.

CDN

Whenever possible, you should add a CDN layer in front of your application server to cache as many requests as possible. If you can cache a page for a few minutes, you can offload a lot of traffic from the composer and increase your web application's performance, thanks to a shorter round trip for the response.

Many frameworks out there will handle the undifferentiated heavy lifting of implementing this type of architecture. To choose the best one for your project, you'll have to understand the project's business goals and evaluate the frameworks accordingly. In the next section, I'll cover some of these frameworks so you can see how they align with the pattern described here. However, every framework emphasizes different aspects, and not all DXs are first-class, so you may end up investing more time streamlining the DX before realizing the expected outcome from your chosen framework.

Another very common server-side rendering architecture that has emerged in recent years with frameworks like Astro.js or Next.js involves splitting micro-frontends per page or per group of pages. These are loaded inside a standard template that also has other common parts loaded at runtime, such as a footer, as shown in [Figure 3-22](#).

In this scenario, we follow almost the same logic as a vertical split, dividing an application by page or group of pages. However, we load one or multiple micro-frontends inside a view, such as the catalog view.

This aligns very well with the latest structures proposed by frameworks like [Next.js](#) and [multi-zone architectures](#).

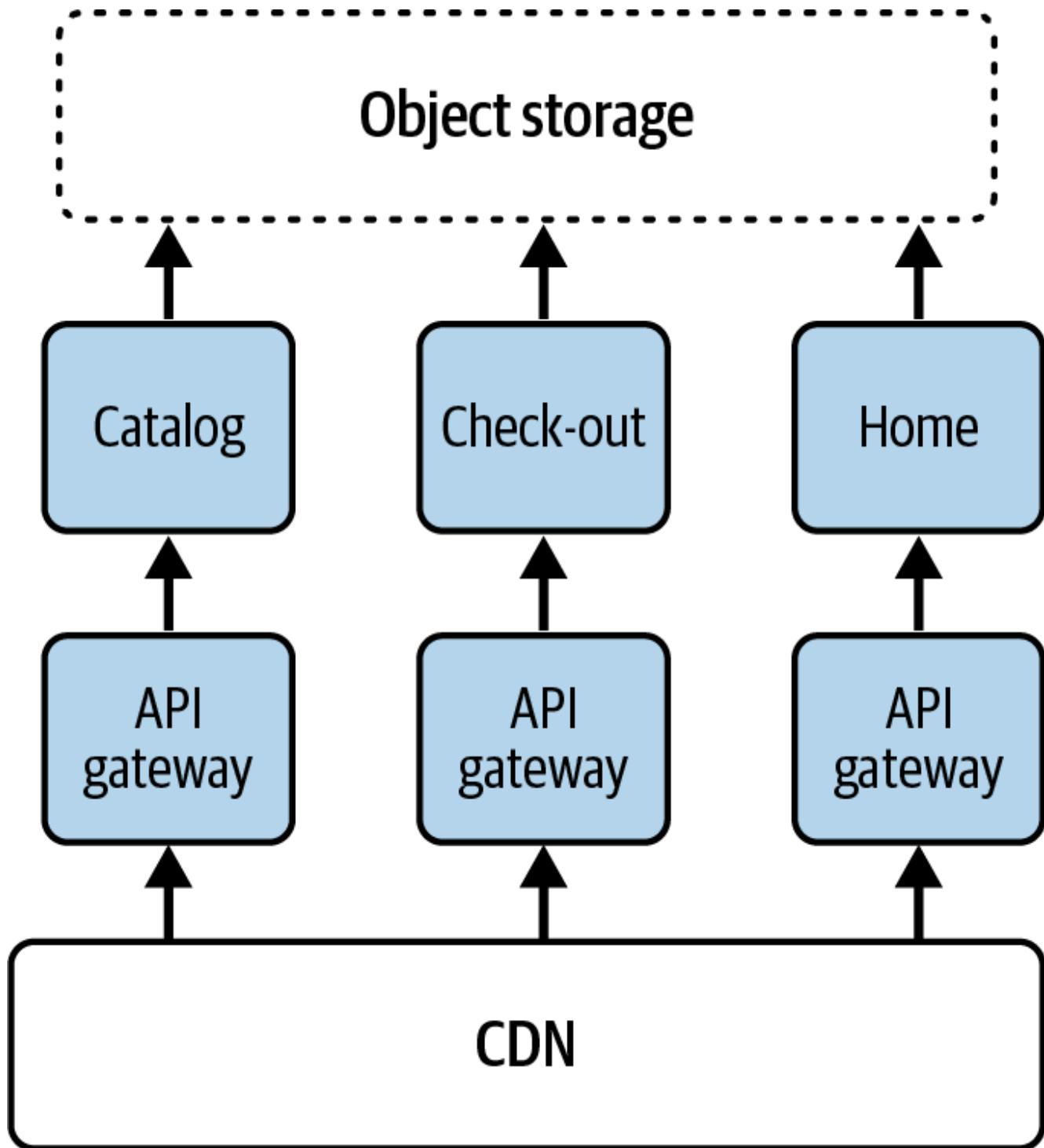


Figure 3-22. Every page or group of pages owned by a single team responsible for that domain end-to-end

Every part of an application is owned by a single team. They can develop and deploy the application independently, and also configure their underlying infrastructure independently to achieve the best performance for their part of the application. They can also cache their page or group of pages independently, making this approach a fantastic candidate for ecommerce or similar solutions.

We will dive deeper into these approaches in [Chapter 6](#).

Micro-Frontend Communication

When you choose the server-side approach, you likely won't have many communications within the view, instead focusing on communication between the view and APIs. This is because the page will reload after every significant user action.

Still, there are situations in which one micro-frontend needs to notify another that something happened in the session, such as a user adding a product to the cart. The micro-frontend that owns the domain will need to update the drop-down to show the user that the change was made. To accomplish this, we can add some logic on the frontend and, using an event emitter or custom event, we will keep the micro-frontends loosely coupled while allowing them to communicate when something happens inside the application.

In [Figure 3-23](#), we can see how this mechanism works in practice:

1. A user adds a product to the cart. This event is communicated to the backend, which acknowledges the added product within the user's session.
2. The "product" micro-frontend notifies the "check-out experience" micro-frontend that a new product was added to the cart.
3. The "check-out experience" micro-frontend fetches the updated list of products in the cart and displays the new information in the UI.

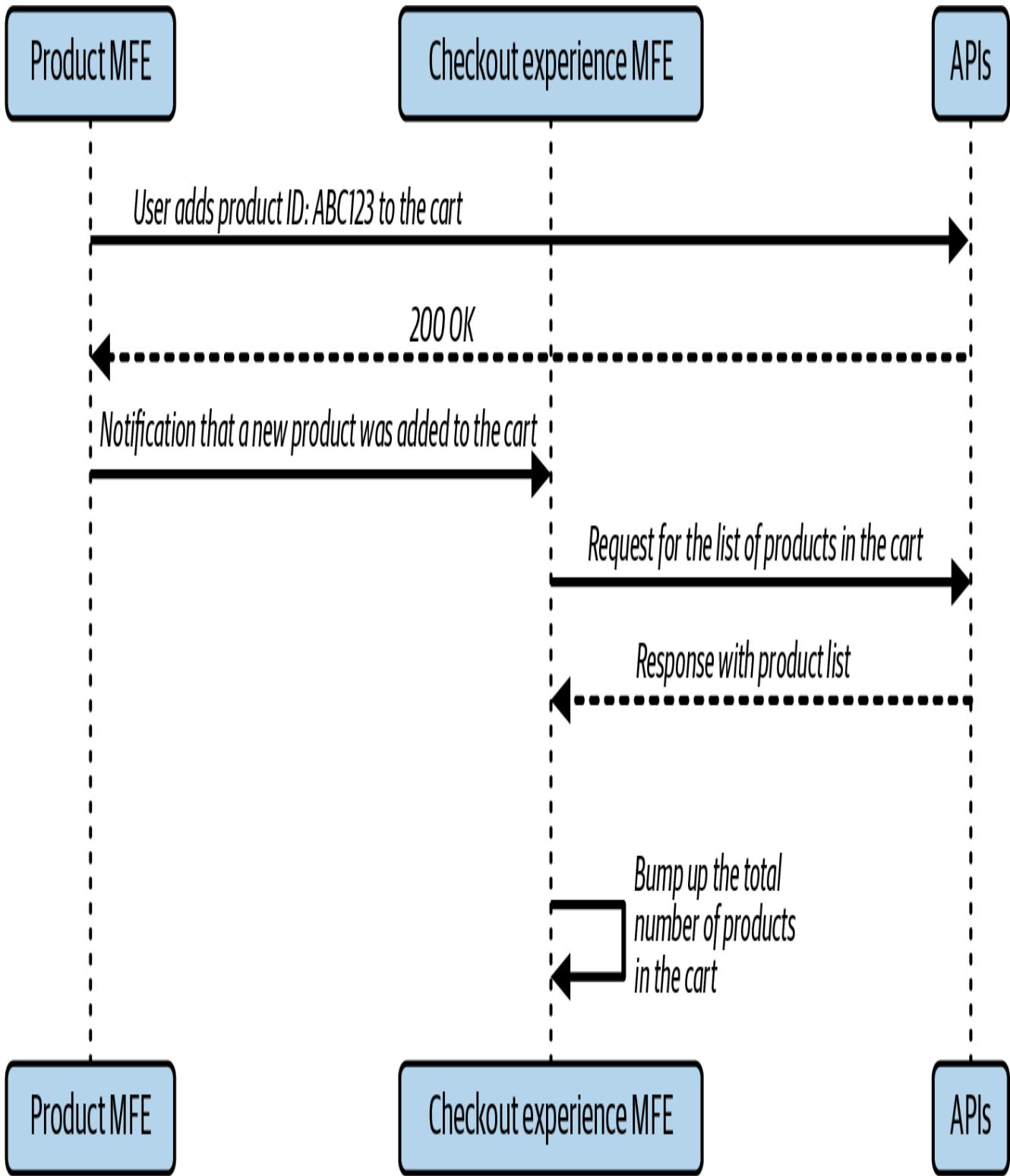


Figure 3-23. An example of how the “product” micro-frontend notifies the “check-out experience” micro-frontend to refresh the cart interface when a user adds a product to the cart

There aren't many of these types of interactions per view in most web applications, so the code won't negatively impact performance or the maintainability of the final solution.

Available Frameworks

[OpenComponents](#) is another micro-frontend framework for server-side horizontal-split architectures. This opinionated framework provides several features out of the box, including prewarming a CDN via runtime agents, a micro-frontend registry, and tools to simplify the DX. Every micro-frontend is encapsulated within a computational layer, completely isolated from the others. This approach enables each team to focus on implementing their own domain without needing to consider the entire application. Moreover, every micro-frontend has a set of utilities, such as observability, monitoring, or dashboards. For managing burst traffic at specific times of the day—for example, from a constant flow of people reserving tables at restaurants—the traffic prewarming the CDN is offloaded whenever a new micro-frontend is created or an existing one is updated. [Figure 3-24](#) showcases this.

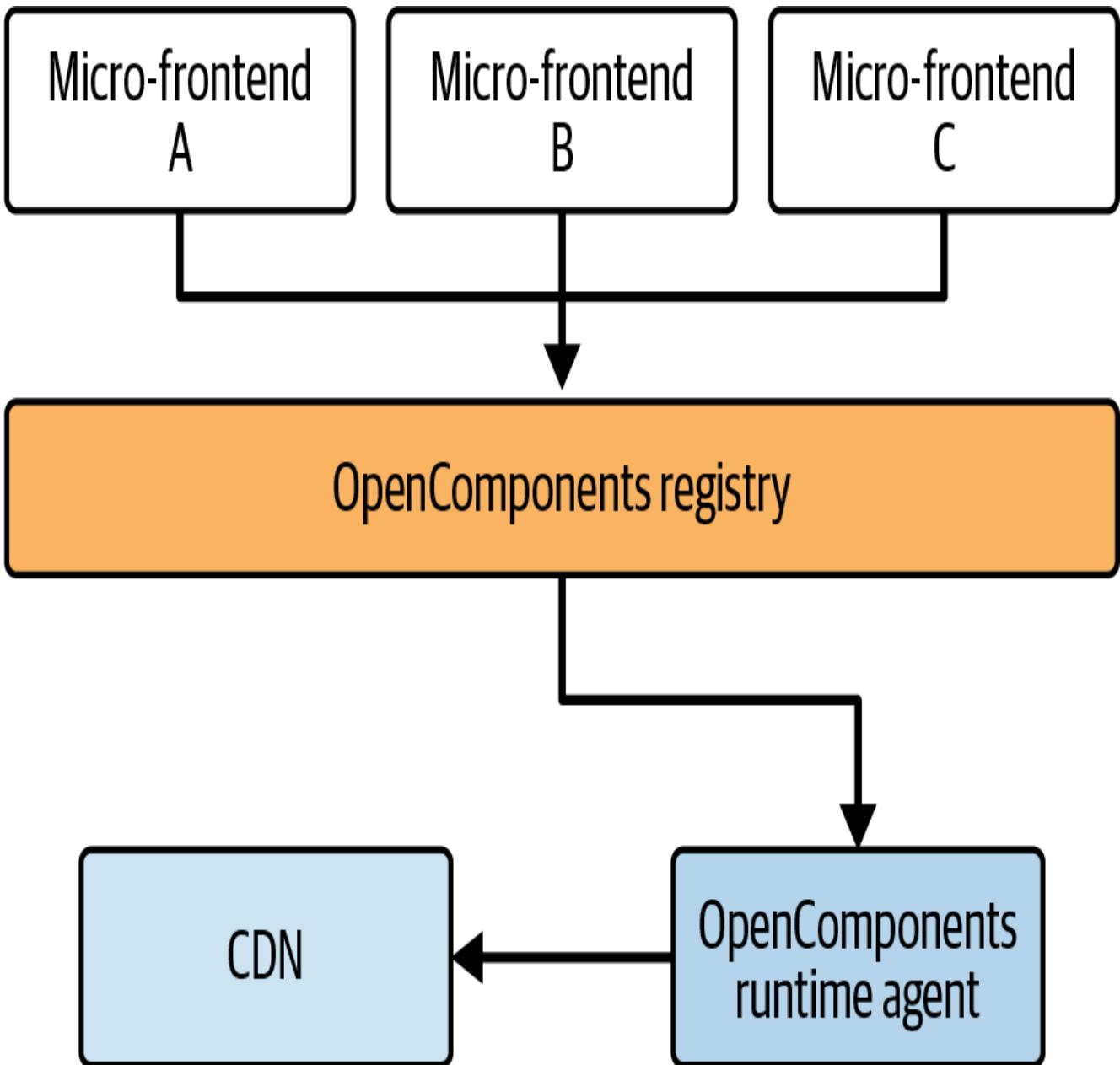


Figure 3-24. The OpenComponents architecture showing how a server-side rendered micro-frontend flows from development to an environment

Interestingly, OpenComponents allows not only server-side rendering but also client-side rendering, so you can choose the right technique for every use case. When SEO is a key goal of a project, for example, you can choose SSR, and when you need a more SPA-like experience, you can use client-side rendering.

Once again, you can see how all these frameworks had to invest in the developer experience to accelerate their adoption.

As you compare frameworks, keep in mind that the vast majority were built by mid- to large-sized organizations, for which the final benefits clearly outweighed the initial investment of resources and time.

Use Cases

The typical use case for this architecture is B2C websites, where content needs to be highly discoverable by search engines, or solutions that require a modular layout, such as dashboards in which the user interface doesn't require significant layout variations, as is common in customer-facing solutions. This architecture also works for B2B applications with many modules that are reused across different views. It's important, however, that full stack or backend developers with the appropriate skills facilitate the introduction of this implementation.

This architecture generally isn't a good choice for very interactive, fluid layouts, mainly due to the coordination needed across teams to create a cohesive final result, or when there are bugs in production that may require more effort in discovering their root cause.

Architecture Characteristics

The following architecture characteristics should be taken into account:

Deployability (4/5)

These architectures may be challenging when you have to handle burst traffic or high-volume traffic. To limit the extra work and avoid production issues, automate repetitive tasks as much as possible.

Modularity (5/5)

This architecture's key characteristic is the control we have over not only how we compose micro-frontends but also how we manage different levels of caching and final output optimization. Because we can control every aspect of the frontend with this approach, it's important to modularize the application to fully embrace it.

Simplicity (3/5)

Using this architecture is by far one of the easiest implementations you'll ever do. Yet, there are many moving parts, and observability tools on both the frontend and backend must be configured so that you'll understand what's happening when the application doesn't behave as expected. Taking into account the architectures discussed so far, this is both the most powerful and the most challenging approach—especially for large projects with burst traffic.

Testability (3/5)

This architecture doesn't differ too much from SSR applications. There may be some challenges when we expect every micro-frontend to hydrate the code on the client side because we'll have some additional logic to test. But, as we're talking about micro-frontends, it won't be too much additional effort.

Performance (5/5)

With this implementation, we have full control over the final result being served to a client, allowing us to optimize every single aspect of our application, down to the byte. That doesn't mean optimization is easier with this approach, but it definitely provides all the tools needed to make a micro-frontends application highly performant.

Developer experience (3/5)

There are frameworks that provide an opinionated way to create a smooth developer experience, but it's very likely you will need to invest time in creating custom tools to improve project management, such as dashboards, additional command-line tools, and so on. Additionally, a frontend developer may need to expand their backend knowledge, learning how to run servers locally, scale them in production, work efficiently in the cloud or on-prem, manage observability of the composition layer, and more. Full stack developers are more likely to embrace this approach, but not always.

Scalability (3/5)

Scalability may be a nontrivial task for high-volume projects, because you'll need to scale the backend that composes the final view to users. A CDN can help, but you will have to deal with different levels of caching. CDNs are helpful for static content, but less so with personalized content. Moreover, when you need to maintain low response latency and you aren't in control of the API you are consuming, you face additional challenges on top of scaling the micro-frontend composition layer.

Coordination (3/5)

Considering all the moving parts in this architecture, the coordination must be carefully designed. The structure should enable different teams to work independently, reducing the risk of too many external dependencies that can jeopardize a sprint and frustrate developers. Additionally, developers must keep both the big picture and the implementation details in mind, making the organizational structure more complicated—especially in large organizations with distributed teams.

Table 3-5 gathers the architecture characteristics and their associated scores for this micro-frontend architecture.

Table 3-5. Architecture characteristics summary for developing a micro-frontend architecture using horizontal split and server-side composition

Architecture characteristics	Score out of 5
Deployability	4/5
Modularity	5/5
Simplicity	3/5
Testability	3/5
Performance	5/5
Developer experience	3/5
Scalability	3/5
Coordination	3/5

Modern Server-Side Rendering Frameworks

In recent years, modern SSR frameworks have revolutionized the way we approach micro-frontend architectures. Frameworks like Astro.js, Qwik, Next.js, and htmx have introduced patterns that make micro-frontend implementation more efficient and manageable. Astro.js pioneered the concept of server islands, where interactive components are selectively hydrated while keeping the rest of the page static, resulting in improved performance and reduced JavaScript payload. This approach enables teams to build independent features that can be seamlessly integrated into a larger application.

Qwik took a similar but distinct approach with its container concept, introducing resumability instead of hydration. Qwik containers enable independent deployment of micro-frontends while maintaining optimal performance through their unique serialization mechanism, which only loads JavaScript when needed for interactivity.

Next.js, however, has emerged as a particularly powerful solution through its multi-zone feature. Multi-zones enable developers to merge multiple Next.js applications into a single main application, with each zone operating independently while sharing the same domain. This approach enables teams to work autonomously on different sections of a website while maintaining a unified user experience.

The introduction of React Server Components (RSC) in Next.js has further enhanced its micro-frontend capabilities. Server Components allow for the rendering of complex UI on the server with zero client-side JavaScript overhead, making them ideal for micro-frontend architectures. These components execute entirely on the server—streaming HTML to the client—and can fetch data directly from the database without additional API layers.

When it comes to horizontal-split micro-frontends, RCS on Next.js and server islands on Astro.js, provide a natural fit. Teams can develop independent features as Server Components, which can be composed into larger applications while maintaining clear boundaries. This approach reduces client-side bundle sizes and improves initial page load performance, as each micro-frontend can be server-rendered independently. The streaming nature of Server Components also ensures that users see content as soon as it's available, rather than waiting for all micro-frontends to load.

In contrast to JavaScript-heavy frameworks, htmx represents a refreshing return to server-centric architecture while still enabling dynamic micro-frontend implementations. HTMX's approach to micro-frontends is particularly elegant in its simplicity, using HTML attributes to handle server communication and DOM updates. This enables teams to create micro-frontends that are incredibly lightweight, as they

don't require complex client-side JavaScript bundles or elaborate build systems. By leveraging standard HTTP requests and server-rendered HTML, htmx micro-frontends can achieve remarkable performance while maintaining clean separation between different application parts. When combined with a reverse proxy like NGINX, teams can route requests to different backend services, each responsible for rendering their own micro-frontend segment. This architecture proves especially powerful for organizations that prefer SSR and want to avoid the complexity of client-side JavaScript frameworks while still maintaining the benefits of a micro-frontend architecture, such as independent deployability and team autonomy.

We are going to discuss some of these implementations in more detail in [Chapter 5](#).

Use Cases

Modern frameworks like Next.js, Astro, Qwik, and htmx all provide robust solutions for building micro-frontend applications. While these frameworks can technically achieve similar results, they differ significantly in their default approaches and optimizations. Next.js, as a full stack React framework, provides comprehensive server and client capabilities with built-in features like Server Components and multi-zones, making it a natural fit for teams already invested in the React ecosystem. Its approach to micro-frontends through multi-zones enables different teams to work independently while sharing the same domain.

On the performance-optimization front, each framework takes a distinct approach. Astro's partial hydration strategy through server islands enables teams to build highly performant applications by default, shipping zero JavaScript where it isn't needed. This makes it particularly effective for micro-frontends where different parts of the application have varying interactivity requirements. Qwik's innovative resumability approach eliminates the traditional hydration step entirely, offering a unique solution to the JavaScript overhead problem in micro-frontend architectures.

It is htmx that takes perhaps the most radical approach by eschewing complex client-side JavaScript frameworks altogether, instead extending HTML itself to handle dynamic updates. This simplicity can be a significant advantage in micro-frontend architectures where reducing complexity and maintenance burden is a priority, or in companies where JavaScript is not the main language.

The key takeaway is that while all these frameworks can build similar applications, choosing one over another is more about aligning with your team's philosophy and leveraging the built-in optimizations that best match your specific needs. Some teams

might prefer the full-featured approach of Next.js, while others might value Astro's content-first strategy or htmx's simplicity—and any of these choices can lead to successful micro-frontend implementations.

Architecture Characteristics

The following architecture characteristics should be taken into account:

Deployability (4/5)

As with any SSR architecture, handling burst traffic or high-volume traffic can be challenging. You will need to properly plan the caching strategy to apply for offloading traffic from origin as much as possible.

Modularity (4/5)

This architecture allows for fine-grained modularity. However, careful attention must be paid to properly divide the application. Most successful teams I've observed started with a coarse-grained approach and gradually moved to a more fine-grained approach where needed.

Simplicity (5/5)

All these frameworks offer a paved path and abstractions to easily implement micro-frontend mechanisms. The main challenge lies in identifying micro-frontend boundaries rather than developing features with these frameworks.

Testability (4/5)

These architectures offer robust testing solutions. From unit to end-to-end testing, the experience doesn't differ drastically from a monolithic application. While end-to-end testing requires more consideration, it presents no significant challenges.

Performance (5/5)

With this implementation, we maintain full control over the final result served to clients, enabling us to optimize every aspect of our application down to the byte.

Developer experience (5/5)

Modern frameworks prioritize developer experience in frontend development. The communities behind these frameworks do an outstanding job of providing tools and plug-ins to address any gaps. Among the four mentioned frameworks, htmx may lag slightly behind due to its intentionally simple nature.

Scalability (3/5)

While these frameworks offer solutions for in-memory caching API results like Next.js, considerable effort is required to design scalable infrastructure. The exception is the platform, which handles the infrastructure complexity, enabling developers to focus solely on feature implementation.

Coordination (4/5)

When structuring an application by assigning pages or groups of pages to individual teams, the need for cross-team coordination remains minimal. Moving to a more fine-grained solution requires additional planning for governance and potentially organizational restructuring to enable teams to succeed in a distributed system.

Table 3-6 gathers the architecture characteristics and their associated scores for this micro-frontend architecture.

Table 3-6. Architecture characteristics summary for developing a micro-frontend architecture using horizontal split and modern server-side frameworks

Architecture characteristics	Score out of 5
Deployability	4/5
Modularity	4/5
Simplicity	5/5
Testability	4/5
Performance	5/5
Developer experience	5/5
Scalability	3/5
Coordination	4/5

Edge Side

Before we close up this chapter, it's worth spending a few words about edge-side web applications.

While edge computing has gained significant traction in recent years—particularly with platforms like Cloudflare Workers, Fastly Compute, and AWS Lambda@Edge functions—it's important to understand why we don't currently see micro-frontend implementations leveraging edge rendering.

The primary reason lies in a fundamental concept known as “data gravity.”

Data gravity, a term coined by Dave McCrory in 2010, suggests that data and services naturally attract each other, similar to a gravitational pull in physics. In practical terms, this means that applications tend to be most effective when they're located close to their

data sources. Most organizations operate their APIs and databases from one or two primary regions—typically near their main business operations or where they initiated their cloud infrastructure.

Let's consider a practical example: an ecommerce company headquartered in New York maintains its primary database and API infrastructure in the AWS us-east-1 region. If a customer in Sydney, Australia, accesses the website through an edge location in Sydney, any server-side rendering happening at the edge would still need to fetch data from New York. The round-trip time (RTT) from Sydney to New York typically ranges from 200–250 ms, meaning that even though the edge compute location is closer to the user, the actual performance improvement is minimal or nonexistent due to the data fetch requirements.

Edge computing environments typically offer limited computational resources compared to traditional cloud computing environments. This constraint exists to maintain the cost-effectiveness and distributed nature of edge networks. Edge computing environments are characterized by strict CPU allocation limits, restricted memory usage that typically ranges from 128 MB to 256 MB, and tight execution time constraints that usually fall between 50 ms and 500 ms. Furthermore, these environments often provide only limited support for certain Node.js APIs and features, making it challenging to run full-featured applications that might be commonplace in traditional server environments.

In the past, Edge Side Includes (ESI) represented a potential solution for micro-frontends. ESI is a markup language that revolutionized dynamic content delivery in the early days of CDNs. This XML-based technology enables the assembly of web pages at the network edge through a system of specialized tags that instruct edge servers how to compose the final content. The fundamental principle behind ESI is the separation of static and dynamic content, allowing for efficient caching strategies while maintaining the ability to serve personalized or frequently updated content.

When a user requests a web page, ESI-enabled edge servers process special markup tags embedded within the HTML. These tags act as instructions, telling the edge server which content fragments to fetch, how to assemble them, and when to incorporate dynamic elements. For instance, a product page might contain static elements like the page layout and navigation, while dynamic elements such as personalized recommendations or real-time inventory status are fetched and inserted at request time.

The ESI processor—typically running on CDN edge servers or reverse proxies—interprets these tags and performs the necessary content assembly before delivering the final page to the user. This approach offers several advantages: it reduces the load on

origin servers, makes efficient use of caching mechanisms, and enables granular control over content updates.

For example, a news website could cache its main layout for 24 hours while updating the headlines every 15 minutes, all without requiring changes to the underlying application architecture.

However, as web architecture has evolved toward more interactive and client-side-heavy applications, ESI has become less prevalent in modern web development, with newer technologies and approaches taking its place in the micro-frontend landscape.

Summary

In this chapter, we have applied the micro-frontend decision framework to multiple architectures. Defining the four pillars offered by this mental model helps us to filter our choices and select the right architecture for a project.

We have analyzed different micro-frontend architectures, highlighting their challenges and scoring the architecture characteristics so that we can easily select the right architecture based on what we need to optimize for.

Finally, because we understand that the perfect architecture doesn't exist, we discovered that we have to find the "least worst" architecture based on the context we operate in.

In the next chapter, we will analyze a technical implementation and focus our attention on the main challenges we may encounter in a micro-frontend implementation.

Chapter 4. Client-Side Rendering Micro-Frontends

In this chapter, we will use the micro-frontend decision framework to build a basic ecommerce website using Module Federation for a client-side rendering approach. As we've discussed, there isn't a one-size-fits-all solution when it comes to architecture. The project's goals, the organization's structure and communications, and the technical skills available in the company are some of the factors we have to consider when we need to choose an approach.

After identifying the context that we'll operate in, we can use the micro-frontend decision framework to help define the key pillars for our architecture's technical direction. Instead of creating the same example in multiple frameworks, I'll focus on helping you build the right mental model, which will enable you to master any micro-frontend framework rather than memorizing only one or two of the options available.

We will definitely explore some code, but I will stress the importance of understanding *why* a decision is made. This way, despite the approach and framework you use in your next project, you will be able to decide what the right direction is, independent of how familiar you are with a specific micro-frontend framework.

Remember the old saying, "Give a man a fish, and you feed him for a day; teach a man to fish, and you feed him for a lifetime"? Let's learn to fish.

The Project

Our project is an internal t-shirt ecommerce website for an enterprise organization. The site is composed of several subdomains:

- Home page
- Login
- Payment
- Catalog
- Account management

- Employee support
- FAQs

For our example in this chapter, we'll use only some of these subdomains: home page, authentication, catalog, and account management. The ecommerce site must have a consistent user interface so that users will have a cohesive experience while they shop for their favorite t-shirt. We will have several teams responsible for delivering this project. To meet the project deadline, the tech department decides to reuse an internal engine developed for its B2C ecommerce solutions. It's a monolithic backend architecture that's been battle-tested after several years of development and hardening within production environments. However, the tech department wants to move away from siloing the frontend and backend expertise, so it decides to use micro-frontends, setting up independent teams responsible for a subdomain of the new ecommerce site. It will also use backend developers to re-architect the backend using microservices, and will incorporate agility at the business and technical levels.

The next step is to assign the teams responsible for the different subdomains:

- *Team Sashimi* will be responsible for the home page subdomains. Because this is an internal ecommerce site, the team will implement the login form using the centralized authentication system available, which employees use to access every system inside the organization. It will also be responsible for user authentication and personal-information gathering for the “account details” micro-frontend. One team member will be the full stack developer, and the rest will focus on the backend integration with Microsoft **Active Directory (AD)**.
- *Team Maki* will own the core domain—the catalog. This is the largest team and will be responsible for the main user experience. The team will be split between frontend and backend developers.
- *Team Nigiri* will cover the payments subdomain. It will integrate different payment methods, such as credit cards and PayPal.

The flow that we'll implement is composed of three sections. [Figure 4-1](#) shows that a user is presented with the home page, and then they authenticate to see the full catalog. Catalog micro-frontends will be a vertical-split micro-frontend, while the “My account” section will be composed of two horizontal-split micro-frontends. We can have just one developer working on the authentication frontend because the heavier part of the implementation is on the backend. For the catalog, however, we want a richer user experience, so we will need a team with in-depth knowledge of frontend practices.

Finally, because account management is an intersection of different subdomains, the two teams responsible for those subdomains will help develop this view.

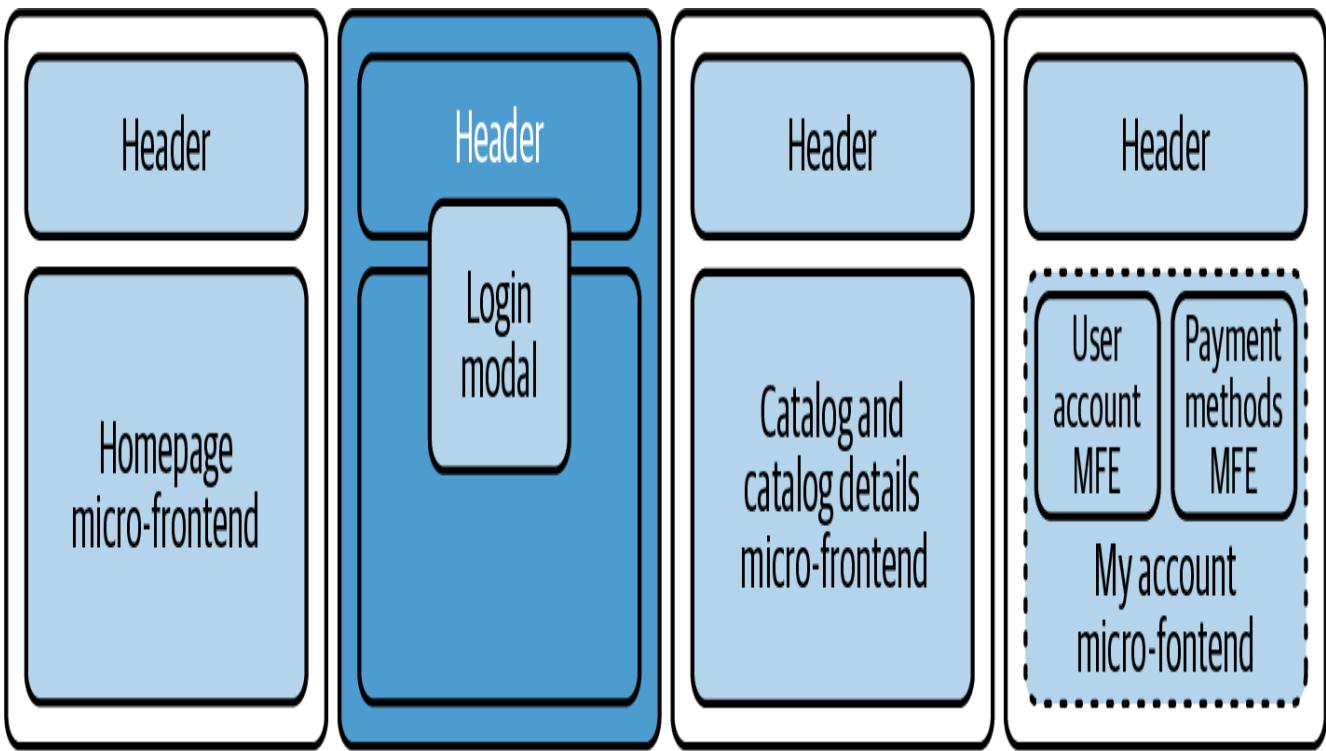


Figure 4-1. The t-shirt ecommerce sections: home page, login, catalog with product details, and account management

Following the micro-frontend decision framework and testing their assumptions with proof of concepts, the teams have decided to use the following:

A hybrid approach for identifying micro-frontends

Instead of using either a horizontal or vertical split for the whole project, the teams decided to select the most appropriate approach for each subdomain. A vertical split is more suitable for achieving the business requirements for authentication and the catalog, considering they are assigned to the same team and they will be capable of managing the entire subdomain without many external dependencies. On the other hand, a horizontal split for the account management subdomain fulfills the business need to have multiple subdomains.

A client-side composition

A client-side composition supports the requirements of the internal ecommerce site and fits within the team's skill sets. This composition

also allows for future evolution to other platforms, such as desktop applications or even progressive web applications.

Client-side routing

Once we decide to use client-side composition, the decision framework helps us easily decide that the routing should happen on the client side as well. We also have to consider that there will be two types of routing: a global routing handled by the micro-frontend container (also called the application shell), which will be responsible for routing between micro-frontends, and a local routing inside the catalog subdomains, where the Maki team will develop micro-frontends with multiple views.

Communication between micro-frontends embracing the decision framework suggestions

Also following the decision framework suggestions, Team Sashimi will store the session token in a cookie and create middleware to augment every request to an API by decorating the header with a bearer token. Because the account management view will have two micro-frontends—and potentially more in the future—we want to maintain the teams and the artifacts independently from each other. As a result, we'll use an event emitter to handle communication between the micro-frontends and the application shell, defining up front the events triggered by every micro-frontend along with their related payloads.

As noted at the start of the chapter, the technology chosen for this ecommerce site is Module Federation in conjunction with React. After several proofs of concept, the teams felt that Module Federation would provide everything they needed to successfully release this project. The main reasons for embracing Module Federation over other solutions are as follows:

Existing Webpack knowledge

Webpack is widely used inside the organization. Many developers have used this JavaScript bundler for other projects, so they likely won't have to learn a new framework. Module Federation fits nicely

within their technology stack, considering it's just a plug-in of a well-known tool for the company.

A client-side composition

With the micro-frontend composition based on the client side, Module Federation will provide a simple way to asynchronously load JavaScript bundles. It was developed initially for this specific use case and then extended to server-side rendering. Therefore, if the requirements change in the future, the teams will be able to change the micro-frontend implementation while maintaining Module Federation as stable assets for the evolution of their platform.

A seamless developer experience

The teams have significant expertise with Webpack. In addition, the implementation in the automation pipeline and the local development tools remain the same, so it's a great way for the teams to be immediately productive.

Module Federation was chosen for specific reasons for this project. For other projects, it may not be the right choice. It's important to analyze your team structures, developers' skills, the tech stack used in other projects by the company, and the project's business goals before deciding which micro-frontend architecture is suitable for your use case. After analyzing the context that you're working in, use the micro-frontend decision framework to create a solid foundation for future decisions for your project.

Module Federation 101

Before jumping into the technical implementation, we need to understand a few basic concepts to appreciate the reasoning behind some technical decisions. Module Federation allows a JavaScript application to dynamically load and run code from another bundle—making micro-frontends a perfect use case for this capability. Version 2.0 marked a massive step forward from the previous version. In fact, Module Federation 2.0 is now available for Webpack, Rollup, Vite, Rspack, and more to come. The previous coupling to Webpack is gone, and it works well for both client-side and server-side applications.

Module Federation provides two key concepts that we must understand before working with it:

Host

The container that loads shared libraries, micro-frontends, or components at runtime

Remote

The JavaScript bundle we want to load inside a host

As we can see in [Figure 4-2](#), a host can load multiple remotes. In our case, the host represents the application shell, while a remote represents a micro-frontend.

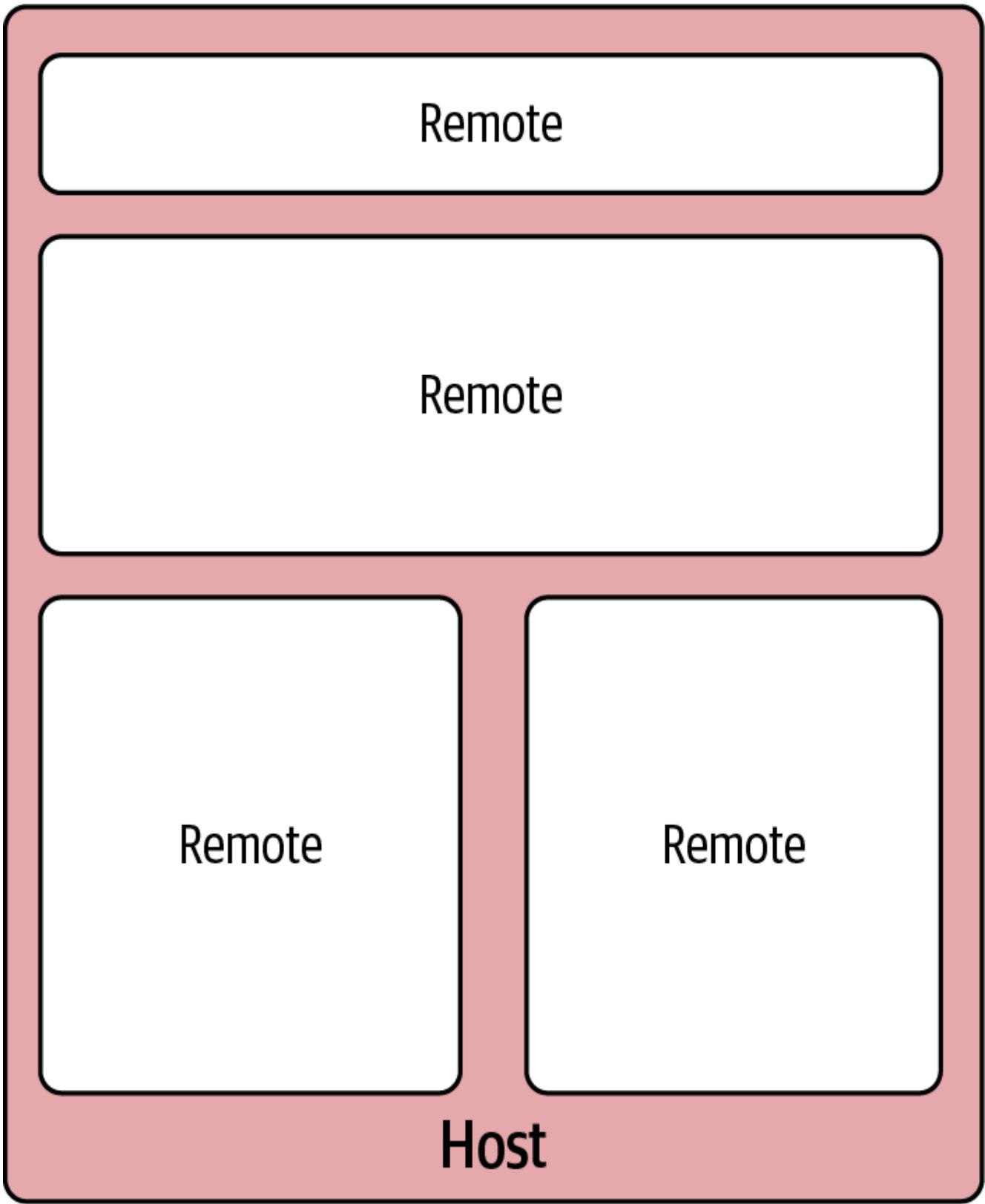


Figure 4-2. Module Federation, composed of two key elements: the host, responsible for loading some JavaScript bundles at runtime, and the remote, responsible for any type of JavaScript bundles, such as shared libraries, micro-frontends, or even components

With Module Federation, sharing can be bidirectional, allowing a remote to share parts of or the whole bundle with a host and vice versa. However, bidirectional sharing can

complicate your architecture very quickly. For example, imagine a “product catalog” micro-frontend exposing a shared product card component, while also depending on a cart micro-frontend for a pricing utility. This mutual dependency creates a circular reference: any change in one micro-frontend can potentially break the other, leading to coordination overhead and increased coupling between teams. In large systems, this type of entanglement reduces autonomy and undermines the benefits of independently deployable artifacts.

The best approach is sharing unidirectionally, so that a host never shares anything with remotes. This makes debugging easier and will reduce the potential for domain leaks to the host, which could cause design coupling between hosts and remotes.

Leveraging this architecture enables us to not only specify remotes in a selected bundler configuration but also to load them using JavaScript in our code. For instance, we can fetch the routes from an API and generate a dynamic view of remotes based on the user’s country or role. This is the right implementation for multi-environment systems like the ones we are used to working with on a daily basis.

Because Module Federation has plug-ins for multiple bundlers, we can use other bundler capabilities to optimize the code in the best way for our project. For instance, Module Federation creates many JavaScript chunk files by default, but we may prefer a less chatty implementation for our remote, loading just two or three files. Let’s assume the project bundler is Webpack. We could use [MinChunkSizePlugin](#), which forces Webpack to divide the chunks with a minimum number of kilobytes per file. We could also use [DefinePlugin](#) to replace variables in our code with other values or expressions at compile time. Using this plug-in, we can easily create some logic to provide the right base path when we are testing code locally or when it’s running on our development environments. Combined with other plug-ins available in the bundler ecosystem, Module Federation can be a powerful, suitable way to tweak your outputs for your context.

We should now have enough Module Federation knowledge to dive deeper into the implementation details. The project we’re exploring here shares many of the configurations available for Module Federation in a micro-frontend project. For more details, check out the [official documentation](#).

Technical Implementation

Now that we've reviewed the context where the application will be developed, and we have applied the decision framework and selected the technical strategy, it's time to look at the implementation details. The t-shirt ecommerce repository is [available on GitHub](#), so you can review the entire project or clone it and play with it. I intentionally developed the example without any server interaction so that you can run it locally without any external dependencies.

Just to recap, the application is composed of an application shell that is available during the entire user's session, loading different micro-frontends—such as the authentication, the catalog, and the “account management” micro-frontends. For the technology stack, the teams chose React with Webpack and Module Federation, enabling every team to create independent micro-frontends. Using Module Federation, they can share common dependencies and load them only once during a user's session. This creates a seamless experience for users without compromising developer experience. Let's dive into the key aspects of the main parts of this application.

Project Structure

Before going ahead with the case study, I want to say a few words on the structure I created for the t-shirt ecommerce project. When you clone the repository, you will see multiple folders, as shown in [Figure 4-3](#).

- >  AppShell
- >  CatalogueMFE
- >  HomeMFE
- >  MyAccount
- >  UserDetails
- >  UserPaymentMethods
-  .gitignore
-  frontend-discovery.json
-  README.md

Figure 4-3. The t-shirt ecommerce project folder structure

Every folder represents an independent project. Because they are present in the same repository. This is a monorepo approach, but they could easily be extracted in a polyrepo approach.

All the folders have a similar structure, as you can see in [Figure 4-4](#).

- ✓  **AppShell**
 - >  **node_modules**
 - >  **public**
 - >  **src**
 -  **package-lock.json**
 -  **package.json**
 -  **README.md**
 -  **webpack.config.js**
- ✓  **CatalogueMFE**
 - >  **node_modules**
 - >  **public**
 - >  **src**
 -  **package-lock.json**
 -  **package.json**
 -  **webpack.config.js**
- ✓  **HomeMFE**
 - >  **node_modules**
 - >  **public**
 - >  **src**
 -  **package-lock.json**
 -  **package.json**
 -  **webpack.config.js**

Figure 4-4. Micro-frontend structure

Application Shell

As previously described, the application shell remains present throughout the user's entire session. As it is essential for orchestrating micro-frontends but does not belong to any specific business subdomain, the teams decide to assign its implementation to a newly formed team of principal engineers: Team Sasazushi. Because building this part of the system requires minimal effort, and ongoing maintenance is expected to be low—given that the application shell does not own any business domain logic—the principal engineers were chosen for this team in addition to their primary roles within their respective teams.

Team Sasazushi is responsible for:

- Preventing domain leakage in the application shell
- Implementing global routing between micro-frontends
- Ensuring micro-frontends are correctly mounted and unmounted
- Managing cross-domain dependencies, bundling them into one or multiple JavaScript chunks

Additionally, given that the team consists of principal engineers, it also oversees the overall performance of the system. This includes establishing recurring meetings with other teams to share optimization best practices and reviewing performance bottlenecks identified during assessments.

Now, let's analyze the Webpack configuration. As Module Federation is a Webpack plug-in, we can import it just like any other JavaScript library:

```
const { ModuleFederationPlugin } = require('@module-federation/enhanced');
```

The most basic Webpack configuration consists of an entry file, an output folder, and a mode that determines how the code is transpiled—typically set to either `development` or `production`:

```
module.exports = {
  entry: "./src/index",
  mode: "development",
  output: {
    publicPath: "auto",
```

```
  },
  // additional configuration
}
```

Beyond this basic setup, we usually define rules to support specific language features or frameworks. In this case, we are using `css-loader` to handle styles and Babel with the React preset to enable JavaScript XML (JSX) support:

```
module: {
  rules: [
    {
      test: /\.css$/,
      use: ['style-loader', 'css-loader'],
    },
    {
      test: /\.jsx?$/,
      loader: 'babel-loader',
      exclude: /node_modules/,
      options: {
        presets: ['@babel/preset-react', '@babel/preset-env'],
        plugins: ['@babel/plugin-transform-runtime'],
      },
    },
  ],
},
```

However, the most critical aspect of our micro-frontend architecture is configuring Module Federation to load remote micro-frontends. As the application shell serves as the container for these micro-frontends, it acts as the *host* in Module Federation terminology.

Many online examples demonstrate how to specify remotes directly within the Module Federation plug-in of the chosen bundler. However, in production environments, this approach is less common—I have rarely seen teams implementing it this way. We will explore this point in more detail later in this chapter. Here is the configuration I used on the Webpack config file:

```
new ModuleFederationPlugin({
  name: 'shell',
  filename: 'remoteEntry.js',
  shared: {
    'react-router-dom': {
      singleton: true,
      requiredVersion: '6.21.3'
    },
    react: {
```

```

        singleton: true,
        requiredVersion: '18.2.0'
    },
    'react-dom': {
        singleton: true,
        requiredVersion: '18.2.0'
    }
}
],
}

```

After defining the name—in this case, *shell*—for a host, we specify the libraries that we want to share across all micro-frontends. When working with other micro-frontend frameworks, handling shared dependencies requires careful consideration.

A common approach is to create an independent repository for shared libraries and dependencies. Developers then establish an automated pipeline for building and deploying this shared code, along with governance policies covering update responsibilities, package size constraints, rollback procedures, and deployment strategies. However, with Module Federation, this process is significantly simplified. Instead of managing a separate repository, we only need to specify the shared dependencies in both the host and remotes—in our case, the application shell and all micro-frontends. Webpack and Module Federation will then generate multiple JavaScript files and ensure that each dependency is downloaded only once per user session, regardless of how many micro-frontends require it.

This may seem straightforward, but in practice, it's often more complex. Module Federation significantly simplifies shared dependency management in micro-frontends compared to traditional external shared library pipelines. Defining shared libraries in the Module Federation configuration is as simple as shown in the following code snippet:

```

//...
shared: {
    'react-router-dom': {
        singleton: true,
        requiredVersion: '6.21.3'
    },
    react: {
        singleton: true,
        requiredVersion: '18.2.0'
    },
}

```

```
'react-dom': {  
    singleton: true,  
    requiredVersion: '18.2.0'  
}  
}  
  
//...
```

In the `shared` object, we specify the libraries that should be shared across micro-frontends. Module Federation also provides advanced APIs that allow us to fine-tune how these dependencies are handled.

One option is to ensure that a library is loaded only once by using the `singleton` property, as demonstrated in our example. This setting signals to Module Federation that when another micro-frontend with the same dependency and version is required, only one instance should be loaded. An additional optimization is to set `import` to `false` on the remotes, so they won't bundle the shared dependency, knowing that the application shell already has a version in memory when a remote is loaded. We can also define a specific version of a library to maintain compatibility across micro-frontends. In our case, we are using a predefined set of React libraries, including `React`, `ReactDOM`, and `React Router DOM`, to ensure consistency.

Additionally, we can configure the `shareScope` property, which determines where these libraries will be appended. We will explore this approach in more detail when examining some of the micro-frontends in this project.

SHARED LIBRARY VERSIONS

When using Module Federation, if you don't specify a required version for a shared library, it defaults to the version declared in your application shell's `package.json` file.

When multiple applications share the same module, Module Federation compares the versions declared in their `shared` configurations. If a dependency is `semver-compatible` with the version already loaded, that version will be reused without issues. If there's a mismatch that falls outside the required range, a warning will appear in the console.

To gain more control over this process, additional configuration options like `requiredVersion` and `singleton` can be used. Defining a `requiredVersion` ensures that your application only accepts versions within a specified range, reducing the risk of compatibility issues. The `singleton` option guarantees that only one instance of the shared module is loaded, maintaining consistency across all micro-frontends.

By leveraging these options, you can prevent version conflicts and ensure seamless interoperability between your micro-frontends.

Module Federation allows dependencies to be loaded either synchronously or asynchronously. Synchronous loading can be enabled using the `eager` property in the plug-in configuration. However, the recommended approach is to load dependencies asynchronously. This prevents users from having to download all dependencies in a large up-front bundle, helping to maintain key performance metrics such as *time-to-first-byte (TTFB)* and *time-to-interactive (TTI)*.

To enable asynchronous loading, we need to split the application's initialization into multiple files. In our setup, the application shell is divided into three main files: `index.js`, `bootstrap.js`, and `app.js`.

The `index.js` file serves as the entry point of our application and contains just a single line of code:

```
import("./bootstrap");
```

The `bootstrap.js` file is responsible for instantiating the application shell and mounting the React application inside a `<div>` element called `root`, which is present in

the HTML template:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './setupFetch';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
);
```

Let's take a moment to discuss the `setupFetch` module found in the application shell bootstrap. In a micro-frontend architecture, authentication and authorization can be managed in different ways. One common approach is to implement a middleware in the application shell that automatically appends a JSON Web Token (JWT) to every fetch request made by the micro-frontends. This centralizes authentication, ensuring a consistent and standardized process across all API calls.

Here's an example of how this might be implemented:

```
const originalFetch = window.fetch;

window.fetch = async (input, init = {}) => {
  const jwtToken = 'your-jwt-token-here';
  const headers = new Headers(init.headers || []);
  headers.append('Authorization', `Bearer ${jwtToken}`);

  const modifiedInit = {
    ...init,
    headers,
  };

  return originalFetch(input, modifiedInit);
};
```

With this middleware in place, every outgoing request from any micro-frontend is automatically enriched with the correct `Authorization` header containing the JWT. This approach works particularly well in horizontal-split micro-frontend architectures, where the shell is responsible for orchestrating shared logic and managing cross-cutting concerns like authentication.

However, this method isn't without its trade-offs. In a vertical-split micro-frontend architecture, where each micro-frontend represents a distinct business domain and is

often owned by different teams, authentication and authorization may need to be handled independently. Overriding `fetch` globally can risk compromising calls to endpoints that don't require authentication or authorization. To avoid this, it can be wise to expand the example so that each micro-frontend decides whether or not to include the token, ensuring security is applied only where it's truly needed.

That said, care must be taken in the implementation. For instance, authentication middleware should not intercept `fetch` calls made by remotes to other domains unless explicitly intended. Doing so risks unintentionally leaking tokens across domain boundaries, violating security expectations, and introducing hard-to-detect bugs.

Additionally, this middleware should be considered a starting point rather than a production-ready solution. Real-world implementations need to account for complexities such as token-expiration handling, error management, and compatibility with different UI frameworks or libraries.

If you choose to adopt this approach, be sure to test it thoroughly within your environment and refine it to handle these edge cases. Not all the frameworks or libraries use the `fetch` API as the base for their HTTP calls. Therefore, make sure you properly identify the method or methods used before overriding.

Finally, `App.js` will contain two key elements. First, the `Main` component defines the basic structure of the user interface, including the application's header. Second, React Router, a widely used routing solution for React applications, will manage global navigation across micro-frontends.

Now, let's take a look at the JSX part of the `App.js` file:

```
return (
  <Router>
    <div>
      <Header />
      <main style={{ maxWidth: '1200px', margin: '0 auto', padding: '2rem' }}>
        {isLoading ? (
          <Loading />
        ) : (
          <Routes>
            {routes.map((route, index) => (
              <Route
                key={index}
                path={route.path}
                element={<System request={route.request}
                  mfInstance={mfInstance} />}
              />
            )))
        )
      </main>
    </div>
  </Router>
)
```

```

<Route
  path="*"
  element={
    <div style={{ textAlign: 'center', padding: '2rem' }}>
      <h2>Page not found</h2>
      <p>Current path: {window.location.pathname}</p>
      <p>Available routes: {routes.map(r => r.path).join(', ')</p>
    </div>
  }
/>
</Routes>
)
)
</main>
<NotificationModal emitter={emitter} />
</div>
</Router>
);

```

As you can see, all the routes are dynamically generated. In mid-to-large-size applications, it's highly unlikely that you'll define all remotes directly in the Webpack configuration. The main reason is that most teams work with a multi-environment strategy, where the application is composed of different endpoints depending on the deployment environment.

Even when testing micro-frontends, having the flexibility to load different versions of micro-frontends—especially those that your team doesn't own—is crucial. Imagine your team has been developing a new feature for weeks. During final testing, you need to ensure that your code doesn't conflict with work done by other teams. To achieve this, you might test the latest version of your micro-frontend in a development environment while pulling stable versions of other micro-frontends from staging or another preproduction environment.

When implementing dynamic routes, make sure to include appropriate error handling for unexpected cases. This includes rendering custom error pages, such as `404 Not Found` for invalid paths and `5xx` errors for server issues. Clear error boundaries improve the user experience and help isolate failures without affecting the rest of the application.

Another key consideration is page routing within a micro-frontend architecture. Let's start with global routing. When using an application shell, routing happens on the client side, making the shell responsible for navigating between micro-frontends. Typically, URLs have multiple levels. For example, in

<https://www.mysite.com/catalog/product/123>, the application shell should handle only the first-level routes (e.g., `/catalog`). Once the user lands on that route, one or more micro-frontends take over to manage deeper navigation locally within their domain.

In this setup, the application shell is responsible for *global routing*—determining which micro-frontend to load based on top-level URL segments. Each micro-frontend then handles its own *local routing*, managing navigation between internal views. This separation ensures clear ownership and encapsulation of routing logic across business domains, reducing the complexity of cross-team coordination.

This approach is critical because it ensures that the application shell remains as domain-agnostic as possible. By dynamically loading only the first-level URLs and delegating the rest of the navigation to individual micro-frontends, we prevent unnecessary dependencies between teams. This accelerates the release process, as each team remains autonomous—one of the primary benefits of adopting a micro-frontend architecture.

To load routes dynamically, we invoke a function called `initializeMFEs`. This function is responsible for fetching the route configuration and registering the micro-frontends with Module Federation:

```
const [routes, setRoutes] = useState([]);
const [isLoading, setIsLoading] = useState(true);
const [isInitialized, setIsInitialized] = useState(false);
const [mfInstance, setMfInstance] = useState(null);

const initializeMFEs = useCallback(async () => {
  if (isInitialized) return;

  try {
    // Get the existing instance created by webpack
    const instance = getInstance();
    setMfInstance(instance);

    const response = await fetch('http://www.mysite.com/MFEDiscovery.json');
    const data = await response.json();

    const remotes = [];
    const routeConfigs = [];

    for (const [_, configs] of Object.entries(data.microFrontends)) {
      const config = configs[0];
      const { name, alias, exposed, route, routes } = config.extras;

      // Register the remote MFE
      remotes.push({
        name,
        alias,
        entry: config.url
      });
    }
  }
});
```

```

// Generate routes for this MFE
const mfeRoutes = generateRoutesForMFE(name, exposed, route, routes);
routeConfigs.push(...mfeRoutes);

}

await instance.registerRemotes(remotes);

setRoutes(routeConfigs);
setIsInitialized(true);
setIsLoading(false);
} catch (error) {
  console.error('Failed to initialize MFEs:', error);
  setIsLoading(false);
},
[isInitialized]);

useEffect(() => {
  initializeMFEs();
}, [initializeMFEs]);

```

There are two key aspects of Module Federation that we need to consider in this method:

Initializing Module Federation

When settings are defined in the bundler, the code can retrieve the generated instance with `getInstance` and append any additional configuration—such as dynamically retrieved remotes. Alternatively, you can create a new Module Federation instance with `createInstance` and configure everything directly within the application code.

Registering remotes at runtime

A remote must be registered inside the `createInstance` method configuration or via the `registerRemotes` function to be loaded dynamically.

At the beginning of this example, we did not specify the remotes statically inside the Webpack Module Federation plug-in. This is why we need to register them dynamically when initializing the application shell.

In [Chapter 8](#), we'll explore how to use a discovery service to retrieve routes dynamically. For now, let's assume we're using a simple JSON configuration hosted within our infrastructure.

As we can see, the `Route` object consists of a path (representing the first-level URL) and the corresponding micro-frontend to be loaded. To streamline this process, I've created a `System` function that reads the JSON configuration and dynamically loads a remote after registering it:

```
const System = ({ request, mfInstance }) => {
  if (!request) {
    return <h2>No system specified</h2>;
  }

  // Use simple Module Federation without React Bridge
  const MFE = React.lazy(() =>
    mfInstance.loadRemote(request)
      .then(module => ({
        default: module.default
      }))
  );

  return (
    <React.Suspense fallback={<div>Loading...</div>}>
      <MFE emitter={emitter} />
    </React.Suspense>
  );
};
```

In this function, I'm using the `loadRemote` method from Module Federation to asynchronously load a micro-frontend. I then return a `Suspense` component with the loaded micro-frontends, injecting an emitter that will be needed for communication between micro-frontends later on.

When the user selects the new ecommerce area from the home page, the router will load the new micro-frontend in a manner similar to how we would lazy-load a regular React component in an SPA. Module Federation will handle the process of importing the remote module for you.

Home Micro-Frontend

The first micro-frontend to be loaded in the application shell is for the home page. It doesn't have any micro-frontend-specific code, so if you navigate through the code, you won't spot any particular micro-frontend implementation. However, I decided to add a bit of spice to this domain.

One of the main benefits of micro-frontends is reducing external dependencies between teams. Imagine your team has a huge backlog of stories to deliver, and you receive a

request from the application shell team to update React to version 18. With tight deadlines, you probably don't want another story to handle this update, as it could risk delaying more important features. Meanwhile, other teams may have already updated their own micro-frontends to the latest version in no time.

To avoid this risk, you can leverage another powerful feature of micro-frontends: the ability to manage different library versions in the same application.

When you look at the home page's Webpack configuration, you'll notice that it differs from the application shell's configuration:

```
new ModuleFederationPlugin({
  name: 'HomeMFE',
  filename: 'remoteEntry.js',
  exposes: {
    './MFE': './src/App'
  },
  shared: {
    'react-router-dom': {
      singleton: true,
      requiredVersion: '6.21.3'
    },
    react: {
      import: 'react',
      shareScope: 'react17',
      singleton: true,
      requiredVersion: '17.0.2'
    },
    'react-dom': {
      import: 'react-dom',
      shareScope: 'react17',
      singleton: true,
      requiredVersion: '17.0.2'
    }
  }
}),
```

In this case, I have created a different scope (or container, in Module Federation terms) by leveraging the `shareScope` property. This property allows the micro-frontend to append the `React` and `ReactDOM` libraries, avoiding a version clashes with the default scope used for the rest of the libraries. In fact, `React Router DOM` is retrieved from the default scope because that version works just fine with both React 17 and React 18.

Module Federation 2.0 provides a way to inspect shared resources through the browser's developer tools. All shared resources are available in the `window.__FEDERATION__.__SHARE__` object.

This object contains:

- All shared dependencies organized by micro-frontend name
- Version information for each shared resource
- The current state of shared modules

For example, you can inspect:

```
// See all shared resources for HomeMFE
window.__FEDERATION__.__SHARE__.HomeMFE
```

This is particularly useful for:

- Debugging shared dependencies
- Verifying correct version sharing
- Confirming proper isolation of React versions (e.g., React 17 versus 18)
- Understanding how Module Federation manages shared resources at runtime

Here's an example of what you might see:

```
window.__FEDERATION__.__SHARE__.HomeMFE
HomeMFE:1.0.0:
  default: {react-dom: {...}, react-router-dom: {...}, react: {...}}
  react17:
    react: {17.0.2: {...}}
    react-dom: {17.0.2: {...}}
    [[Prototype]]: Object

window.__FEDERATION__.__SHARE__.UserPaymentsMFE
default:
  react: {18.2.0: {...}}
  react-dom: {18.2.0: {...}}
  react-router-dom: {6.21.3: {...}, 6.30.0: {...}}
  [[Prototype]]: Object
```

I think it's a very handy method of verifying your configurations without delving too deep into the internals of Module Federation!

NOTE

Remember, the ability to isolate libraries is also available with other approaches like using scopes with `import maps` or `SystemJS`. This capability is great because it is spread across multiple libraries leveraging web standards like Native Federation or `single-spa`, for example.

Although you can handle multiple library versions in the same application, I urge you not to optimize for this approach. When I was building this demo, I encountered several incompatibilities across libraries, and it wasn't a fun experience at all. Moreover, designing an application that downloads more code than users need is far from being user-centric.

The following subsections are situations, like the one I described, that buy you time to update the libraries, but don't abuse this feature.

Catalog Micro-Frontend

The catalog domain is probably the most complex and largest of all the micro-frontends. It's the reason that users are going to the website, so it has to not only be simple to use but also provide all the information the user is looking for. Team Maki is responsible for this micro-frontend, and its goal is to implement multiple views so users can discover what's available in the catalog and get the details of each product. In the future, the team may have to add new functionalities, such as sharing product images taken by a buyer or adding a review score with comments.

The team will implement all these features and prepare the codebase in a modular fashion, such that in the future it will be easy to hand over part of the domain to another team if needed. Strong encapsulation and solid modularity will enable Team Maki to easily decouple parts of the domain and collaborate with other teams to provide a great user experience.

This vertical-split micro-frontend's peculiarity is that we have to handle multiple views inside the same micro-frontend—a kind of SPA specific to the catalog domain. This shouldn't preclude the possibility of adding shared or domain-specific components, such as a personalized products component, implemented by other teams inside this domain. Although the application shell is responsible for the global routing, this micro-frontend requires us to implement local routing—that is, a routing implemented at a micro-frontend level—that works in conjunction with the global one. The routing implementation code is as follows:

```

function Catalog() {
  const navigate = useNavigate();
  const { productId } = useParams();

  const handleViewDetails = (productId) => {
    navigate(`catalog/${productId}`);
  };

  // If we have a productId parameter, show the product details
  if (productId) {
    return <ProductDetails productId={productId} />;
  } else {
    console.log('No productId found');
  }
}

```

Using the React Router library, we initially retrieve the first level of URL depth. We then incrementally append the product ID to the depth level when a user selects a specific product. From the second level onward, the structure and management are usually fully handled inside the micro-frontend, so the domain can evolve autonomously, eliminating the need to coordinate its enhancements with other teams. This also prevents domains from creating overlapping first-level URLs because only one team is responsible for the global routing.

In this way, we prepare our codebase for potential future splits without too much effort. Query strings are a good way to hand over ephemeral information to another view or even to another micro-frontend. Because this type of data is consumed immediately by another part of the system and doesn't need to be stored for long, using query strings for passing them across the system is strongly recommended.

Account Management Micro-Frontend

Team Nigiri, responsible for the payment subdomain, and Team Sashimi, responsible for the user's account, have to collaborate for the account management view, with part of the payment information and part of the user's details converging in the same view. Because these domains are assigned to different teams, we need a different approach to the other micro-frontends. Instead of a vertical-split micro-frontend architecture, we'll use a horizontal split to compose the final view and allow the two domains to communicate with each other for specific user interactions. To achieve this, we'll need to create a new host, which I usually call "meta shell," and two remotes.

We know every remote is associated with a team, but what about the new host? Together, Teams Nigiri and Sashimi define the strategy for evaluating this common

container of their subdomains. The new host has a clear implementation path that consists of loading two micro-frontends: a user’s details and payment details.

Because of this, Team Nigiri decides to take ownership of the new host and collaborate with Team Sashimi to define mechanisms to ensure the developer experience is as smooth as possible and the releases of the host don’t cause any issues with Team Sashimi’s work.

You may be asking yourself where the new host will actually be presented to the user. Module Federation allows us to nest multiple hosts, and there isn’t a strict hierarchical structure. In fact, there is a very thin line between remote and host, because a remote can expose some libraries used by the host and vice versa. Remember, we need to pay attention to this thin line: when it’s crossed and we start to share dependencies bidirectionally, we risk creating unmaintainable code that creates more problems than benefits in the long run. My recommendation is to enforce a hierarchical relation between host and remote, where the sharing is always unidirectional, so that the host will never expose any modules to its remotes. This simple but effective practice simplifies the micro-frontend architecture implementation and reduces the risk of potential bugs. Moreover, it improves application debugging by reducing the coupling between modules, avoiding a big ball of mud where multiple modules depend on each other. When we reduce the coupling and external dependencies in this way, each team will have the power to make the right decisions for the project, while keeping in mind that we may sometimes have to make compromises to achieve the organization’s business goals.

On the technical side, we have to account for some small changes to handle the horizontal approach. First, we need a container for the two micro-frontends—also called “meta shell” because it acts like a lightweight application shell. We’ll create a host called `MyAccount`, which will load the two remotes and have a similar configuration of a Module Federation plug-in like the other micro-frontends. The main difference is that `MyAccount` has to be a host, because it concerns the user’s details and payment micro-frontends, but it also must be a remote for the application shell. To do this, we add the `exposed` objects, as seen in the following code snippet:

```
// additional code before

new ModuleFederationPlugin({
  name: 'MyAccountMFE',
  filename: 'remoteEntry.js',
  exposes: {
    './MFE': './src/MyAccount'
```

```
},
shared: {
  react: {
    singleton: true,
    requiredVersion: '18.2.0',
  },
  'react-dom': {
    singleton: true,
    requiredVersion: '18.2.0',
  }
}
// additional code after
```

With just this change, we can have a host that is also a remote. However, we should avoid overusing this practice, as it can quickly lead to many small applications that represent individual components rather than entire business domains. Designing boundaries correctly is crucial. When we are unsure whether to create a new micro-frontend or incorporate a feature into an existing one, we should revisit the drawing board and ensure our decision aligns with the core principles of micro-frontends.

A key aspect of this implementation is the communication between micro-frontends. As the micro-frontend decision framework states, we need to share information between micro-frontends without directly sharing code across different domains, as would be the case with a global state. An ideal approach is leveraging the pub/sub pattern to keep domains decoupled. When a modification occurs within a domain, the corresponding micro-frontend emits an event to notify the change. The application shell, which includes a toast notification system accessible to all micro-frontends, listens for these events and displays notifications as needed. This ensures seamless communication without introducing unnecessary dependencies. See [Figure 4-5](#) for reference.

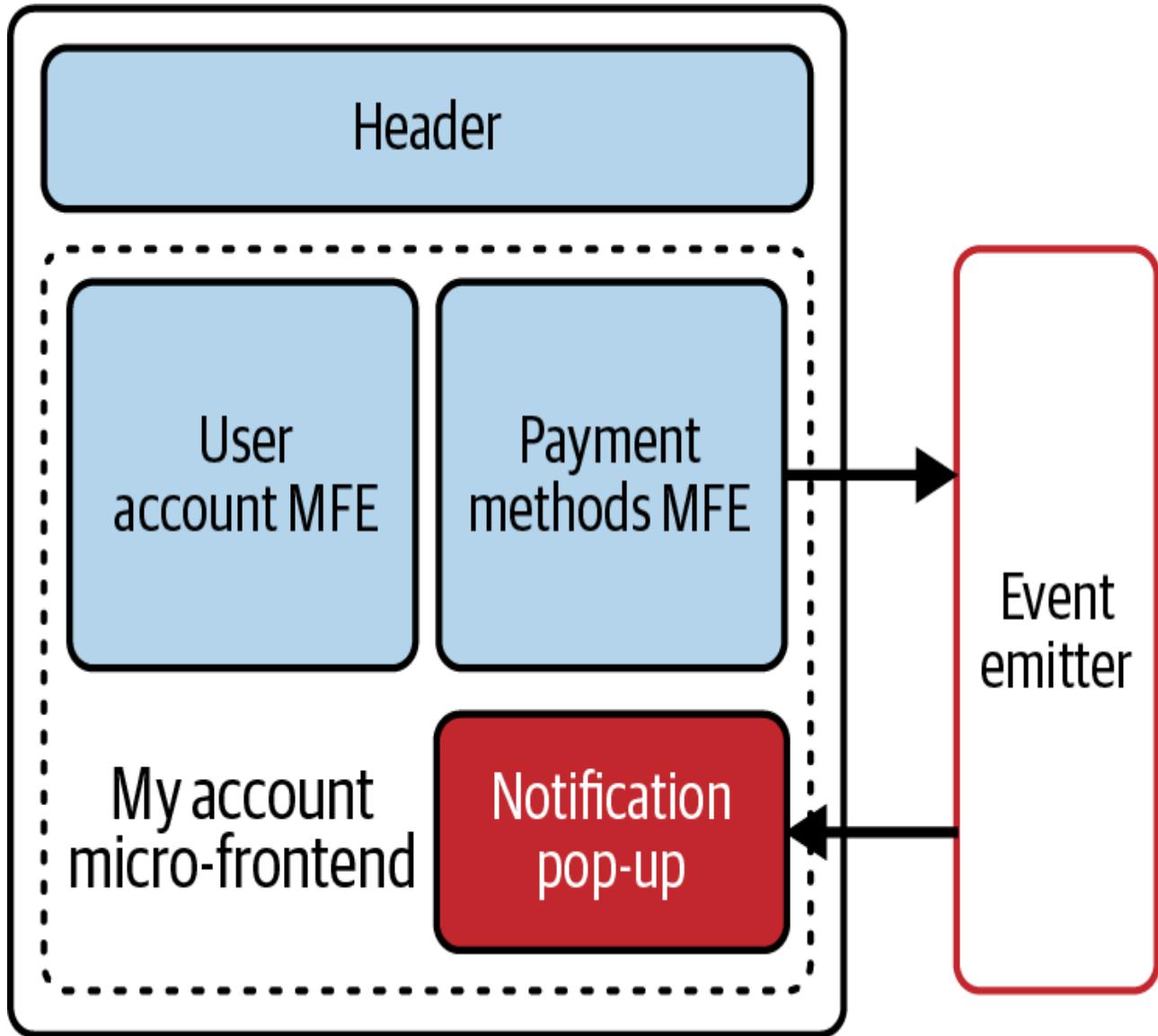


Figure 4-5. The event emitter shared between micro-frontends, allowing for communication across domain boundaries and maintaining their independence

In the host, we create the same mechanism used in the application shell, but this time we are loading the “user details” and “payment methods” micro-frontends:

```

function MyAccount({ emitter }) {
  const [mfeRequests, setMfeRequests] = useState([]);
  const [error, setError] = useState(null);
  const [isLoading, setIsLoading] = useState(true);

  const loadMfes = useCallback(async () => {
    try {
      const response = await fetch('http://127.0.0.1:8080/MFEDiscovery.json');
      const data = await response.json();
    }
  }, []);
}

```

```

const userMfes = [
  data.microFrontends.UserDetailsMFE[0],
  data.microFrontends.UserPaymentMethodsMFE[0],
];

await mfInstance.registerRemotes(
  userMfes.map(mfe => ({
    name: mfe.extras.name,
    entry: mfe.url,
  }))
);

const requests = userMfes.map(
  mfe => `${mfe.extras.alias}/${mfe.extras.exposed}`);
setMfeRequests(requests);
 setIsLoading(false);
} catch (err) {
  console.error('Error in loadMfes:', err);
  setError(err.message);
  setIsLoading(false);
}
}, []);

useEffect(() => {
  loadMfes();
}, [loadMfes]);

```

An optimization for this approach would be injecting the list of available micro-frontends into the “my account” micro-frontend, retrieved by the application shell. However, let’s consider the context. Making an additional call that is highly cacheable will not cause any performance issues. Moreover, this is the only micro-frontend that leverages a meta-shell approach. Therefore, if the system evolves, we can consider making this change. Otherwise, we can maintain the separation between the application shell and the “My account” data, keeping them independent and avoiding the need for team coordination.

After retrieving the information for the two micro-frontends, we render them inside the “My account” view:

```

return (
  <div style={{ padding: '20px' }}>
    <h1 style={{ marginBottom: '20px' }}>My Account</h1>
    <div style={{
      display: 'grid',
      gridTemplateColumns: 'repeat(auto-fit, minmax(300px, 1fr))',
      gap: '20px'
    }}>

```

```

{mfeRequests.map(request => (
  <System key={request} request={request} emitter={emitter} />
))
</div>
</div>
);

```

Now, let's dive into two other micro-frontends: user details and user payment methods.

The “user details” micro-frontend has the same configuration as the home micro-frontend. Because it uses React 17, it leverages the same Module Federation configuration and reuses the same dependencies if they are already loaded by the home micro-frontend:

```

new ModuleFederationPlugin({
  name: 'UserDetailsMFE',
  filename: 'remoteEntry.js',
  exposes: {
    './MFE': './src/UserDetails'
  },
  shared: {
    react: {
      import: 'react',
      shareScope: 'react17',
      singleton: true,
      requiredVersion: '17.0.2'
    },
    'react-dom': {
      import: 'react-dom',
      shareScope: 'react17',
      singleton: true,
      requiredVersion: '17.0.2'
    }
  },
}),

```

Looking at the “user payment methods” micro-frontend, we can see the implementation of micro-frontend communication. As explained in this chapter, the chosen approach is an event emitter because of its simplicity, compared to reactive streams, and its independence from the DOM hierarchy, unlike custom events. It should be your go-to approach for communication within micro-frontends.

I cannot stress enough the importance of avoiding global state managers across micro-frontends. State management should be handled inside each micro-frontend to keep them independent from one another. When a micro-frontend is loaded into the application

shell, an emitter is injected to facilitate communication between micro-frontends or between a micro-frontend and the application shell:

```
return (
  <React.Suspense fallback={<div>Loading...</div>}>
    <MFE emitter={emitter} />
  </React.Suspense>
);
```

That emitter is simply a singleton that cannot be extended, thanks to the use of the `Object.freeze` method:

```
import { EventEmitter } from 'tseep';

class EventEmitterSingleton {
  constructor() {
    if (EventEmitterSingleton.instance) {
      return EventEmitterSingleton.instance;
    }

    this.emitter = new EventEmitter();
    EventEmitterSingleton.instance = this;
  }

  emit(event, data) {
    this.emitter.emit(event, data);
  }

  on(event, callback) {
    this.emitter.on(event, callback);
  }

  off(event, callback) {
    this.emitter.off(event, callback);
  }
}

const emitter = new EventEmitterSingleton();
Object.freeze(emitter);

export default emitter;
```

Now, from every micro-frontend, I can emit or listen to an event without relying on a specific position inside the DOM, as is the case with custom events. In fact, every time the user changes the default payment method, we emit an event called `notification` with the message that the toast notification should display:

```

handleMakeDefault = (id) => {
  const { emitter } = this.props;

  this.setState(prevState => ({
    paymentMethods: prevState.paymentMethods.map(method => ({
      ...method,
      isDefault: method.id === id
    }))
  }), () => {
    const defaultMethod = this.state.paymentMethods.find(m => m.id === id);
    if (emitter) {
      const notification = {
        type: 'success',
        title: 'Payment Method Updated',
        message: `${defaultMethod.brand} ending in
          ${defaultMethod.last4} is now your default payment method.`
      };
      console.log('notification msg', notification);

      emitter.emit('notification', notification);
    }
  });
};

```

Bear in mind that, in this case, the event is fairly simple—consisting of just a title, a type, and a message. We keep it generic to ensure it's useful across all domains, considering the communication will be with a shared toast notification present in the application shell. However, this doesn't mean that all your events should be structured in the same way.

On the frontend, designing events for an event emitter in a micro-frontend architecture requires a lightweight, flexible, and decoupled approach to communication. Events should be namespaced to avoid collisions (e.g., `cart:itemAdded`, `auth:userLoggedIn`), and the event payloads should be consistent and predictable to ensure smooth integration across micro-frontends. The event emitter should support both global and local event handling, where global events (e.g., authentication changes) are broadcast at the application shell level, while local events (e.g., UI state changes) are scoped to individual micro-frontends. Events should be designed to avoid tight coupling; micro-frontends should listen for events rather than directly calling functions in other micro-frontends.

Then, we create a component in the application shell that handles the notification:

```

function NotificationModal({ emitter }) {
  const [isOpen, setIsOpen] = useState(false);
  const [type, setType] = useState('info');

```

```

const [title, setTitle] = useState('');
const [message, setMessage] = useState('');

useEffect(() => {
  const handleNotification = ({ type, title, message }) => {
    console.log('Notification received:', { type, title, message });
    setType(type);
    setTitle(title);
    setMessage(message);
    setIsOpen(true);
  };

  emitter.on('notification', handleNotification);

  return () => {
    emitter.off('notification', handleNotification);
  };
}, [emitter]);

const handleClose = () => {
  setIsOpen(false);
};

if (!isOpen) return null;

```

I hope this example helps you understand that Module Federation can be a great companion for your client-side rendering applications. One of the key benefits is the ease of sharing dependencies between different micro-frontends. In this example, we have a React 17 micro-frontend coexisting on the same view as a React 18 micro-frontend, with the “my account” micro-frontend using React 18. Module Federation allows both versions of React to be loaded without conflict, ensuring that each micro-frontend can operate independently while sharing common resources efficiently.

Leveraging micro-frontends is not as complex as it may seem when you have a clear understanding of what needs to be expressed within your web or mobile application. The example shared demonstrates that the code structure doesn’t differ significantly from building a standard client-side application with the framework of your choice. The key difference lies in defining clear boundaries and responsibilities for each micro-frontend and establishing an efficient communication strategy between them. This is where the real shift happens—understanding how micro-frontends interact while maintaining independence. This challenge is addressed by the micro-frontend decision framework, which provides a structured approach to designing, organizing, and implementing micro-frontends. By applying this framework to every micro-frontend application, teams can make informed decisions, ensuring scalability, maintainability, and seamless collaboration.

The use of event emitters, custom events, or reactive streams should be intentional and minimal, ensuring each micro-frontend remains loosely coupled and independently deployable. Additionally, careful attention must be given to routing strategies, distinguishing between global routing (handled by the application shell) and local routing (managed within each micro-frontend). By following these principles and relying on the micro-frontend decision framework, teams can design modular, adaptable frontends that align with the distributed nature of modern applications while maintaining a seamless user experience.

You can explore the complete codebase for this example in the accompanying [GitHub repository](#).

Project Evolution

We don't want to create a project that works just for a short period of time. We want to create one that can evolve, eliminating the need to start from scratch to advance our organization's business goals. Let's explore, then, some potential ways this project could evolve and how we can create a coherent implementation across different subdomains.

Embedding a Legacy Application

Imagine that we have to add a tool for customizing existing products—like socks, hoodies, and mugs—to our ecommerce project. The tool is a legacy application that was developed several years ago with an old version of Angular. Only one person from the team that developed this solution is still with the company, and they are keeping the lights on for this project, fixing bugs and optimizing the codebase where possible. To reduce the feature's time to market, the business and the tech department decide to integrate the legacy tool with the existing micro-frontend architecture and to ship it for a limited time. Later, a new team will take ownership of the project and revamp it to natively embrace micro-frontends. The application is well encapsulated, not requiring any particular information about the environment that is running, and we can pass the configuration needed to render a file to the configurator via a query string. Additionally, we want to minimize possible clashes with other parts of the codebase, such as the application shell.

We can solve this problem by wrapping the legacy application inside an iframe, as in [Figure 4-6](#), which will prevent any possible clash with the existing micro-frontend system.

However, if we want to communicate with the legacy application—such as by displaying errors across the entire interface instead of only in the iframe—we should create a communication bridge between the legacy application and the application shell in order to reuse the alerting system. We could directly integrate the application shell with the legacy application, but this would mean polluting the application shell codebase. We can implement a better strategy than that. Instead, we will apply an adapter pattern using a micro-frontend as a container for the iframe that contains the legacy application. The micro-frontend will be responsible for orchestrating the iframe using query strings and intercepting any messages from the legacy application, translating it into events emitted in the event bus.

Header

Navigation

Legacy app wrapped in an iframe

Legacy app container

Application shell

Figure 4-6. The application shell loading a micro-frontend that acts as an adapter between the new and old worlds, with the legacy application wrapped in an iframe to minimize the impact on the existing micro-frontend codebase

NOTE

The *adapter pattern* is a software design pattern (also known as a wrapper) that allows an existing class's interface to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

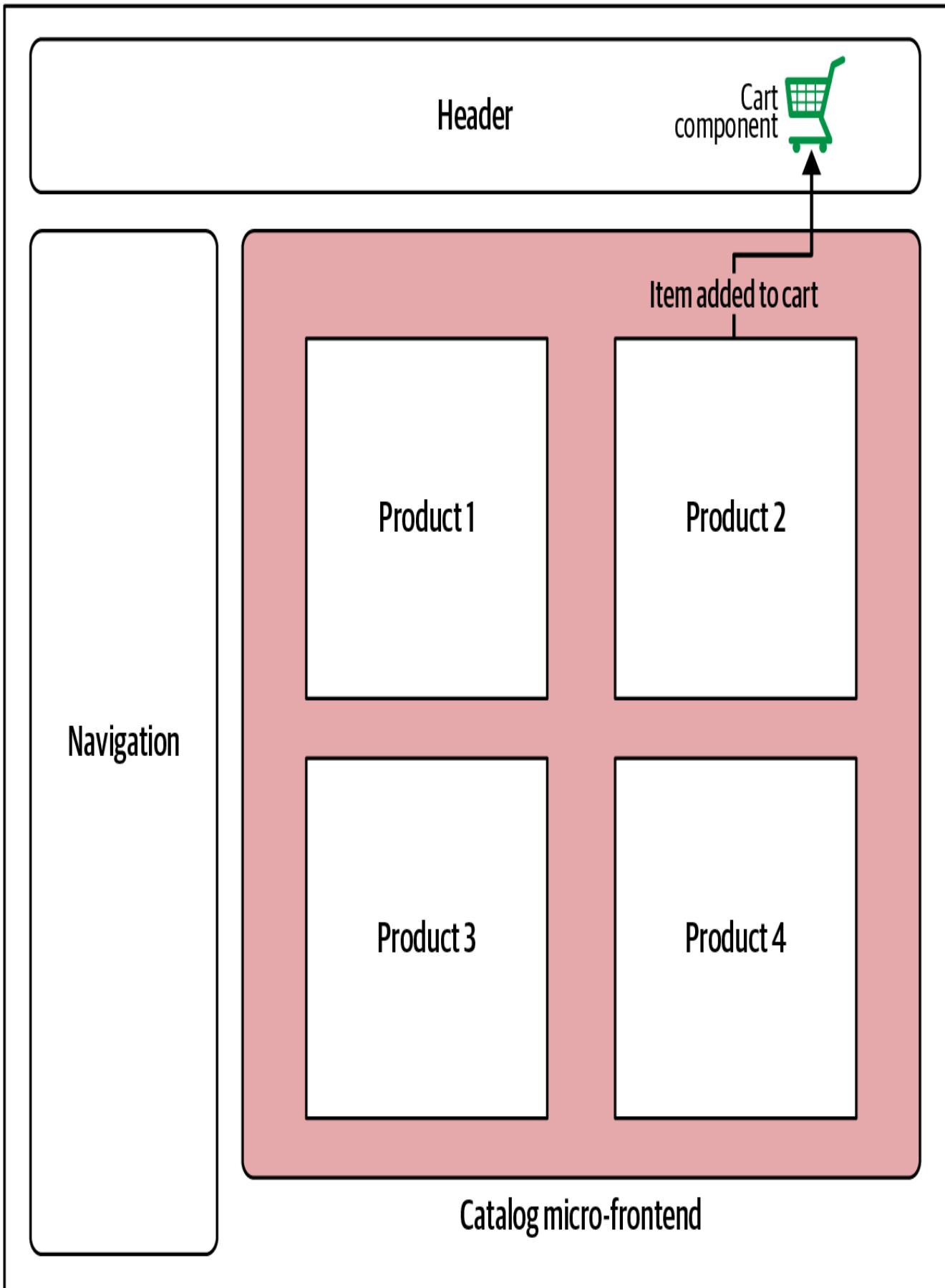
By using a micro-frontend as an adapter, we can prepare our project for future evolutions. We can also reduce any refactoring in the application shell, first for integrating the legacy application and then for substituting it with the new micro-frontend implementation. Within the application shell, we will maintain a business-unaware logic, since the communication will be translated to events via the event emitter. This process acts as an anticorruption layer between the inner and the outer systems. This pattern also comes in handy when we want to consolidate multiple applications under the same system and slowly but steadily replace every legacy application with micro-frontends, implementing a **strangler fig pattern**, which allows the micro-frontend application to live alongside the legacy ones.

Developing the Check-Out Experience

The project is finally getting traction inside the organization, and Team Nigiri is developing the check-out process. The product owner and the UX team have decided to place a cart inside the application shell header. The cart should be shown only when a user is authenticated in the shop, so that component should be visible only in certain views. When the cart button is clicked, the new “check-out” micro-frontend will guide the user to process the order correctly inside the system.

The cart component (see [Figure 4-7](#)) has different responsibilities:

- Hide and reveal the cart based on the area a user is navigating to.
- Display the total number of items in the cart.
- Start the check-out experience.



Application shell

Figure 4-7. The cart component, loaded inside the application shell, using an event emitter to listen for when an object should be added to the cart, and showing the total number of elements the user has selected

Because the cart component will be present in the application shell, we have to create a logic for hiding it when a user is not authenticated and for sharing it when they are. We could have the application shell orchestrate the component's visibility, but the check-out domain logic would leak into the application shell, which could pollute the codebase. Additionally, every time we want to change the visibility logic, we'd need to release a new application shell version. And because the check-out experience and the application shell are owned by different teams, creating such dependencies can only cause more trouble than benefit.

A better solution is to ask the team responsible for the application shell to add the component, and the component itself will handle its own visibility based on a set of conditions, such as the page URL. In this way, any logic change or improvement happens inside the component, and does not leak these implementation details to the application shell. The application shell team will need to upgrade the library used for loading the component if they used a compile-time implementation. In the case of Module Federation, they won't need to do anything else because the new component will be loaded at runtime.

To display the total number of items in the cart, we first need to add the product to the cart via an API exposed by the backend and then notify the cart component to update the value displayed in the interface. The best way to achieve this is by emitting an event via the event emitter instance. When the cart component receives the event, it will consume an API to retrieve the number of items currently in the cart.

Finally, when the user starts the check-out experience by clicking on the cart component, all that is required is changing the URL, which notifies the application shell to move on to the “check-out” micro-frontend.

As you can see, investing a bit of time up front to think through the implementation of a simple element like a cart component can make a great difference in the long run. This cart component will maintain strong encapsulation despite living inside a different domain (the application shell). It will receive events from other parts of the system via the event emitter and route the user to the check-out experience. By following the principles of micro-frontends, we can reason in new ways, enhancing our developer experience and avoiding domain leaks in other application areas.

Hosting a Client-Side Rendering Micro-Frontend Project

When hosting client-side rendering micro-frontends, I've found that teams typically follow three main approaches:

- Using a single storage with a CDN
- Employing multiple storages with a unified CDN
- Hosting micro-frontends within secure containers for highly regulated industries

When I mention storage, think about solutions like [AWS Simple Storage Service \(Amazon S3\)](#). Each approach has its own advantages and trade-offs, and the right choice often depends on the organization's structure and requirements.

The approach of using single storage with a CDN involves storing all micro-frontends in a centralized location, such as a cloud storage bucket, and serving them via a CDN. This setup is simple to implement and manage, as it centralizes deployment pipelines and ensures consistency across teams. For example, imagine an ecommerce platform where the home page, product pages, and check-out flow are separate micro-frontends. In this setup, all micro-frontends are stored in one bucket (e.g., AWS S3) and served globally through a CDN like Amazon CloudFront. This simplifies governance and makes it easy to reason about deployments while ensuring high performance through caching at edge locations. This approach works best for organizations with tightly integrated teams under unified governance.

The approach of multiple storages with a unified CDN gives each team its own storage for micro-frontends while using a single CDN to deliver the content. This setup offers greater autonomy to individual teams while still leveraging the performance benefits of a shared delivery mechanism. For instance, consider a travel booking platform where separate teams manage flights, hotels, and car rentals as independent micro-frontends. Each team can maintain its own storage bucket for artifacts, but rely on the same CDN for delivery. This allows teams to operate independently while maintaining consistency in user experience. However, this approach requires more coordination between teams to ensure compatibility and avoid duplication of effort in setting up and maintaining the infrastructure.

The approach of using secure container hosting is typically adopted by organizations in highly regulated industries like finance or health care, where accessing static files via the public internet is not permitted. Micro-frontends are hosted in containers (e.g., Docker) within secure networks accessible only via VPNs or similar mechanisms. For example, a financial institution might use this method to host internal dashboards for

account management or compliance reporting. While this ensures strict security controls, it introduces significant complexity in infrastructure management and wastes compute resources, because containers are not optimized for serving static files. I strongly discourage the use of containers for this purpose unless absolutely necessary due to regulatory requirements.

For most organizations, the choice between single storage with a CDN and multiple storages with a unified CDN boils down to their structure and workflows. My recommendation is to adopt the approach of single storage with a CDN whenever possible, because it is easier to set up, simpler to manage, and reduces operational overhead. It provides clear governance and ensures consistency across teams while delivering excellent performance.

While the approach of using multiple storages with a unified CDN offers more flexibility for decentralized teams, it requires careful coordination to avoid duplication and to maintain compatibility across artifacts. It's best suited for larger organizations or those with autonomous teams that need greater control over their micro-frontends.

To reiterate, I strongly discourage using containers to serve static files unless absolutely necessary (e.g., in highly regulated industries). Containers are designed for running applications rather than serving static assets, making them an inefficient and expensive choice for this purpose. They consume unnecessary compute resources and significantly increase infrastructure complexity without providing meaningful benefits compared to simpler solutions like CDNs or cloud storage.

By carefully evaluating these approaches based on your organizational structure and requirements, you can make an informed decision that balances simplicity, performance, and scalability while avoiding unnecessary complexity.

Caching

One of the key optimizations for hosting client-side rendering micro-frontends is configuring different time-to-live (TTL) values for the CDN cache based on the rate of change of individual micro-frontends. This strategy allows you to balance performance and freshness effectively, ensuring that frequently updated micro-frontends remain current while less volatile ones benefit from longer caching durations.

Micro-frontends with a low rate of change can have higher TTL values. For example, an ecommerce platform might have a micro-frontend displaying 1950s Formula 1 standings or archived product details. These components rarely change, so setting a high TTL

ensures they are cached for extended periods, reducing server load and improving response times for users.

On the other hand, micro-frontends with frequent updates—such as breaking news sections or live inventory displays—require lower TTL values to ensure users always see the latest information. For instance, the same ecommerce platform might have a micro-frontend showing real-time stock availability or promotional banners. These components need rapid updates, so shorter TTL values ensure the CDN fetches fresh content more often. In these cases, even a one-minute TTL can help reduce the traffic toward the origin and improve the response time to the clients, so don't forget to set up the cache correctly.

Summary

In this chapter, we have seen the micro-frontend decision framework in action. Every team identified the right approach to achieve the requirements presented by the product teams. They maintained a decoupled approach, knowing that this path would guarantee independence and faster response time to any business shift. During this journey, we have also seen the technical implementation of a micro-frontend architecture using Webpack and Module Federation.

This is one of the many approaches mentioned in [Chapter 3](#). Every framework and technology will have its own implementation challenges; walking you through the reasoning behind certain decisions is far more valuable than evaluating each implementation. With this approach, you will have a mental model that enables you to move from one framework to another easily, just by following what you learned during this journey.

Chapter 5. Server-Side Rendering Micro-Frontends

You've been there. You open a website, and it takes forever to load. Or worse—content flashes, jumps around, and leaves you wondering if it's even working. If you're building a business-to-consumer (B2C) platform, especially in ecommerce, you can't afford for that to happen. Users will bounce. SEO suffers. Trust evaporates. That's why server-side rendering (SSR) matters. It's not merely a technical choice—it directly impacts key web performance metrics like largest contentful paint (LCP), cumulative layout shift (CLS), and search engine optimization (SEO) crawlability, making it a business-critical architectural decision:

LCP

Measures how quickly the main content of a page loads, shaping the user's perception of speed.

CLS

Tracks unexpected layout shifts that can frustrate users by moving visible elements during page load.

SEO crawlability

Ensures that search engines can easily access and index your content, directly affecting your app's visibility and discoverability.

By improving these metrics, SSR helps your app feel fast, stable, and reliable from the very first user interaction—delivering measurable value that resonates with both users and leadership.

When combined with micro-frontends, SSR unlocks greater scalability of team structures and independent deploys, not just user traffic.

However, blending these two approaches is not without its challenges.

In this chapter, we'll explore when SSR is the right choice for your frontend, the unique hurdles it introduces in a micro-frontend landscape, and practical strategies for dividing, composing, and scaling your application. Along the way, we'll look at real-

world patterns, performance tips, and case studies that show how leading teams are building robust, modern web platforms with SSR and micro-frontends.

When to Use Server-Side Rendering

SSR architectures shine in several key scenarios. First, they're excellent for content-heavy websites where SEO is crucial, such as news sites, blogs, and ecommerce platforms. Search engines can more effectively crawl and index server-rendered content, improving visibility and rankings. Second, SSR provides better performance for users with poor network conditions or less powerful devices, as it delivers HTML content immediately without waiting for JavaScript to execute. Lastly, SSR is valuable for applications requiring fast initial page loads to reduce bounce rates and improve user retention.

Ecommerce platforms particularly benefit from SSR during high-traffic events like flash sales. The initial product information loads quickly, giving users immediate feedback while interactive elements hydrate progressively.

However, if you are building a business-to-business (B2B) application where the majority of the system is behind authentication, SSR should be the last architecture you consider, due to various challenges.

Scalability Challenges

Scaling SSR applications during traffic spikes presents unique challenges. Unlike static sites or client-side applications, SSR requires significant server resources to render HTML for each request. During events like Black Friday, an ecommerce platform might see a traffic increase 10–20 times above normal levels, which could overwhelm servers.

There are several techniques that we can employ to reduce the strain on our SSR systems as well as the APIs that these systems consume:

Caching

Caching plays a pivotal role in scaling SSR effectively. Multiple in-memory caches—strategically placed throughout the architecture—store rendered HTML, API responses, and business data, significantly reducing the need to regenerate content on repeated requests. An always-warm cache approach, where data is proactively stored rather

than fetched on every request, ensures minimal latency during peak loads. Careful analysis often reveals that not all data requires absolute freshness; many elements can leverage short-lived time-to-live (TTL) values measured in seconds. Ultimately, effective cache design, robust invalidation strategies, and appropriate consistency models are the true levers that enable SSR to scale seamlessly during traffic spikes. Without these, SSR alone cannot meet high-demand scenarios reliably, making caching architecture a business-critical consideration for performance and scalability.

Serverless or prepositioning compute

There are various ways to handle compute scalability, ranging from serverless solutions (like AWS Lambda) to pre-provisioning compute resources (such as keeping containers running in advance of predictable traffic spikes). The choice between these approaches—or a hybrid of both—enables organizations to optimize for cost, performance, and operational control based on their specific application needs and traffic patterns.

Precomputation

Recommendation engines and content stores prepare data ahead of time, reducing processing during peak loads. For instance, during a flash sale, product pages can be prerendered and cached at the content delivery network (CDN) level, while inventory updates are handled through smaller, targeted API calls.

Later in this chapter, we will cover a case study where these techniques were used on a well-known and large-scale website.

Dividing Micro-Frontends

Most successful SSR micro-frontend implementations use a coarse-grained horizontal split. This minimizes orchestration overhead and complexity in error handling across fragments, which becomes unmanageable in fine-grained splits. The header and footer (rarely changing components) are often shared across the application, while the main content area is divided by page or domain.

Modern SSR frameworks like Next.js or Astro naturally support this division through their page-based structure. The page boundary—where content changes frequently—forms a natural split point between teams. For example, one team might manage product pages while another handles check-out flows.

Some companies attempt more granular splitting, where individual components on a page come from different micro-frontends. While this provides maximum team autonomy, it introduces some significant challenges:

- Coordination complexity increases exponentially.
- Performance can suffer from multiple service calls.
- Debugging becomes more difficult across service boundaries.
- There is a higher possibility of dependency clashes.
- Cache invalidation strategies grow more complex.

Composition Approaches

Overall, the main decision you need to make is how to compose micro-frontends. This decision affects runtime performance, team autonomy, and operational complexity, and teams often underestimate possible orchestration challenges. There are several approaches for composing server-rendered micro-frontends, each with their own trade-offs and ideal use cases:

HTML fragments

This remains a classic approach, where backend services return HTML chunks that are assembled on the server before being sent to the client. This method is straightforward and works well for coarse-grained splits, but can become difficult to manage as the number of fragments and teams grows.

Framework-specific solutions

This approach is becoming increasingly popular. For example, Next.js supports a multi-zone approach, allowing different parts of your app to be handled by separate deployments, while Astro.js introduces the concept of server islands, letting you mix rendering strategies within a single page. Each framework brings its own conventions for

composition, often influenced by the underlying UI technology or runtime.

Web components with shadow roots

This is a robust option for building micro-frontends. By encapsulating each micro-frontend as a web component using a shadow DOM, you ensure strong isolation of styles and logic, which helps prevent CSS and JavaScript conflicts between teams. Thanks to broad support for declarative shadow DOM in all major modern browsers (from 2023 onward), this approach is now practical even with SSR, making it easier to integrate micro-frontends built with different frameworks and to support progressive enhancement strategies.

Module Federation

As we've seen in previous chapters, with Module Federation, teams can independently build and deploy micro-frontends, sharing code and dependencies at runtime. This enables true runtime composition, reducing duplication and making it easier to roll out updates across teams without redeploying the entire application.

Vertical split with independent infrastructure

This is a simpler but effective strategy, especially for organizations with clear domain boundaries. Here, each major section of the application—often mapped to first-level URLs—is handled by a separate, independently deployed web application.

It's important to note that edge computing isn't always suitable, particularly for single-region applications with strict compliance requirements. The overhead of distributing workloads to edge locations may not justify the benefits, especially when your data remains centralized in another region. Additionally, compute power at the edge is often less than in-region resources, which can mean dealing with limitations in Node.js features or other runtime constraints.

The Main Challenge

Composing server-side rendered micro-frontends is one of the most nuanced challenges in modern web architecture. Over the years, working with hundreds of teams, I've seen a wide variety of composition strategies—each with their own trade-offs in terms of performance, team autonomy, and operational complexity.

There are many options available for composing micro-frontends, ranging from simple server-side assembly of HTML fragments (transclusion) to more advanced techniques like runtime Module Federation; web components with shadow DOM; leveraging modern JavaScript framework capabilities like Next.js, Astro, or Qwik; or orchestrating vertical splits through independent applications using your infrastructure. The choice often depends on your organization's structure, the technologies you use, and the level of independence you want to grant your teams. In the following sections, we'll analyze some of the most popular and effective composition patterns I've encountered in real-world projects. By understanding their strengths and limitations, you'll be better equipped to select the right approach for your own architecture.

HTML Fragments

The first option we are going to explore is the HTML fragments approach. With this model, each team is responsible for a specific business capability—such as the product catalog, shopping cart, or user reviews—and exposes its UI as an HTML fragment via an HTTP endpoint. These fragments are not full standalone pages, but rather partial HTML snippets that represent a bounded context within the larger application.

To illustrate this, consider an ecommerce platform. The product details page might be composed of several micro-frontends: one for product information, another for customer reviews, and a third for personalized recommendations. Each of these micro-frontends is developed by a separate team, possibly using different technologies or programming languages. For example, the “product details” micro-frontend could be built in JavaScript, the reviews in Python, and the recommendations in .NET. Each team is responsible for delivering its fragment as HTML, ensuring it adheres to a contract—such as a specific placeholder or API—that the overall application expects.

Ideally, the frontend part should use the same library so that dependencies are loaded only once for the entire application, preserving user experience and performance. Libraries like htmx provide a solid foundation for this multi-language approach.

The assembly of these fragments into a cohesive page is handled by a UI composer—a server-side orchestrator that leverages transclusion. See, for example, [Figure 5-1](#).

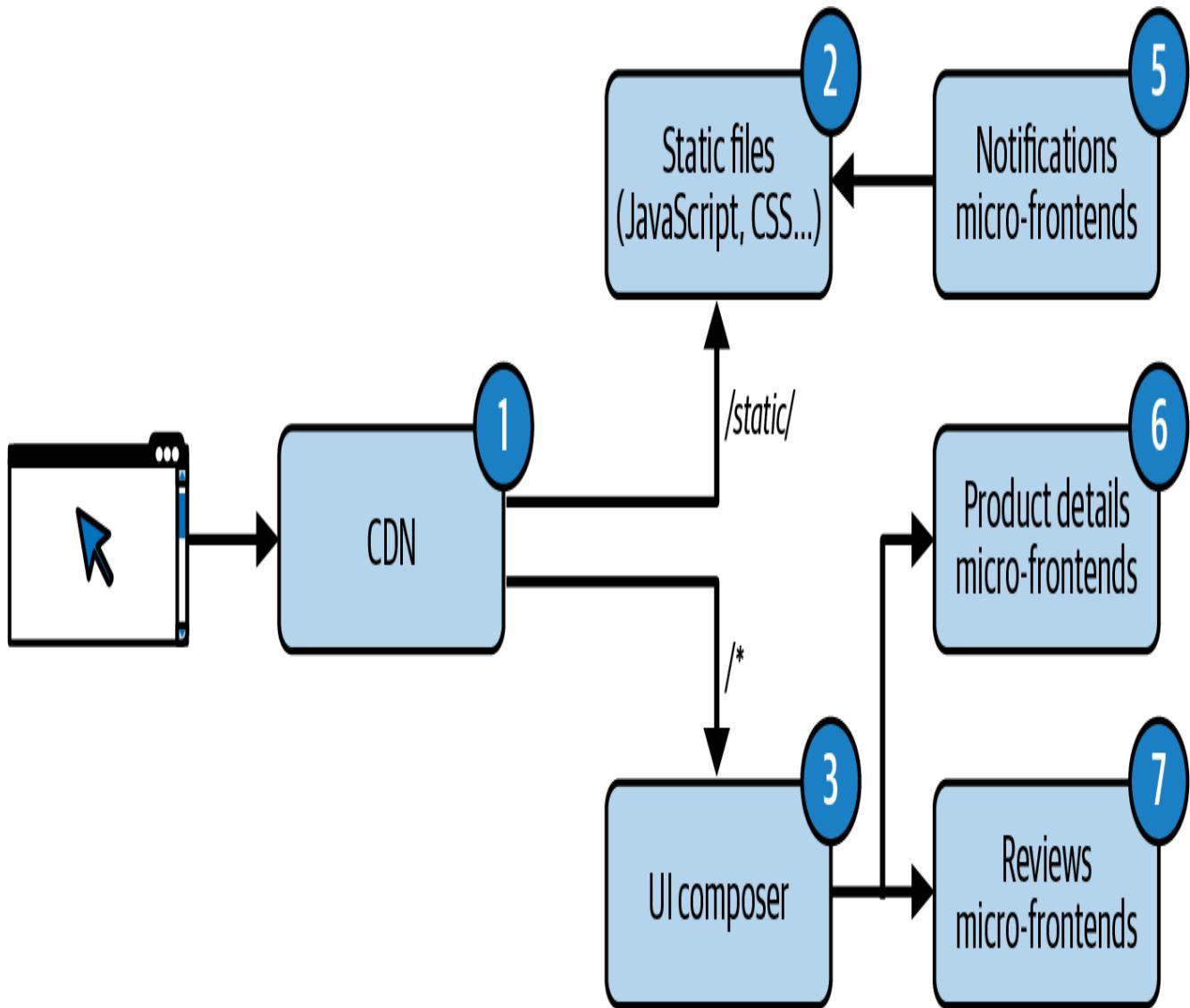


Figure 5-1. An example of HTML fragments implementation

The UI composer (3) loads a base HTML template (2) containing placeholders for each micro-frontend. At runtime, it fetches the relevant HTML fragments from the various upstream services (6 and 7) and injects them into the corresponding placeholders.

For instance, when a user navigates to a product page, the UI composer retrieves the product details fragment from the product team's service, the reviews fragment from the reviews team's service, and so on, and then stitches them together on the server side before streaming the complete HTML page to the browser.

This approach brings several benefits. It allows teams to work autonomously, as each team can update, test, and deploy its micro-frontend independently without impacting others. The UI composer remains business-logic unaware—it simply knows where to fetch each fragment and where to place it, but not how each fragment is built or what business rules it implements. This separation of concerns enables robust testing, as

teams can validate their fragments in isolation, confident that the composition mechanism will remain stable.

In the context of ecommerce, this model is particularly powerful. Imagine another Black Friday scenario: the promotions team can update the recommendations fragment to highlight flash sales, while the product team rolls out a new product details layout, all without needing to coordinate releases or risk breaking the overall page. If the reviews service experiences high load, the UI composer can gracefully degrade by showing a cached fragment or a simple placeholder, ensuring the rest of the page remains functional and performant.

This architecture is especially attractive for organizations with strong backend and full stack engineering teams, as it enables them to leverage their existing expertise in languages and frameworks beyond JavaScript, while still delivering a modern, scalable, and maintainable frontend experience.

Let's explore how the composition would look with this approach. When a user requests a page, the journey begins at the edge of the system, typically at a CDN. The CDN handles caching and security, routing the request to the appropriate backend if the content isn't already cached. If the requested page requires dynamic assembly, the CDN forwards the request to the system's entry point—a load balancer or gateway that directs traffic to the UI composer service.

The UI composer is the central orchestrator responsible for constructing the final HTML page:

```
fastify.get('/productdetails', async(request, reply) => {
  try{
    const catalogDetailspage = await transformTemplate(catalogTemplate)
    return reply
      .code(200)
      .header('content-type', 'text/html')
      .send(catalogDetailspage)
  } catch(err){
    request.log.error(createError('500', err))
    throw new Error(err)
  }
})
```

It starts by retrieving a static HTML template, which includes common dependencies in the `head` and placeholders or custom elements in the `body`, indicating where each micro-frontend fragment should be inserted. Here's an example:

```

<!DOCTYPE html>
<html>
<head>
  <title>AWS micro-frontends</title>
  <script src=".static/nanoevents.js" type="module"></script>
  <script src=".static/preact.min.js"></script>
  <script src=".static/htm.min.js"></script>
  <style>
    ...
  </style>
</head>
<body>
  ...
  <div id="noitificationscontainer">
    <script src=".static/notifications.js" defer></script>
  </div>

  <micro-frontend id="catalog" errorbehaviour="error"/>
  <micro-frontend id="review" errorbehaviour="hide" />
</body>
</html>

```

The `head` tag contains the common dependencies, and the `body` has the placeholders or custom elements indicating where each micro-frontend fragment should be inserted.

Another interesting aspect is that the micro-frontend placeholder can also specify an error-handling strategy. This signals to the UI composer how to manage error responses from a service. When you consider a view in a web application, it becomes clear that not all parts are always essential. For instance, in an ecommerce platform, the product detail page requires information about the product itself, but if (for any reason) the reviews are not available at that moment, the application can provide a degraded experience by displaying just the essential part—the product details. This flexibility allows a micro-frontend system to fail gracefully if a service becomes unavailable, returns an error, or if its response time is too high.

This decouples failure domains, enabling partial page loads instead of total failures, which is necessary for high-traffic ecommerce environments. For example, if reviews are unavailable, the application can provide a degraded experience by displaying just the essential product details.

To fill these placeholders, the UI composer uses a service discovery mechanism—a key-value store or registry that maps logical names to the actual locations of the corresponding services. This enables teams to deploy and update their micro-frontends independently, ensuring rapid iteration within their bounded contexts. Once endpoints are resolved, the UI composer makes parallel requests to each micro-frontend service.

The discovery pattern will be covered in great detail in the next chapter.

Each service generates an HTML fragment—often by querying its own data sources and applying its own rendering logic. These fragments are returned to the UI composer, which injects them into the appropriate placeholders within the template:

```
const transformTemplate = async (html) => {
  try{
    const root = parse(html);
    const mfeElements = getMfeElements(root);
    let mfeList = [];

    // generate VOs for MFEs available in a template

    if(mfeElements.length > 0) {
      mfeElements.forEach(element => {
        mfeList.push(
          getMfeVO(
            element.getAttribute("id"),
            element.getAttribute("errorbehaviour")
          )
        );
    } else {
      return ""
    }

    // Retrieves the micro-frontends endpoint from the discovery service

    mfeList = await getServices(mfeList);

    // retrieve HTML frgments

    const fragmentsResponses = await Promise.allSettled(mfeList.map(
      element => element.loader(element.service)))
    // analyze responses

    const mfeToRender = fragmentsResponses.map((element, index) =>
      analyseMFEresponse(element, mfeList[index].errorBehaviour))

    // transclusion in the template

    mfeElements.forEach((element, index) =>
      element.replaceWith(mfeToRender[index]))

    return root.toString();
  } catch(error) {
    console.error("page generation failed", error)
  }
}
```

}

HTML is easy to parse with JavaScript, and transclusion becomes a powerful mechanism for replacing placeholders with real HTML fragments that include HTML, styles, and JavaScript. The composition process is strictly mechanical: the UI composer does not interpret or alter the content of the fragments, preserving a clear separation of concerns and keeping the composition logic business-agnostic. In case of errors, the UI composer can apply logic to handle errors gracefully—either displaying a user error or hiding the affected micro-frontend:

```
const analyseMFEresponse = (response, behaviour) => {
  let html = "";

  if(response.status !== "fulfilled"){
    switch (behaviour) {
      case behaviour.ERROR:
        throw new Error()
      case behaviour.HIDE:
      default:
        html = ""
        break;
    }
  } else {
    html = response.value.body
  }

  return html;
}
```

After all the fragments are assembled, the UI composer streams the completed HTML page back through the CDN to the user's browser. Streaming enables the system to deliver critical content as soon as it's available, improving perceived performance and reducing time-to-interactive (TTI). If any fragment is delayed or fails, the UI composer can insert a fallback or cached version, ensuring resilience and a seamless user experience.

Not all micro-frontends must be rendered on the server side; some can be fully client-side rendered, offering flexibility in how each part of the application is delivered and interacts with the user. For example, the notifications micro-frontend is loaded on the client and listens for events emitted by other micro-frontends, providing real-time feedback such as messages when an item is added to the cart. This event-driven communication pattern enables loose coupling between micro-frontends.

By thoughtfully choosing the rendering mode for each micro-frontend, teams can optimize both performance and user experience—leveraging CDN caching for static sections and dynamic updates for interactive features. This flexibility is a key strength of the micro-frontend architecture, empowering organizations to tailor their approach based on the needs of each business domain and user interaction pattern.

Finally, it is generally recommended to maintain a clear separation between the APIs that serve raw data and the services that perform frontend rendering. Backend APIs can evolve independently, optimized for data delivery and business logic, while frontend rendering services focus solely on transforming that data into presentational HTML fragments. This approach was notably adopted by Zalando, the German fashion ecommerce platform, in their early implementation with Tailor.js. Today, the approach is increasingly popular among organizations with strong backend or full stack engineering teams working in languages such as JavaScript, Python, or .NET. By focusing on HTML fragments as the integration point, organizations can leverage their existing skills and infrastructure while reaping the benefits of modern, decoupled frontend development.

An example of this approach using AWS serverless services can be found in the [GitHub repository](#). It's important to highlight that this approach is not AWS-centric; it can be implemented with any compute solution, from on-premises infrastructure to other cloud providers.

Splitting by URL

This approach trades granular runtime composition for deployment and migration simplicity, which aligns well with most organizations' real-world constraints during modernization. This technique directs traffic to distinct, independently deployed services. Rather than composing multiple fragments on a single page, this model assigns entire top-level routes (such as `/news`, `/standings`, or `/video`) to specific applications—each maintained and deployed by a dedicated team, as shown in the high-level architecture diagram in [Figure 5-2](#).

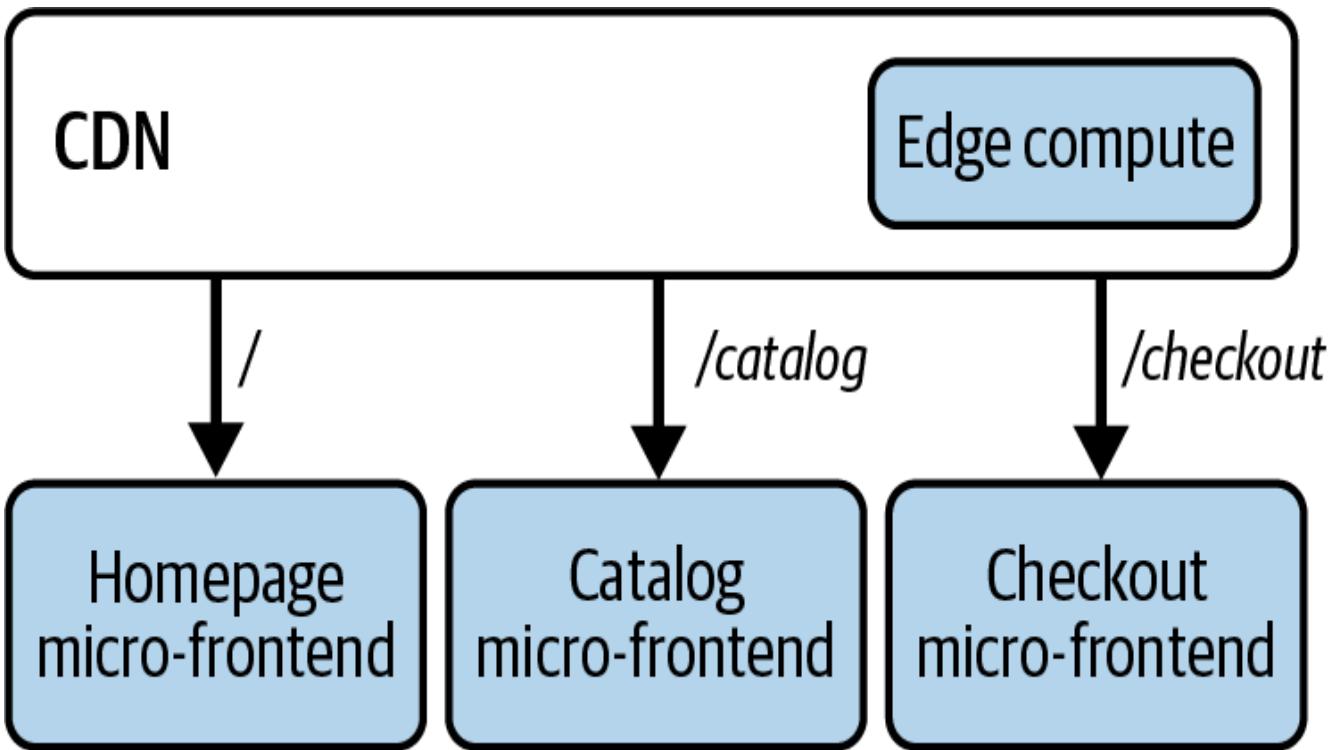


Figure 5-2. The CDN configuration helping associate different first-level URLs with different clusters that take care of SSR for a specific subdomain of a system

At the top of the diagram in [Figure 5-2](#), a CDN receives all incoming user requests. Based on the first segment of the URL (such as `/`, `/catalog`, or `/check-out`), the CDN routes each request directly to a dedicated micro-frontend application responsible for that part of the site. For example, requests to the root path are handled by the “home page” micro-frontend, `/catalog` by the catalog micro-frontend, and `/check-out` by the check-out micro-frontend. Alternatively, you can use edge compute to intercept requests and rewrite them to specific endpoints. This approach is especially useful during migrations, as it keeps temporary redirection logic out of your main codebase. By isolating these rules at the edge, you ensure the micro-frontend logic is well encapsulated and doesn’t require additional tests every time a micro-frontend changes.

This architecture offers remarkable simplicity and resilience. Each micro-frontend is a fully independent application—developed, deployed, and scaled by its own team. If one micro-frontend becomes unavailable, the rest of the system continues to function without interruption. This isolation of concerns ensures that failures are contained and do not cascade across the platform, which greatly improves overall system reliability.

Another key advantage is the flexibility in caching strategies. Because each micro-frontend is served independently, teams can tailor caching policies to the needs of their domain. However, be mindful that misaligned cache policies across micro-frontends can lead to inconsistent user experiences, requiring coordinated cache invalidation

patterns. For instance, the “contact us” micro-frontend—which may change infrequently—can be aggressively cached at the CDN to optimize performance and reduce backend load. In contrast, the home page or catalog micro-frontends—which handle dynamic data—might require fresher content and thus use more conservative caching or may even bypass caching entirely. This granularity allows the business to balance performance, cost, and data freshness for each area of the site.

Shared dependencies, such as utility libraries or design system components, are distributed through a private package repository. Ideally, these shared elements are implemented as web components, enabling teams to reuse and update UI elements without rewriting each application. This approach ensures visual and functional consistency across micro-frontends, while allowing each team to evolve independently and adopt design system updates incrementally.

From a compute perspective, the deployment model is flexible and straightforward. Each micro-frontend can be hosted using containers or serverless functions, depending on the organization’s infrastructure preferences and scalability requirements. This empowers teams to select the most suitable technology stack and deployment strategy for their needs.

Authentication is also streamlined in this model. By storing the authentication token in a cookie scoped to the base domain, every micro-frontend served under that domain has access to the token and can reuse it for secure API calls or session validation. This avoids the need for complex cross-application authentication mechanisms and ensures a seamless user experience as users navigate between different sections of the site.

The Formula 1 Website is Powered by Micro-Frontends

The split of the first-level URL was central to the transformation of [F1.com](#), where the challenge was to modernize a legacy monolithic web platform into a scalable, high-performing, and independently deployable system. The initial architecture, like many enterprise websites, was a classic three-tier monolith: a single application server generated all HTML, managed all business logic, and handled every user request. This setup, while straightforward, imposed significant constraints on scalability, release velocity, and team autonomy.

Any change—no matter how minor—required regression testing and redeployment of the entire application, leading to bottlenecks and increased risk.

The migration began by identifying clear domain boundaries within the site's structure. For F1.com, this meant recognizing that pages like the subscription journey, news listings, and statistics could be isolated and owned by different teams. Instead of rewriting the entire site in one go, the team adopted an incremental strangler fig pattern: new features or domains were extracted one by one into standalone applications. A key enabler was the use of first-level URL routing at the edge or gateway layer. Incoming requests were inspected at the top-level path—such as `/tv-proposition` for subscription pages—and routed directly to the appropriate independent service. Pages still served by the monolith continued to be handled as before, while new or migrated pages were seamlessly served by their dedicated micro-frontend applications.

This routing strategy offers remarkable simplicity and speed for migration. There's no need for a complex composition layer or runtime orchestration: each micro-frontend is a self-contained application, responsible for its own assets, dependencies, and rendering. Shared elements—such as the header, footer, or design system components—can be distributed as libraries and reused across applications, ensuring visual consistency without tightly coupling deployments.

Over time, the Formula 1 web platform evolved from a legacy Adobe Experience Manager (AEM) monolith into a modern micro-frontend architecture. These micro-frontends leveraged a combination of shared components—used consistently across the platform—and unique “snowflake” components for specialized features.

According to their implementation results, F1 experienced a significant business impact:

- It saw a 34% increase in subscriptions and sign-ups to F1 Unlocked and F1 TV (measured from January to September 2024).
- It experienced a 26% reduction in platform costs by shifting workloads from Amazon Elastic Compute Cloud (EC2) to container-based solutions.

Similarly, F1 saw various performance improvements:

- Significant web vitals improvements on the “Latest F1 News” page:
 - Its first contentful paint (FCP) reduced from 2.0 to 0.8 seconds (desktop).
 - Its Speed Index improved from 4.3 to 3.0 seconds on desktop (and as low as 0.6 when excluding nonessential code).

- LCP on mobile dropped from 8.7s to 3.2 seconds.
- Its mobile Speed Index improved from 7.3 to 5.2 seconds (and 2.2 seconds without nonessential code).
- Its overall Lighthouse performance score increased from 53 to 83, a 56% improvement on mobile and 30% on desktop.

Finally, the developer experience improved drastically:

- Engineers focused on individual micro-frontends, reducing cognitive load.
- Independent development, testing, and deployment minimized platform-wide regression testing.
- Tight integration with design systems and automated accessibility testing accelerated delivery quality.

This data is a fantastic testimony to how well this architecture could suit organizations, enhance their core metrics, and improve business outputs. However, there is always more room for improvement. The most important thing is to have an architecture that works for you, rather than the other way around. Formula 1 is on a promising trajectory to further improve its future web applications. If you are interested in learning more about this case study, you can engage with the [AWS re:Invent 2024 presentation](#) offered by Ivan Rigoni, the digital solutions architect in the F1 digital technology team.

Next.js Multi-Zones

Next.js multi-zones are another approach for building micro-frontends, especially for large-scale applications that need to be split into independently deployable sections. Sounds like micro-frontends, doesn't it?

In this model, each “zone” is a standalone Next.js application responsible for a specific set of routes or features-like `/blog/*`, `/dashboard/*`, or the main site, while all zones are seamlessly merged under a single domain for the end user.

Navigation in a multi-zone setup is designed to feel smooth and unified for users. When moving between pages within the same zone, navigation is handled by Next.js, resulting in fast and soft navigations that don't require a full page reload. However, navigating from one zone to another (for example, from `/` to `/dashboard`) triggers a hard

navigation, unloading resources from the current zone and loading those for the new zone.

While hard navigations are sometimes unavoidable in a multi-zone or micro-frontend architecture, there are several best practices you can adopt to minimize their impact and preserve a seamless user experience. Vercel recommends leveraging modern browser features like prefetching and prerendering to reduce perceived latency when crossing zones. By prefetching resources for likely destinations in advance—or even prerendering the next page in a hidden tab—you ensure that most of the content is already loaded and ready to display when the user initiates a hard navigation. The Speculation Rules API, for example, allows you to declaratively specify which links should be prefetched or prerendered, making transitions between zones noticeably faster.

It's also important to use plain `<a>` tags for links that cross zone boundaries, rather than the Next.js `<Link>` component. This prevents Next.js from attempting to soft-navigate to a route that belongs to a different application, and ensures the browser handles the transition as a full page load. For zones that are frequently visited together, consider grouping them within the same application to reduce the number of hard navigations required.

Finally, thoughtful routing configuration with the use of rewrites or middleware can help optimize request handling and further streamline the user journey. By combining these techniques—prefetching, prerendering, correct link usage, and intelligent routing—you can greatly reduce the friction of hard navigations and deliver a user experience that feels fast and cohesive, even in complex multi-zone environments.

The real power of multi-zones comes from how rewrites are configured. Next.js enables you to set up rewrites in your `next.config.js` so that requests to certain paths are transparently proxied to the appropriate zone, even if that zone is running on a different server or port. This acts like a reverse proxy, avoiding cross-origin resource sharing (CORS) issues and allowing all the zones to appear as a single, cohesive application to the user. For example, you might rewrite `/dashboard/:path*` to `https://dashboard.example.com/:path*`, letting the dashboard team deploy and manage their app separately from the rest of the site.

Bear in mind that with this configuration, you can always create your own reverse proxy and configure it similarly to how you would in a multi-zone setup. So, if you decide to implement a multi-zone approach, switching from the built-in proxy to a custom reverse proxy requires minimal changes.

Middleware in Next.js adds another layer of flexibility. Middleware lets you run code before a request is completed, allowing you to rewrite, redirect, or even inject custom logic based on the incoming request. In a multi-zone architecture, middleware can be used to handle authentication, apply feature flags, or perform A/B testing, ensuring that requests are routed to the right zone or handled according to your business rules, all before the main application code runs.

Let's explore a real example. I created a [t-shirt shop project](#) to offer a clear, practical example of how to implement a micro-frontend architecture using Next.js 15's multi-zone capabilities. In this setup, the application is divided into three distinct zones—home, catalog, and account—each represented by their own independent Next.js application.

The home zone acts as the landing page, welcoming users and showcasing featured products. The catalog zone manages product listings and detailed product views, handling the core shopping experience. The account zone is responsible for user authentication and profile management, ensuring that sensitive operations are kept logically and technically separate from the rest of the application.

Each zone maintains its own routing and configuration, which means teams can make decisions inside their own area of responsibility independently without impacting the rest of the platform.

Shared components are managed through a shared library, promoting consistency and reducing duplication. Styling across all zones is unified with Tailwind CSS—ensuring a cohesive look and feel, no matter which part of the site a user visits.

Home Zone: The Entry Point of Your Web Application

The `next.config.js` file for the home zone in the t-shirt shop project is a great example of how Next.js multi-zones orchestrate independent applications into a unified experience. The most critical aspect here is the `rewrites` configuration, which acts as the traffic director for requests that don't belong to the home zone itself.

When a user visits the site, requests for the root path or any route handled by the home zone are processed locally. However, if a user navigates to a path that begins with `/catalog/` or `/account/`, the home zone's Next.js server uses the rewrites defined in `next.config.js` to transparently forward these requests to the appropriate remote zone. This is achieved by mapping the incoming path to the destination URL and base

path of the catalog or account zones, as specified by environment variables. The result is a seamless experience for the user (who remains in the same domain) while, under the hood, the request is routed to a completely separate Next.js application.

This approach ensures that each zone can be developed, deployed, and scaled independently, while the home zone acts as the entry point and traffic coordinator. The rewrites are handled at the server level, so the browser is unaware that the handoff navigation to `/catalog/shirts` or `/account/profile` appears as a natural part of the site, even though those pages are rendered by different applications.

Although this mechanism might initially seem problematic, the rewrite code actually offers enough flexibility to define the first-level URL once, requiring only minimal changes throughout the project's life cycle:

```
async rewrites() {
  return {
    fallback: [
      {
        source: '/catalog/:path*',
        destination: `${process.env.NEXT_PUBLIC_CATALOG_URL}
${process.env.NEXT_PUBLIC_CATALOG_BASE_PATH}/:path*`,
      },
      {
        source: '/account/:path*',
        destination: `${process.env.NEXT_PUBLIC_ACCOUNT_URL}
${process.env.NEXT_PUBLIC_ACCOUNT_BASE_PATH}/:path*`,
      },
    ],
  };
},
```

To visualize how this works, consider the sequence diagram in [Figure 5-3](#), written in Mermaid.js syntax, which illustrates what happens when a user requests the `/catalog/shirts` page.

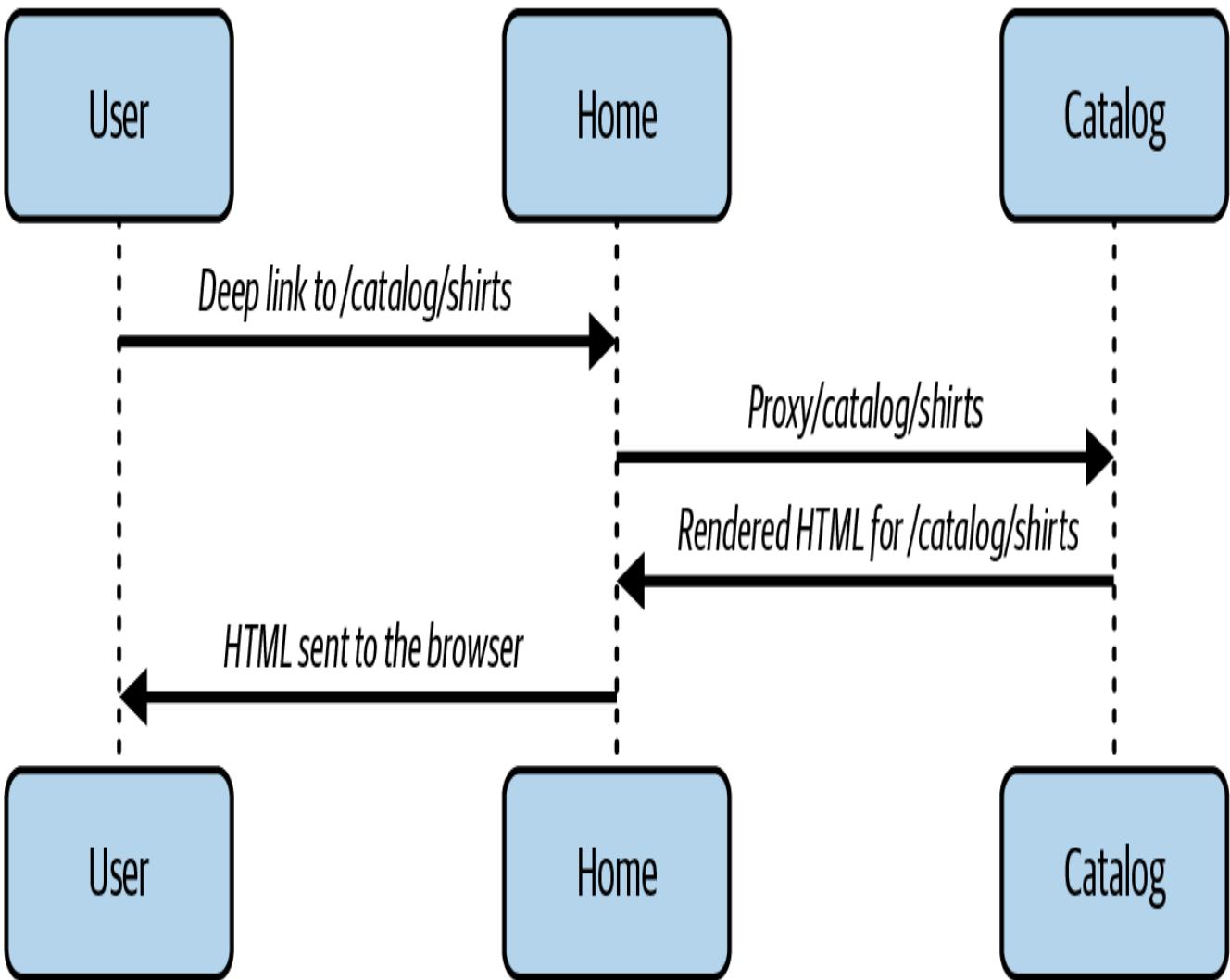


Figure 5-3. Sequence diagram highlighting how the home zone acts as a proxy for the other zones

In this flow, the user's browser sends a request to the home zone's Next.js server. The server, recognizing that the path matches the `/catalog/:path*` pattern, proxies the request to the catalog zone as defined in the rewrites. The catalog zone renders the requested page and returns the HTML back to the home zone, which then delivers it to the user. This pattern is repeated for any route that belongs to a different zone, enabling the application to scale horizontally and maintain clear separation of concerns.

How to Handle Shared Components

In a multi-zone micro-frontend architecture like the one used in the t-shirt shop project, the way you share components across zones is a crucial design decision. In this setup, shared components—such as headers or footers that span across all the different zones—are included at build time rather than at runtime. This approach means that each zone—whether it's home, catalog, or account—imports the shared components directly from

a common library as part of its build process. As a result, the components are bundled into the final output of each independent Next.js application.

Choosing build-time sharing brings several advantages. Most importantly, it avoids duplication of logic and styling at runtime, which can otherwise lead to inconsistent user experiences and unnecessarily large JavaScript bundles. By ensuring that all zones use the same version of shared components at build time, you maintain a consistent look and feel throughout the application. This also simplifies debugging and maintenance, as any updates to a shared component are picked up the next time each zone is built and deployed.

However, this approach does require some discipline around package management and versioning. Automation tools like Dependabot are particularly useful, because they can simplify the review of zones that use shared components and can automatically update dependencies by opening a PR for team review. This is a manageable trade-off for the benefits of speed and simplicity at runtime.

In this example, we updated the `layout.jsx` leveraging the shared components available in this monorepo approach:

```
import "@t-shirt-shop/shared/src/styles/globals.css";
import { Header, Footer } from "@t-shirt-shop/shared";

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <div className="min-h-screen flex flex-col">
          <Header />
          <main className="flex-grow">{children}</main>
          <Footer />
        </div>
      </body>
    </html>
  );
}
```

Everything we've said about code reusability still holds true here. If you're unsure, just start by duplicating the code. Later, evaluate whether it really makes sense to create (and maintain!) an abstraction like a shared component. Try not to share too much too soon—it can add unnecessary complexity and make life harder for other teams.

Alternatively, some organizations experiment with runtime sharing of components using technologies like Module Federation or the injection of HTML fragments. This allows zones to load shared components dynamically at runtime, which can make it easier to roll out updates without rebuilding every zone. While this offers flexibility, it also introduces additional complexity in dependency management and can lead to version mismatches or unexpected behaviors if different zones load incompatible versions of a component.

For most teams, especially those prioritizing performance and reliability, build-time sharing is the preferred approach. It keeps runtime environments lean and predictable, and ensures that styling and behavior remain consistent across all parts of the application, regardless of which zone a user is navigating.

How to Manage Data

Managing ephemeral data and cross-zone communication is a subtle but vital aspect of building a robust multi-zone micro-frontend architecture. In this model, each zone is an independent application, so sharing information between them requires careful consideration of both user experience and security.

For lightweight, transient information—such as search terms, product IDs, or promo codes—query strings are an effective and straightforward solution.

For example, if a user enters a promo code on the home page and then navigates to a specific product in the catalog zone, the promo code can be appended to the URL as a query parameter. When the catalog zone receives the request, it can extract the promo code from the query string and apply any relevant discounts or messaging. This approach is simple and transparent, and it works well for data that doesn't need to persist beyond a single navigation or session. You could also consider using session storage, but be sure to keep its security model in mind. Session and local storage are tied to their subdomains, so the session storage on `www.mysite.com` can't be seen by `catalog.mysite.com`, and vice versa.

However, not all data is suitable for this kind of ephemeral, client-side transfer. For persistent or sensitive information—such as the number of items in a user's cart, authentication tokens, or user preferences—it's best to rely on backend storage or APIs to synchronize state across zones. For instance, the cart icon displayed in the header across all zones should reflect the current state of the user's shopping cart, regardless of which zone the user is in. This ensures consistency, security, and a seamless experience as users move between different parts of the application.

You might be wondering: what about a user session token? In a multi-zone setup—where each part of the application (such as the home, account, and catalog zones) is independently deployed and serves different functionality—authentication must be handled efficiently to maintain a seamless user experience.

Authentication is often managed using tokens, typically stored in cookies, which can be securely accessed by each independent zone to validate the user's session. This method enables seamless authentication as users navigate between zones while ensuring secure access to resources across the entire application.

At the core of a secure authentication system is the use of tokens, usually in the form of JSON Web Tokens (JWTs), which are issued by an authentication service when a user logs in. These tokens are set as cookies in the browser to maintain the user's session across multiple requests and pages.

To ensure the security of these cookies, it's important to set several key attributes:

`HttpOnly`

This prevents the cookie from being accessed via JavaScript, safeguarding it from cross-site scripting (XSS) attacks.

`Secure`

Ensures the cookie is transmitted only over HTTPS, protecting it from interception attacks.

`SameSite`

By setting this attribute to `Lax` or `Strict`, you can prevent cross-site request forgery (CSRF) attacks by restricting when cookies are sent with cross-site requests.

With these security measures in place, whenever a user navigates between zones—whether it's moving from the home page to the account page, or accessing personalized content in the catalog zone—the authentication token is included automatically in the HTTP headers. This allows the server of each zone to validate the token, perform SSR checks, and ensure the user is authorized to access protected resources, such as their profile information or past order history.

When a user requests a page in a multi-zone architecture, middleware can intercept the request before it reaches the page's rendering logic. This is an effective way to ensure

the following:

- The user's session is valid and the authentication token is present.
- The token is verified using the secret key, ensuring it hasn't been tampered with.

By centralizing this logic in middleware, you eliminate the need to duplicate authentication and authorization code across multiple zones. This results in a more maintainable and secure system. It also ensures that sensitive resources—like user profiles or admin panels—are only accessible by authorized users, while other users are redirected to login pages or access-denied pages as needed.

To ensure persistent user sessions, it's good practice to use refresh tokens alongside short-lived access tokens. Access tokens typically have an expiration time (e.g., 15 minutes to an hour), while refresh tokens are long-lived and can be used to obtain new access tokens when the current one expires. This mechanism helps to maintain user sessions without forcing users to log in frequently.

The refresh token can be securely stored in an `HttpOnly` cookie, and the backend API can issue a new access token when the old one expires, based on the valid refresh token. This strategy provides a balance of security and usability, preventing users from being logged out frequently while still ensuring that stolen tokens are only useful for a limited time.

Beyond using cookies, another important consideration is where and how you store tokens. `HttpOnly` cookies remain the most secure choice for session management, as they cannot be accessed by JavaScript running in the browser, protecting them from XSS attacks. It's important to never store sensitive tokens in `localStorage` or `sessionStorage`, as these are vulnerable to JavaScript access and can be easily exploited if an attacker gains control over the client-side application.

Tokens should always be transmitted over HTTPS to prevent interception by attackers on insecure networks. The use of the `Secure` cookie attribute ensures that tokens are only sent over secure connections.

Whether you are using Next.js, Astro.js, or any other SSR solution, following these principles will help you build a robust and secure authentication and authorization system in your micro-frontend architecture.

Vercel: A Glance into the Future

After leveraging multi-zone architecture for vertically split micro-frontends, it's clear that Vercel is pushing innovation even further. Chief technology officer Malte Ubl has confirmed that micro-frontends will become first-class primitives on the platform, and recent discussions with Vercel's engineering team reveal concrete steps toward this vision. They're developing tools to streamline federated rendering, environment management, and remote component integration—key pain points in micro-frontend architectures.

A standout innovation is Vercel's approach to environment-agnostic Module Federation, where relative paths for remote entries allow for seamless switching between production, staging, and local setups. This is managed via a centralized `microfrontends.json` configuration file, paired with a developer toolbar for dynamic environment simulation.

The beauty of this approach is that any developer can swap micro-frontends directly from the UI in any environment, including production, without having to deploy them separately in each environment. In my opinion, this will significantly simplify the development and testing of micro-frontends before running the usual automation for deploying a new version.

For SSR-specific challenges, Vercel uses rewrite rules and cookie-based routing overrides to maintain security without compromising user experience.

React Server Components (RSCs) are poised to play a pivotal role. Vercel's engineers are prototyping remote components, a powerful abstraction that enables code transclusion into Next.js applications. This mechanism allows hosts to embed server-rendered content from external services with minimal client-side JavaScript. While development is currently in early stages, the long-term vision is to open source remote components and make them framework-agnostic—supporting not only React, but also other frontend frameworks capable of leveraging RSCs, Incremental Static Generation (ISG), or even plain HTML. This positions Vercel as a central player in enabling native integration of distributed UIs, regardless of framework choice.

As this book is being written, it's exciting to note that the discussions between Vercel and the Module Federation teams have resumed. And Vercel platform's strategy is not limited to Module Federation; support is expanding to include single-spa and any other existing frameworks, including custom micro-frontend frameworks. This underscores Vercel's commitment to flexibility and composability across diverse technical stacks.

In the second half of 2025, Vercel is preparing a public beta rollout, with plans for conference demos and ecosystem partnerships. While full details remain under wraps,

their focus on developer experience (local proxying, hybrid deployments) and observability (cache health monitoring, dependency tracking) signals a holistic approach to micro-frontends—one that could redefine how teams build and scale composable web applications in the coming years.

API Strategies

We cannot think about server-side rendered micro-frontends without considering API strategies—where the data comes from and how teams are structured to provide and consume that data. In SSR architectures, the way you design, aggregate, and cache data is just as important as how you compose your UI.

GraphQL has become just as popular as REST in modern micro-frontend architectures. While most companies still use REST APIs—often for historical reasons—GraphQL is increasingly appreciated, especially by full stack and frontend developers. Its flexibility allows teams to fetch exactly the data they need, which is particularly useful for supporting different screen sizes and device types, such as mobile apps and desktop web applications.

One of the main benefits of GraphQL in a micro-frontend setup is the ability to maintain a single unified graph for the entire application, rather than splitting GraphQL into multiple endpoints for each domain. This unified approach simplifies the data layer and allows backend services to evolve independently, without forcing frontend teams to constantly update their data-fetching logic.

For teams that prefer REST, the backend for frontend (BFF) pattern remains a solid choice. With BFF, each micro-frontend can have its own tailored backend service that aggregates data from various domains, maintaining a clear separation between the micro-frontend user interfaces and the underlying microservices.

However, perfectly dividing a system into subdomains and assigning each to a separate team is rarely straightforward.

For example, consider an ecommerce website: the product listing, shopping cart, and user profile might seem like distinct areas, but features such as personalized recommendations or promotional banners often require data from multiple domains. This overlap means teams must collaborate and share clear API contracts, which can introduce complexity and coordination challenges.

It's important to recognize that these boundaries are rarely perfect, and some cross-domain dependencies are inevitable.

Throughout this book, we'll explore API strategies in greater depth, helping you navigate the practical realities of building scalable and maintainable micro-frontend systems.

Don't Fear the Cache: Scaling SSR with Smart Caching Strategies

Caching doesn't need to be intimidating. Even implementing a small five-second TTL can dramatically reduce the strain on your backend systems during traffic spikes. When a popular sports event or breaking news story drives thousands of visitors to your site simultaneously, these brief caching windows prevent your databases and APIs from being overwhelmed.

Types of Caches Every Developer Should Know

Before diving into real-world implementations, let's understand the different types of caches you should consider when building server-side rendered micro-frontends.

Content Delivery Networks

CDNs store content at edge locations around the world, serving it directly to users without hitting your origin servers. CDNs excel at caching with:

- Static assets (images, CSS, JavaScript)
- Full HTML pages that don't require personalization
- API responses that can be shared across users

A CDN is your first line of defense against traffic spikes, often handling 80–90% of requests for popular content. I have had customers who were able to reduce their transactions per second (TPS) from tens of thousands to mere dozens by leveraging CDNs properly.

In-Memory Database Cache

In-memory solutions like Redis provide microsecond response times, making them perfect for SSR scenarios. The cache-aside pattern is most commonly used in micro-frontend applications.

This caching pattern is pretty straightforward. Your application checks the cache first before querying backends, updating the cache with fresh data when needed. This pattern works exceptionally well for read-heavy workloads, such as serving web pages.

For example, imagine a news website that needs to render a “top stories” component on multiple pages. Instead of making repeated API calls to the content service for each user request, you can cache the HTML fragment:

```
async function getTopStoriesComponent(req, res) {
  // First check if we have it in Redis
  const cachedHTML = await redisClient.get('top-stories-fragment');

  if (cachedHTML) {
    // We found it in cache, use it directly
    return cachedHTML;
  }

  // Not in cache, call the content service
  const stories = await contentService.getTopStories();
  const htmlFragment = renderToString(<TopStoriesComponent stories={stories} />);

  // Store in Redis with a 5-minute TTL
  await redisClient.set('top-stories-fragment', htmlFragment, 'EX', 300);

  return htmlFragment;
}
```

This approach can reduce page rendering time from hundreds of milliseconds to just a few milliseconds, significantly improving both user experience and server throughput during traffic spikes. In some projects that I have worked on, I was able to achieve query times of around 5 milliseconds for gathering 1 MB of HTML—an absolute lifesaver!

Warm Cache

Instead of waiting for the first user request to populate a cache (potentially creating a poor experience for that user), warm caches proactively load data before it’s needed. They do so in the following ways:

- Periodic jobs populating caches with frequently accessed data
- Background processes refreshing cache entries before expiration
- Precomputation of expensive operations during low-traffic periods

The BBC Architecture: Caching in Action

During QCon London 2025, the BBC shared its implementations using multiple caching layers. In fact, the BBC's architecture demonstrates how these caching strategies work together in a real-world, high-scale system. [Figure 5-4](#) follows a request through the BBC's architecture to showcase caching in action.

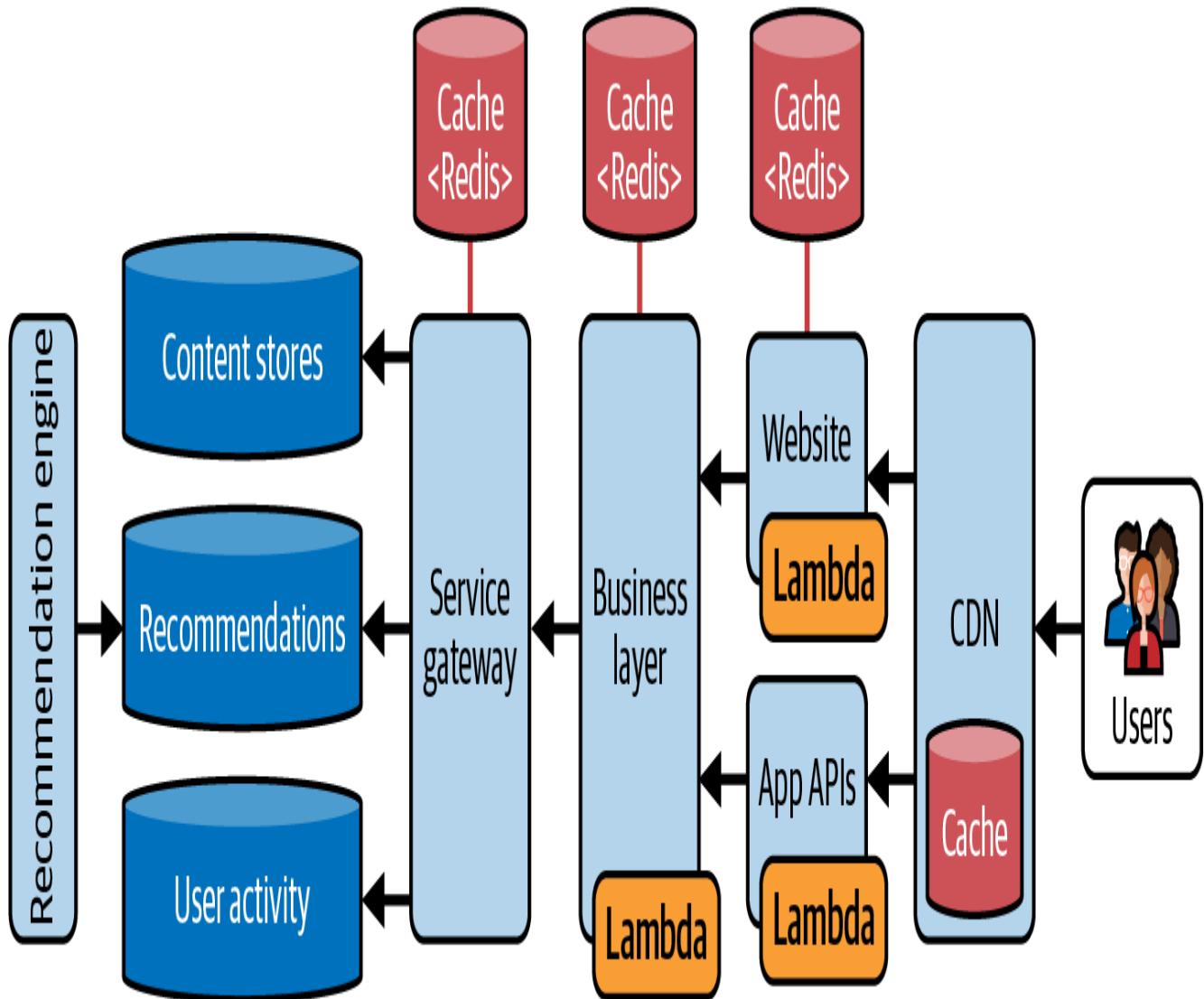


Figure 5-4. The BBC website's high-level architecture

When someone wants to check the latest football scores or breaking news on the BBC, their request first hits the CDN. For popular content like major news stories or sports events, the CDN likely already has the response cached, allowing it to serve the content immediately without touching any of the BBC's internal systems. This cache absorbs the majority of repeated requests, ensuring that only truly unique or uncached requests travel deeper into the system.

Moreover, this approach enables you to configure different cache TTLs based on content volatility. “Breaking news” pages might cache for just 30 seconds, while archived content could be cached for days. Even those brief 30-second windows dramatically reduce origin load during traffic surges.

If the CDN doesn’t have the content cached (a “cache miss”), the request reaches either the website or app APIs layer. Here, lambda functions (shown by the orange λ icons) handle SSR.

Notice how each of these components connects to its own Redis cache. These caches store rendered HTML fragments and API responses, creating a buffer between the rendering layer and deeper backend services.

For instance, for highly dynamic but frequently accessed components like sports scores or weather updates, the BBC could set short TTLs of 5–15 seconds. This brief window helps the system handle massive concurrent requests while still keeping content relatively fresh.

The business layer, which aggregates and applies business logic to data from multiple sources, also leverages serverless functions for scalability and flexibility. At this stage, a dedicated Redis cache helps ensure that frequently requested business data is served quickly, reducing the need for repeated computation and further protecting backend services from spikes in demand.

Thanks to the caching approach on this layer, you can implement circuit breakers to make the system more resilient. If backend services become slow or unresponsive, the system can serve slightly “stale” cached data instead of failing completely, prioritizing availability over perfect freshness.

The service gateway connects the business logic to the underlying data sources. This crucial layer has its own Redis cache to minimize calls to backend systems.

As an integrated best practice, the BBC implements staggered cache expiration times at this layer. Instead of all cache entries expiring simultaneously (potentially causing a “thundering herd” of requests to backends), each entry gets a slightly randomized TTL, spreading the load more evenly.

When a major news event breaks or a significant sports match concludes, the BBC might see traffic increase by tenfold within minutes. Without this layered caching strategy, their servers would likely collapse under the load.

Instead, the CDN handles most requests, while the various Redis caches ensure that even dynamic, personalized content can be delivered quickly without overwhelming

backend systems.

Even implementing a brief 5–10 second cache for data that changes infrequently can make the difference between a smooth user experience and system failure during peak traffic. For instance, when England scores in a World Cup match, millions of users might check BBC Sport simultaneously—these brief caching windows ensure the underlying content systems don't receive millions of identical requests.

Performance: The Key Reason for Server-Side Rendering

Performance is at the heart of why organizations choose SSR, and it's especially critical when you're building with micro-frontends. The journey to a fast, interactive web experience doesn't end with delivering HTML quickly; it's about what happens next—how and when your JavaScript loads, how much of it runs, and how users actually experience your site across devices and network conditions.

One of the most effective strategies is to use partial hydration or resumability. Instead of hydrating the entire page, you focus on making only the interactive parts of your UI come alive for the client. For example, a news home page might render the article content and navigation instantly, but only hydrate the comments section or live score widgets when a user scrolls them into view. Frameworks like Preact, Astro, and Qwik are leading the way, allowing you to deliver static content immediately while deferring or even skipping hydration for parts of the page that don't need it. This approach dramatically reduces the amount of JavaScript that needs to run on initial load, which is especially important for users on slower devices or networks.

To ensure these optimizations are working, you need to measure performance continuously and objectively. Lighthouse is an invaluable tool for this. By running Lighthouse audits on your micro-frontends throughout development and as part of your CI/CD pipeline, you can track key metrics like LCP, FCP, and TTI. Setting a performance budget for each micro-frontend—such as a maximum JavaScript bundle size or a target LCP—helps teams catch regressions early. If a new feature pushes your bundle over budget or slows down the first paint, you'll know right away, before it reaches production.

A technique I've found both easy to implement and extremely valuable for improving the developer feedback loop is to generate the final micro-frontend artifact using hooks provided by your version control system. When a developer opens a pull request, the

hook triggers a service that clones the project, builds it, and checks if the final artifact size exceeds a specified threshold. If it does, you can choose to block the merge or notify the team in another way. This approach helps teams catch potential performance issues early and maintain high standards for bundle size and efficiency.

But synthetic tests alone aren't enough; real-world observability in production is essential. You need to see how your application performs for actual users, across a variety of devices, browsers, and network conditions. Real user monitoring (RUM) tools can capture this data, showing you not just averages but outliers—those users on older phones or using spotty connections who may be having a very different experience. With this insight, you can prioritize optimizations that matter most, like improving hydration speed on mobile or reducing JavaScript for users in regions with slower networks.

Performance work is never “done”; it’s a cycle—optimize, measure, observe, and repeat.

By combining SSR, smart hydration strategies, continuous Lighthouse checks, and robust observability, you can ensure your micro-frontends deliver not just fast initial loads but consistently great experiences for all your users.

Summary

As you move forward with SSR and micro-frontends, keep your focus on what matters most: delivering a fast, seamless experience for your users while empowering your teams to work efficiently and independently.

Don’t feel pressured to chase every new framework or architectural pattern—instead, pick the tools and strategies that best fit your team’s strengths and your project’s needs. Whether you’re experimenting with Next.js multi-zones, Astro islands, or classic HTML fragment compositions, remember that clear ownership, thoughtful boundaries, and regular performance checks will take you much further than any one piece of technology.

Start small, measure often, and don’t be afraid to iterate. Use real user data to guide your optimizations, and encourage your teams to share what works—and what doesn’t.

Most importantly, keep the lines of communication open between teams so your micro-frontends grow together, not apart. Building at scale is always a journey, but with the right mindset and a willingness to learn, you’ll be well equipped to deliver robust, modern web applications that delight your users and stand the test of time.

Chapter 6. Micro-Frontend Automation

In this chapter, we discuss another key topic for distributed systems like micro-frontends: the importance of a solid automation strategy. The microservices architecture adds great flexibility and scalability to our architecture, allowing our APIs to scale horizontally based on the traffic our infrastructure receives. This enables us to implement the right pattern for the right job rather than applying a common solution to all our APIs, as in a monolithic architecture. Despite these great capabilities, microservices increase the complexity of managing the infrastructure, requiring an immense number of repetitive actions to build and deploy them. Any company embracing the microservices architecture, therefore, must invest a considerable amount of time and effort in its continuous integration (CI) or continuous deployment (CD) pipelines—which we explore further in this chapter. Given how quickly a business can change direction nowadays, improving a CI/CD pipeline is not only a concern at the start of a project; it’s a constant incremental improvement throughout the entire project life cycle. One of the key characteristics of a solid automation strategy is that it creates confidence in the replicability of artifacts and provides fast feedback loops for developers.

This is also true for micro-frontends. Having solid automation pipelines will enable our micro-frontend projects to succeed, creating a reliable solution for developers to experiment, build, and deploy. In fact, for certain projects, micro-frontends could proliferate to the point where it would become nontrivial to manage them. One of the key decisions outlined in the micro-frontend decision framework, discussed in [Chapter 2](#), is the possibility of composing multiple micro-frontends in the same view (horizontal split) or having just one micro-frontend at a time (vertical split). With the horizontal split, we could end up with tens or even hundreds of artifacts to manage in our automation pipelines. Therefore, we need to invest in solutions to handle such scenarios. Vertical splits also require work, but they are similar to the traditional method of setting up automation for single-page applications (SPAs). The major difference is that you’ll have more than one artifact and potentially different ways to build and optimize your code.

We will delve into these challenges in this chapter and the subsequent ones, starting with the principles behind a solid and fast automation strategy and how we can improve the developer experience (DX) with some simple but powerful tools. Then we'll analyze best practices for CI and micro-frontend deployment. Finally, we conclude with an introduction to fitness functions for automating and testing architecture characteristics during different stages of the pipelines.

Automation Principles

Working with micro-frontends requires constant improvement of the automation pipeline. Skipping this work may hinder the delivery speed of every team working on the project and decrease their confidence in production deployment—or, worse, frustrate both developers and the business when the project fails. Nailing the automation part is fundamental if you're going to have a clear idea of how to build a successful CI, continuous delivery, or continuous deployment strategy.

CONTINUOUS INTEGRATION VERSUS DELIVERY VERSUS DEPLOYMENT

An in-depth discussion about continuous integration, continuous delivery, and continuous deployment is beyond the scope of this book. However, it's important to understand the differences between these three strategies. *Continuous integration* defines a strategy where an automation pipeline kicks in for every merge into the main branch, extensively testing the codebase before the code is merged in the release branch. *Continuous delivery* is an extension of CI where, after the tests, we generate the artifact ready to be deployed with a simple click from a deployment dashboard. *Continuous deployment* goes one step further, deploying the artifacts built after the code is committed in the version control system directly into production. If you are interested in learning more, I recommend reading *Continuous Delivery* by Jez Humble and David Farley.

To get automation speed and reliability correct, we need to keep the following principles in mind:

- Keep the feedback loop as fast as possible.
- Iterate often to enhance the automation strategy.

- Empower your teams to make the right decisions for the micro-frontends they are responsible for.
- Identify some boundaries—also called guardrails—where teams operate and make decisions while maintaining tool standardization.
- Define a solid testing strategy.

Let's discuss these principles to get a better understanding of how to leverage them.

Keep the Feedback Loop Fast

One of the key features of a solid automation pipeline is fast execution. Every automation pipeline provides feedback for developers. Having a quick turnaround on whether our code has broken the codebase is essential for developers to establish confidence in what they have written. Good automation should run often and provide feedback in a matter of seconds—or minutes, at the most. It's important for developers to receive constant feedback to encourage them to run the tests and checks within the automation pipeline more frequently. It's essential, then, to analyze which steps may be parallelized and which are serialized. A technical solution that allows both is ideal. For example, we may decide to parallelize the unit testing step so we can run our tests in small chunks instead of waiting for hundreds, if not thousands, of tests to pass before moving to the next step. However, some steps cannot be parallelized, so we should know how to optimize these steps to be as fast as possible.

Working with micro-frontends, by definition, should simplify the optimization of the automation strategy. Because we are dividing an entire application into smaller parts, there is less code to test and build, for instance, and every stage of a CI should be very fast.

Iterate Often

An automation pipeline is not a piece of infrastructure that, once defined, remains as it is until the end of a project life cycle. Every automation pipeline must be reviewed, challenged, and improved. It's essential to maintain a fast automation pipeline to enable our developers to receive quick feedback loops. In order to constantly improve, we need to visualize our pipelines. Dashboards can show how long it will take to build artifacts, making it clear to the team how healthy the pipelines are (or aren't) and immediately indicating whether a job failed or succeeded. When pipelines take more than 8–10 minutes, it's time to review them and identify opportunities to optimize the

automation strategy. Review this strategy regularly—monthly if the pipelines are running slowly, and then every three to four months once they’re healthy. Don’t stop reviewing after defining the automation pipeline. Continue to improve it, aiming for better performance and faster feedback loops; this investment of time and effort will pay off very quickly.

Empower Your Teams

At several companies where I worked in the past, the automation strategy was kept out of capable developers’ hands. Only a few people inside the organization understood how the entire automation system worked, and even fewer were allowed to change the infrastructure or take steps to generate and deploy an artifact. This is the worst nightmare of any developer working in an organization with one or more teams. The developer’s job shouldn’t be just writing code; it should include a broad range of tasks, including how to set up and change the automation pipeline for the artifacts they are working on, whether it’s a library, a micro-frontend, or the entire application.

Empowering our teams when we are working with micro-frontends is essential, as we cannot always rely on every micro-frontend using the same build pipeline—especially when maintaining multiple stacks at the same time. Certainly, the deployment phase will be the same for all micro-frontends in a project. However, the build pipeline may use different tools or optimizations, and centralizing these decisions could lead to a worse outcome than allowing developers to manage their own automation pipelines.

Ideally, the organization should provide some guardrails for the development team. For instance, the CI/CD tool should be the platform team’s responsibility, but all the scripts and steps to generate an artifact should be owned by the team because they know the best way to produce an optimized artifact with the code they have written. This doesn’t mean creating silos between a team and the rest of the organization, but rather empowering them to make certain decisions that would result in a better outcome for everyone.

Last but not least, encourage a culture of sharing and innovation by creating opportunities for the teams to share their ideas, proof of concepts, and solutions. This is especially important when you work in a distributed environment. Remote meetings lack the everyday casual work conversations that we have around the coffee machine. Creating a virtual opportunity to enjoy these conversations may seem excessive at the beginning, but it helps keep morale up and establishes connections between team members.

Define Your Guardrails

An important principle for empowering teams and having a solid automation strategy is creating some guardrails for them to make sure they are heading in the right direction.

Guardrails for the automation strategy are boundaries identified by tech leadership, in collaboration with architects and/or platform or cloud engineers, within which teams can operate and add value for the creation of micro-frontends.

In this situation, guardrails might include the tools used for running the automation strategy, the dashboard used for deployment in a continuous delivery strategy, or the fitness functions for enforcing architecture characteristics that we discuss extensively during this chapter.

Introducing guardrails doesn't mean reducing developers' freedom. Instead, they guide developers toward using the company's standards, shielding them as much as possible from unnecessary complexity, while still allowing teams to innovate within these boundaries. It's important to find the right balance when defining these guardrails and to make sure everyone understands the "why" behind them, rather than just the "how." Usually, creating documentation helps to scale and spread this knowledge across teams and to new employees. As with other parts of the automation strategy, guardrails shouldn't be fixed; they need to be revised, improved, or even removed, as the business evolves.

Define Your Test Strategy

Investing time in a solid testing strategy is essential—especially end-to-end testing—when multiple micro-frontends contribute to a single view and multiple teams collaborate to deliver the final user experience. It's critical to ensure that transitions between views are thoroughly tested and that they function correctly before deploying to production.

However, end-to-end tests in a micro-frontend architecture should be selective and fast, focusing on critical user journeys. Avoid large, brittle test suites that slow down feedback loops and reduce developer productivity. By targeting the most important interactions and relying on a mix of unit and integration tests for individual micro-frontends, teams can maintain confidence while keeping the testing process efficient.

While unit and integration testing remain important, micro-frontends do not introduce unique challenges at this level. Instead, end-to-end testing needs to be adapted to this architecture. Because every team owns a part of the application, we need to make sure

the critical path of this application is extensively covered to achieve the final desired outcome. End-to-end testing will help ensure those things.

Developer Experience

A key consideration when working with micro-frontends is DX. While not all companies can support a dedicated DX team, even a virtual team—composed of members from different teams who collaborate periodically as needed—can be valuable. Such a team is responsible for creating tools and improving the experience of working with micro-frontends, helping to prevent friction when developing new features.

At this stage, it should be clear that every team is responsible for part of the application, not the entire codebase. Creating a frictionless DX will help our developers feel comfortable with building, testing, and debugging the part of the application they are responsible for. We need to guarantee a smooth experience for testing micro-frontends both in isolation and within the overall web application, as there are always touchpoints between micro-frontends, regardless of the architecture used. A bonus would be creating an extensible experience that isn't closed to the possibility of embracing new or different tools during the project life cycle.

WHAT DOES DX MEAN?

DX usually refers to having one or more teams dedicated to studying, analyzing, and improving how developers get their work done. Specifically, such teams observe which tools and processes developers use to accomplish their daily tasks, providing support for improving the development life cycle across the entire organization. One of a DX team's goals is to simplify the process of building, testing, and deploying artifacts across different environments, while also providing tools that enhance delivery speed and accelerate developers' feedback loops when automating guardrails.

Many companies have created end-to-end solutions that they maintain alongside the projects they are working on, effectively filling the gaps left by existing tools when needed. This may seem like a great way to create the perfect DX for an organization; however, businesses aren't static, and neither are tech communities. As a result, we need to account for the cost of maintaining a custom DX, as well as the cost of onboarding new employees. It may still be the right decision for your company,

depending on its size or the type of project, but I encourage you to analyze all the options before committing to building an in-house solution to ensure your investment is maximized.

Horizontal Versus Vertical Split

The decision between a horizontal and vertical split in your micro-frontend project will significantly impact DX. Vertical splits often align naturally with team boundaries, enabling clearer ownership and autonomy. In contrast, horizontal splits—divided by technical layers or shared concerns—require stronger governance and coordination to prevent dependency chaos and integration challenges. A vertical split will represent the micro-frontends as single HTML pages or SPAs owned by a single team, resulting in a DX very similar to the traditional development of an SPA. All the tools and workflows available for SPA will suit the developers in this case. You may want to create some additional tools specifically for testing your micro-frontend under certain conditions as well. For instance, when you have an application shell loading a vertical micro-frontend, you may want to create a script or tool for testing the application shell version available on a specific environment to make sure your micro-frontend works with the latest or a specific version.

The testing phases are very similar to a normal SPA, where we unit, integration, and end-to-end testing can be set up without any particular challenges. Therefore, every team can test its own micro-frontends, as well as the transitions between micro-frontends—for example, ensuring that the next micro-frontend is fully loaded. However, we also need to make sure that all micro-frontends are reachable and loadable within the application shell. One solution I've seen work very well is having the team that owns the application shell perform end-to-end testing for routing between micro-frontends, allowing them to test interactions across all micro-frontends.

Horizontal splits entail a different set of considerations. When a team owns multiple micro-frontends that are part of one or more views, we need to provide tools for testing within these views, assembling the page at runtime. These tools need to allow developers to review potential dependency clashes, communication with micro-frontends developed by other teams, and the overall picture. These aren't standard tools, and many companies have had to develop custom tools to solve these challenges. The right tools will vary, depending on the environment and context we operate in, so what worked for one company may not work for another. Some solutions associated with the chosen framework will work, but more often than not, we will need to customize tools to provide our developers with a frictionless experience.

Another challenge with a horizontal split is how to run a solid testing strategy. We will need to identify which team will run the end-to-end testing for every view and how specifically the integration testing will work, given that an action happening with one micro-frontend may trigger a reaction with another. We do have ways to solve these problems, but the governance behind them may be far from trivial. The DX with micro-frontends is not always straightforward. The horizontal split in particular is challenging because we need to answer far more questions and make sure our tools are constantly up to date to simplify the lives of our developers.

Frictionless Micro-Frontend Blueprints

The micro-frontend DX isn't only about development tools; we must also consider how the new micro-frontends will be created. The more micro-frontends we have and the more we have to create, the more mandatory it will become to speed up and automate this process. Creating a command-line tool for scaffolding a micro-frontend will not only cover implementation—providing a team with all the dependencies to start writing code—but also take care of collecting and providing best practices and guardrails within the company. For instance, if we are using a specific library for observability or logging, adding the library to the scaffolding can speed up the creation of a micro-frontend—and it guarantees that your company's standards will be in place and ready to use.

An alternative that has gained more traction in recent years is a centralized portal where developers can retrieve blueprints for micro-frontends and other resources. Open source tools like [Spotify's Backstage](#) help centralize blueprints, documentation, and more in a single platform. It provides developers with a one-stop destination for their day-to-day needs, eliminating the need to search through scattered resources across multiple internal systems such as wikis, repositories, and other tools. I have seen many companies implementing this strategy successfully for their distributed systems, including for micro-frontends.

Another important item to provide out of the box would be a sample of the automation strategy, with all the key steps needed for building a micro-frontend. Imagine that we have decided to run static analysis and security testing inside our automation strategy. Providing a sample of how to configure it automatically for every micro-frontend would increase developers' productivity and help get new employees up to speed faster. This scaffolding would need to be maintained in collaboration with developers learning the challenges and solutions directly from those on the ground. A sample can help communicate new practices and specific changes that arise during the development of

new features or projects, further saving your team time and helping them work more efficiently.

Environments Strategies

Another important consideration for DX is enabling teams to work within the company's environment strategy. The most commonly used strategy across mid- to large-sized organizations is a combination of testing, staging, and production environments. The testing environment is often the most unstable of the three because it's used for quick attempts made by the developers. As a result, staging should resemble the production environment as much as possible, and the production environment should be accessible only to a subset of people. In addition, the DX team should create strict controls to prevent manual access to this environment and provide a swift solution for promoting or deploying artifacts in production.

An interesting twist on the classic environment strategy is spinning up environments with a subset of the system for testing purposes (for example, end to end or visual regression) and then tearing them down once the operation is complete. This strategy of on-demand environments is a great addition for a company because it supports not only micro-frontends but also microservices, enabling testing of end-to-end flows in isolation. With this approach, it's also possible to conduct end-to-end testing in isolation for an entire business subdomain, deploying only the necessary microservices and running multiple on-demand environments, which can save a considerable amount of money.

Another feature of on-demand environments is the ability to offer the business or a product owner a preview of an experiment or a specific feature. Nowadays, many cloud providers like AWS can provide significant cost savings by using **spot instances**—a middle-ground approach where infrastructure is more cost-effective than standard offerings because it leverages spare capacity for a limited time. Spot instances are a perfect fit for on-demand environments.

Version Control

When we start to design an automation strategy, a mandatory step is to select a version control and branching strategy to adopt. While there are valid alternatives, like Mercurial, Git is the most popular version control system. I'll use Git as a reference in my examples that follow, but know that all the approaches are applicable to Mercurial as well.

Working with version control means deciding which approach to use in terms of repositories. Usually, the debate is between a monorepo and a polyrepo, also called a multirepo. There are benefits and pitfalls with both approaches, but you can employ both in your micro-frontend project to make use of the right technique for your context.

Monorepo

A monorepo (see [Figure 6-1](#)) is based on the concept that all teams are using the same repository. Therefore, all the projects are hosted together.

My Project



Sign-in MFE



Sign-up MFE



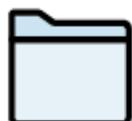
Catalog MFE



iOS app



Android app



Auth service



Catalog service



Search service

...

Figure 6-1. A monorepo example, where all the projects live inside the same repository

The main advantages of using a monorepo are as follows:

Code reusability

Sharing libraries becomes very natural with this approach. Because all of a project's codebase lives in the same repository, we can seamlessly create a new project, abstracting some code and making it available for all the projects that can benefit from it.

Easy collaboration across teams

Because the discoverability is completely frictionless, teams can contribute across projects. Having all the projects in the same place makes it easy to review a project's codebase and understand the functionality of another project. This approach facilitates communication with the team responsible for the maintenance in order to improve or simply change the implementation by pointing to a specific class or line of code without kicking off an abstract discussion.

Cohesive codebase with less technical debt

Working with a monorepo encourages every team to be up to date with the latest dependency versions—specifically APIs, but also the latest solutions developed by other teams. This means that the project might break and require refactoring when a breaking change occurs. A monorepo forces teams to continually refactor the codebase, which improves the code quality and reduces technical debt.

Simplified dependencies management

With monorepos, all the dependencies used by several projects are centralized, so we don't need to download them every time for all our projects. When we want to upgrade to the next major release, all the teams using a specific library must work together to update their codebase, reducing the technical debt that, in other cases, a team may accumulate. Updating a library may cost a bit of coordination overhead, especially when you work with distributed teams or large organizations.

Large-scale code refactoring

A monorepo is very useful for large-scale code refactoring. Because all the projects are in the same repository, refactoring them at the same time is trivial. Teams will need to coordinate or work with technical leaders, who have a strong high-level picture of the entire codebase and are responsible for refactoring or coordinating the refactor across multiple projects.

Easier onboarding for new hires

With monorepos, a new employee can quickly find code samples from other repositories. Additionally, a developer can find inspiration from other approaches and quickly shape them inside the codebase.

Despite the undoubted benefits, embracing a monorepo also brings some challenges:

Constant investment in automation tools

Monorepos require a constant, critical investment in automation tools, especially for large organizations. There are plenty of open source tools available, but not all are suitable for a monorepo—particularly after some time on the same project, when the monorepo starts to grow exponentially. Many large organizations must constantly invest in improving their automation tools for a monorepo to avoid their entire workforce being slowed down by intermittent commitment to improving the automation pipelines and reducing the feedback loop time for the developers.

Scaling tools when the codebase increases

Another important challenge is that automation pipelines must scale alongside the codebase. Many reports from Google, Facebook, and X (Twitter) claim that after a certain threshold, the investment in having a performant automation pipeline increases until the organization has several teams working exclusively on it.

Unsurprisingly, every aforementioned company has built its own version of build tools and released them as open source to deal with the unique challenges they face with thousands of developers working in the same repository.

Projects are coupled together

Given that all the projects are easy to access and, more often than not, share libraries and dependencies, we risk having tightly coupled projects that can exist only when they are deployed together. We may, therefore, not be able to share our micro-frontends across multiple projects for different customers when the codebase lives in a different monorepo. This is a key consideration to think about before embracing the monorepo approach with micro-frontends.

Trunk-based development

Trunk-based development is the only branching strategy that makes sense with a monorepo. It is based on the assumption that all the developers commit to the same branch, called the trunk. Considering that all the projects live inside the same repository, the main trunk branch may receive thousands of commits per day, so it's essential to commit often with small increments rather than developing an entire feature per day before merging. This technique encourages developers to commit smaller chunks of code, helping avoid the “merge hell” common in other branching strategies. Although I am a huge fan of trunk-based development, it requires discipline and maturity across the entire organization to achieve good results.

Disciplined developers

We must have disciplined developers in order to maintain a healthy codebase. When tens, or even hundreds, of developers work in the same repository, the Git history—along with the codebase—could become messy very quickly. Unfortunately, it's almost impossible to have senior developers on all teams, and that lack of knowledge or discipline could compromise the repository quality and extend the impact from one project inside the monorepo to many, if not all, of them.

Using a monorepo for micro-frontends is definitely an option, and tools like **Lerna** help with managing multiple projects inside the same repository. In fact, if needed, Lerna can install and **hoist** all dependencies across packages together and publish a package when a new version is ready to be released. However, we must understand that one of the main monorepo strengths is its code-sharing capability, which requires a significant commitment to maintain the quality of the codebase. Therefore, we must be careful not

to couple too many of our micro-frontends, because we risk losing their nature of independently deployable artifacts.

Git has started to invest in reducing the operation times when a user invokes commands like Git history or Git status in large repositories. And, as monorepos have become more popular, Git has been actively working on delivering additional functionalities for filtering what a user wants to clone to their machine without needing to clone the entire Git history and all the project's folders.

Obviously, these enhancements will also be beneficial for CI/CD, helping to overcome one of the main challenges of embracing a monorepo strategy.

SPARSE-CHECKOUT

In early 2020, Git introduced the `sparse-checkout` command (for version 2.25 and later) for cloning only part of a repository instead of all the files and history. This reduction in the amount of data to clone for running fast automation pipelines addresses one of the main challenges of adopting the monorepo approach.

Using the monorepo approach means investing in our tools, evangelizing and building discipline across our teams, and committing to a constant investment in improving the codebase. If these characteristics suit your organization, a monorepo would likely enable you to successfully support your projects. Many companies are using monorepos, specifically large organizations like Google and Meta, where the investment in maintaining this paradigm is totally sustainable. One of the most famous [papers on monorepos](#) was written by Google's Rachel Potvin and Josh Levenberg. In its concluding paragraph, they write:

Over the years, as the investment required to continue scaling the centralized repository grew, Google leadership occasionally considered whether it would make sense to move from the monolithic model. Despite the effort required, Google repeatedly chose to stick with the central repository due to its advantages.

The monolithic model of source code management is not for everyone. It is best suited to organizations like Google, with an open and collaborative culture. It would not work well for organizations where large parts of the codebase are private or hidden between groups.

Polyrepo

The opposite of a monorepo strategy is the polyrepo (see [Figure 6-2](#)), or multirepo, where every single application lives in its own repository.

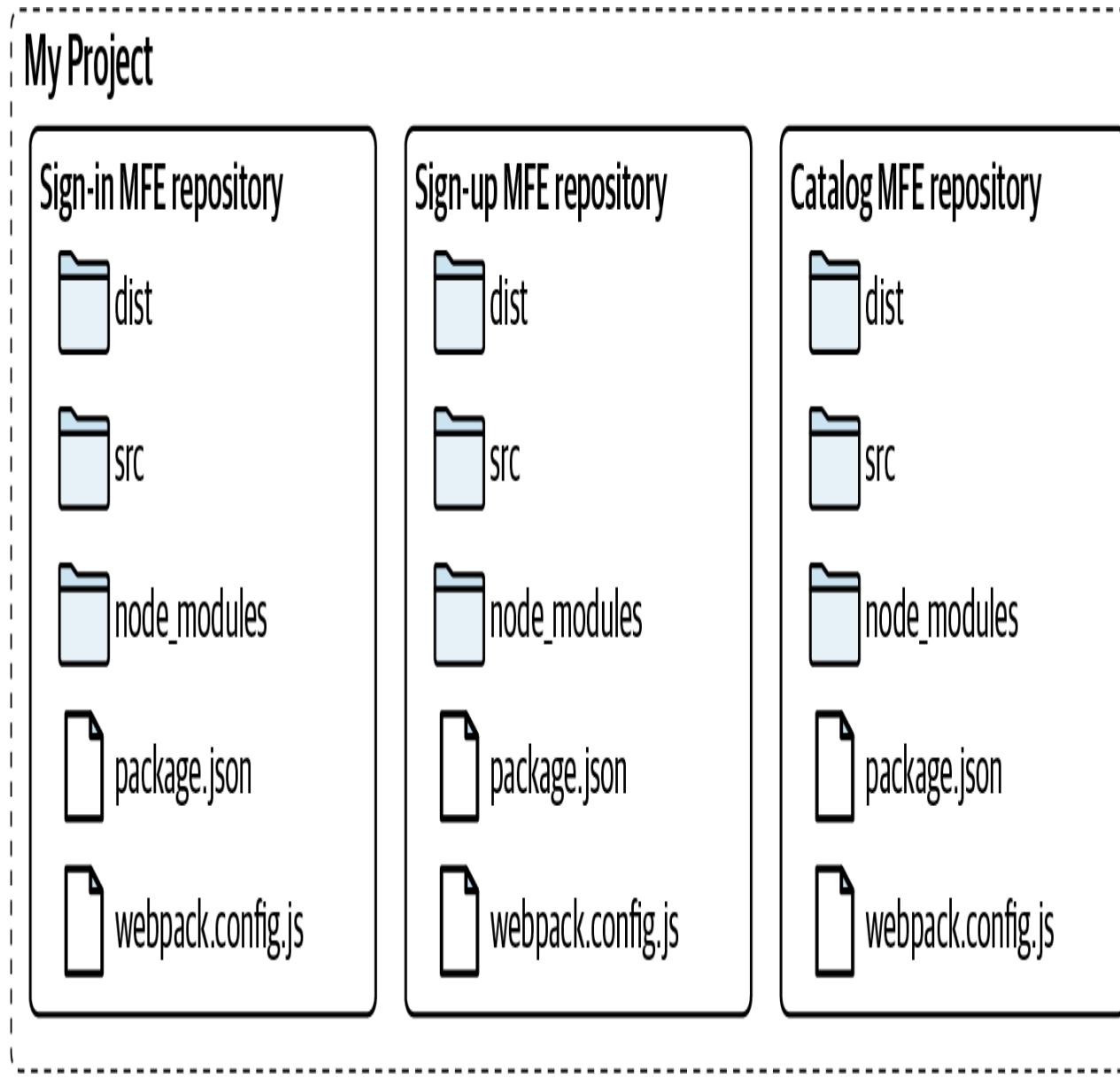


Figure 6-2. A polyrepo example, where we split the projects across multiple repositories

Some benefits of a polyrepo strategy are as follows:

Different branching strategy per project

With a monorepo strategy, we should use trunk-based development, but with a polyrepo, we can use the right branching strategy for the project that we are working on. Imagine, for instance, that we have a legacy project with a different release cadence than other projects

that are in continuous deployment. With a polyrepo strategy, we can use [Git flow](#) in just that project, providing a branching strategy specific to that context.

No risk of blocking other teams

Another benefit of working with a polyrepo is that the blasting radius of our changes is strictly confined to our project. There isn't any possibility of breaking other teams' projects or negatively affecting them, because they live in another repository.

Encourages thinking about contracts

In a polyrepo environment, the communication across projects has to be defined via APIs. This means that every team must identify the contracts between producers and consumers, and create governance to manage future releases and breaking changes.

Fine-grained access control

Large organizations are more likely to work with contractors, who should see only the repositories they are responsible for, or to implement a security strategy where only certain departments can access and work on specific areas of the codebase. Polyrepo is the perfect strategy for achieving that fine-grained access control on different codebases, introducing roles, and identifying the level of access needed for every team or department.

Less up-front and long-term investment in tooling

With a polyrepo strategy, we can easily use any tool available out there. For instance, using managed solutions like CircleCI or Drone CI would be more than sufficient due to the contained size of a micro-frontend repository. Usually, the repositories are not expanding at the same rate as monorepo repositories because fewer developers are committing to the codebase. This means that your investment up front and in maintaining the build of a polyrepo environment is far less, especially when you automate your CI/CD pipeline using infrastructure as code or command-line scripts that can be reused across different teams.

Polyrepo, however, also has some caveats:

Difficulties with project discoverability

By its nature, polyrepo makes it more difficult to discover other projects because every project is hosted in its own repository. Creating a solid naming convention that allows every developer to discover other projects easily can help mitigate this issue. Unquestionably, however, polyrepo can make it difficult for new employees or for developers who are comparing different approaches to find other projects suitable for their research.

Code duplication

Another disadvantage of polyrepo is code duplication. For example, a team might create a library that will be used by other teams for standardizing certain approaches, but the tech department may not be aware of that library. Often, there are libraries that should be used across several micro-frontends, like logging or observability integration, but a polyrepo strategy doesn't facilitate code sharing if there isn't good governance in place. It's helpful, then, to identify the common aspects that may be beneficial for every team and to coordinate the code-sharing effort across teams. Architects and tech leaders are in the perfect position to do this, as they work with multiple teams and have a high-level picture of how the system works and what it requires.

Naming convention

In polyrepo environments, I've often seen a proliferation of names without any specific convention; this quickly compounds the issue of tracking what is available and where. Regulating a naming convention for every repository is critical in a polyrepo system because working with micro-frontends—and possibly also with microservices—could result in a huge number of repositories inside our version control system.

Best practice maintenance

In a monorepo environment, we have just one repository to maintain and control. In a polyrepo environment, however, it may

take a while before every repository is in line with a newly defined best practice. Again, efficient communication and processes may mitigate this problem, but polyrepo requires you to think this through up front because finding out these problems during development will slow down your team's throughput.

Polyrepo is definitely a viable option for micro-frontends, though it carries the risk of a proliferation of repositories. This complexity should be handled with clear and strong governance around naming conventions, repository discoverability, and processes.

Micro-frontend projects with a vertical split have far fewer issues when using polyrepo than those with a horizontal split, where the application is composed of many different parts. In the context of micro-frontends, polyrepo also makes it possible to use different approaches from a legacy project. For example, we may introduce new tools or libraries specifically for the micro-frontend approach while keeping the existing ones for the legacy project, without affecting the best practices already in place on the legacy platform. This flexibility must be weighed against potential communication overhead and governance defined within the organization. Therefore, if you decide to use polyrepo, be aware of where you focus your initial investment; it should be on governance and communication flows across teams.

A Possible Future for a Version Control System

Any of the different paths we take with a version control system will not be a perfect solution, just the solution that works best in a given context. It's always a trade-off. However, we may want to try a hybrid approach (see [Figure 6-3](#)), where we can minimize the pitfalls of both approaches and leverage their benefits. Because micro-frontends and microservices should be designed using domain-driven design, we may follow the subdomain and bounded context divisions for bundling all the projects that are included in a specific subdomain.

My Project

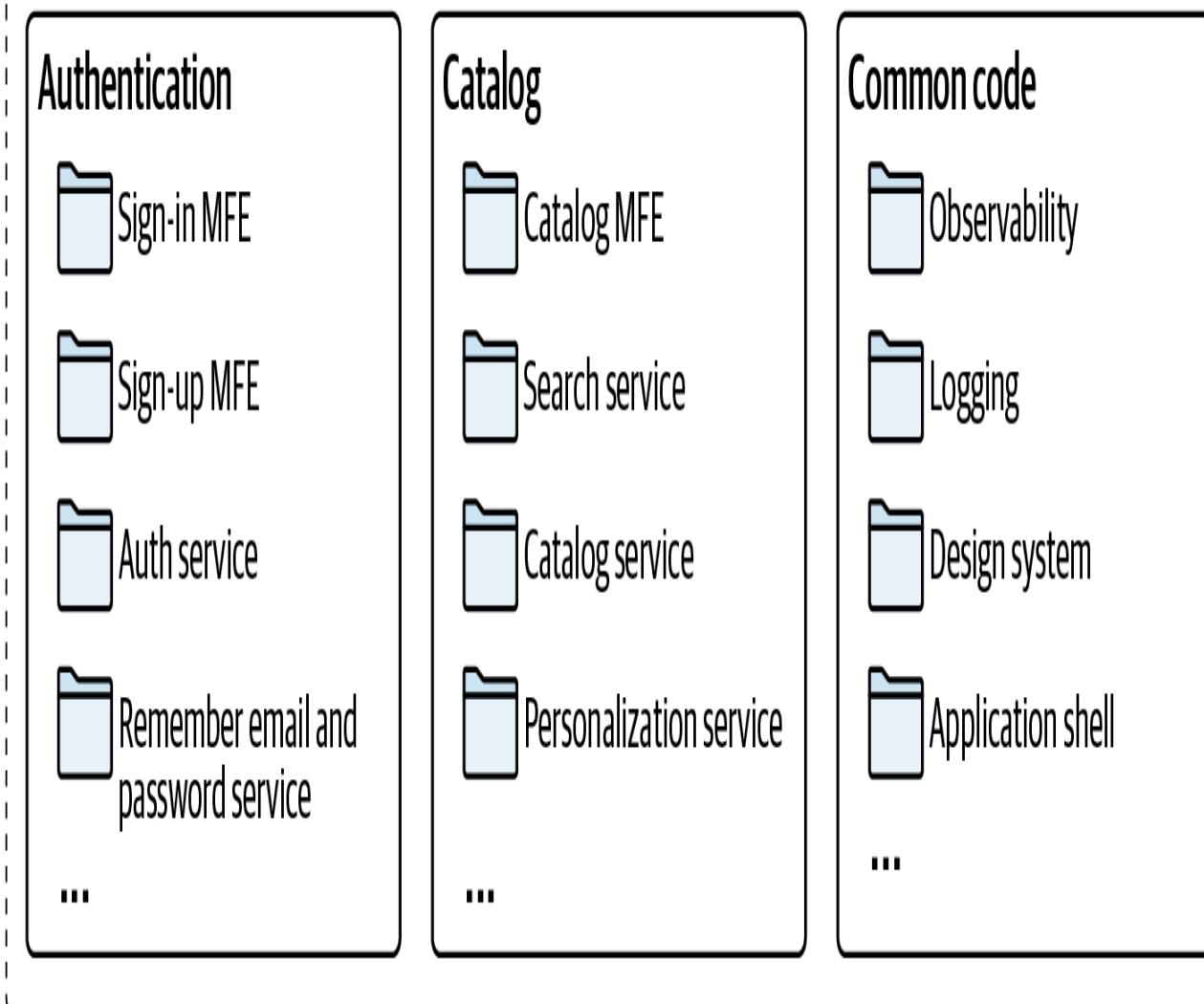


Figure 6-3. A hybrid repositories approach, combining monorepo and polyrepo strengths in a unique solution

In this way, we can enforce the collaboration across teams responsible for different bounded contexts and work with contracts while benefiting from the monorepo's strengths across all the teams working in the same subdomain. This approach might result in new practices, new tools to use or build, and new challenges, but it's an interesting solution worth exploring for microarchitectures.

Continuous Integration Strategies

After identifying the version control strategy, we need to think about the CI method. Different CI implementations in different companies are the most successful and effective when managed by the developer teams rather than by an external guardian of the CI machines.

A lot of things have changed in the past few years. For one thing, developers—including frontend developers—have had to become more aware of the infrastructure and tools needed for running their code because, in reality, building the application code in a reliable and fast pipeline is part of their job. In the past, I've seen many situations where the CI was delegated to other teams in the company, denying the developers a chance to change anything in the CI pipeline. As a result, the developers treated the automation pipeline as a closed system—impossible to change, but needed for deploying their artifacts to an environment. More recently, thanks to the DevOps culture spreading across organizations, these situations are becoming increasingly rare.

ABOUT DEVOPS

DevOps is the combination of cultural philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity. Under a DevOps model, development and operations teams are no longer siloed. Sometimes, they're merged into a single team, where the engineers work across the entire application life cycle—from development and test to deployment and operations—and develop a range of skills that aren't limited to a single function.

Nowadays, many companies are giving developers ownership of their automation pipelines. That doesn't mean developers should be entitled to do whatever they want in the CI, but they definitely should have some skin in the game, because the speed of the feedback loop depends mainly on developers. The tech leadership team (architects, platform team, DX, tech leaders, engineers, managers, and so on) should provide the guidelines and the tools within which the teams operate, while also allowing some flexibility within those boundaries.

In a micro-frontend architecture, the CI is even more important because of the number of independent artifacts we need to build and deploy reliably. The developers, however, are responsible for running the automation strategy for their micro-frontends, using the right tool for the right job. This approach may seem excessive, considering that every micro-frontend may use a different set of tools. However, we usually end up having a

few tools that perform similar tasks, and this approach also allows for a healthy comparison of tools and approaches, helping teams to develop best practices.

More than once, I have walked the corridors and overheard conversations between engineers about how building tools like Rollup have certain features or performance advantages that Webpack doesn't have in specific scenarios, and vice versa. This, for me, is a sign of a great confrontation between tools tested in real-world scenarios rather than in a sandbox.

It's also important to recognize that there isn't a unique CI implementation for micro-frontends; a lot depends on the project, company standards, and the architectural approach. For instance, when implementing micro-frontends with a vertical split, all the stages of a CI pipeline would resemble normal SPA stages. End-to-end testing may be done before deployment if the automation strategy allows for the creation of on-demand environments. After the test is completed, the environment can be turned off. However, a horizontal split would require more consideration regarding the right moment to perform a specific task. When performing end-to-end testing, we'd have to conduct this phase in staging or production; otherwise, every single pipeline would need to be aware of the entire composition of an application, retrieving every latest version of the micro-frontends and pushing to an ephemeral environment—a solution very difficult to maintain and evolve.

Testing Micro-Frontends

Plenty of books discuss the importance of testing our code and catching bugs or defects as early as possible, and the micro-frontend approach is no different.

TESTING STRATEGIES

I won't cover all the different possible testing strategies—such as unit testing, integration testing, or end-to-end testing—in this book. Instead, I'll cover the differences from a standard approach we are used to implementing in any frontend architecture. If you would like to become more familiar with different testing strategies, I recommend studying the materials shared by incredible authors like Kent Beck or Robert C. Martin (a.k.a. Uncle Bob), especially Beck's *Test-Driven Development* and Martin's *Clean Code*.

Working with micro-frontends doesn't mean changing the way we are dealing with frontend testing practices, but they do create additional complexity in the CI pipeline

when we perform end-to-end testing. As unit testing and integration testing are not changing compared to other frontend architectures, we'll focus here on end-to-end testing, as this is the biggest challenge for testing micro-frontends.

End-to-End Testing

End-to-end testing is used to test whether the flow of an application from start to finish is behaving as expected. We perform tests to identify system dependencies and ensure that data integrity is maintained between various components and systems. End-to-end testing may be performed before deploying our artifacts in production in an on-demand environment created at runtime just before tearing the environment down. Alternatively, when we don't have this capability in-house, we should perform end-to-end tests in existing environments after the deployment or promotion of a new artifact. In this case, the recommendation would be to embrace testing in production when the application has implemented feature flags, allowing a feature to be toggled on and off and granting access to test for a set of users.

Testing in production brings its own challenges, especially when a system is integrating with third-party APIs. However, it can save a lot of money on environment infrastructure, maintenance, and developers' resources because we don't have to configure and maintain multiple environments simultaneously. I'm conscious that not all companies or projects are suitable for this practice; therefore, using the environments that you have available should be considered a last resort. When you start a new project or it's possible to modify an existing one, take into consideration the possibility of introducing feature flags not only to reduce the risk of bugs in front of users but also for testing purposes.

Some of the complexity brought by micro-frontends may be mitigated with effective coordination across teams and solid governance overseeing the testing process. As discussed multiple times in this book, the complexity of end-to-end testing varies depending on whether we embrace a horizontal or vertical split for our application.

Vertical-Split End-to-End Testing Challenges

When we work with a vertical split, one team is responsible for an entire business subdomain of the application. In this case, testing all the logic paths inside the subdomain is not far from what you would do in an SPA. But we have some challenges to overcome when we need to test use cases outside of the team's control, such as the scenario in [Figure 6-4](#).

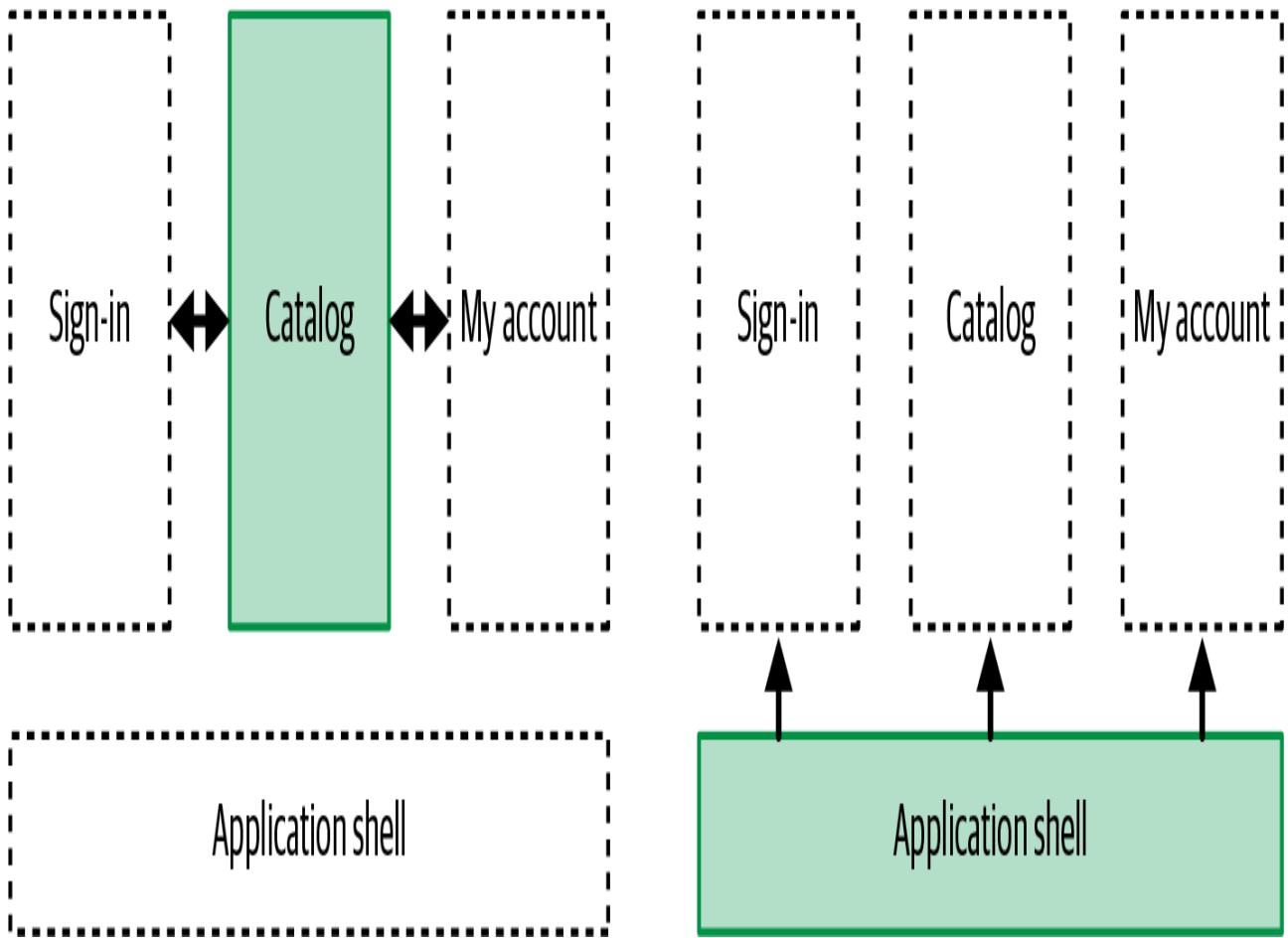


Figure 6-4. An end-to-end testing example with a vertical split architecture

The catalog team is responsible for testing all the scenarios related to the catalog. However, some scenarios involve areas not controlled by the catalog team, such as when the user signs out of the application and should be redirected to the “sign-in” micro-frontend, or when a user wants to change something in their profile and should be redirected to the “my account” micro-frontend. In these scenarios, the catalog team will be responsible for writing tests that cross their domain boundary and ensure that the appropriate micro-frontend loads correctly. In the same way, the teams responsible for the “sign-in” and “my account” micro-frontends will need to test their business domain and verify that the catalog loads correctly as the user expects.

Another challenge is making sure our application behaves in cases of deep-linking requests or when we want to test different routing scenarios. It always depends on how we have designed our routing strategy, but let’s take the example of having the routing logic in the application shell, as in [Figure 6-4](#). The application shell team should be responsible for these tests, ensuring that the entire route of the application loads

correctly, that the key behaviors like signing in or out work as expected, and that the application shell can load the correct micro-frontend when a user requests a specific URL.

Horizontal-Split End-to-End Testing Challenges

Using a horizontal-split architecture raises the question of who is responsible for end-to-end testing of the final solution. Technically speaking, what we have discussed for the vertical-split architecture still stands, but we have a new level of complexity to manage. For example, if a team is responsible for a micro-frontend in multiple views, is the team then responsible for the end-to-end testing of all the scenarios where their micro-frontends are present? Let's try to shed some light on this with the example in Figure 6-5.

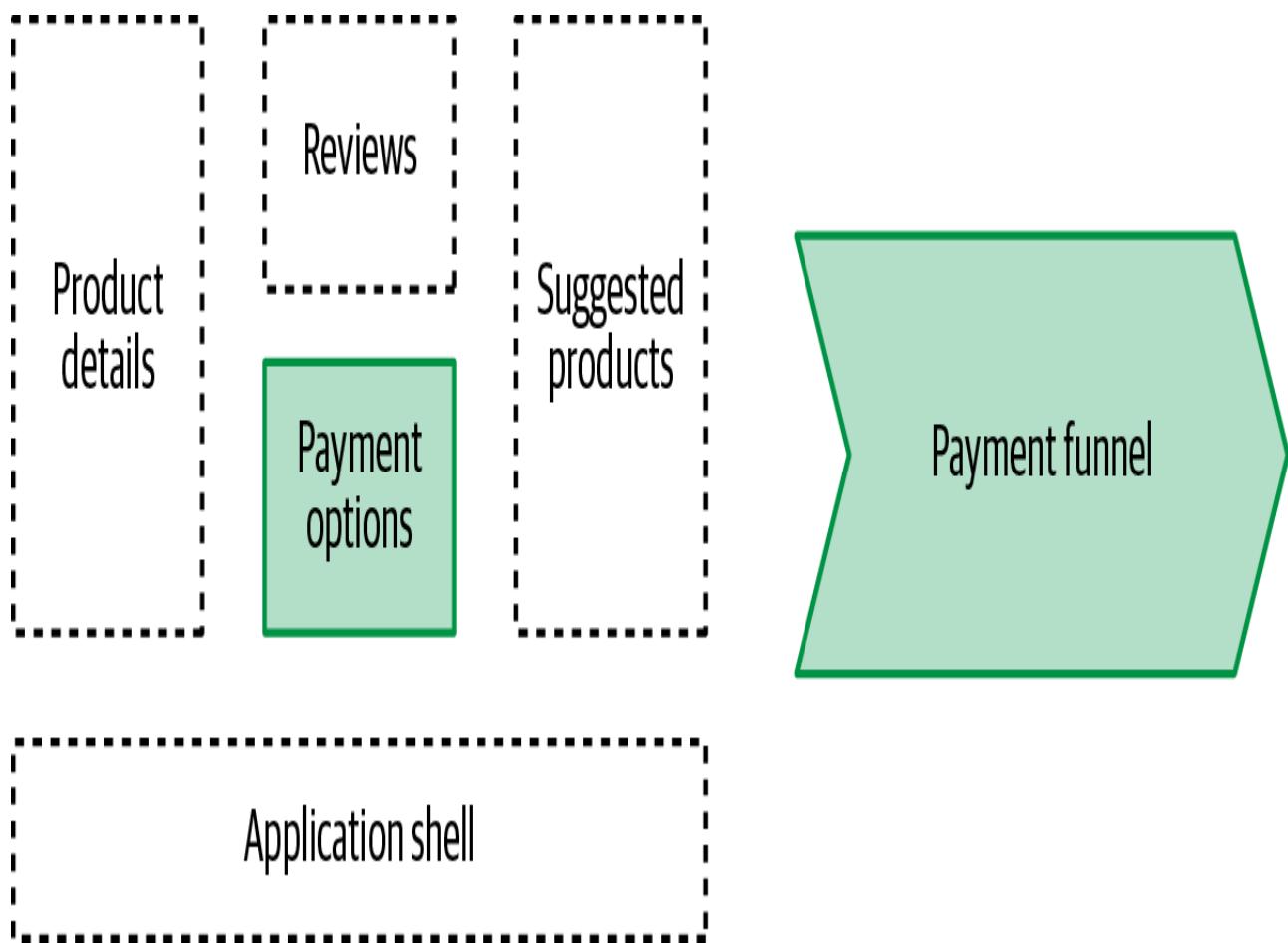


Figure 6-5. An end-to-end testing example with a horizontal-split architecture

The payments team is responsible for providing all the micro-frontends needed for making a payment within the project. In the horizontal-split architecture, their micro-frontends are available on multiple views. In [Figure 6-5](#), we can see the “payment option” micro-frontend that guides the user to choose a payment method and finalize the payment when they’re ready to check out. Therefore, the payments team is responsible for ensuring the user can pick a payment option and complete the check-out, showing the interface needed for performing the monetary transaction with the selected payment option in the following view. In this case, the payments team can perform the end-to-end test for this scenario, but will need significant coordination and governance to analyze all the necessary end-to-end tests and assign them to the appropriate teams to avoid duplication of effort, which is even more difficult to maintain in the long run.

The end-to-end tests also become more complex to maintain as different teams are contributing to the final output of a view, and some tests may become invalid or broken the moment other teams change their micro-frontends. That doesn’t mean we aren’t capable of performing end-to-end testing with a horizontal split, but it does require better organization and more consideration before implementing.

Testing Technical Recommendations

For technically implementing end-to-end tests successfully for horizontal- or vertical-split architecture, there are three main options. The first is running all the end-to-end tests in a stable environment where all the micro-frontends are present. We delay the feedback loop if our new micro-frontend works as expected, end to end.

Another option is using on-demand environments where we pull together all the resources needed for testing our scenarios. This option may become complicated in a large application, however, especially when we use horizontal-split architecture. This option may also cost us a lot when it’s not properly configured (as explained earlier).

Finally, we may decide to use a proxy server that will allow us to conduct end-to-end testing of the micro-frontend we are responsible for. When we need to use any other part of the application involved in a test, we’ll just load the parts needed from an environment, either staging or production—in this case, the micro-frontends and the application shell not developed by our team.

In this way, we can reduce the risk of unstable versions or optimization for generating an on-demand environment. The team responsible for the end-to-end testing won’t have any external dependencies to manage either, but they will be able to test all the scenarios needed to ensure the quality of their micro-frontend.

WEBPACK DEV SERVER PROXY CONFIGURATION

For completeness of information, Webpack—as with many other building tools—allows us to configure a proxy server that retrieves external resources from specific URLs, such as static files, or even consumes APIs in a specific environment. This feature may be useful for setting up our end-to-end testing in scenarios where we want to run it during the CI pipeline. The configuration is trivial, as you can see in the following example:

```
// In webpack.config.js
{
  devServer: {
    proxy: {
      '/catalog-mfe': {
        target: 'https://other-server.example.com',
        secure: false
      }
    }
  }
}

// Multiple entry
proxy: [
  {
    context: ['/catalog-mfe/**', '/myaccount-mfe/**'],
    target: 'https://other-server.example.com',
    secure: false
  }
]
```

Additional information about the Webpack proxy setup is available in the [Webpack documentation](#).

When the tool used for running the automation pipeline allows it, you can also set up your CI to run multiple tests in parallel instead of in sequence. This will speed up the results of your tests, specifically when you are running many of them at once; it can also be split into parts and grouped in a sensible manner. If we have a thousand unit tests to run, for example, splitting the effort into multiple machines or containers may save us time and get us results faster. This technique may also be applied to other stages of our CI pipeline. With just a little extra configuration by the development team, you can save time testing your code and gain confidence in it sooner. Even tools that work well for us can be improved, and systems evolve over time. Be sure to analyze your tools and any potential alternatives regularly to ensure you have the best CI tools for your purposes.

Fitness Functions

Considering the inherent complexity of distributed systems where multiple modules make up an entire platform, the architecture team should have a way to measure the impact of their architecture decisions and make sure these decisions are followed by all teams, whether they are co-located or distributed. In the book *Building Evolutionary Architecture*, **Neal Ford, Rebecca Parsons, and Patrick Kua** discuss how to test an architecture's characteristics in CI with fitness functions. They state that a fitness function “provides an objective integrity assessment of some architectural characteristic(s).”

Many of the steps defined inside an automation pipeline are used to assess architecture characteristics, such as static analyses in the shape of cyclomatic complexity or the bundle size in the micro-frontend use case. Having a fitness function that assesses the bundle size of a micro-frontend is a good idea when a key characteristic of your micro-frontend architecture is the size of the data downloaded by users. The architecture team may decide to introduce fitness functions inside the automation strategy, guaranteeing the agreed-upon outcome and trade-off that a micro-frontend application should have. Here are some key architecture characteristics to pay attention to when designing the automation pipeline for a micro-frontend project:

Bundle size

Allocate a budget size per micro-frontend and analyze when this budget is exceeded and why. In the case of shared libraries, also review the size of all the libraries shared, not only the ones built with micro-frontends.

Performance metrics

Tools like Lighthouse and WebPageTest allow us to validate whether a new version of our application has the same or higher standards as the current version.

Static analysis

There are plenty of tools for static analysis in the JavaScript ecosystem, with SonarQube probably being the most well-known. Implemented inside an automation pipeline, this tool will provide us with insights such as the cyclomatic complexity of a project (in our case, a micro-frontend). We may also want to enforce a high code-

quality bar when setting a cyclomatic complexity threshold over which we don't allow the pipeline to finish until the code is refactored.

Architecture-characteristics testing

In a distributed architecture like micro-frontends, it's not enough to only test for functionality—it's equally important to ensure that the system's structural boundaries and architectural decisions are respected as the codebase evolves. Architecture-characteristics testing allows you to codify these decisions as automated checks (fitness functions) in your CI pipeline, providing fast, objective feedback on whether your implementation aligns with the intended architecture.

By implementing architecture tests, you can verify that boundaries between micro-frontends remain clear (for example, ensuring there are no direct dependencies between micro-frontends or that only approved shared libraries are used), and that architectural rules around modularity, isolation, and separation of concerns are upheld. This approach helps prevent unwanted coupling, architectural drift, and technical debt—especially as teams grow or become more distributed. Ultimately, architecture-characteristics testing empowers architects and tech leads to enforce standards and guide the system's evolution without micromanaging every pull request.

Code coverage

Another example of a fitness function is making sure our codebase is tested extensively. Code coverage provides a percentage of tests run against our project. However, keep in mind that this metric doesn't provide us with the quality of the test, just a snapshot of tests written for public functions.

Security

Finally, we want to ensure our code won't violate any regulations or rules defined by the security or architecture teams.

These are some architecture characteristics that we may want to test in our automation strategy when we work with micro-frontends. In a distributed architecture like this one, these metrics become fundamental for architects and tech leads to understand the quality of the product developed and where the tech debt lies, and to enforce key architecture characteristics without having to chase every team or be part of any feature development.

Introducing and maintaining fitness functions inside the automation strategy will offer several benefits for helping the team provide a fast feedback loop on architecture characteristics and helping the company achieve better code quality standards.

Micro-Frontend-Specific Operations

Some automation pipelines for micro-frontends may require additional steps compared to traditional frontend automation pipelines. The first one worth mentioning would be checking that every micro-frontend is integrating specific libraries flagged as mandatory for every frontend artifact by the architecture team. Let's assume that we have developed a design system and we want to ensure that all our artifacts must contain the latest major version. In the CI pipeline, we should take a step to verify the `package.json` file, making sure the design system library contains the right version. If it doesn't, it should notify the team or even block the build, failing the process.

The same approach may be feasible for other internal libraries that we want to make sure are present in every micro-frontend, like analytics and observability. Considering the modular nature of micro-frontends, this additional step is highly recommended—no matter the architecture style we decide to embrace in this paradigm—to guarantee the integrity of our artifacts across the entire organization.

Another interesting approach, mainly available for vertical-split architecture, is the possibility of a server-side render at compile time instead of runtime when a user requests the page. The main reason for doing this is saving computation resources and costs, such as when we have to merge data and user interfaces that don't change very often. Another reason is to provide a highly optimized and fast-loading page with inline CSS and maybe even some JavaScript.

When our micro-frontend artifact results in an SPA with an HTML page as the entry point, we can generate a page skeleton with minimal CSS and HTML nodes inlined to suggest how a page would look, providing immediate feedback to the user while we are loading the rest of the resources needed for interacting with micro-frontends. This isn't

an extensive list of reasons that an organization may want to evaluate for micro-frontends, because every organization has its own gotchas and requirements. However, these are all valuable approaches that are worth thinking about when we are designing an automation pipeline.

Observability

The last important part to take into consideration in a successful micro-frontend architecture is the observability of our micro-frontends. Moreover, observability closes the feedback loop when our code runs in a production environment; otherwise, we would not be able to react quickly to any incidents happening during prime time. In the last few years, many observability tools have started to appear for the frontend ecosystem, such as Sentry, New Relic, and LogRocket. These tools allow us to identify the user journey before encountering a bug that may or may not prevent the user from completing the action. Observability is a must-have feature; nowadays, it should be part of any release strategy and is even more important when we are implementing distributed architectures such as micro-frontends.

Every micro-frontend should report errors—custom and generic—for providing visibility when a live issue happens. In that regard, Sentry, New Relic, or LogRocket can help by providing the visibility needed. In fact, these tools retrieve the user journey, collect the JavaScript stack trace of an exception, and cluster errors into groups. We can configure alerts for every type of error or warning in these tools' dashboards and even integrate them with alerting systems like PagerDuty.

It's very important to think about observability at a very early stage of the process because it plays a fundamental role in closing the feedback loop for developers, especially when dealing with multiple micro-frontends that compose the same view. These tools will help us to debug and understand which part of our codebase is affected, and quickly guide a team toward resolution by providing user context information such as browser, operating system, country, and so on. All this information, in combination with the stack trace, provides a clear investigation path for any developer to resolve the problem without spending hours trying to reproduce a bug in the developer's machine or in a testing environment.

Summary

We've covered a lot of ground here, so a recap is in order. First, we defined the principles we want to achieve with automation pipelines, focusing on fast feedback and constant review based on the evolution of both tech and the company. Then we talked about DX. If we cannot provide a frictionless experience, developers may try to game the system or use it only when strictly necessary, reducing the benefits of a well-designed CI/CD pipeline. Next, we discussed implementing the automation strategy, including all the best practices, such as unit, integration, and end-to-end testing; bundle-size checks; fitness functions; and many others that can be implemented in our automation strategy to guide developers toward the right level of software quality.

Automation is a very interesting topic, especially when we are implementing microarchitectures. There are plenty of additional topics we could cover, but if you include these practices in your automation strategy, you will be in really great shape, and you can always extend and evolve based on business and technical needs. The main takeaway is that automation is not a one-off action but an iterative process that has to be reviewed and improved with the life cycle of a product.

In the next chapter, we will discuss micro-frontend deployment and discoverability, another key aspect of de-risking deployments by multiple independent teams in distributed systems.

Chapter 7. Discover and Deploy Micro-Frontends

The last stage of any automation strategy is the delivery of the artifacts created during the build phase. Whether we decide to deploy our code via continuous deployment, a shell script running on premises, in a cloud provider, or via a user interface, understanding how we can deploy micro-frontends independently from each other is fundamental.

By their nature, micro-frontends must be independent. The moment we have to coordinate a deployment with multiple micro-frontends, we should question the decisions that we made in identifying their boundaries. Coupling risks can jeopardize the entire effort of embracing this architecture, generating more issues than value for the company. With microarchitectures, we deploy only a small portion of code without impacting the entire codebase. As with micro-frontends and microservices, we may decide to move forward to avoid the possibility of breaking the application and, therefore, the user experience. We'll present the new version of a micro-frontend to a smaller group of users instead of doing a big-bang release to all our users. For this scope, the microservices world uses techniques like blue-green deployment and canary releases, where a portion of the traffic is redirected to a new microservice. Adapting these key techniques for any micro-frontend deployment strategy is worth considering.

Blue-Green Deployment Versus Canary Releases

Blue-green deployment starts with the assumption that the last stage of our tests should be done in the production environment that we are running for the rest of our platform. After deploying a new version, we can test our new code in production without redirecting users to the new version, while still getting all the benefits of testing in the production environment. When all the tests pass, we are ready to redirect 100% of our traffic to the new version of our micro-frontend.

This strategy reduces the risk of deploying new micro-frontends, because we can perform all the necessary testing without impacting our user base.

Another benefit of this approach is that we may decide to provision only two environments: testing and production. Considering that all the tests are running in

production with a safe approach, we’re cutting infrastructure costs without having to support the staging environment. As you can see in [Figure 7-1](#), we have a router that should aim for shaping the traffic toward the right version.

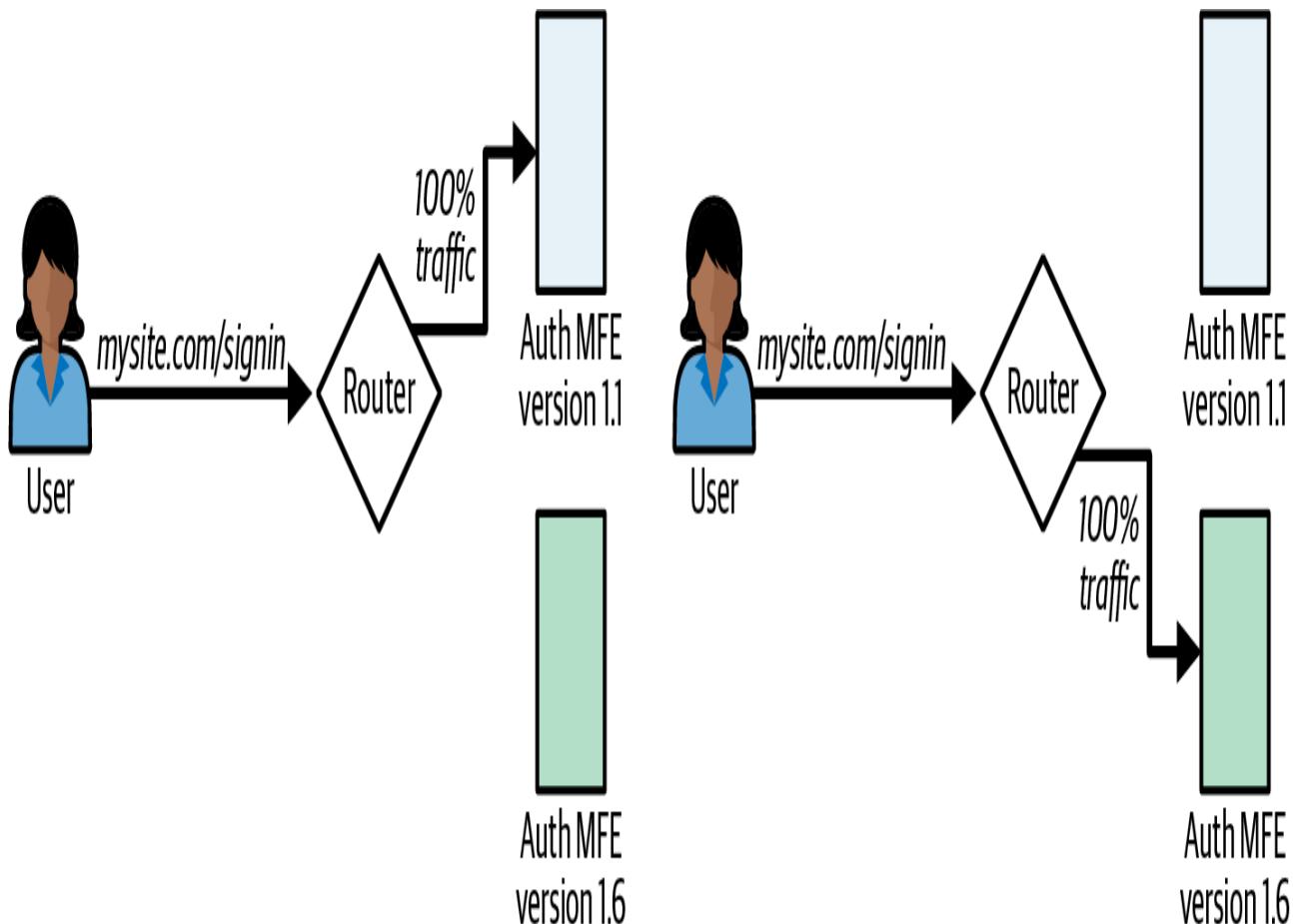


Figure 7-1. Blue-green deployment

In canary releases, we don’t switch all the traffic to a new version after all tests pass; instead, we gradually ease the traffic to a new micro-frontend version. As we monitor the metrics from the live traffic consuming our new frontend—such as increased error rates or less user engagement—we may decide to increase or decrease the traffic accordingly (see [Figure 7-2](#)).

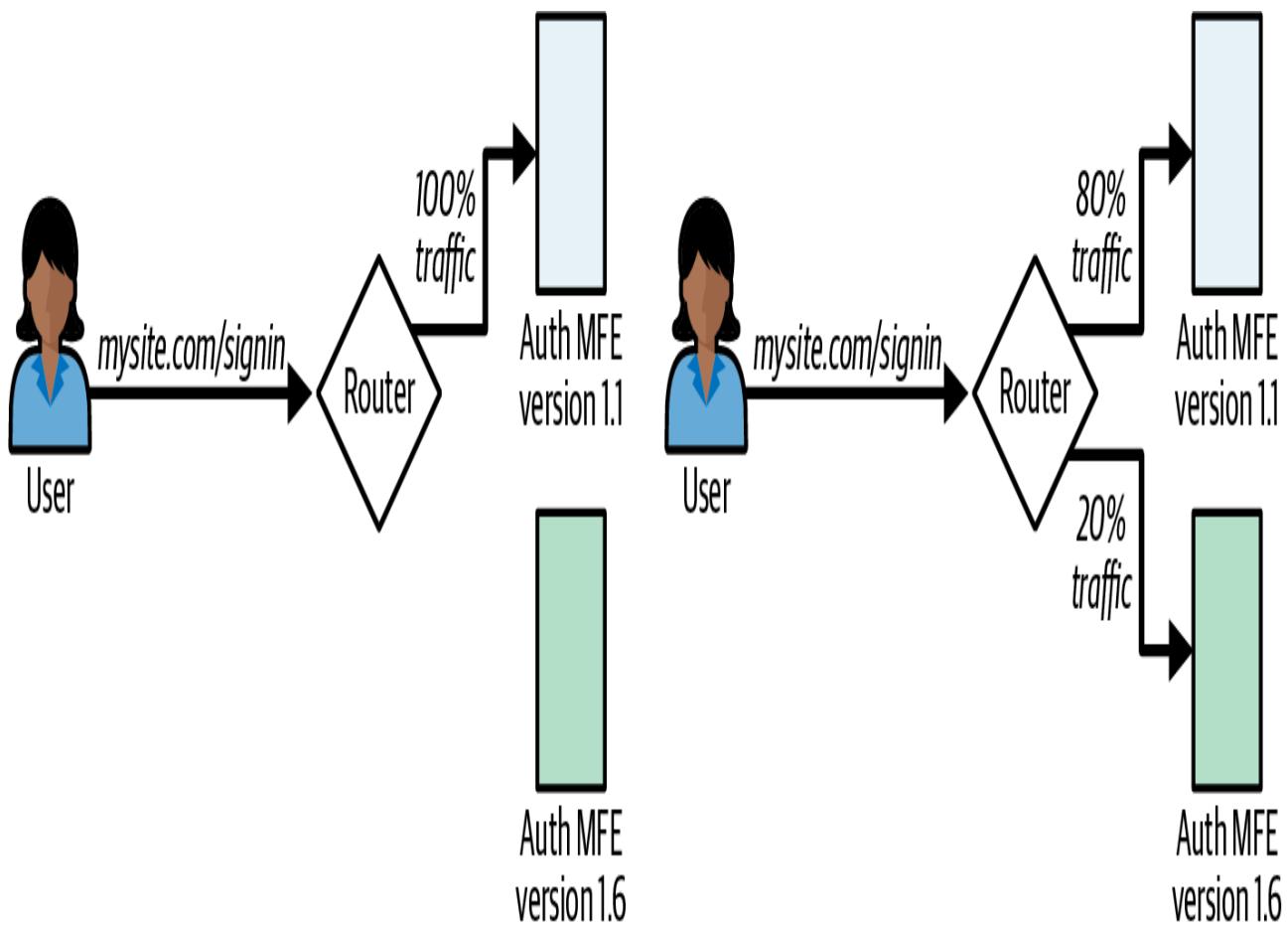


Figure 7-2. Canary release

In both approaches, we need a router that shapes the traffic (for a canary release) or switches the traffic from one version to another (for blue-green deployment). The router could be some logic handled on the client side, server side, or edge side, depending on the architecture chosen.

But there is another dimension to consider when we talk about the deployment of micro-frontends: the multiple environments that every company uses these days.

You're likely already familiar with the concept of the multi-environment strategy. It's a common practice that most companies utilize, typically involving environments like development, testing, staging (or preproduction), and production. Sometimes, you might even deal with more than three environments when you work in large companies.

This approach allows teams to vet changes thoroughly before they reach end users, ensuring the quality and stability of their applications. At the same time, this strategy adds complexity that we need to handle for distributed systems.

When it comes to micro-frontends, this multi-environment approach introduces a unique set of challenges that add layers of complexity to an already intricate architecture.

The Problem Space

Imagine that you're working on an ecommerce platform with separate micro-frontends for product listings, shopping cart, and check-out. Each of these independent components might be at different stages of development, requiring deployment to different environments. The product listing might be ready for production, while the shopping cart is still in testing, and the check-out is in staging.

This scenario raises critical questions:

- How do you ensure that the right version of each micro-frontend is loaded in the correct environment?
- How do you make sure you're loading a stable version of a micro-frontend that is not owned by your team while testing the latest functionality inside your micro-frontend?
- Can you test the latest API created by another team in an ephemeral environment while loading more stable versions of other micro-frontends?

In this scenario, your system needs to be smart enough to load the production version of the product listing, the test version of the shopping cart, and the staging version of the check-out—all within the same application.

The challenge becomes even more pronounced when multiple micro-frontends need to coexist on the same view in a horizontal split. For instance, your product page might need to display the “product listing” micro-frontend, owned by your team, and the related “product recommendations” micro-frontend—each being a separate micro-frontend at different stages of development.

This situation can lead to several issues:

Consolidated strategy

Designing a solid and replicable strategy to load micro-frontends across multiple environments represents a significant challenge for many companies due to the complexity and distributed ownership of the various components involved. Creating a self-service system for

each team adds another layer of intricacy to this already complex task.

Integration complexity

It becomes a significant challenge to keep ensuring that you load the latest version of the micro-frontends not owned by your team.

Testing challenges

Comprehensive testing becomes more complex, as you need to ensure that different versions of micro-frontends work together correctly in various environment combinations.

Moreover, perhaps the most significant challenge in the micro-frontend discoverability space is the lack of standardization. Each company tends to build its own custom solution for discovering micro-frontends across environments. This leads to a fragmented ecosystem, where knowledge and best practices aren't easily shared or transferred.

Imagine you're a developer moving from one company to another. At your previous job, micro-frontends were discovered using a centralized registry service. In your new role, you find that micro-frontends are loaded based on environment variables set at build time. This lack of consistency across the industry makes it difficult to establish common patterns and tools.

Addressing these challenges requires careful planning and implementation of strategies for version management and dependency isolation. It also necessitates a robust system for discovering and loading the correct versions of micro-frontends based on the current environment and user context.

Welcome to the Frontend Discovery Schema

As an expert in micro-frontends and a passionate advocate for software architecture, I analyzed the micro-frontend landscape and realized that we could improve not only environment management but also deployment risk mitigation. After all, micro-frontends form a distributed system—much like microservices—allowing us to leverage successful backend architecture practices. This realization came after years of experience working with hundreds of teams worldwide, helping them migrate to micro-frontend architectures.

In my search for a solution, I looked into various patterns that could address this challenge, and I discovered the *service discovery pattern*. This pattern is commonly used in microservices architectures, and it provided the perfect match to overcome the discoverability issues in micro-frontends.

The service discovery pattern is a crucial component in distributed systems, particularly in microservice architectures. It uses a centralized server, often called a “service registry,” to maintain a global view of service network locations. In this pattern, services register themselves with the registry, providing information about their network location (IP address and port). Clients can then query the registry to discover the locations of services that they need to communicate with. The diagram in [Figure 7-3](#) illustrates an example.

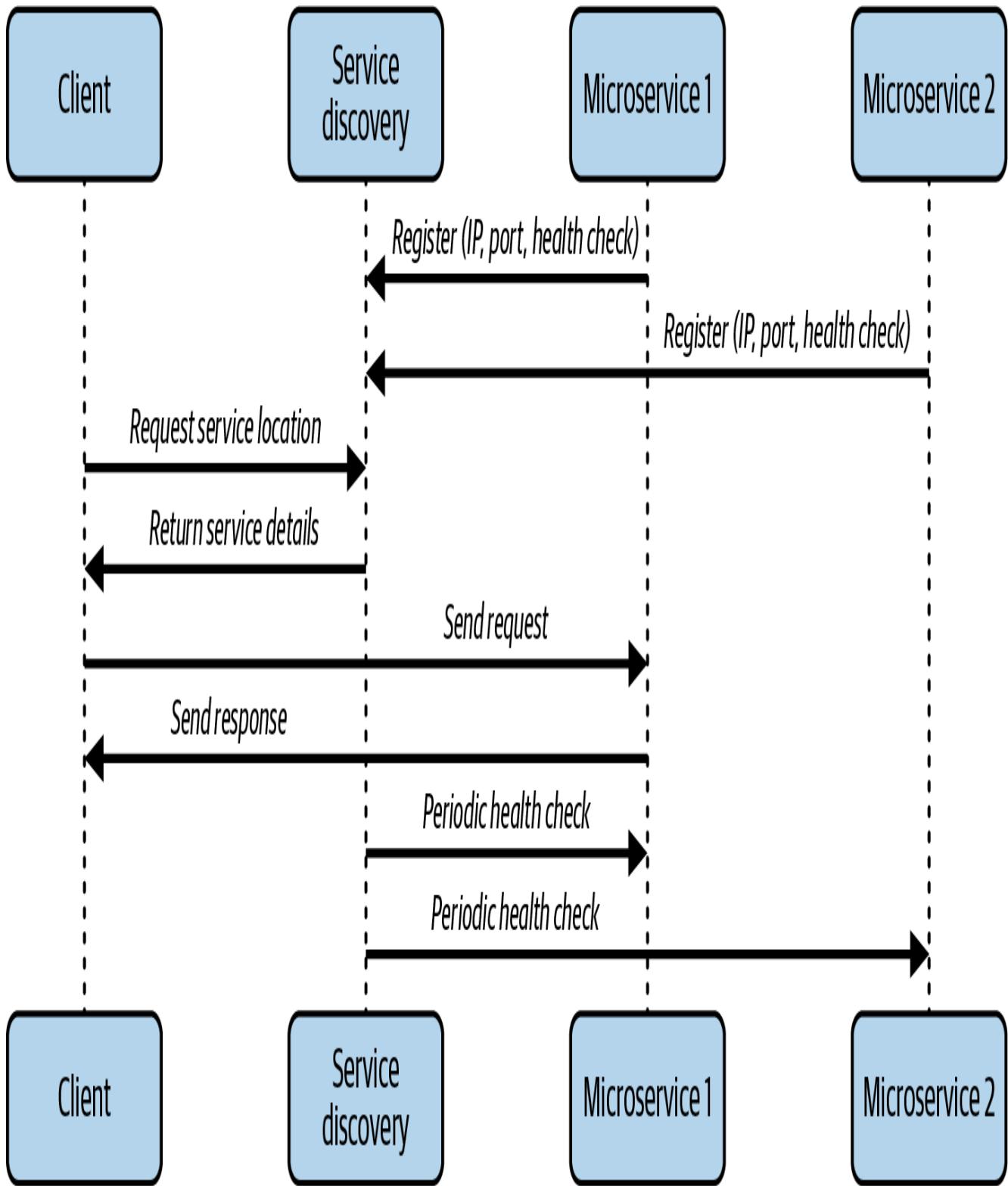


Figure 7-3. Microservices service discovery sequence

This sequence diagram shows the basic flow of a service discovery working with microservices:

1. Microservices register themselves with the service discovery.
2. A client requests the service location from the service discovery.

3. The service discovery returns the service details.
4. The client can then directly communicate with the microservice.
5. The service discovery performs periodic health checks on the registered microservices.

This pattern offers several key benefits:

Dynamic updates

Services can update their locations in real-time, allowing for flexible scaling and deployment.

Load balancing

The pattern can facilitate client-side or server-side load balancing among multiple instances of a service.

Fault tolerance

By maintaining a list of available services, the system can quickly adapt to service failures or network issues.

Decoupling

Service consumers are decoupled from the specific network locations of service providers, providing more flexible and resilient systems.

Recognizing the parallels between microservices and micro-frontends, I saw an opportunity to adapt this pattern to solve the discoverability challenges in micro-frontend architectures—and not only this problem! This insight led me to envision a system that would not only enable the retrieval of micro-frontend entry points but also improve the deployment process with mechanisms such as canary releases or blue-green deployments.

Recognizing the importance of the challenge and the potential impact of a solution, I knew I couldn't tackle this alone. In January 2022, I reached out to key figures in the micro-frontend community:

- Zack Jackson, the creator of Module Federation
- Joel Denning, the creator of single-spa

- Matteo Figus, one of the pioneers of micro-frontends and part of the OpenComponents core team

Together, we formed a working group dedicated to addressing the discoverability challenge in micro-frontends. Over the course of eight months, our group of experts engaged in intense discussions and brainstorming sessions. Despite the challenges of coordinating across different time zones—with Matteo and I in London, and the others on the West Coast of the United States—we persevered.

Initially, we considered creating a plug-in or writing specific code. However, as discussions progressed, we realized the need for a more universal solution—something bulletproof that would cover not only today’s libraries and frameworks but also those of the future. I proposed the idea of creating a standard that could help everyone, regardless of the specific frameworks they were using. This led to the conception of a JSON schema—a language-agnostic format that could serve as a common ground for micro-frontend discoverability.

The result of our collaborative efforts was the Frontend Discovery schema. This JSON-based schema provides a standardized way to describe micro-frontends and their deployment strategies:

```
{
  "schema": "https://mfewg.org/schema/v1-pre.json",
  "microFrontends": [
    "@my-project/catalog": [
      {
        "url": "https://static.website.com/my-catalog-1.3.5.js",
        "fallbackUrl": "https://alt-cdn.com/my-catalog-1.3.5.js",
        "metadata": {
          "integrity": "e0d123e5f316bef78bfd5a008837577",
          "version": "1.3.5"
        },
        "extras": {
          "modulefederation": {
            "prefetch": ["/dependencies/chunk1.js"]
          }
        }
      }
    ],
    "@my-project/myaccount": [
      {
        "url": "https://static.website.com/my-account-1.2.2.js",
        "fallbackUrl": "https://alt-cdn.com/my-account-1.2.2.js",
        "metadata": {
          "integrity": "e0d123e5f316bef78bfd5a008837577",
          "version": "1.2.2"
        }
      }
    ]
  ]
}
```

```
        },
        "deployment": {
            "traffic": 30,
            "default": false
        }
    },
    [
        {
            "url": "https://static.website.com/my-account-2.0.0.js",
            "fallbackUrl": "https://alt-cdn.com/my-account-2.0.0.js",
            "metadata": {
                "integrity": "e0d123e5f316bef78bfd5a008837577",
                "version": "2.0.0"
            },
            "deployment": {
                "traffic": 70,
                "default": true
            }
        }
    ]
}
```

The key features of the schema include:

Unique identifiers

Each micro-frontend is given a unique identifier within the schema.

URL information

The schema includes the URL where each micro-frontend can be fetched.

Fallback URLs

For improved reliability, fallback URLs can be specified.

Metadata

Additional information about each micro-frontend, such as version and integrity hashes, can be included.

Deployment strategies

The schema supports specifying deployment strategies, such as traffic splitting for canary releases.

Extensibility

An “extras” field allows for framework-specific or company-specific extensions to the schema.

With this schema, every micro-frontend can have one or more versions associated with the same micro-frontend identifier. In this way, you can iteratively release newer versions and apply your preferred logic to roll out the new version in the production environment.

The Benefits of This Pattern

The Frontend Discovery pattern offers several key advantages. It is versatile, functioning on both frontend and backend systems. This flexibility stems from its use of JSON, a widely adopted format in modern development. The approach is not overly opinionated, making it easy to implement across various systems. What I realized is that not every enterprise can change its process entirely or even specific points of its automation pipelines. Therefore, we aimed to create a solution that could be easily integrated into any system, regardless of whether it uses cutting-edge pipelines or an on-premises solution that might be older but still effectively serves its original purpose. Our goal was to develop a flexible approach that would work across a wide range of environments, from modern cloud-based infrastructures to legacy systems. This solution’s extensibility is a significant benefit, with the “extras” field allowing for custom additions to meet specific needs.

Importantly, it can be integrated seamlessly with existing APIs, eliminating the need for additional calls to retrieve micro-frontend information, but just adding information for the application shell to act upon them. While the approach does require fetching the configuration before loading any content, and it may be considered somewhat loose due to its lack of strict enforcement, it provides a solid foundation for managing micro-frontends. Clearly, for security and performance reasons, backend implementation is highly recommended.

Real-World Implementation

At this point, the real question is how we can accelerate the adoption of this solution, or at least provide a solution that can easily be tweaked to implement in every system.

First, let’s analyze how a potential implementation would work in practice. Let’s assume we are building a client-side micro-frontend application with an application

shell that loads the micro-frontends by leveraging a frontend service discovery from an API exposed by a microservice. The interaction would look like the sequence diagram in [Figure 7-4](#).

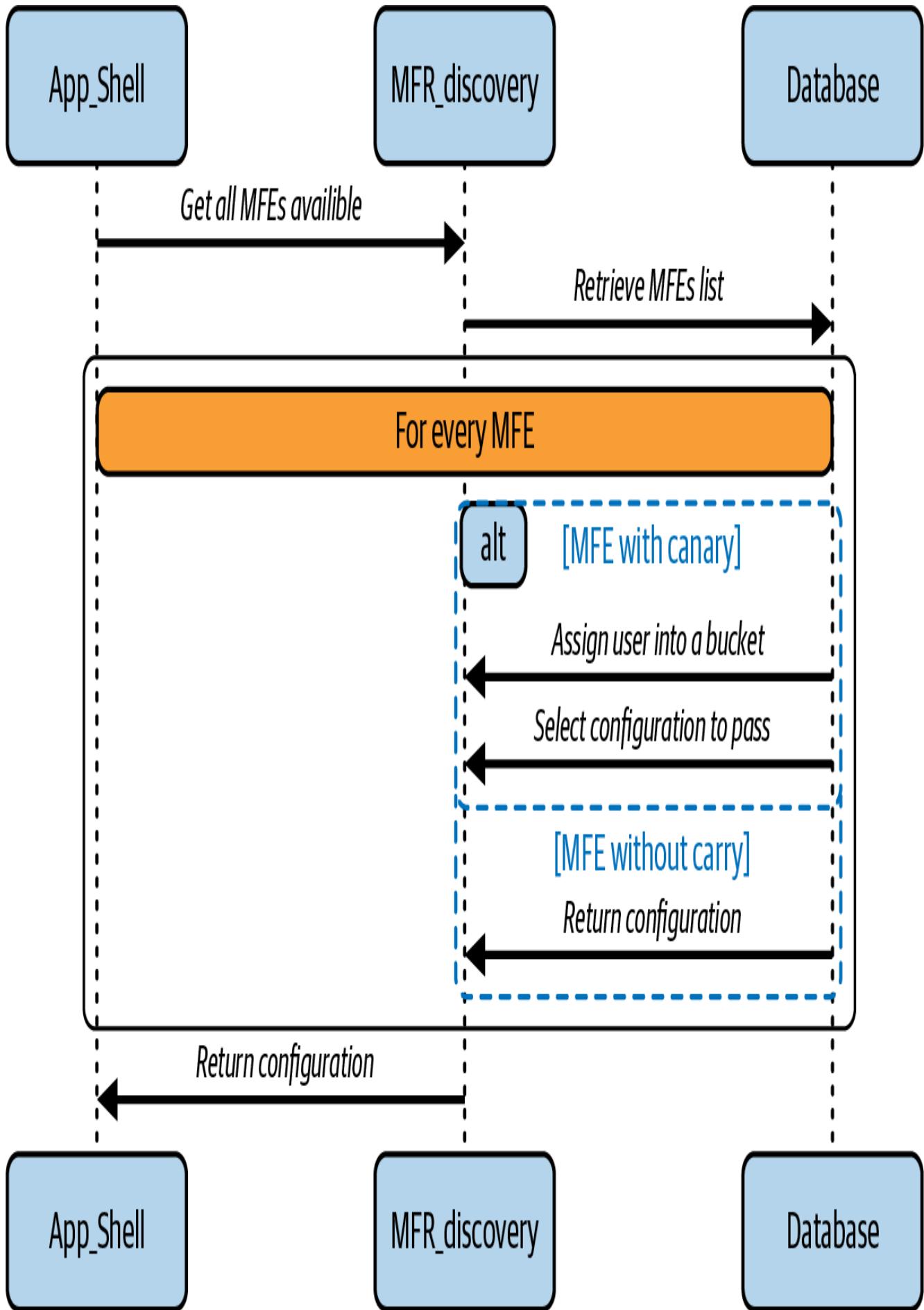


Figure 7-4. Integration of a service discovery inside a micro-frontend client-side application

The discovery mechanism has three main actors: the `App_Shell` acts as the client, the `MFE_discovery` service as the orchestrator, and the DB as the configuration store. When a user initiates the application, the `App_Shell` sends a comprehensive request to fetch all available micro-frontends, triggering a cascade of configuration resolution steps.

The resolution strategies in micro-frontend architecture encompass two approaches for handling configuration delivery. The canary resolution or blue-green deployment pathways implement a multistep process that begins with precise user segmentation through bucket assignment. When a user accesses the application, they are deterministically assigned to a specific bucket based on criteria such as user ID or other identifying factors. Once bucketed, the system evaluates the appropriate configuration based on predefined bucket criteria, which may include exposure percentages or user characteristics. The process continues with dynamic version mapping that enables gradual rollouts of new features or versions. This mapping ensures that different user segments receive different versions of the micro-frontend, allowing for controlled deployment and risk mitigation.

The standard resolution pathway provides a more straightforward approach for micro-frontends that don't require progressive deployment strategies. This process involves direct configuration retrieval from the database, where a single, stable version is served to all users. The system implements static version mapping, ensuring consistency across all user sessions. This approach maintains a default feature set application, where all users receive the same feature configuration without variation. This method is particularly useful for stable, well-tested micro-frontends that don't require experimental deployments or gradual rollouts.

An Integration Example with Module Federation

Let's expand the Module Federation ecommerce example we created in [Chapter 4](#) by implementing the service discovery. In this example, I'll use a static JSON file, but you can, of course, use an API to serve a dynamic list of micro-frontends for the client.

The first thing I created is a React custom hook for retrieving the micro-frontends available from the frontend discovery:

```
import { useState, useEffect } from 'react';
import { registerRemotes, init } from '@module-federation/runtime';
```

```
const useMfeInitialization = () => {
  const [routes, setRoutes] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    const initializeMFEs = async () => {
      try {
        await init({
          name: 'shell'
        });
        // fetch micro-frontends from the service discovery
        const response = await fetch('http://localhost:8080/discovery.json');
        const data = await response.json();

        const remotes = [];
        const routeConfigs = [];

        for (const [, configs] of Object.entries(data.microFrontends)) {
          const config = configs[0];
          const { name, alias, exposed, route } = config.extras;
          // register Module Federation remotes
          remotes.push({
            name,
            alias,
            entry: config.url
          });
          // create dynamic routes for micro-frontends
          routeConfigs.push({
            path: route,
            request: `${name}/${exposed}`
          });
        }
      }
      registerRemotes(remotes);
      setRoutes(routeConfigs);

    } catch (error) {
      console.error('MFE Error: micro-frontend configuration:', error);
    } finally {
      setIsLoading(false);
    }
  };

  initializeMFEs();
}, []);

return { routes, isLoading };
};

export default useMfeInitialization;
```

This code defines a custom React hook called `useMfeInitialization` that initializes and sets up micro-frontends in a Module Federation 2.0 architecture. Here's a breakdown of what the code does:

1. It starts by initializing the Module Federation runtime.
2. It fetches a configuration file (`frontend-discovery.json`) from a local server in this example, but it can be a remote endpoint when your application runs in any of your environments.
3. It processes the configuration data to:
 - a. Register remote modules for Module Federation
 - b. Create route configurations for each micro-frontend
4. It sets up these routes for use in the main application.
5. It handles loading states and errors during this process.

The hook returns two pieces of information:

`routes`

An array of route configurations for the micro-frontends

`isLoading`

A boolean, indicating whether the initialization process is complete

This setup allows the main application to dynamically load and route to different micro-frontends.

Let's now integrate the hook in the application shell:

```
import React, { lazy, Suspense } from 'react';
import { loadRemote } from '@module-federation/runtime';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import Header from './Header';
import Loading from './Loading';
import useMfeInitialization from './hooks/useMfeInitialization';

const System = ({ request }) => {
  if (!request) {
    return <h2>No system specified</h2>;
  }
}
```

```

const MFE = lazy(() => loadRemote(request));

return (
  <Suspense fallback="Loading...">
    <MFE />
  </Suspense>
);
};

const App = () => {
  const { routes, isLoading } = useMfeInitialization();

  return (
    <Router>
      <div>
        ...
      </div>
    </Router>
  );
};

export default App;

```

As you can see from this snippet, I created a custom React hook, `useMfeInitialization`, within the application shell:

- ➊ The App component uses the `useMfeInitialization` hook to manage the initialization of micro-frontends:

```
const { routes, isLoading } = useMfeInitialization();
```

- ➋ The hook returns two values: `routes` and `isLoading`.
- ➌ The component uses these values to conditionally render content:
 - If `isLoading` is true, it displays a loading component.
 - Otherwise, it renders the routes based on the `routes` array.
- ➍ The `routes` array is used to dynamically generate Route components:

```

{routes.map((route, index) => (
  <Route
    key={index}
    path={route.path}
    element={<System request={route.request} />}
))

```

```
/>  
))}
```

- ⑤ Each route renders a `System` component, which lazy-loads the corresponding micro-frontend.

This integration allows the app to dynamically set up routing for micro-frontends based on the initialization process handled by the custom hook, providing a flexible and modular structure for the application. Moreover, the integration is hidden inside a custom React hook that makes it easier to reuse across projects if needed.

Integrating the Discovery Pattern with Other Solutions

Module Federation is a popular solution for building micro-frontends, but it's not the only option. As we discussed in this book, import maps have become increasingly popular over the years, especially in the micro-frontend community.

Single-spa and Native Federation leverage these web standards for loading remote micro-frontends and their related dependencies, maintaining a mental model similar to that of Module Federation.

However, while Module Federation offers native integration of dynamic modules, import maps are treated as DOM elements. Therefore, if we don't use any library, we can integrate the discovery pattern with a bit more code.

An implementation example could be as follows:

```
async generateImportsMap() {  
  const originalScript = document.getElementById("map");  
  const theMap = JSON.parse(originalScript.textContent);  
  
  let id, i;  
  for (i in this.imports) {  
    id = `@${this.imports[i].id}`;  
    theMap.imports[id] = this.imports[i].url;  
  }  
  
  const script = document.createElement("script");  
  script.type = "systemjs-importmap";  
  script.textContent = JSON.stringify(theMap);
```

```
document.head.appendChild(script);
}
```

The `generateImportsMap()` function dynamically creates and updates an import map. Here's how it works in relation to import maps:

1. It starts by retrieving an existing import map from a script element with the ID `map`.
2. The function then iterates through a collection of imports, likely defined elsewhere in the code. For each import:
 - a. It creates a new key in the import map using the format `@{id}`.
 - b. It sets the value of this key to the URL of the import.
3. After processing all imports, the function creates a new `<script>` element with the type `systemjs-importmap`.
4. The updated import map is stringified and set as the content of this new script element.
5. Finally, the new script element is appended to the document's head, effectively updating the SystemJS import map at runtime.

More or less every framework or library using import maps for loading micro-frontends provides a similar approach to abstract this capability. Single-spa has a utility library called `import-map-injector`, while Native Federation for Angular provides an API very similar to Module Federation that leverages import maps. In both cases, you can generate the list of modules dynamically. Therefore, after fetching the list of micro-frontends and related modules to load, you can add them dynamically to the DOM using your preferred micro-frontend approach.

What about Feature Flags?

By now, you might be wondering why we are not using feature flags instead of building this approach.

The micro-frontend discovery approach illustrated so far demonstrates an architecture that complements, rather than conflicts with, feature-flag systems. This synergy creates a powerful system for managing both deployment and feature delivery.

The service discovery's bucket-assignment mechanism naturally aligns with feature-flag implementations. While canary deployments control which version of a micro-frontend users receive, feature flags can simultaneously determine which features are active within that version. This layered approach provides granular control over both deployment and functionality.

Traditional feature-flag implementations might create a web of complexity within the codebase. Consider a real-world ecommerce scenario where a team wants to introduce a new check-out experience. With feature flags, developers would need to implement conditional logic across multiple components: the cart page, payment processing, order confirmation, and potentially user account sections. Each micro-frontend would require flag coordination, resulting in scattered conditional statements that must be maintained and eventually removed—and without taking into account other existing feature flags present in the system.

With a micro-frontend architecture, we have clear boundaries that help to scope feature flags within what a team controls, rather than across multiple areas of a system. This simplifies the implementation and elimination of feature flags without requiring coordination across multiple teams.

Using the service discovery approach, the same ecommerce enhancement becomes more manageable. Instead of implementing feature flags across multiple components, the team deploys a complete version of the check-out micro-frontend with the new experience. The service discovery then directs 20% of users to this new version while maintaining the original version for others.

If an issue arises, such as an unexpected payment-processing error, the team can immediately redirect traffic back to the stable version through the service discovery configuration. This is significantly faster and safer than coordinating the disabling of multiple feature flags across different micro-frontends.

The key advantage becomes apparent when considering the cleanup phase:

With feature flags

Developers must carefully remove all conditional logic, flag checks, and alternative implementations across multiple codebases. And we all know that, oftentimes, this is a constraint.

With service discovery

Once the new version is proven successful, simply increasing the traffic allocation to 100% completes the rollout, with no code cleanup required.

This approach particularly shines in complex scenarios where features have cross-cutting concerns across multiple micro-frontends, as it eliminates the need for orchestrating feature-flag states across different teams and codebases. Moreover, with this approach, we reduce the number of feature flags implemented in the system because smaller features can be easily tested just using a canary release or blue-green deployment without causing issues on the application stability.

Nevertheless, a hybrid approach combining micro-frontend service discovery with selective feature-flag usage presents a sophisticated solution for modern web architectures. Let's examine it in more detail.

Teams can reserve feature flags for genuinely necessary runtime configurations inside a single or multiple micro-frontends that the team owns, while using the service discovery for broader deployment strategies. This separation of concerns allows organizations to maintain fine-grained control over feature availability while simplifying the overall deployment and rollout process. The result is a more robust system that can adapt to changing requirements while maintaining code quality and reducing or better managing technical debt.

Consider the previous example of the ecommerce platform implementing a new check-out experience. In a traditional feature-flag approach, developers would need to implement flags across multiple micro-frontends: "cart," "payment," "order confirmation," and "user account." Each component would require conditional logic, creating a complex web of feature flags.

Using the service discovery, the implementation becomes more streamlined. The cart micro-frontend deploys two versions:

Version A

Current check-out flow

Version B

New check-out experience

The service discovery manages user distribution:

- 80% of users receive Version A
- 20% of users receive Version B through canary deployment

Feature flags are reserved only for critical runtime toggles within the new version, such as:

- Payment processor failover options
- Promotional discount logic
- New streamlined check-out with less steps to reach the payment

Instead of managing multiple feature flags across four different micro-frontends, teams only maintain feature flags for essential runtime configurations while letting the service discovery handle the broader deployment strategy. When the new check-out experience proves successful, transitioning to 100% deployment requires no code—just a simple configuration change in the service discovery.

The result is a cleaner codebase, simpler testing requirements, and more manageable deployments while maintaining flexibility to quickly respond to critical runtime needs through strategic feature-flag usage.

While it might seem unnecessary, there are some situations where I see the benefits of using both approaches.

Availability in the Market

As of 2025, there are several solutions in the market that are implementing the service discovery pattern for micro-frontends. These include an open source project by AWS, Piral Cloud, and Zephyr Cloud.

AWS has developed an open source project called [Frontend Service Discovery on AWS](#). This solution helps handle frontend releases using the frontend discovery pattern. It allows developers to deploy micro-frontends using canary releases, which can improve availability and operations. The project provides an Admin API for managing micro-frontend releases and a Consumer API that applications can use to fetch the list of micro-frontends and their metadata dynamically. Because of this API-based approach, this solution is pluggable into any existing automation pipeline, as it doesn't require changes to your current workflow. This approach enables individual micro-frontends to be discoverable by other previously deployed micro-frontends orchestrated by a shell

app. It is framework-agnostic; it can work with client-side, server-side, or even edge-side micro-frontends; and it works with frontend architectures as well.

Piral is another solution that provides a service discovery out of the box. The framework simplifies the management of micro-frontends through its opinionated platform. Through several features available in Piral, teams are encouraged to focus more on building micro-frontends rather than spending excessive time on defining the governance of a micro-frontend platform. The framework allows any service discovery to be used—even though it advocates its generic service, Piral Cloud, which can be installed on-premises. With feature flags, vulnerability scanning, CSS conflict detection, multi-environment support, and much more, it is a comprehensive solution that goes beyond the Piral frontend framework.

Zephyr Cloud is a cloud-agnostic and framework-agnostic platform that supports micro-frontends and Module Federation. Zephyr Cloud allows developers to manage dependencies for micro-frontends and provides visualization tools to discover relationships between them. It also enables the publishing of environments and modules independently.

Summary

In this chapter, we explored the challenges and solutions for managing micro-frontends across multiple environments and examined how to de-risk the deployment of new versions in a production environment.

The key takeaway is the introduction of the Frontend Discovery schema, a standardized approach to describe and deploy micro-frontends. This schema helps solve common problems like version management, integration complexity, and testing challenges in distributed frontend architectures. In the chapter, we also discussed the benefits of using a service discovery pattern, which allows for dynamic updates, load balancing, and improved fault tolerance.

We explored how to introduce the discovery pattern within an existing application shell using Module Federation, which can be applied similarly with other frameworks or libraries. The process involves consuming an endpoint and transforming the received JSON into routes to load the right micro-frontend. We provided examples of implementing this pattern using Module Federation and React hooks, demonstrating how to create a flexible and modular frontend system.

Overall, the Frontend Discovery schema offers a powerful approach for teams to manage and scale their micro-frontend architectures more effectively.

Chapter 8. Automation Pipeline for Micro-Frontends: A Case Study

Now that we've discussed the theory of a micro-frontend automation pipeline, let's review a fictional use case example, including the different steps that should be taken into consideration based on the topics we have covered. Keep in mind that not all the steps or the configuration described in this example have to be present in every automation strategy, because companies and projects are different.

Setting the Scene

ACME Inc., a video-streaming service, empowers its developers and trusts them to know better than anyone else in the organization which tools they should use for building the micro-frontends needed for the project. Every team is responsible for setting up its micro-frontend build, so the developers are encouraged to choose the tools that meet the technical requirements of their micro-frontends and to operate within the boundaries, or guardrails, defined by the company.

The company uses a custom cloud automation pipeline with Docker containers, with the cloud team providing the tools required to run these pipelines. This Docker-based approach ensures environmental parity across local development, continuous integration (CI), and production—an essential factor for maintaining consistency and reliability across distributed micro-frontends.

The project is structured using micro-frontends with a vertical-split architecture, where each micro-frontend is technically represented by an HTML page, a JavaScript file, and a CSS file. Every development team in the organization works with unit, integration, and end-to-end testing—a decision made by the tech leaders and the head of engineering to ensure the quality and reliability of code deployed in production.

The architecture team, which is the bridge between product and engineers, requests using fitness functions within the pipeline to ensure the artifacts delivered in the production environment contain the architecture characteristics they desire. The team is responsible for translating product people's business requirements into technical requirements that the developers can implement.

The development teams decide to use a monorepo strategy, so all the micro-frontends will be present in the same repository. The team will use trunk-based development for its branching strategy and release directly from the main branch instead of creating a release branch.

The project won't use feature flags. The team decides to defer this decision to reduce the number of moving parts, so manual and automated testing will be performed in existing environments already created by the DX team.

Finally, for bug fixing, the teams will use a fix-forward strategy, where they fix bugs directly in the trunk branch and then deploy. The company's environment strategy consists of three environments: development (DEV), staging (STAGE), and production (PROD), as shown in [Figure 8-1](#).

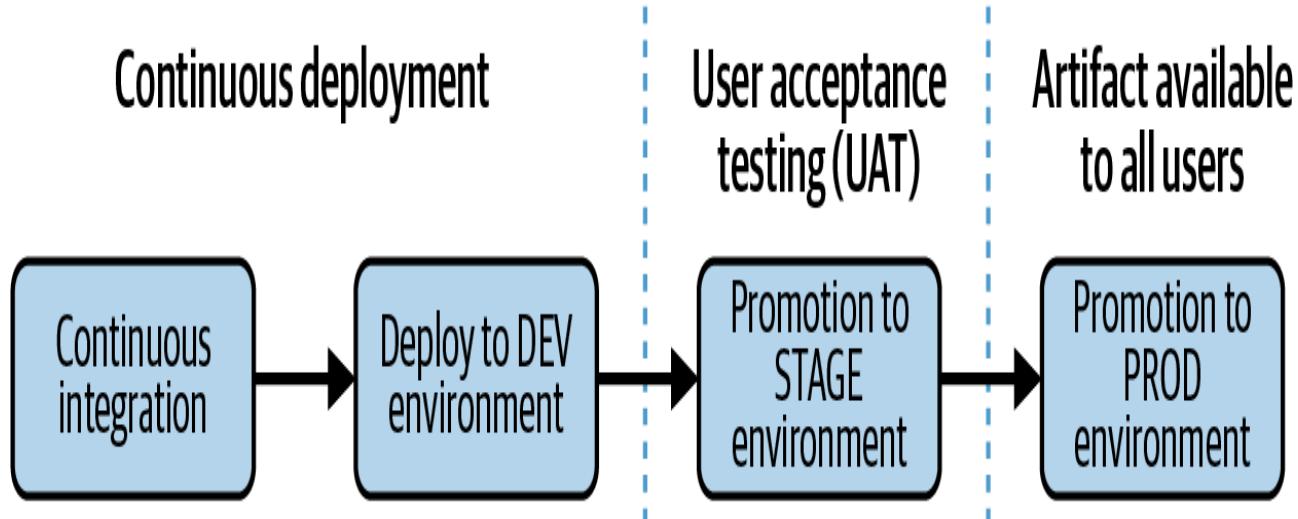


Figure 8-1. An example of an environment strategy

The DEV environment is in continuous deployment so that the developers can see the results of their implementations as quickly as possible. When a team feels ready to move to the next step, it can promote the artifact to user acceptance testing (UAT). At this stage, the UAT team will make sure the artifact respects all the business requirements before promoting the artifact to production, where it will be consumed by the end user. Based on all this, [Figure 8-2](#) illustrates the automation strategy for our use case project up to the DEV environment. It's specifically designed for delivering the micro-frontends at the desired quality.

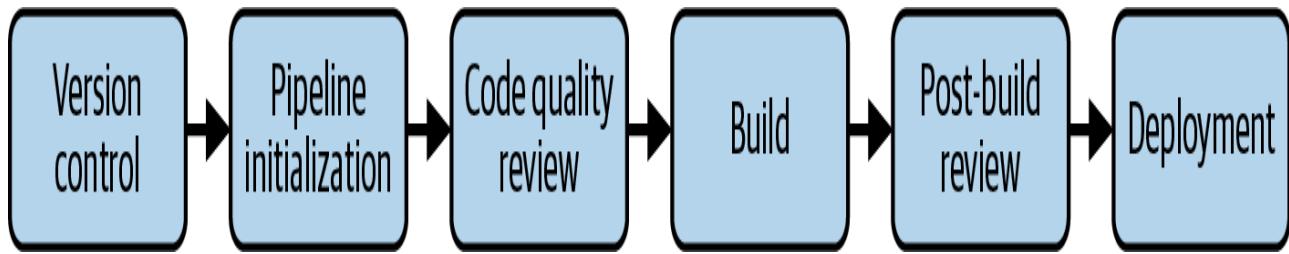


Figure 8-2. High-level automation strategy design

A dashboard built in-house will promote artifacts across environments. In this way, the developers and quality assurance have full control of the different steps for reviewing an artifact before it is presented to users. Such an automation strategy will create a constant, fast feedback loop for the developers, catching potential issues as soon as possible during the CI phase instead of further down the line and making bug fixing as cheap as possible.

DEFECT COSTS RISE OVER TIME

Remember, the cost of detecting and fixing defects in software increases exponentially over time in the software development workflow. That's because when a developer is working on a feature, the code developed is fresh in their mind; a code change is fairly trivial. When a developer catches bugs in production, however, months may have passed since the developer worked on that code. In the meantime, the developer will have worked on several other projects or features, so remembering the team's entire logic and approach will take time. Finding bugs in production will cost you more than just time; it hurts the company's credibility and costs more money than just investing in a fast feedback loop at the beginning. The National Institute of Standards and Technology estimates the **cost of fixing bugs in production** to be 25 times more expensive than catching them during the development phase.

The automation strategy in this project is composed of six key areas, within which there are multiple steps:

1. Version control
2. Pipeline initialization
3. Code-quality review
4. Build
5. Post-build review
6. Deployment

Let's explore these areas in detail.

Version Control

The project will use a monorepo for version control, so the developers decided to use [Lerna](#), which enables them to manage all the different micro-frontend dependencies at the same time. Lerna also allows for hosting all the shared modules across projects in the same `node_modules` folder in the root directory, so that if a developer has to work on multiple projects, they can download a resource for multiple micro-frontends just once. Dependencies will be shared, so a unique bundle can be downloaded once by a user and will have a high time-to-live (TTL) time at the content delivery network (CDN) level. Considering the vendors aren't changing as often as the application's business logic, we'll avoid an increase in traffic to the origin.

ACME Inc. uses GitHub as a version control system, partially because there are always interesting automation opportunities in a cloud-based system like GitHub. In fact, [GitHub has a marketplace](#) with many scripts available to be run at different branching life cycles. For instance, we may want to apply linting rules at every commit or when someone opens a pull request. We can also decide to run our own scripts if we have particular tasks to apply in our codebase during the opening of a pull request, like scanning the code to avoid any library secrets being presented or for other security reasons.

In addition, the core team decided to configure Dependabot for managing shared libraries across micro-frontends. Dependabot is an automated dependency-management tool that plays a vital role in maintaining the health and security of JavaScript applications, particularly within micro-frontend architectures. It streamlines the process of keeping dependencies up to date across multiple repositories or modules by regularly scanning package files to identify outdated or vulnerable dependencies. When it detects any issues, Dependabot automatically creates pull requests to update these dependencies, significantly reducing the manual effort required by developers. This automation not only enhances the security posture of the application by addressing vulnerabilities promptly but also allows development teams to focus on building features rather than getting bogged down in the tedious task of manual package updates.

In a micro-frontend architecture, Dependabot is especially valuable for managing shared dependencies. For instance, when a team responsible for a design system releases updates, Dependabot can automatically generate pull requests in all micro-frontends that utilize this shared library. This eliminates the need for manual

notifications and coordination between teams, allowing for a smoother workflow. Similarly, Dependabot can apply automated updates to other shared libraries—such as analytics or observability tools—ensuring that all micro-frontends benefit from the latest versions.

Pipeline Initialization

The pipeline initialization stage includes several common actions to perform for every micro-frontend, including:

- Cloning the micro-frontend repository inside a container
- Installing all the dependencies needed for the following steps

In [Figure 8-3](#), we can see the first part of ACME Inc.’s automation pipeline where the developers perform two key actions: cloning the micro-frontend repository and installing the dependencies via the `yarn` or `npm` command, depending on each team’s preference.

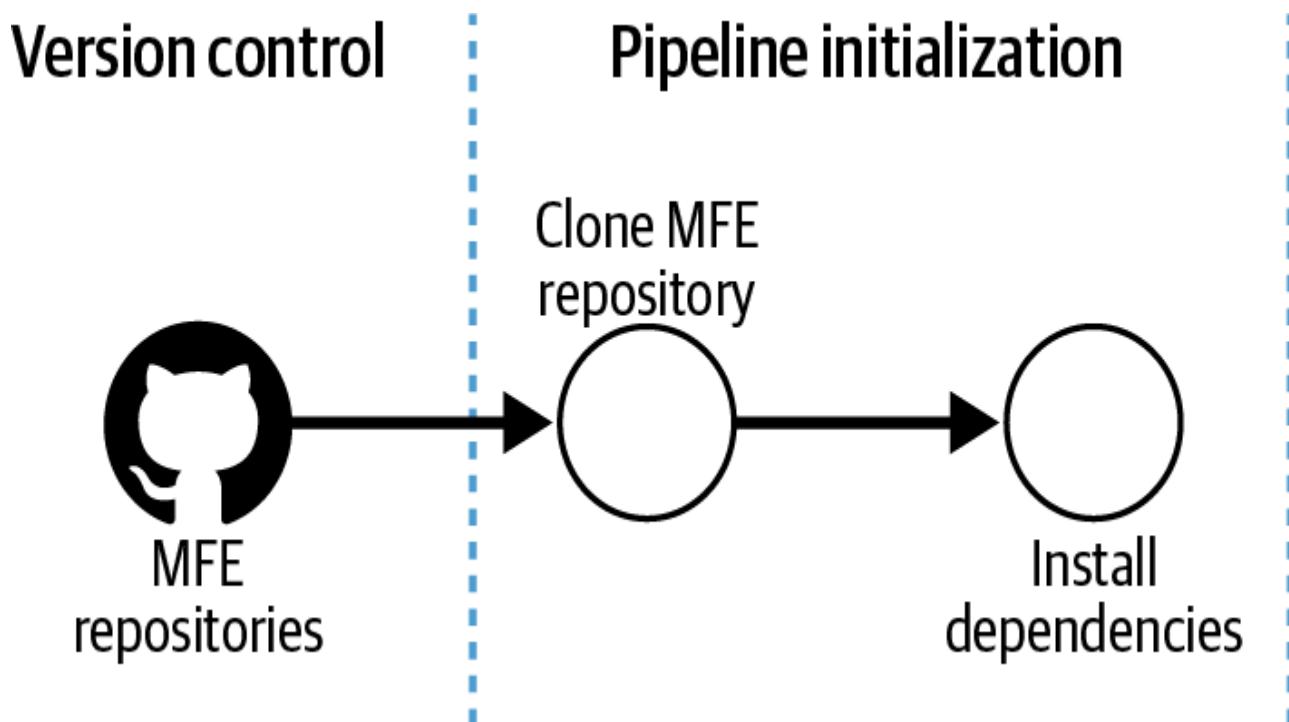


Figure 8-3. Pipeline initialization stage, showing two actions: cloning the repository and installing the dependencies

The most important thing that the developers need to remember is to make the repository cloning as fast as possible. They don't need the entire repository history for a CI process, so it's good practice to use the command `depth` for retrieving just the last commit, especially when using a monorepo approach, considering the repository may grow in size very quickly. The cloning operation will speed up when they are dealing with repositories with years of history tracked in the version control system:

```
git clone --depth [depth] [remote-url]
```

An example would be:

```
git clone --depth 1 https://github.com/account/repository
```

Code-Quality Review

During this phase, the developers are performing all the checks to make sure the code implemented respects the company standards. [Figure 8-4](#) shows several stages, from static analysis to visual tests. For this project, the company decided not only to cover unit and integration testing but also to ensure that the code was maintainable in the long term, the user interface integration respects the design guidelines from the UX team, and the common libraries developed are present inside the micro-frontends and respect the minimum implementations.

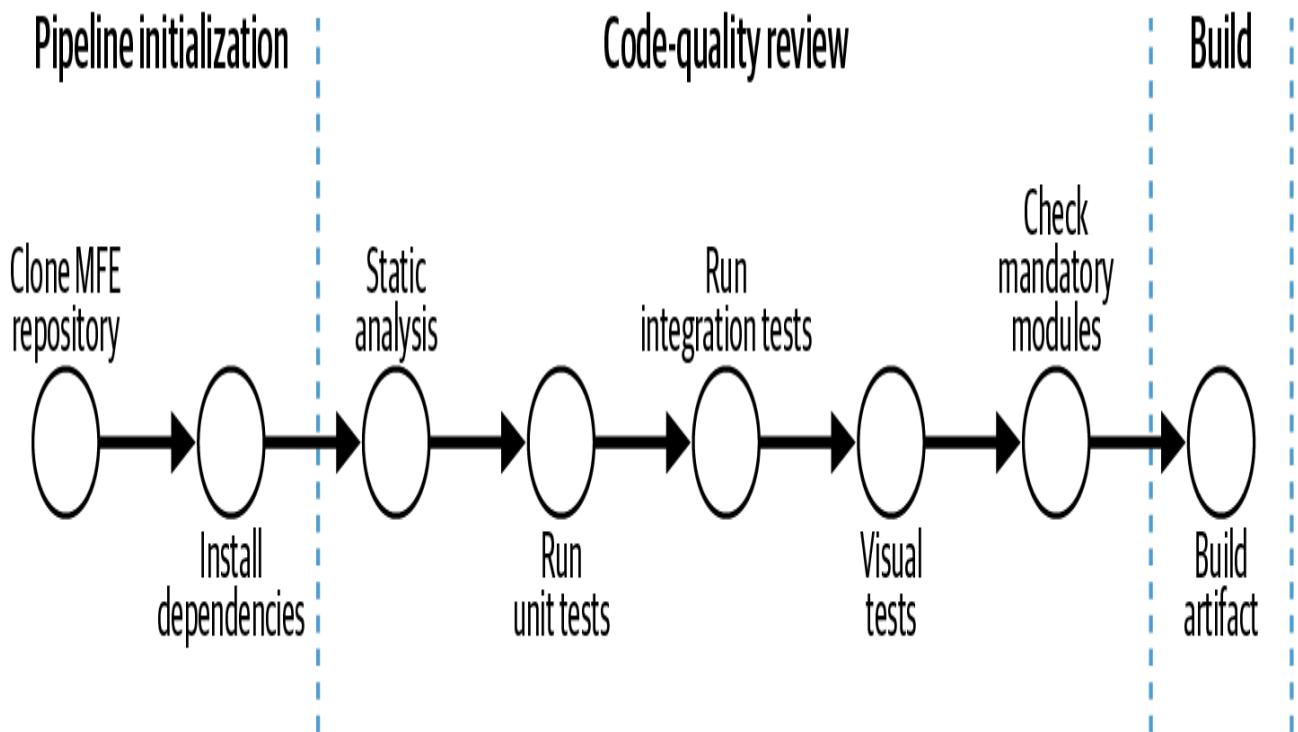


Figure 8-4. Code-quality checks like unit testing, static analysis, and visual regression tests

For static analysis, ACME Inc. uses [SonarQube](#) with the JavaScript plug-in. SonarQube is a tool for static analysis, and it retrieves many metrics, including cyclomatic complexity (CYC), which tech leaders and architects who aren't working every day in the codebase need in order to understand the code quality produced by a team. Often underestimated, CYC can provide a lot of useful information about how healthy your project is. It provides a score on the code complexity based on the number of branches inside every function, which is an objective way to understand if the micro-frontend is simple to read but harder to maintain in the long run.

Let's consider this example:

```

const myFunc = (someValue) =>{
    // variable definitions

    if(someValue === "1234-5678"){ //CYC: 1 - first branch
        // do something
    } else if(someValue === "9876-5432"){ //CYC: 2 - second branch
        // do something else
    } else { //CYC: 3 - third branch
        // default case
    }
}

```

```
// return something  
}
```

This function has a CYC score of 3, which means we will need at least three unit tests for this function. It may also indicate that the logic managed inside the function starts to become complex and harder to maintain.

By comparison, a CYC score of 10 means a function definitely requires some refactoring and simplification; we want to keep our CYC score as low as possible so that any change to the code will be easier, both for us and for other developers inside or outside our team.

Unit and integration testing are becoming more important every day, and the tools for JavaScript are becoming better. Developers, as well as their companies, must recognize the importance of automated testing before deploying in production. With micro-frontends, we should invest in these practices, mainly because the area to test per team is far smaller than a normal SPA, and the related complexity should be lower. Also, considering the size of the business logic, testing micro-frontends should be very quick. There aren't any excuses for avoiding this step.

ACME Inc. decided to use **Jest** for unit and integration testing, which is standard within the company. As there isn't a specific tool for testing micro-frontends, the company's standard tool will be fine for unit and integration tests.

The final step is specific to a micro-frontend architecture: checking on the implementation of specific libraries—like logging or observability—across all the micro-frontends inside a project, and making sure the design of your system fits the implementation. When we develop a micro-frontend application, there are some parts we want to write once and use in all our micro-frontends. A check on the shared libraries present in every micro-frontend will help enforce these controls, making sure that all the micro-frontends respect the company's guidelines and we aren't reinventing the wheel. Controlling the presence inside the `package.json` file present in every JavaScript project is a simple way to do this; however, we can go a step further by implementing more complex reviews and analysis on the implementation.

It's very important to customize an automation pipeline, introducing these kinds of fitness functions to ensure the architectural decisions are respected despite the distributed nature of this architecture. Moreover, with micro-frontends—where sharing code across them may require much more coordination than in a monolithic codebase—these kinds of steps are fundamental for having a positive end result.

This is where architecture testing tools like `ts-arch` come into play. `ts-arch` is an architectural unit test framework for TypeScript and JavaScript projects, designed to help you codify and automatically enforce architectural conventions. With `ts-arch`, you can write tests to ensure, for example, that the shared library is the only folder used in a monorepo by micro-frontends, or that there are no direct dependencies between individual micro-frontends. This kind of testing is especially valuable in distributed systems like micro-frontends, where independent teams might otherwise accidentally introduce unwanted coupling or bypass agreed boundaries.

By integrating tools like `ts-arch` into your workflow, you'll gain confidence that your micro-frontend boundaries remain clean, your shared code is used as intended, and your overall architecture stays healthy as your teams and codebase grow.

Build

The artifact is created during the build stage. For this project, the teams are using `Webpack` to perform any code optimizations (like minification and dead code elimination). Micro-frontends allow us to use different tools for building our code; in fact, it may be normal to use Webpack for building and optimizing certain micro-frontends and using another tool for others. The important thing to remember is to provide freedom to the teams within certain boundaries.

If you have any particular requirements that should be applied at build time, raise them with the teams and make sure that when a new tool is introduced inside the build phase—and generally inside the automation pipeline—it has the capabilities required for maintaining the boundaries. Introducing a new build tool is not a problem per se, because we can experiment and compare the results from the teams. We may even discover new capabilities and techniques we wouldn't have found otherwise. However, we don't *have* to use different tools. It's perfectly fine if all the teams agree on a set of tools to use across the entire automation pipeline. But don't block innovation; sometimes we discover interesting results from an approach that is different from the one agreed to at the beginning of the project.

Post-Build Review

The post-build stage (shown in [Figure 8-5](#)) is the last opportunity for the developers to confirm that their artifact has all the performance characteristics and requirements ready to be deployed in production.

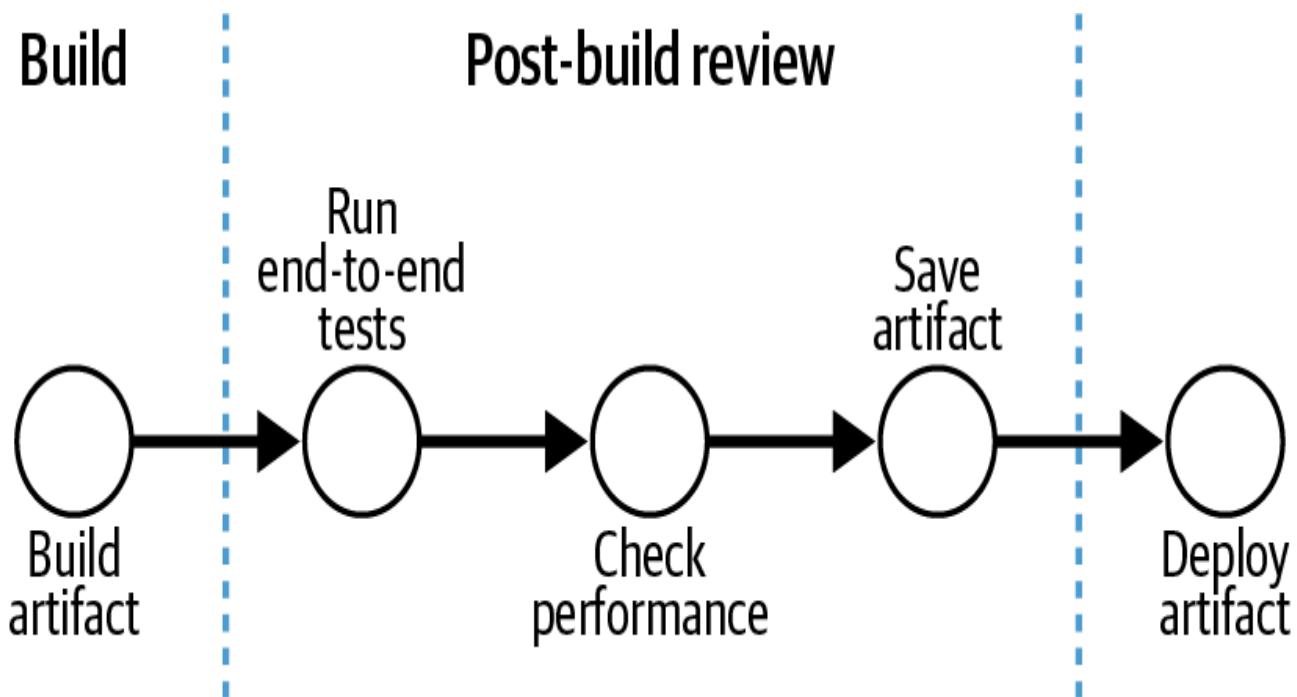


Figure 8-5. Additional checks performed in the post-build review before deploying an artifact to an environment

A key step is storing the artifact in an artifact repository, like Sonatype Nexus Repository or Jfrog Artifactory. You may also decide to use a simpler storage solution, like an AWS S3 bucket. The important thing is to have a unique source of truth where all your artifacts are stored.

ACME Inc. decided to introduce additional checks during this stage: end-to-end testing and performance reviews. Whether these two checks are performed at this stage depends on the automation strategy in place and the capability of the system. In this example, we are assuming that the company can spin up a static environment for running end-to-end testing and performance checks and then tear it down when these tests are completed.

End-to-end testing is critical for micro-frontends. In this case, where we have a vertical split and the entire user experience is inside the same artifact, testing the entire micro-frontend like we usually do for SPAs is natural. However, if we have multiple micro-frontends in the same view with a horizontal split, we should postpone end-to-end testing to a later stage so we can test the entire view.

When we cannot afford to create and maintain on-demand environments, we might use web servers that are proxying the parts not related to a micro-frontend. For instance, Webpack's dev server plug-in can be configured to fetch all the resources requested by

an application during end-to-end tests locally or remotely, specifying from which environment to pull the resources when not related to the build artifact. If a micro-frontend is used in multiple views, we should check whether the code will work end to end in every view the micro-frontend is used.

Although end-to-end testing is becoming more popular in frontend development, there are several schools of thought about when to perform the test. You may decide to test in production—as long as all the features needed to sustain testing in that environment are present. Therefore, be sure to include feature flags, potential mock data, and coordination when integrating with third parties to avoid unexpected and undesirable side effects.

Performance checks have become far easier to perform within an automation pipeline, thanks to command-line interface (CLI) tools that can be wrapped inside a Docker container and easily integrated into any automation pipeline. There are many alternatives, but I recommend starting with [Lighthouse CLI](#).

With one of these tools implemented in our automation strategy, we can make sure our artifact respects key metrics, like performance, accessibility, and best practices. Ideally, we should be able to gather these metrics for every artifact in order to compare them during the lifespan of the project. In this way, we can review the improvements and regressions of our micro-frontends and organize meetings with the tech leadership to analyze the results and determine potential improvements, creating a continuous learning environment inside our organization.

With these steps implemented, we make sure our micro-frontends deployed in production are functioning (through end-to-end testing) and performing as expected when the architectural characteristics are identified.

Deployment

The last step in our example is the deployment of a micro-frontend. An AWS S3 bucket will serve as the final platform for the user, and CloudFront will be the CDN. As a result, the CDN layer will take the traffic hit, and there won't be any scalability issues to take care of in production, regardless of the shape of user traffic hitting the web platform. An AWS Lambda—an event-driven, serverless computing platform provided by Amazon as part of AWS—will be triggered to decompress the `tar.gz` file present in the artifact repository. Then, the content will be deployed inside the dev environment bucket. Remember that the company built a deployment dashboard for promoting the

artifacts through different environments. In this case, for every promotion, the dashboard triggers an AWS Lambda for copying the files from one environment to another.

ACME Inc. decided to create a very simple infrastructure for hosting its micro-frontends, neatly avoiding additional investments initially to understand how to scale the additional infrastructure needed for serving micro-frontends. Obviously, this is not always the case. But I encourage you to find the cheapest, easiest way to host and maintain your micro-frontends. This approach reduces complexities in production and results in fewer moving parts that may fail.

To mitigate risks associated with large-scale deployments and potential bugs in new artifacts, the ACME Inc. teams implemented a micro-frontend service discovery pattern. This strategic move, as discussed in [Chapter 7](#), provides a robust mechanism for managing the rollout of new micro-frontends. After installing the [Frontend Discovery Service](#), the teams began utilizing its API to create, update, and delete micro-frontend deployments. This service is designed to increase traffic incrementally to new versions based on a predefined deployment strategy, typically over a specified time interval.

The gradual rollout facilitated by the Frontend Discovery Service allows for a controlled and monitored deployment process. Developers can observe their dashboards to quickly identify any issues arising from the new version in production. If problems are detected, the system enables rapid rollback capabilities, allowing traffic to be swiftly redirected to the previous stable version. Additionally, the teams implemented automated safety measures that trigger a traffic switch if error rates exceed predefined thresholds during the initial deployment hours. This creates a safety net for developers, encouraging them to deploy more frequently in production without the fear of widespread impact in case of issues.

The positive impact of this automation has also resonated with the technical leadership at ACME Inc. They have begun to reduce deployment gates, recognizing that this system inherently improves the quality of artifacts shipped to production. The ability to quickly identify and mitigate issues has led to a more agile and responsive development environment. By adopting this micro-frontend discovery pattern, ACME Inc. has not only enhanced its deployment safety but also fostered a culture of continuous improvement and frequent, low-risk releases. This approach aligns well with modern DevOps practices, emphasizing rapid iteration and feedback loops while maintaining high standards of reliability and performance.

Automation Strategy Summary

Every area of this automation strategy (shown in [Figure 8-6](#)) is composed of one or more steps designed to provide a feedback loop to the development teams for different aspects of the development process, using different testing strategies—such as unit testing, end-to-end testing, visual regression, bundle-size check, and many others.

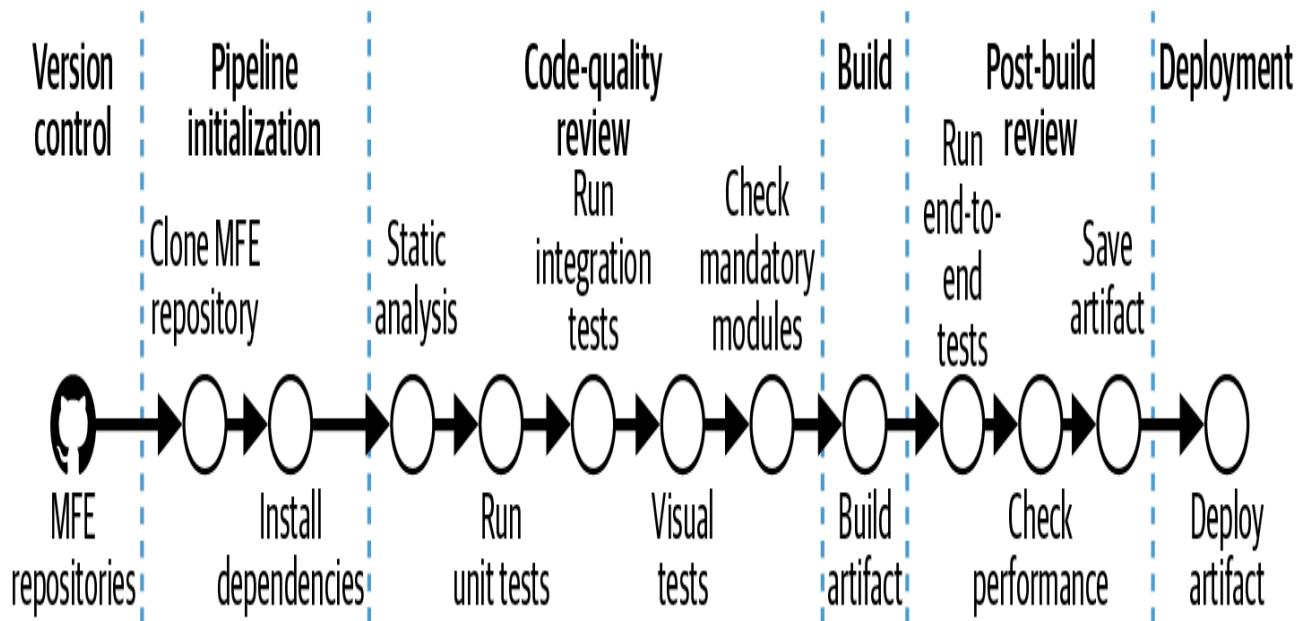


Figure 8-6. The end-to-end automation strategy

All of these controls create confidence in the delivery of high-quality content. This strategy also provides developers with a useful and constant reminder of the best practices leveraged inside the organization, guiding them to deliver what the business wants.

The automation strategy shared in this chapter is one of many that a company may decide to use. Different micro-frontend architectures will require additional or fewer steps than the ones described here. However, this automation strategy covers the main stages for ensuring a good result for a micro-frontend architecture.

Remember that the automation strategy evolves with the business and the architecture; therefore, after the first implementation, review it often with the development teams and the tech leadership. When automation serves the purpose of your micro-frontends well, implementation has a greater chance of being successful.

As we have seen, an automation strategy for micro-frontends doesn't differ too much from a traditional one used for an SPA. I recommend organizing some retrospectives every other month with architects, tech leaders, and representatives of every team to

review and enhance such an essential cog in the software development process. And since every micro-frontend should have its own pipeline, the DX team is perfectly positioned to automate the infrastructure configurations as much as possible in order to have a frictionless experience when new micro-frontends arise. Using containers enables a DX team to focus on the infrastructure, providing the boundaries needed for a team implementing its automation pipeline.

Summary

In this chapter, we've reviewed a possible automation strategy for micro-frontends, building on concepts introduced earlier in this book. Your organization may adopt some of these stages, but it's essential to regularly review the goals you want to achieve with your automation efforts. This is a critical success factor for micro-frontends. Avoid it, and you may jeopardize the entire project.

The distributed nature of micro-frontends demands a frictionless, continuously evolving automation pipeline. When a company begins to struggle with regular builds or deployments, that's often a clear signal that the automation strategy needs reassessment. Specifically, when teams face slow delivery, flaky deployments, or frequent rollback needs, the pipeline is often the primary leverage point for addressing these systemic inefficiencies.

Don't underestimate the importance of a robust automation strategy—it can shape the outcome of your entire initiative.

Chapter 9. Backend Patterns For Micro-Frontends

You may think that micro-frontends are only a possible architecture when you combine them with microservices, because we can have end-to-end technology autonomy. Maybe you're thinking that your monolith architecture would never support micro-frontends, or even that having a monolith on the API layer would mean mirroring the architecture on the frontend as well. However, that's not the case. There are several nuances to take into consideration, and micro-frontends can definitely be used in combination with microservices and monoliths.

In this chapter, we review some possible integrations between the frontend and backend layers. In particular, we analyze how micro-frontends can work in combination with a monolith, with microservices, and even with the backend for frontend (BFF) pattern.

Also, we will discuss the best patterns to integrate with different micro-frontend implementations, such as the vertical split, horizontal split with a client-side composition, and horizontal split with server-side composition.

Finally, we will explore how GraphQL can be a valid solution for micro-frontends as a single entry point for our APIs.

API Integration and Micro-Frontends

Let's start by defining the different API approaches we may use in a web application. As shown in [Figure 9-1](#), we will focus on the most used and well-known patterns.

This doesn't mean micro-frontends work only with these implementations. For example, you can choose the right approach for a WebSocket (a two-way computer communication protocol over a single TCP connection) or hypermedia. In the case of hypermedia—for example, using REST with hypermedia links in the response contents—the client consuming the API can dynamically navigate to the appropriate resources by traversing these links. The key is learning how to work with BFF, API gateway, or service dictionary patterns.

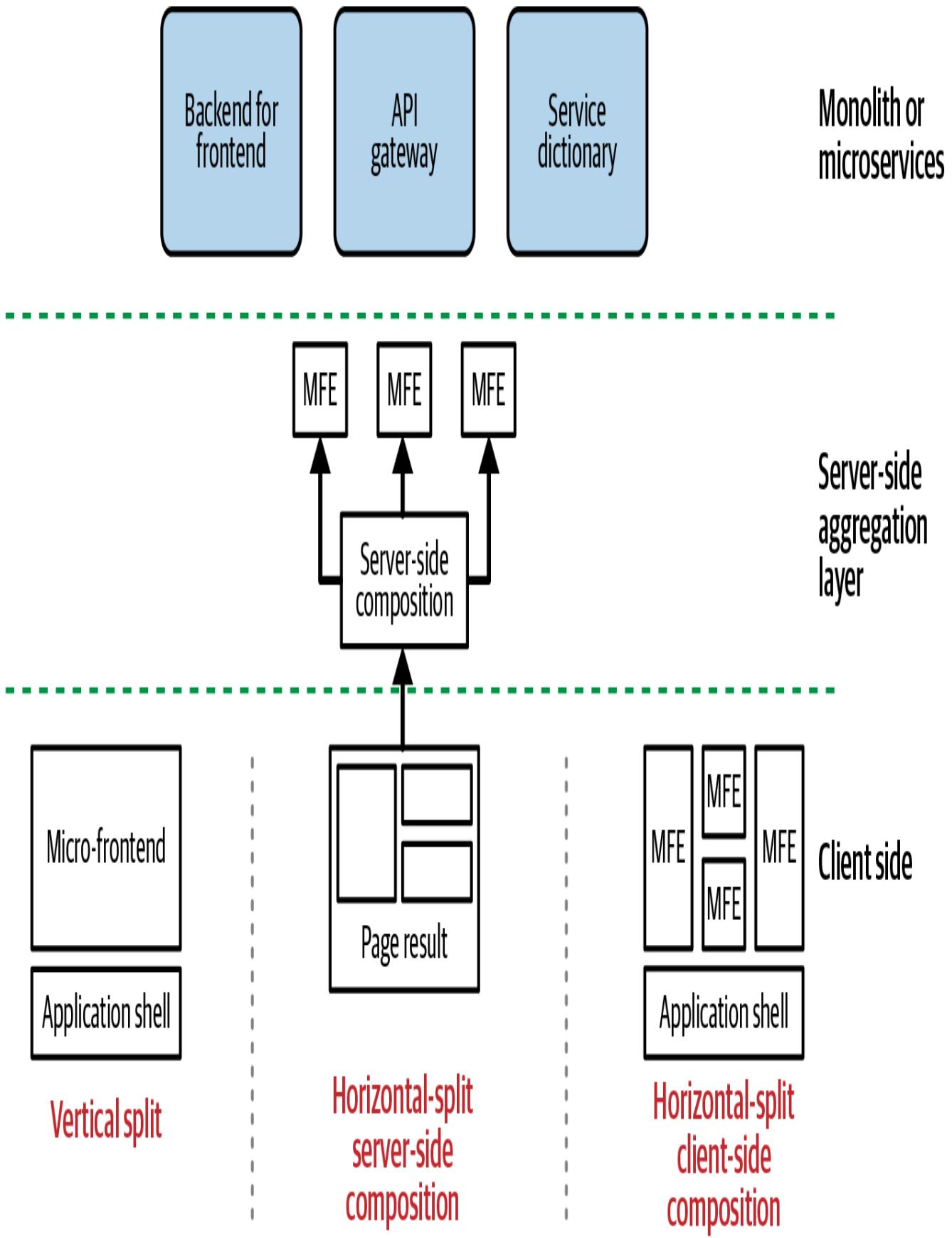


Figure 9-1. Micro-frontends and API layers

There are three patterns we will analyze in this chapter:

Service dictionary

The service dictionary is just a list of services available for the client to consume. It's used mainly when we are developing an API layer with a monolith or modular monolith architecture; however, it can also be implemented with a microservices architecture with an API gateway, among other architectures. A service dictionary avoids the need to create shared libraries, define environment variables, or inject configurations during the continuous integration (CI) process, and it eliminates the requirement to hardcode all the endpoints in the frontend codebase. The dictionary is loaded for the first time when the micro-frontend loads, allowing the client to retrieve the URLs for consumption directly from the service dictionary.

API gateway

Known well in the microservices community, an API gateway is a single entry point for a microservice architecture. The clients can consume the APIs developed inside microservices through one gateway. The API gateway also enables the centralization of a set of capabilities:

- Token validation, which entails validating the signature of a token before passing the request to a microservice
- Visibility and reporting, where we have a centralized means of verifying all inbound and outbound traffic
- Rate limiting, where the API Gateway rejects the request after exceeding a specific threshold—e.g., setting 100 requests per second as the limit from a client, so when the limit is exceeded, the API gateway returns errors instead of calling the microservice to fulfill the request

BFF

The BFF is an extension of the API gateway pattern, creating a single entry point per client type. For instance, we may have a BFF for the web application, another for mobile, and a third for the Internet of Things (IoT) devices we are commercializing. BFF reduces the

chattiness between client and server, aggregating the API responses and returning an easy data structure for the client to be parsed and rendered inside a user interface. This allows for a great degree of freedom to shape APIs dedicated to a client and reduces the round trips between a client and the backend layer.

These patterns are not mutually exclusive, either; they can be combined to work together.

An additional possibility worth mentioning is writing an API endpoint library for the client side. However, I discourage this practice with micro-frontends because we risk embedding an older library version in some of them and, therefore, the user interface may have some issues like outdated information or even API errors due to the dismissal of some APIs. Without strong governance and discipline around this library, we risk having certain micro-frontends using the wrong version of an API. It is far better to rely on the service discovery pattern or similar mechanisms that provide a list of end points at runtime.

Domain-driven design (DDD) also influences architecture and infrastructure decisions, especially in distributed, end-to-end systems. We can divide an application into multiple business domains, applying the most appropriate approach for each business domain.

This level of flexibility offers architects and developers a variety of choices that were not possible before. At the same time, however, we need to be careful not to fragment the client-server communication unnecessarily, and instead introduce a new pattern when it provides a tangible benefit for our application. A beneficial approach I've observed over the years is to start by developing independent solutions for each team and then gradually consolidating them into unified entry points.

As teams deploy multiple micro-frontends into production, the necessity to consolidate the API layer becomes apparent, and governance naturally emerges from practical experience. Platform teams that attempt to design everything up front run a higher risk of overcomplicating the entire process, creating friction, and slowing the flow that a distributed system requires—especially in the early stages of the journey.

Working with a Service Dictionary

A service dictionary is simply a list of endpoints available in the API layer provided to a micro-frontend. This allows the API to be consumed without the need to bake the endpoints within the client-side code, injecting them during a CI pipeline, or maintain them in a shared library.

Usually, a service dictionary is provided via a static JSON file or an API that should be consumed as the first request for a micro-frontend (in the case of a vertical-split architecture) or an application shell (in the case of a horizontal split).

A service dictionary may also be integrated into existing configuration files or APIs to reduce the round trips to the server and to optimize the client startup.

In this case, we can have a JSON object containing a list of configurations needed for our clients, where one of the elements is the service dictionary. An example of a service dictionary structure would be:

```
{
  "my_amazing_api": {
    "v1": "https://api.acme.com/v1/my_amazing_api",
    "v2": "https://api.acme.com/v2/my_amazing_api",
    "v3": ">https://api.acme.com/v3/my_amazing_api"
  },
  "my_super_awesome_api": {
    "v1": "https://api.acme.com/v1/my_super_awesome_api"
  }
}
```

As you can see, we are listing all the APIs supported by the backend. Thanks to API versioning, we can handle cross-platform applications without introducing breaking changes, because each client can use the API version that best suits its needs.

One thing we can't control in such scenarios is the presence of a new version on every mobile device. When we release a new version of a mobile application, updating may take several days, if not weeks or even longer.

Therefore, versioning the APIs is important to ensure we don't harm our user experience. Reviewing the cadence of when to dismiss an API version, then, is important. One of the main reasons is that potential attacks may harm our platform's stability.

Usually, when we upgrade an API to a new version, we are improving not only the business logic but also its security. But unless this change can be applicable to all the versions of a specific API, it would be better to assess whether the APIs are still valid for legitimate users, and then decide whether to discontinue the support of an API.

To create a frictionless experience for our users, implementing a forced upgrade in every application released via an executable—consider, for example, React Native applications—may be a solution. This would prevent the user from accessing older applications due to drastic updates in our APIs or even in our business model. Therefore, we must think about how to mitigate these scenarios in order to create a smooth user experience for our customers. Endpoint discoverability is another reason to use a service dictionary.

Not all companies work with cross-functional teams; many still work with component teams, with some teams fully responsible for the frontend of an application and others for the backend. Using a service dictionary enables every frontend team to be aware of what's happening in other teams. If a new version of an API is available or a brand-new API is exposed in the service dictionary, the frontend team will be aware.

This is also a valid argument for cross-functional teams when we develop a cross-functional application. In fact, it's very unlikely that a “two-pizza team” would have all the knowledge needed to develop web, backend, mobile (iOS and Android), and even smart TV and console applications, considering that many of these devices support HTML and JavaScript.

A TWO-PIZZA TEAM

According to [Jeff Bezos](#), chief executive officer of Amazon, if a team can't be fed with just two pizzas, it's too big. The introduction of the two-pizza rule at Amazon meant that each team should be no larger than eight or nine people—with two pizzas being enough to feed them!

The reasoning behind this rule isn't to save money on pizzas; it's based on the number of links between people within a team.

There is a formula for calculating the links between members in a group: $n(n-1)/2$, where n corresponds to the number of people. For instance, if a team has six people, there will be 15 links between members. Double the team to 12 members, and there will be 66 links. Complexity grows exponentially, not linearly, creating a higher risk of missing information across the team.

Using a service dictionary gives every team a list of available APIs in every environment, simply by checking the dictionary. We often assume the problem is just a communication issue that can be resolved with better communication. However, look again at the number of links in a 12-person team: forgetting to update a team about a new API version can happen frequently. A service dictionary also helps initiate discussions with the team responsible for the API, particularly in large organizations with distributed teams.

Last but not least, a service dictionary is valuable for testing micro-frontends with new endpoint versions in production. A company that uses a testing-in-production strategy can expand this to its micro-frontend architecture, thanks to the service dictionary, all without affecting the standard user experience.

We can test new endpoints in production by providing a specific header recognized by the service dictionary. The service interprets the header value and responds with a custom service dictionary used for testing new endpoints directly in production.

We would choose to use a header instead of a token or any other type of authentication, because it supports both authenticated and unauthenticated use cases. [Figure 9-2](#) presents a high-level design of what the implementation would look like.

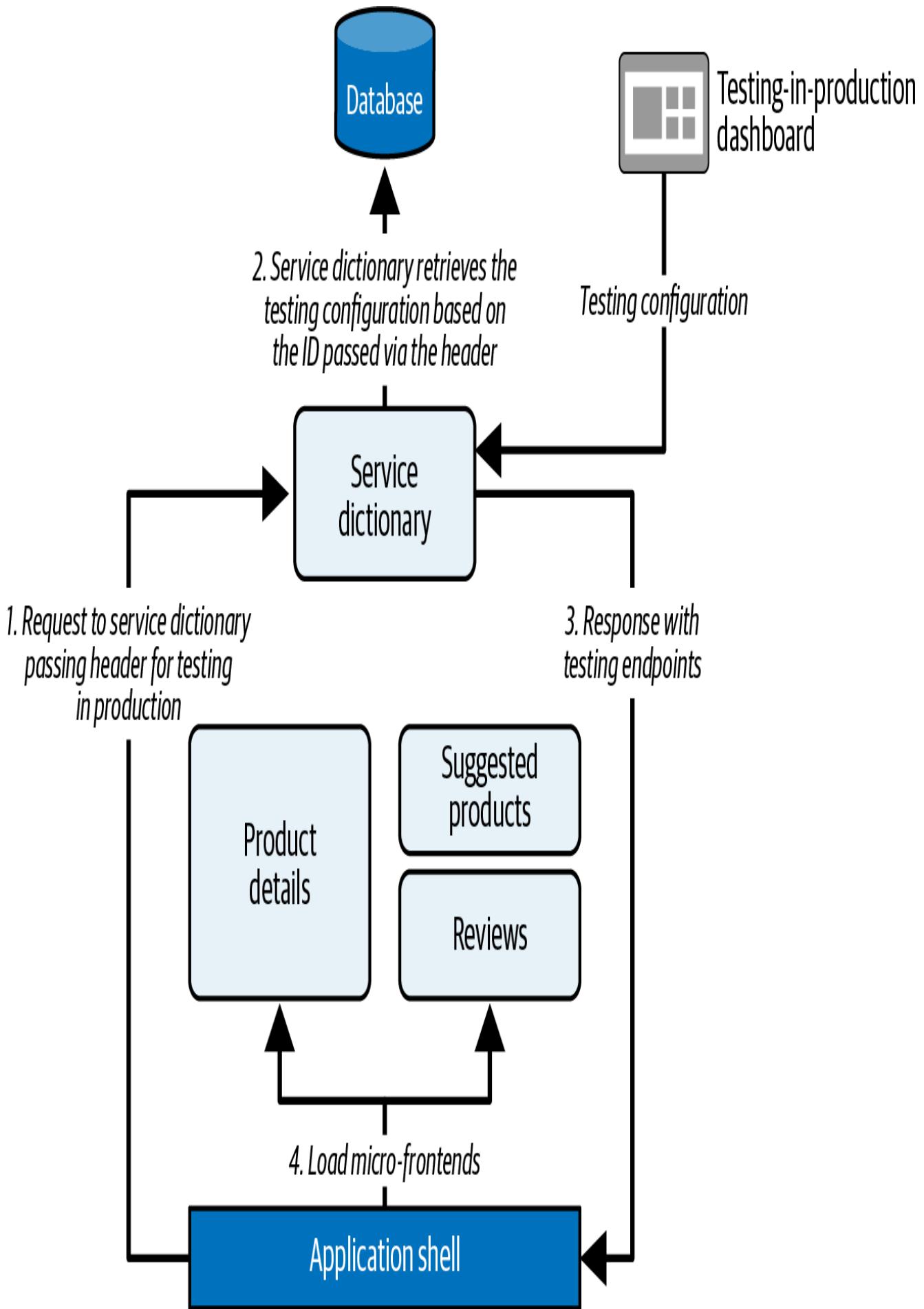


Figure 9-2. A high-level architecture on how to use a service dictionary for testing in production

In [Figure 9-2](#), we can see that the application shell consumes the service dictionary API as the first step. But this time, the application shell passes a header with an ID related to the configuration that needs to be loaded. In this example, the ID was generated at runtime by the application shell. When the service dictionary receives the call, it will check for a header in the request. If present, it will load the associated configuration from the database. It then returns the response to the application shell with the specific service dictionary requested. The application shell is now ready to load the micro-frontends to compose the page. Finally, the custom endpoint configuration associated with the client ID is produced via a dashboard (top-right corner of the diagram) used only by the company's employees. We may even extend this mechanism for other use cases inside our backend, providing a great level of flexibility for micro-frontends and beyond.

The service dictionary can be implemented with either a monolith or a modular monolith. The important thing to remember is to allow categorization of the endpoints list based on the micro-frontend that requests the endpoints. For instance, we can group the endpoints related to a business subdomain or a bounded context. This is the strategic goal we should aim for.

A service dictionary makes more sense with micro-frontends composed on the client side rather than on the server side. BFFs and API gateways are better suited for the server-side composition, considering the coupling between a micro-frontend and its data layer.

MODULAR MONOLITH

A modular monolith is a concept from the 1960s where the code is actually compartmentalized into separate modules. Moving to a modular monolith may be enough for some companies to continue evolving the API layer instead of doing a full migration to microservices. In his book [Monolith to Microservices](#), [Sam Newman provides many insights](#) into migrating a monolithic backend to microservices and discusses the concept of the modular monolith as a potential first step for our migration journey.

Let's now explore how to implement the service dictionary in a micro-frontend architecture.

Implementing a Service Dictionary in a Vertical-Split Architecture

The service dictionary pattern can easily be implemented in a vertical-split micro-frontend architecture, where every micro-frontend requests the dictionary corresponding to its business domain.

However, it's not always possible to implement a service dictionary per domain. For example, when transitioning from an existing SPA to micro-frontends, the SPA requires the full list of endpoints because it won't reload the JavaScript logic until the next user session. In this case, we may decide to implement a tactical solution, providing the full list of endpoints to the application shell instead of a business-domain-specific endpoints list to each micro-frontend. With this approach, we assume the application shell exposes or injects the list of endpoints for every micro-frontend.

When it becomes possible to divide the services list by domain, only minimal effort is required to remove the logic from the application shell and move it into each micro-frontend, as shown in [Figure 9-3](#).

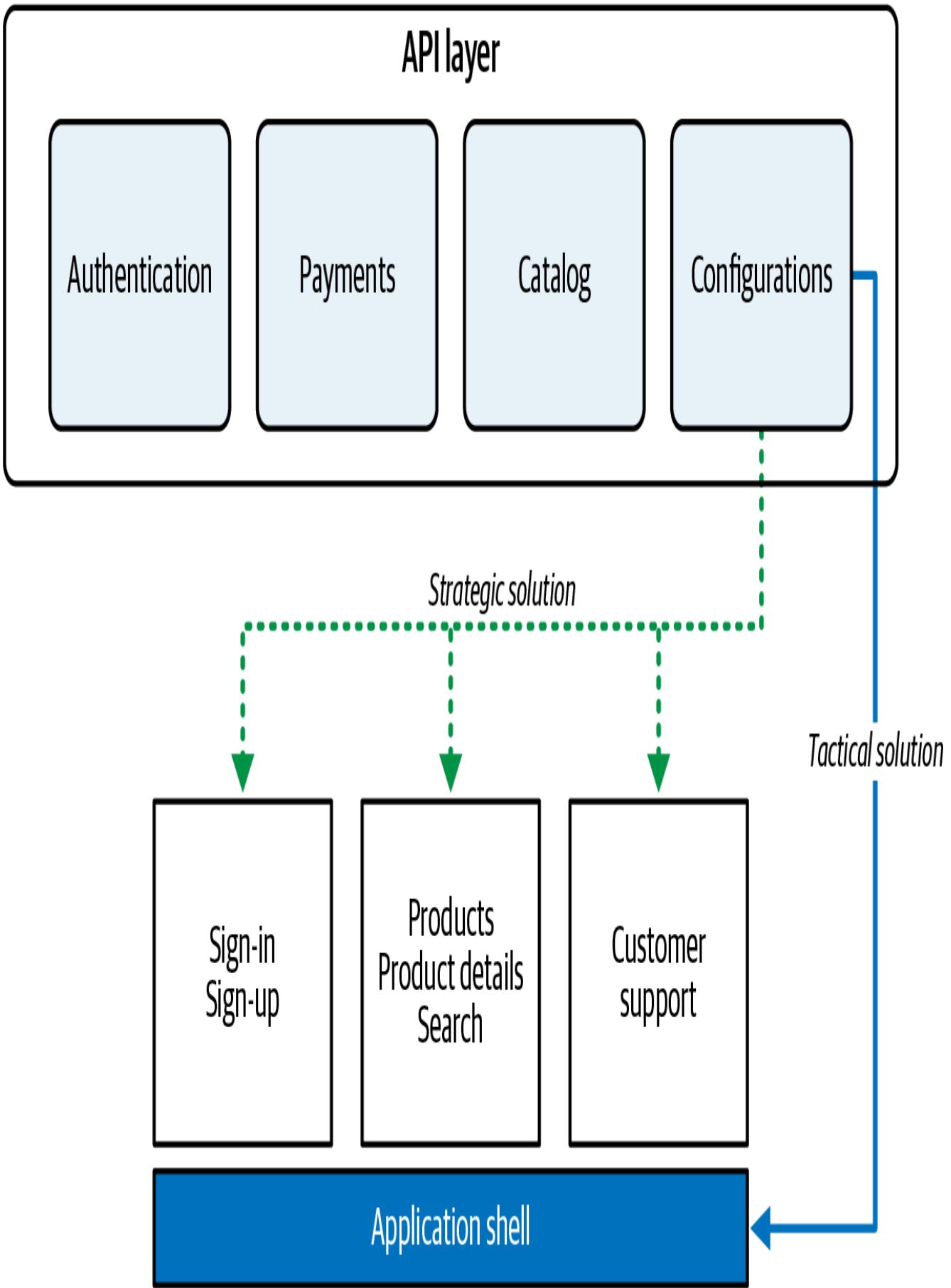


Figure 9-3. Vertical-split architecture allowing each micro-frontend to retrieve its service dictionary from an endpoint—here “Configurations”—with endpoints divided by business domain to align team structure

The service dictionary approach may also be used with a monolith backend. If we determine that our API layer will never move to microservices, we can still implement a service dictionary divided by domain for each micro-frontend, especially if we implement a modular monolith.

Considering [Figure 9-3](#), we can derive a sample of sequence diagrams, like the one shown in [Figure 9-4](#). Bear in mind, there may be additional steps to perform either in the application shell or in the micro-frontend loaded, depending on the context we operate in. Take the sequence diagram in [Figure 9-4](#) as an example.

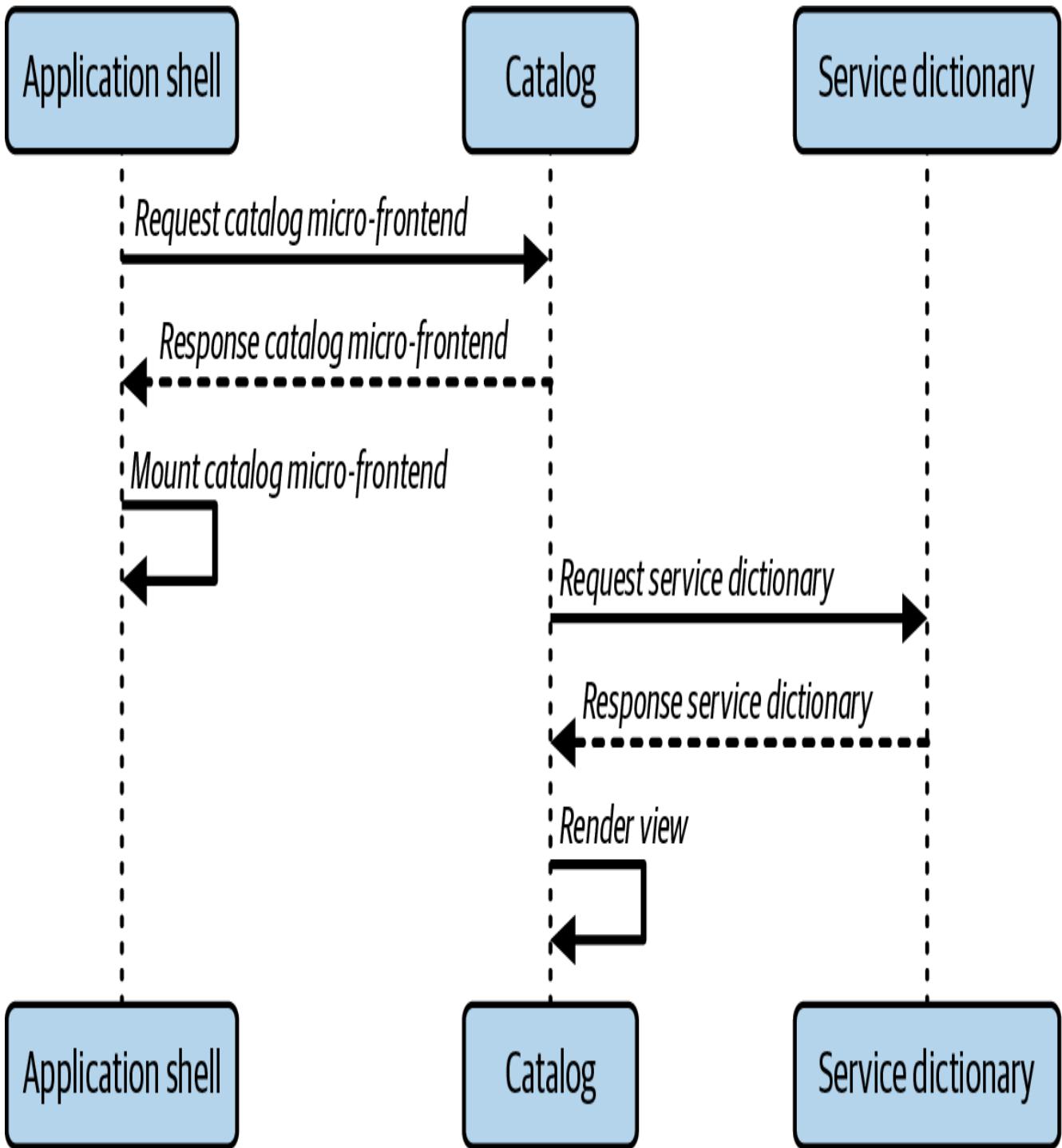


Figure 9-4. Sequence diagram to implement a service dictionary with a vertical-split architecture

As the first step, the application shell loads the micro-frontend requested—in this example, the catalog micro-frontend. After mounting the micro-frontend, the catalog initializes and consumes the service dictionary API for rendering the view. It can consume any additional APIs, as necessary. From this moment on, the catalog micro-frontend has access to the list of endpoints available and uses the dictionary to retrieve the endpoints to call. In this way, we are loading only the endpoints needed for a micro-

frontend, reducing the payload of our configuration and maintaining control of our business domain.

Implementing a Service Dictionary in a Horizontal-Split Architecture

To implement the service dictionary pattern with a micro-frontend architecture using a horizontal split, we have to pay attention to where the service dictionary API is consumed and how to expose it for the micro-frontends within a single view.

When the composition is managed client side, the recommended way to consume a service dictionary API is within the application shell or host page. Because the container has visibility into every micro-frontend to load, we can perform just one round trip to the API layer to retrieve the APIs available for a given view and expose or inject the endpoints list to each loaded micro-frontend.

Consuming the service dictionary APIs from every micro-frontend would negatively impact our applications' performance, so it's strongly recommended to stick the logic in the micro-frontend container as shown in [Figure 9-5](#).

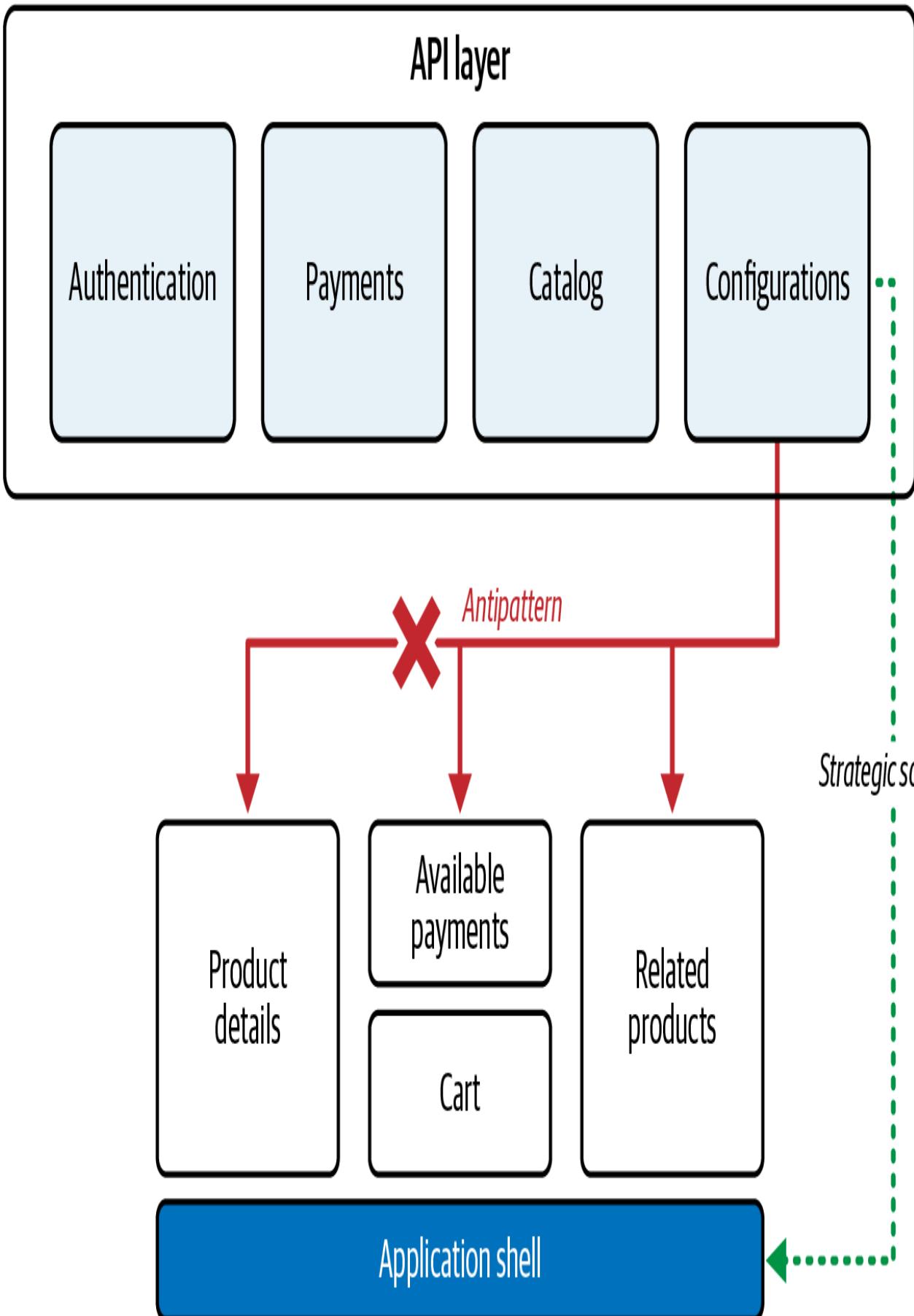


Figure 9-5. Service dictionary loaded from the micro-frontend container in a horizontal-split architecture

The application shell should expose the endpoint list via the `window` object, making it accessible to all the micro-frontends when the technical implementation allows us to do it. Another option is to inject the service dictionary, alongside other configurations, after loading each micro-frontend. For example, using Module Federation in a React application requires sharing this data using [React context APIs](#). The context API allows you to expose a context—in our case, the service dictionary—to the component tree without having to pass props down manually at every level. The decision to inject or expose our configurations is driven by the technical implementation. The sequence diagram in [Figure 9-6](#) showcases how we can express this use case.

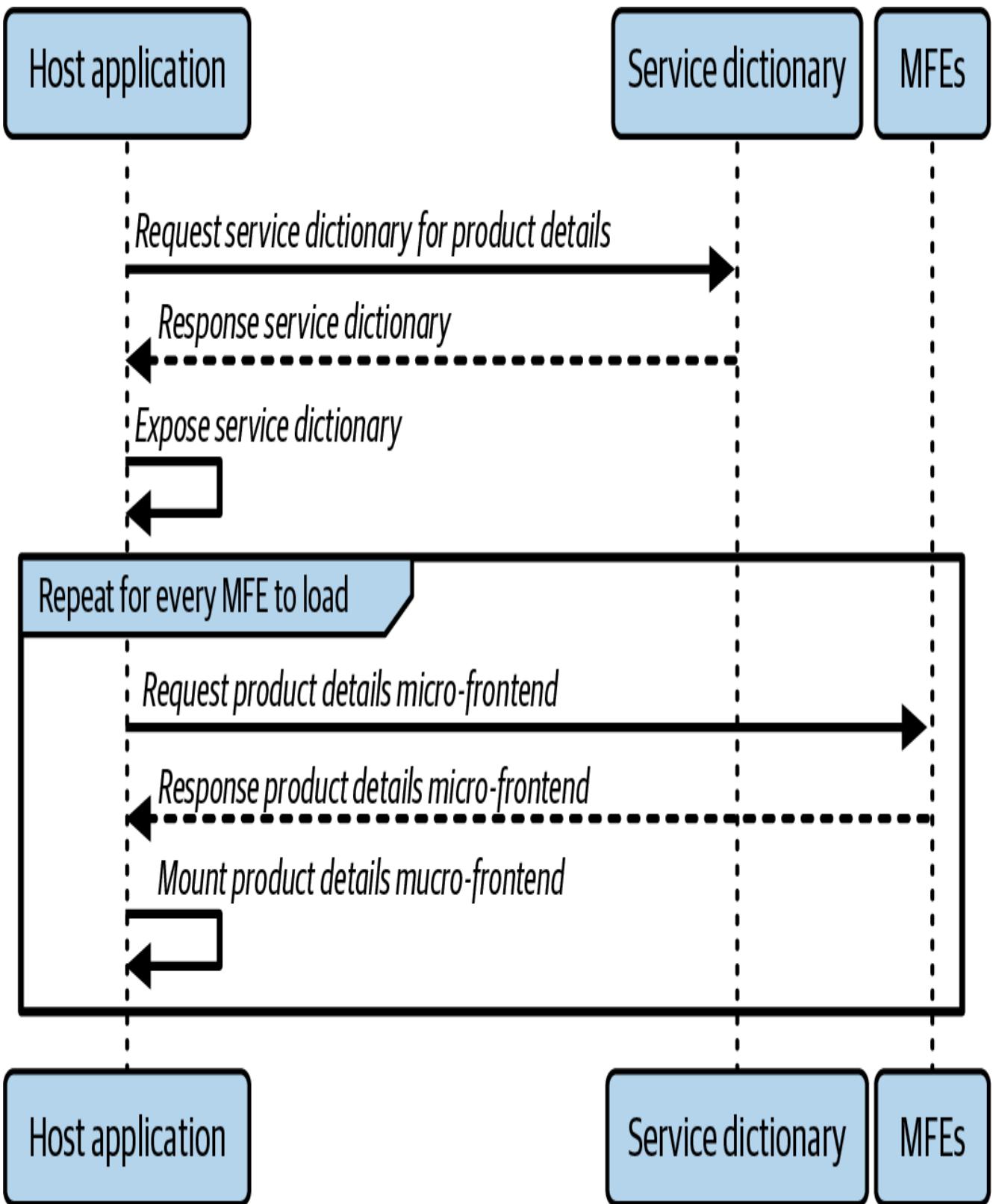


Figure 9-6. How a horizontal-split architecture with client-side composition may consume the service dictionary API

In this sequence diagram, the request from the host application, or application shell, to the service dictionary is at the very top of the diagram. The host application then

exposes the endpoints list via the `window` object and starts loading the micro-frontends that compose the view.

Again, in real scenarios we may have a more complex situation. Adapt the technical implementation and business logic to your project needs accordingly.

Working with an API Gateway

An API gateway pattern represents a unique entry point for the outside world to consume APIs in a microservices architecture. Not only does an API gateway simplify API access for any frontend by providing this entry point, but it's also responsible for request routing, API composition and validation, and other edge functions—namely authorization, logging, rate limiting, and any other centralized functionality needed before the API gateway sends the request to a specific microservice. An API gateway also allows us to maintain the same communication protocol between clients and the backend, while the gateway routes a request in the background in the format requested by the microservice (see [Figure 9-7](#)).

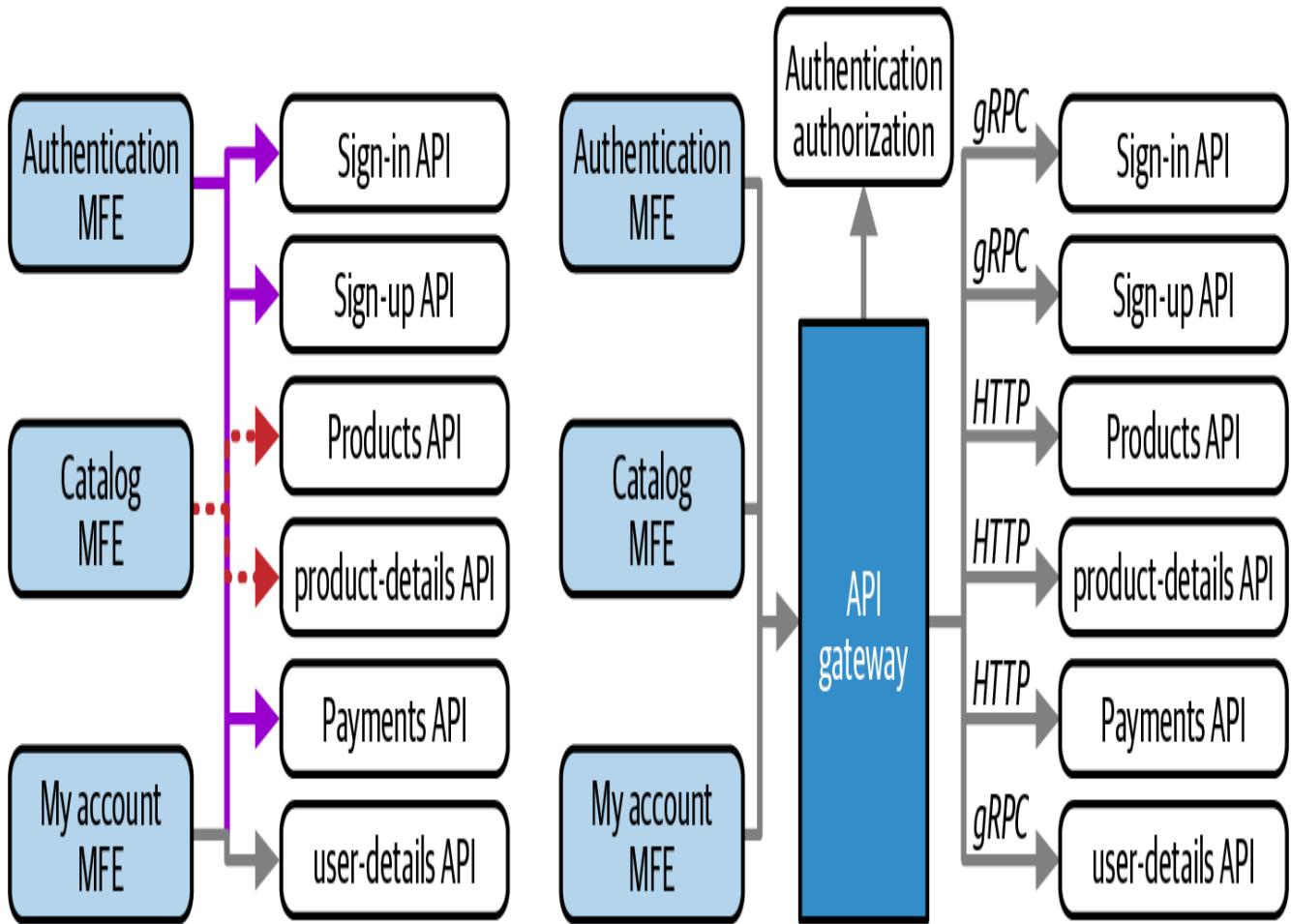


Figure 9-7. An API gateway pattern simplifying the communication between clients and server, and centralizing functionalities (such as authentication and authorization) via edge functions

Imagine a microservice architecture composed with HTTP and gRPC protocols. Without implementing an API gateway, the client won't be aware of each API or all the communication protocol details. Instead, by using the API gateway pattern, we can hide the communication protocols behind the API gateway, allowing the client to focus on the API contracts and implement the business logic needed on the user interface.

Other capabilities of edge functions include rate limiting, caching, metrics collection, and logging requests. Without an API gateway, all these functionalities would need to be replicated in every microservice, instead of being centralized through a single entry point.

Still, the API gateway has some downsides. As a unique entry point, it could become a single point of failure, so we need a cluster of API gateways to add resilience to our application. Cloud providers typically offer services that easily address this challenge, providing solutions designed to handle high traffic and built for resilience.

Another challenge is more operational. In a large organization with hundreds of developers working on the same project, many services may sit behind a single API

gateway. We'll need to provide solid governance for adding, changing, or removing APIs in the API gateway to prevent teams from being frustrated with a cumbersome workflow.

In addition, implementing an API gateway introduces some latency to the system between the client and the microservice. The process for updating the API gateway must be as lightweight as possible, making investment in governance mandatory. Otherwise, developers will be forced to wait in line to update the gateway with a new version of their endpoint. The API gateway can work in combination with a service dictionary, combining the benefits of both a service dictionary and the API gateway pattern.

Finally, with microarchitectures, it becomes possible and easier to manage and control because we are splitting our APIs by domain, enabling multiple API gateways to handle groups of related APIs.

One API Entry Point per Business Domain

Another possibility to consider is creating one API entry point per business domain instead of having one entry point for all the APIs, as with an API gateway. Multiple API gateways enable you to partition your APIs and policies by solution type and business domain.

In this way, we avoid having a single point of failure in our infrastructure. Parts of the application can fail without impacting the rest of the infrastructure. Another important characteristic of this approach is that we can choose the best entry point strategy per bounded context based on the requirements, as shown in [Figure 9-8](#).

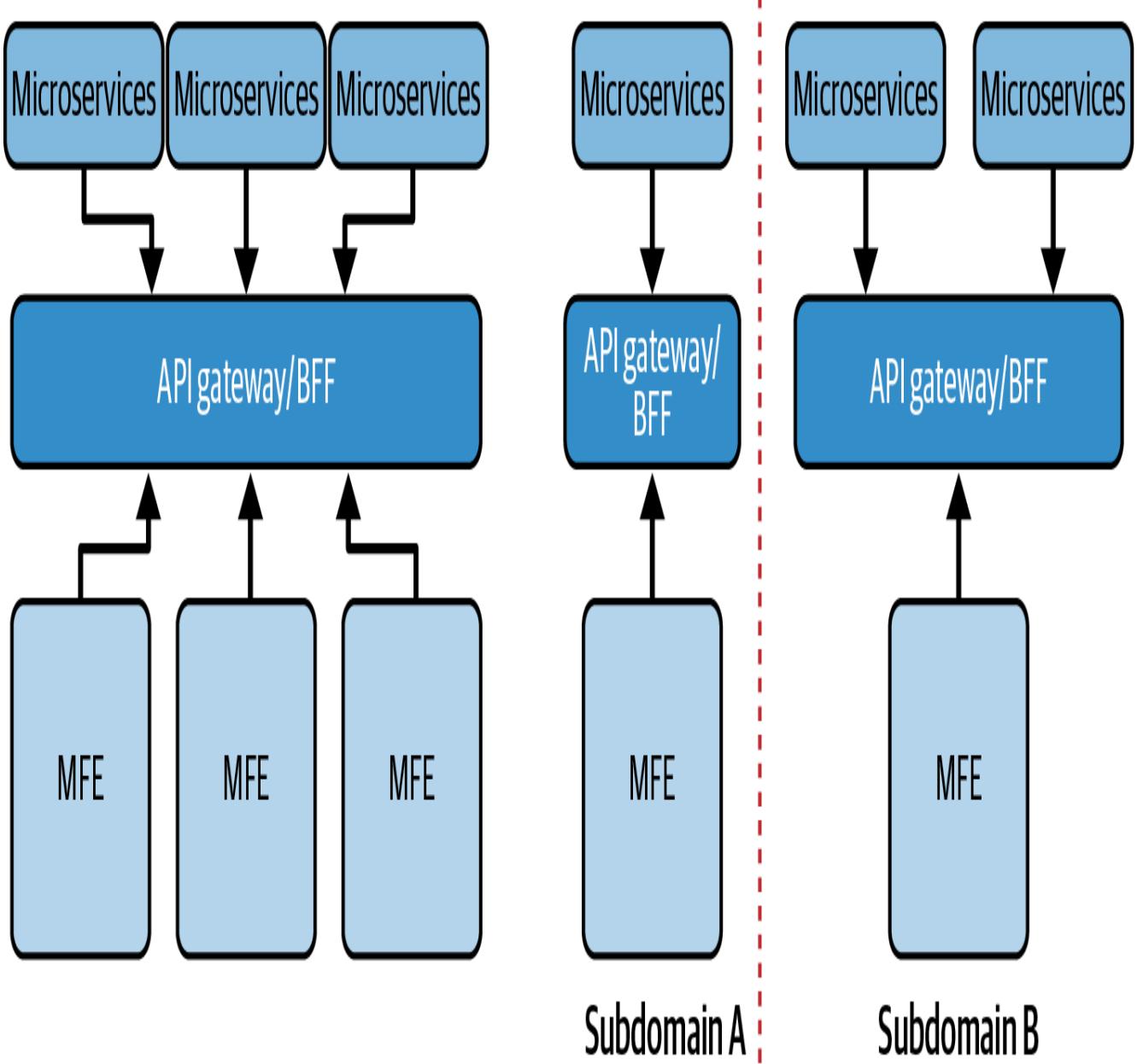


Figure 9-8. A unique entry point for the API layer (left) and multiple entry points, one per subdomain (right)

So, let's say we have a bounded context that needs to aggregate multiple APIs from different microservices and return a subset of each microservice's response body. In this case, a BFF would be a better fit for consumption by a micro-frontend, rather than having the client perform multiple round trips to the server and filter the API responses to display the final result to the user.

However, in the same application, we may have a bounded context that doesn't need a BFF.

Let's go one step further and say that in this subdomain, we have to validate the user token in every call to the API layer to check whether the user is entitled to access the

data. In this case, using an API gateway pattern with validation at the API gateway level will allow you to fulfill the requirements simply.

With infrastructure ownership, choosing different entry points for our API layer means every team is responsible for building and maintaining the entry point chosen, reducing potential external dependencies across teams, and allowing them to own the subdomain they are responsible for, end to end. Therefore, we can potentially have a one-to-one relationship between subdomain and entry point.

While this approach may demand more initial effort, it empowers your team with granular control, enabling you to choose the best tool for each task, rather than compromising between flexibility and functionality. It also fosters true end-to-end autonomy, freeing engineers to evolve the frontend, backend, or infrastructure independently, without risking cross-domain side effects.

However, be aware of potential pitfalls. Achieving a perfectly clean API separation across business domains is rarely practical; some APIs will inevitably span multiple domains. If you find yourself with too many cross-cutting APIs, consider consolidating them behind a single entry point. This pattern can greatly simplify governance and reduce operational overhead.

Client-Side Composition with an API Gateway and a Service Dictionary

Using an API gateway with a client-side micro-frontend composition (either vertical or horizontal split) is not that different from implementing a service dictionary in a monolith backend. In fact, we can use the service dictionary to provide our micro-frontends with the endpoints to consume, following the same suggestions we provided previously. The main difference, in this case, is that the endpoints list will be provided by a microservice responsible for serving the service dictionary, or by a more generic client-side configuration, depending on our use case.

Another interesting option is that, with an API gateway, authorization may happen at the API gateway level, removing the risk of introducing libraries at the API level, as we can see in [Figure 9-9](#).

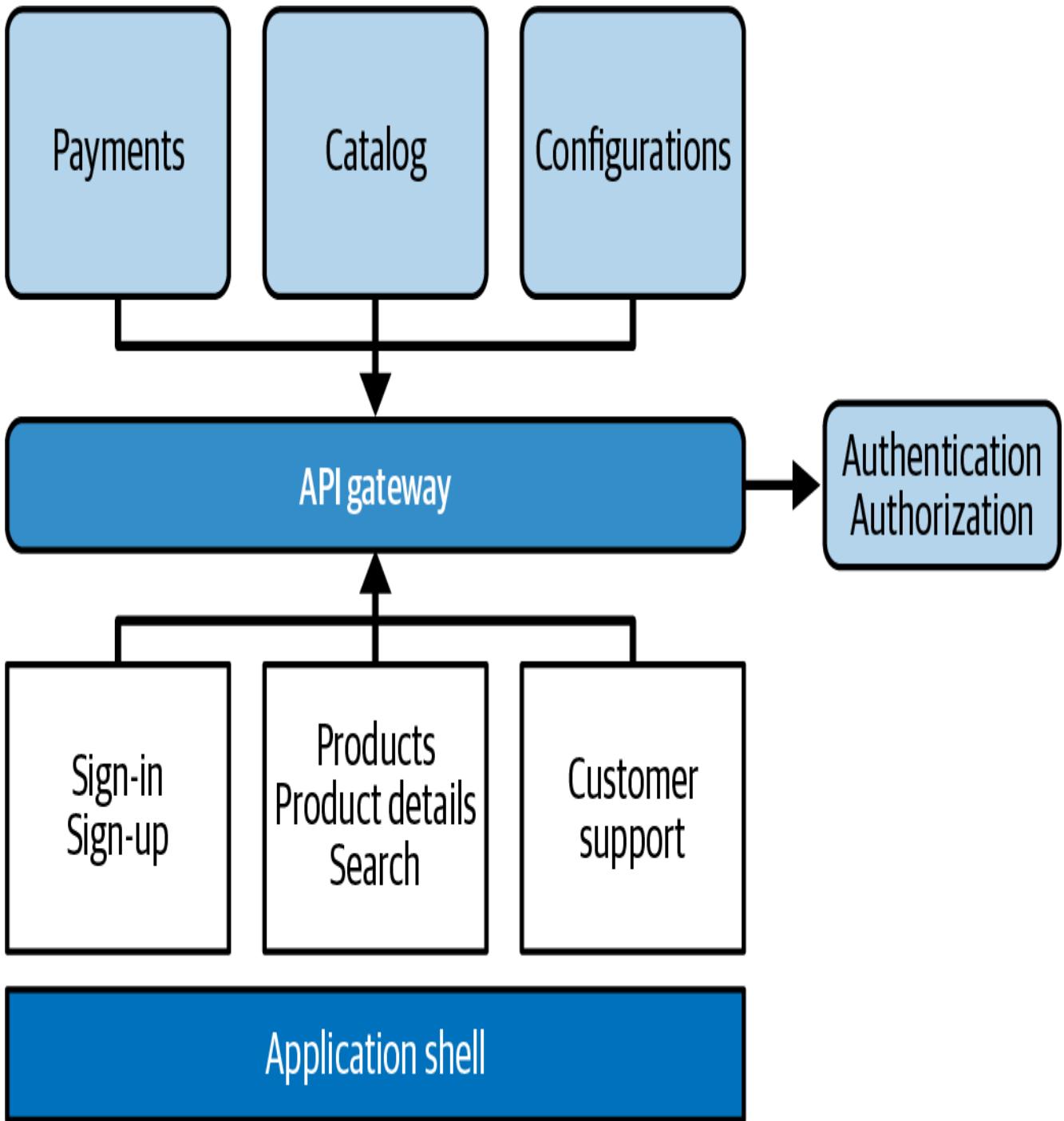


Figure 9-9. A vertical-split architecture with a client-side composition requesting data to a microservice architecture with an API gateway as entry point

Based on the concepts shared with the service dictionary, the backend infrastructure has changed, but not the implementation side. As a result, the same implementations applicable to the service dictionary are also applicable in this scenario with the API gateway.

Let's look at one more interesting use case for the API gateway. Some applications allow us to use a micro-frontend architecture to provide different flavors of the same product to multiple customers, such as customizing certain micro-frontends on a

customer-by-customer basis. In such cases, we tend to reuse the API layer for all customers, using part or all of the microservices based on user entitlement. But in a shared infrastructure, we risk some customers consuming more of our backend resources than others. In such scenarios, using API throttling at the API gateway will mitigate this problem by assigning the appropriate limits per customer or per product. At the micro-frontend level, we only need to handle the errors triggered by the API gateway for this use case.

Server-Side Composition with an API Gateway

A microservices architecture opens up the possibility of using a micro-frontend architecture with a server-side composition, as detailed in [Chapter 6](#).

As we can see in [Figure 9-10](#), after the browser request to the API gateway, the gateway handles the user authentication/authorization first, and then allows the client request to be processed by the UI composition service responsible for calling the microservices needed to aggregate multiple micro-frontends inside a template, with their relative content fetched from the microservices layer.

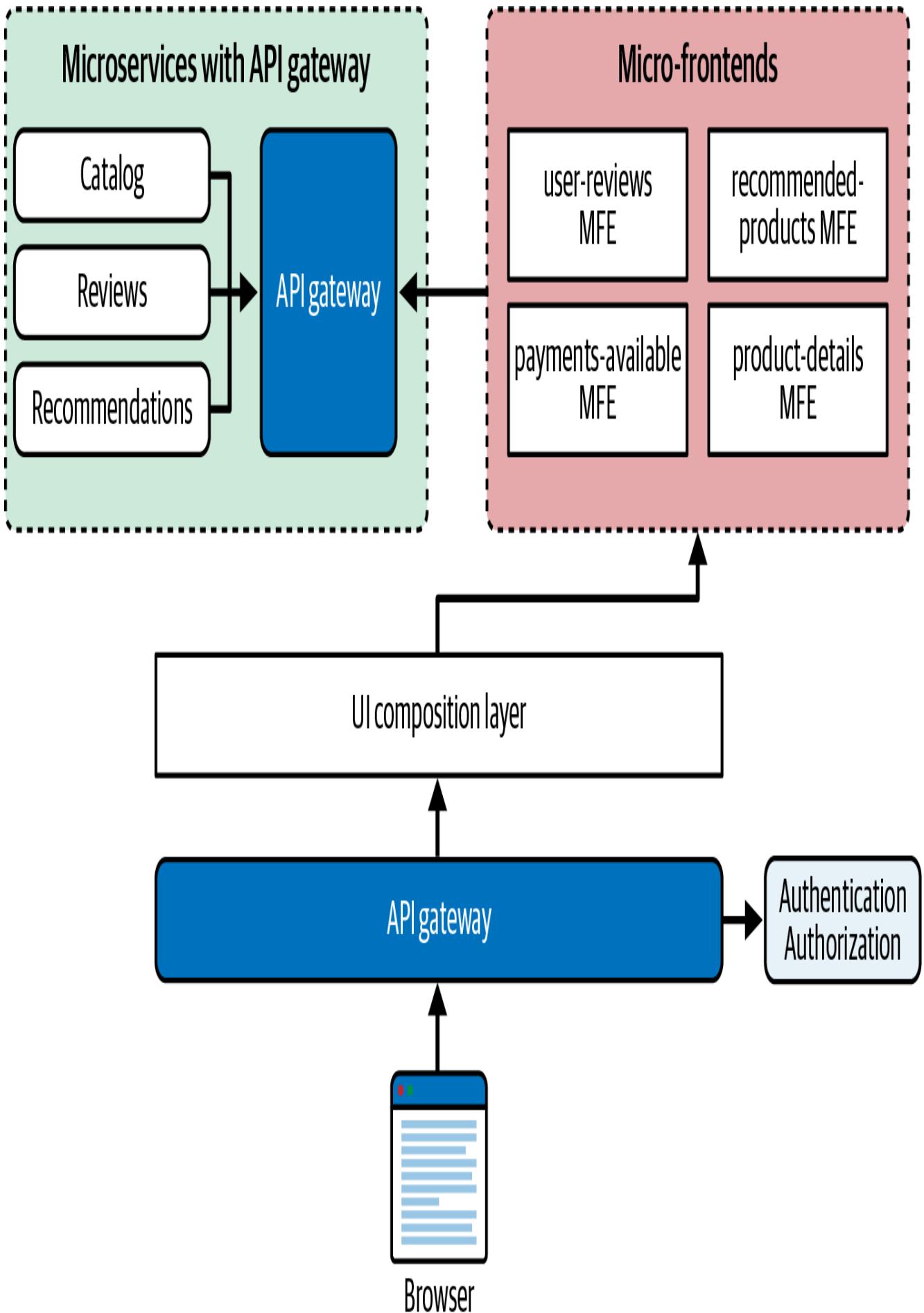


Figure 9-10. Example of a server-side composition with a microservices architecture

For the microservices layer, we use a second API gateway to expose the API for internal services, in this case, used by the micro-frontend services for fetching the related API.

Figure 9-11 illustrates a hypothetical implementation with the sequence diagram related to this scenario.

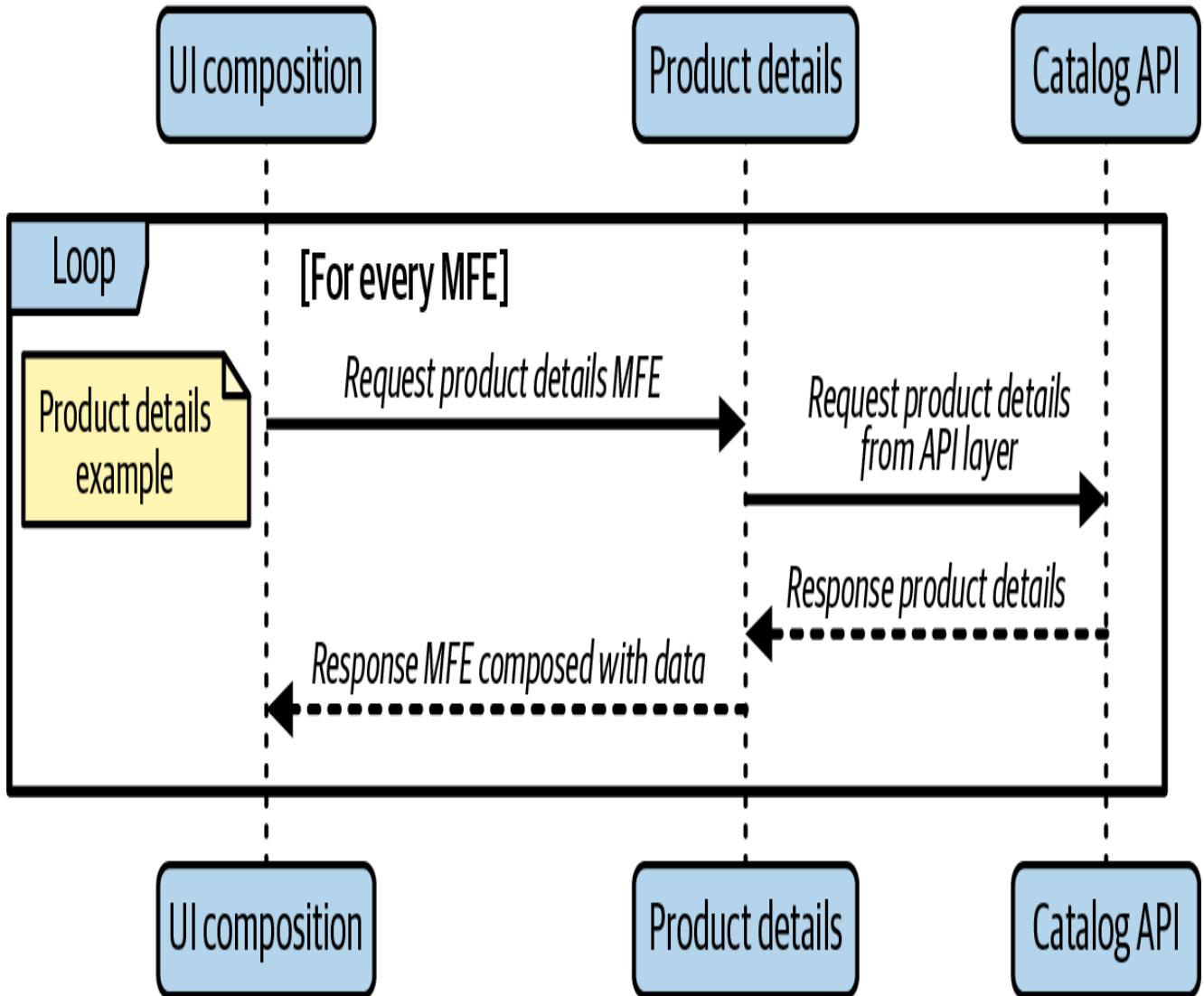


Figure 9-11. Example of server-side composition with API gateway

After the API gateway token validation, the client-side request lands at the UI composition service, which calls the micro-frontend to load. The micro-frontend service is then responsible for fetching the data from the API layer and the relative template for the UI, and serving a fragment to the UI composition layer that will compose the final result for the user. While Figure 9-11 presents an example with a

micro-frontend, it's also applicable for all the other micro-frontends available in a system that should be retrieved for composing a user interface.

Usually, the microservice used for fetching the data from the API layer should have a one-to-one relation with the API it consumes, which enables end-to-end team ownership of a specific micro-frontend and microservice.

Working with the BFF Pattern

Although the API gateway pattern is a very powerful solution for providing a unique entry point to our APIs, in some situations, we have views that require the aggregation of several APIs to compose the user interface. For example, a financial dashboard may require several endpoints for gathering the data to display inside a unique view.

Sometimes, we aggregate this data on the client side, consuming multiple endpoints and interpolating data for updating our view with the diagrams, tables, and useful information that our application should display. The question arises as to whether we can do something better than that. Here, BFF comes to the rescue.

Another interesting scenario where an API gateway may not be suitable is in a cross-platform application where our API layer is consumed by web and mobile applications. Moreover, the mobile platforms often require displaying the data in a completely different way from the web application, especially taking screen size into consideration.

In this case, many visual components and relative data may be hidden on mobile in favor of providing a more general, high-level overview and allowing a user to drill down to a specific metric or piece of information that interests them, instead of waiting for all the data to download.

Finally, mobile applications often require a different method for aggregating data and exposing it in a meaningful way to the user. APIs on the backend are the same for all clients, so for mobile applications, we need to consume different endpoints and compute the final result on the device instead of changing the API responses based on the device that consumes the endpoint. In such cases, the BFF pattern, as coined by [Phil Calçado](#) (former employee of SoundCloud), can provide a solution.

The BFF pattern develops niche backends for each user experience. This pattern will only make sense if and when you have a significant amount of data coming from different endpoints that must be aggregated to improve the client's performance, or when you have a cross-platform application that requires different experiences for the

user based on the device used. This pattern can also help solve the challenge of introducing a layer between the API and the clients, as we can see in [Figure 9-12](#).

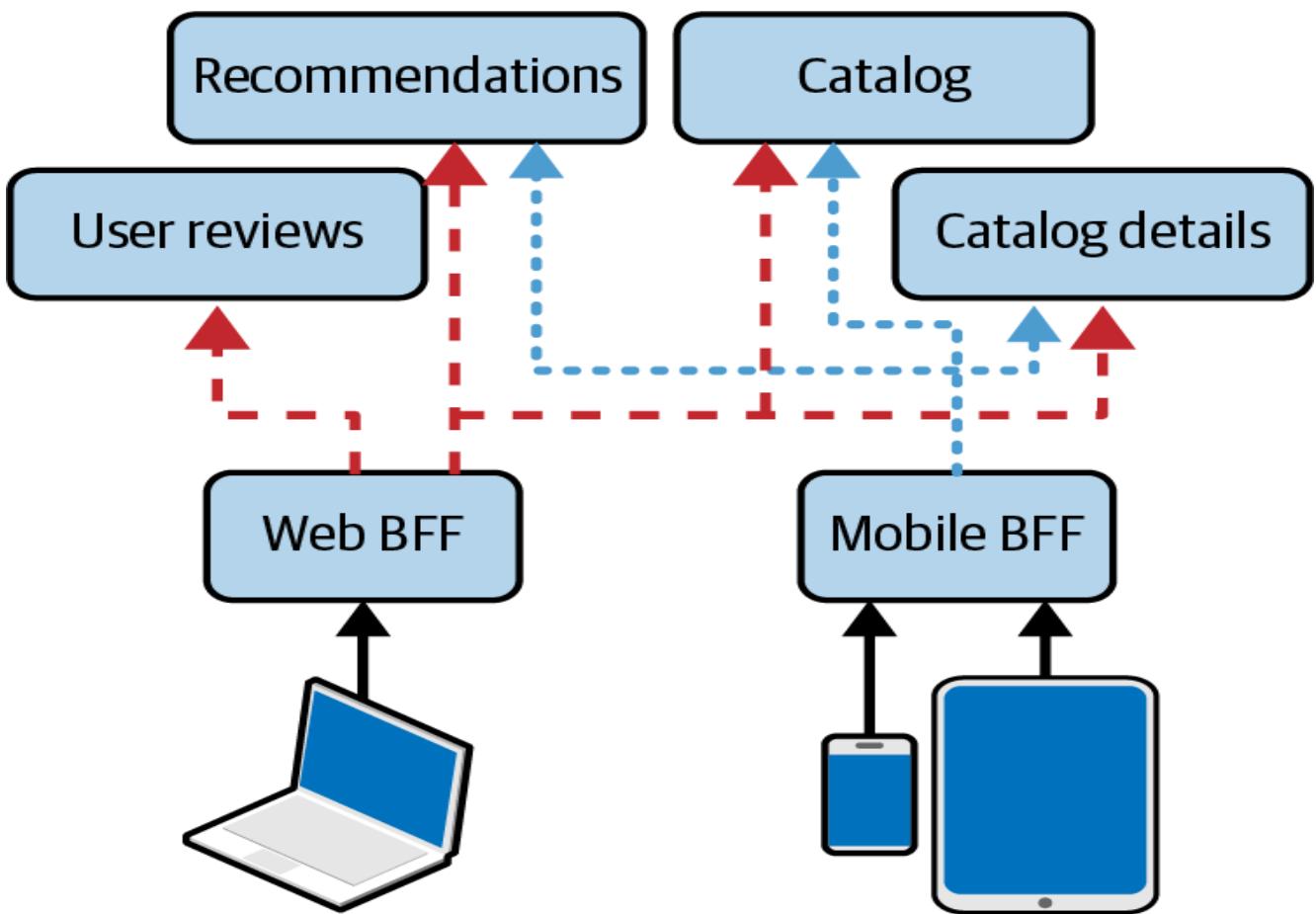
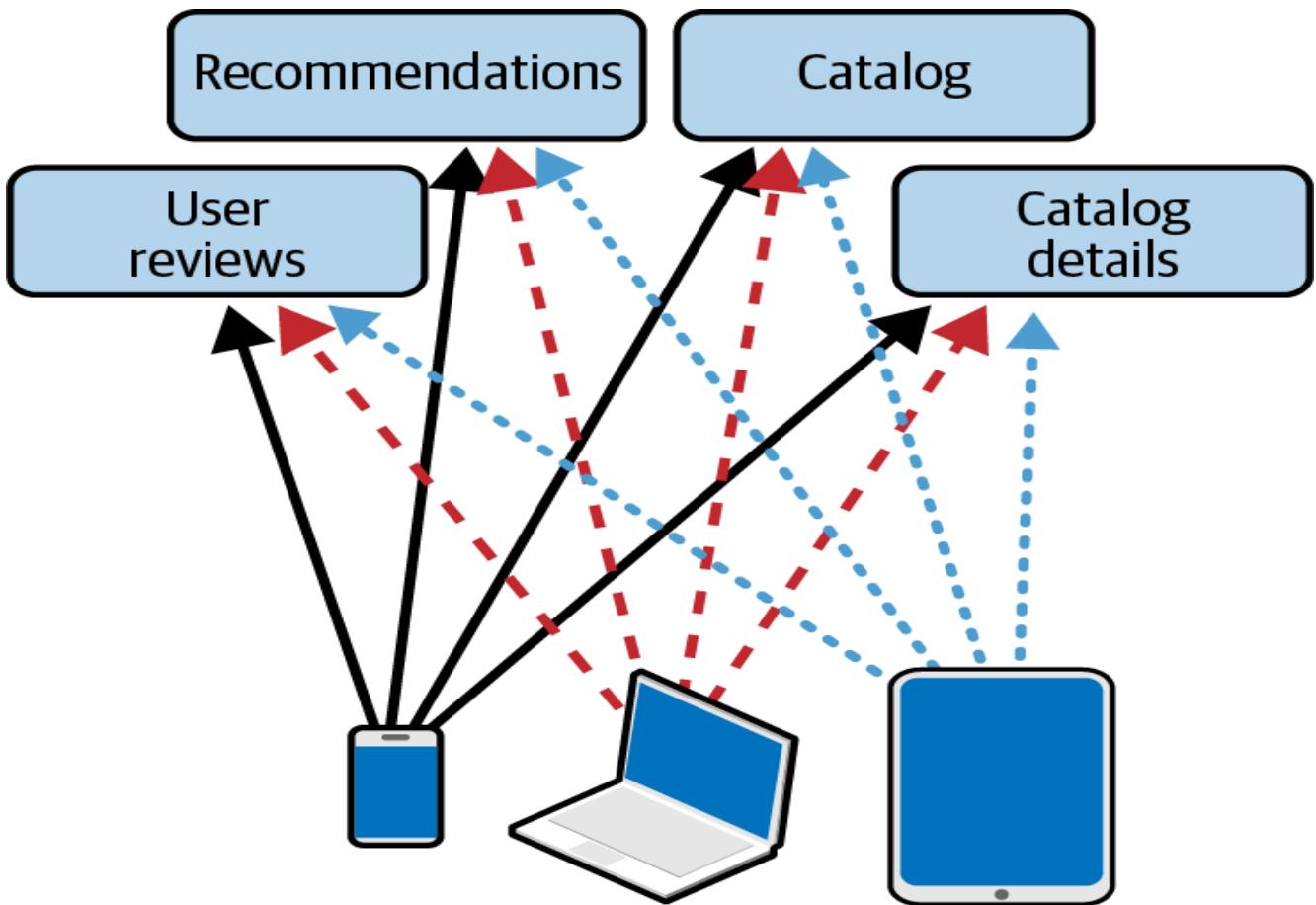


Figure 9-12. On the left, a microservices architecture consumed by different clients; on the right, a BFF layer exposing only the APIs needed for a given group of devices (here, mobile and web BFF)

Thanks to BFF, we can create a unique entry point for a given device group, such as one for mobile and another for a web application. However, this time we also have the option of aggregating API responses before serving them to the client and, therefore, generating less chatter between clients and the backend because the BFF aggregates the data and serves only what is needed for a client, with a structure reflecting the view to populate. Interestingly, the microservice architecture's complexity sits behind the BFF, creating a single entry point for the client to consume the APIs without needing to understand the complexity of the microservices architecture.

BFF can also be used when migrating a monolith to microservices. In fact, thanks to the separation between clients and APIs, we can use the strangler fig pattern to retire the monolith in an iterative way, as illustrated in [Figure 9-13](#). This technique is also applicable to the API gateway pattern.

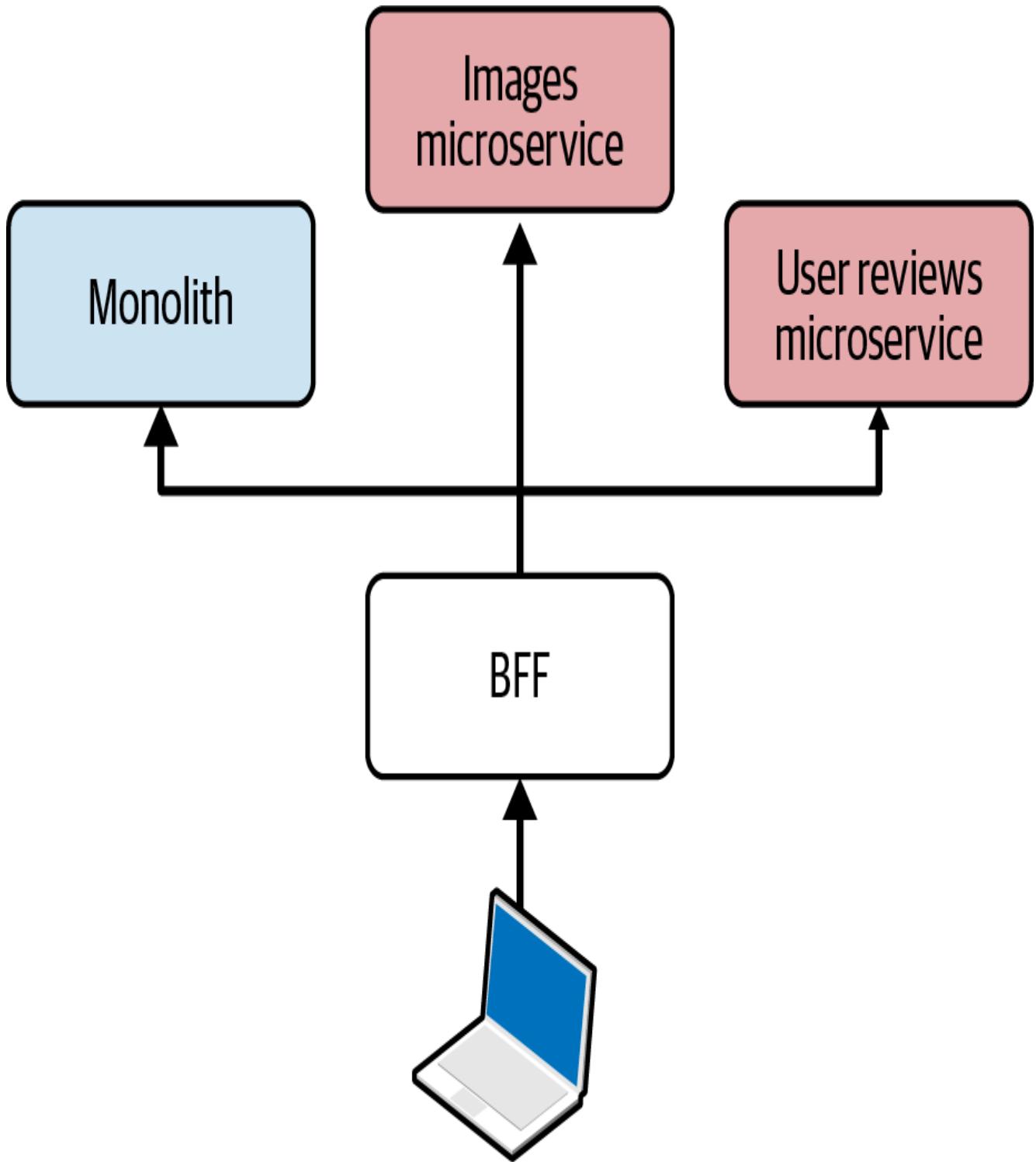


Figure 9-13. Red boxes representing services extracted from the monolith and converted to microservices, with the BFF layer allowing the client to be unaware of the change happening in the backend, maintaining the same contract at the BFF level

Another use case that often comes to mind when we combine BFF and micro-frontends is aggregating APIs by domain—similar to what we have seen for the API gateway. Following our subdomain decomposition, we can identify a unique entry point for each subdomain, grouping all the microservices for a specific domain together instead of taking into consideration the type of device that should consume the APIs. This would

allow us to control the response to the clients in a more cohesive way, and allow the application to fail more gracefully than having a single layer responsible for serving all the APIs, as in the previous examples.

Figure 9-14 illustrates how we can have two BFFs—one for the catalog and one for the account section—for aggregating and exposing these APIs to different clients. In this way, we can scale the BFFs based on their traffic.

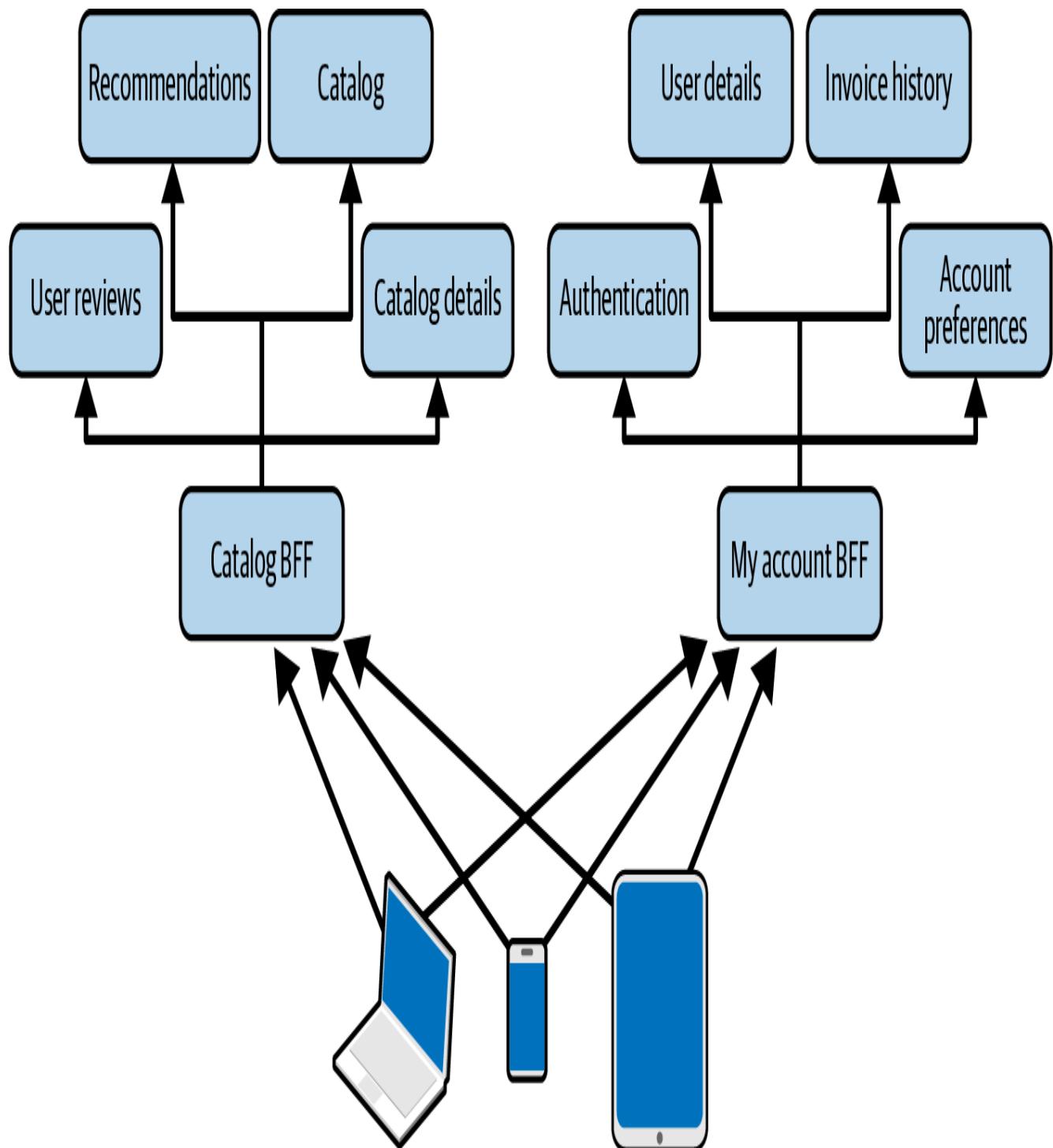


Figure 9-14. How to separate different DDD subdomains

However, gathering all the APIs behind a unique layer may lead to an application's popular subdomains requiring a different treatment compared to less-accessed subdomains. Dividing by subdomain, then, allows us to apply different infrastructure requirements based on the traffic and characteristics of each domain.

Sometimes, BFF raises concerns due to some inherent pitfalls such as reusability, code duplication, and cross-boundary APIs. In fact, we may need to duplicate some code for implementing similar functionalities across different BFFs, especially when we create one per device family. In these cases, we need to assess whether the burden of having teams implementing similar code twice is greater than abstracting (and maintaining) the code.

It is no surprise that identifying domain boundaries for completely self-sufficient APIs is difficult. Consider, for example, an ecommerce platform where a product's service is used in multiple domains. In this case, we need to make sure that every BFF implements the latest API version of the `products` service. Moreover, every time a new version of the `products` service is released, we will need to coordinate the release of the BFF layers. Alternatively, we can support multiple versions of the `products` API for a period, allowing each BFF to update independently at its own pace.

Client-Side Composition with a BFF and a Service Dictionary

Because a BFF is an evolution of the API gateway, many of the implementation details for an API gateway are valid for a BFF layer as well—plus, we can aggregate multiple endpoints, reducing client chatter with the server. It's important to iterate this capability, because it can drastically improve application performance.

However, there are some caveats when implementing either a vertical or horizontal split. For instance, [Figure 9-15](#) shows a product details page that has to fetch the data for composing the view.

Logo

Menu

Product details

Payment
options

Related
products

User reviews

Footer

Figure 9-15. A wireframe of a product page

When we want to implement a vertical-split architecture, we may design the BFF to fetch all the data needed for composing this view. This is shown in [Figure 9-16](#).

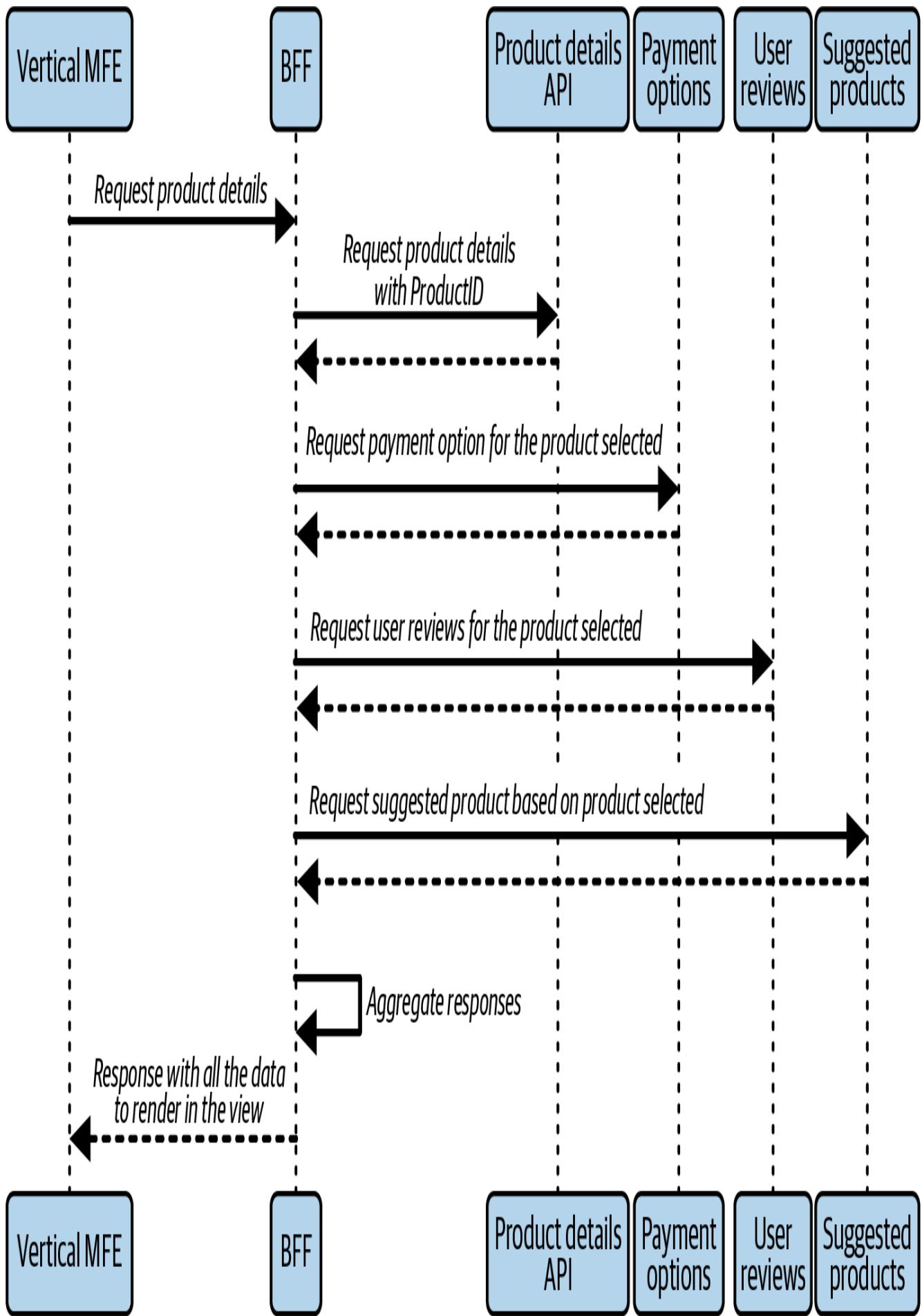


Figure 9-16. Benefits of the BFF pattern used in combination with a vertical split composed on the client side

In this example, we assume the micro-frontend has already retrieved the endpoint for performing the request via a service dictionary and that it consumes the endpoints, leaving the BFF layer to compose the final response. In this use case, we can also easily use a service dictionary for exposing the endpoints available in our BFF to our micro-frontends, similar to the way we do it for the API gateway solution.

However, when we have a horizontal split composed on the client side, things become trickier because we need to maintain the micro-frontends' independence and keep the host page domain as unaware as possible. In this case, we need to combine the APIs differently, delegating each micro-frontend to consume its related API. If not, the host page would need to be responsible for fetching the data for all the micro-frontends, which could create a coupling and force us to deploy the host page alongside the micro-frontends, breaking the intrinsic independence between them. Considering that these micro-frontends and the host page may be developed by different teams, this setup would slow down feature development rather than leverage the benefits of this architecture.

Moreover, this could lead to the creation of a global state, implemented at the micro-frontends' container to simplify access for all micro-frontends present in the view, creating unnecessary coupling. A BFF with a horizontal split composed on the client side could introduce more challenges than benefits in this scenario. It's wise to analyze whether this pattern's benefits will outweigh its challenges.

Server-Side Composition with a BFF and a Service Dictionary

When we implement a horizontal-split architecture with server-side composition and we have a BFF layer, our micro-frontend implementation resembles the API gateway one. The BFF exposes all the APIs available for every micro-frontend, so using the service dictionary pattern will allow us to retrieve the endpoints for rendering our micro-frontends ready to be composed by a UI composition layer.

Using GraphQL with Micro-Frontends

In a chapter about APIs and micro-frontends, we can't avoid discussing [GraphQL](#). GraphQL is a query language for APIs and a server-side runtime for executing queries by using a type system you define for your data. GraphQL was created by Facebook and

released in 2015. Since then, it has gained a lot of traction inside the developer community.

Particularly for frontend developers, GraphQL represents a great way to retrieve the data needed for rendering a view, decoupling the complexity of the API layer, rationalizing the API response in a graph, and allowing any client to reduce the number of round trips to the server when composing the UI.

The paradigm for designing an API schema with GraphQL should be based on the structure of the view we need to render, rather than the data exposed by the backend services. This view-first approach flips the traditional REST design on its head—prioritizing UI needs over resource modeling—and makes GraphQL particularly well-suited for micro-frontends, where each fragment of the UI often has distinct and localized data requirements. This is a very key distinction compared to how we design our database schemas or REST APIs.

Two projects in the GraphQL community stand out as providing great support and productivity with the open source tools available: [Apollo Server](#) and [Relay](#). Both projects leverage GraphQL, adding an opinionated view on how to implement this layer within our application and increasing our productivity due to the features available in one or both, such as authentication, rate limiting, caching, and schema federation.

GraphQL can be used as a proxy for microservices, orchestrating the requests to multiple endpoints and aggregating the final response for the client.

Remember that GraphQL acts as a unique entry point for your entire API layer. By design, GraphQL exposes a unique endpoint where clients can perform queries against the GraphQL server. Because of this, we tend not to version our GraphQL entry point—although, if the project requires versioning because we don't have full control of the clients consuming our data, we can version the GraphQL endpoint. [Shopify](#) does this by adding the date in the URL and supporting all the versions up to a certain period.

It's important to highlight that GraphQL works best when it's created as a unique entry point for the client and not split by domains, as seen with BFF. The graph implementation allows every client to query whatever part of the graph is exposed. Splitting it into multiple domains would only make life harder for developers integrating with multiple graphs, as they would now have to compose the different responses on the client side.

You might wonder how to scale the development of GraphQL across multiple teams—the answer is schema federation.

The Schema Federation

Schema federation is a feature that allows multiple **GraphQL schemas** to be composed declaratively into a single data graph.

When we work with GraphQL in a mid- to large-sized organization, we risk creating a bottleneck because all the teams are contributing to the same schema. But with schema federation, we can have individual teams working on their own schemas and exposing them to the client as unique entry points, just like a traditional data graph.

Apollo Server exposes a gateway with all associated schemas from other services, allowing each team to remain independent and not change the way the frontend consumes the data graph. This technique comes in handy when we work with microservices, though it comes with a caveat.

A GraphQL schema should be designed with the UI in mind, so it's essential to avoid silos within the organization. We must facilitate the initial analysis, engaging with multiple teams and following all improvements in order to have the best implementation possible.

Figure 9-17 shows how schema federation works, using the gateway as an entry point for all the implementing services, and providing a unique entry point and data graph for clients to query.

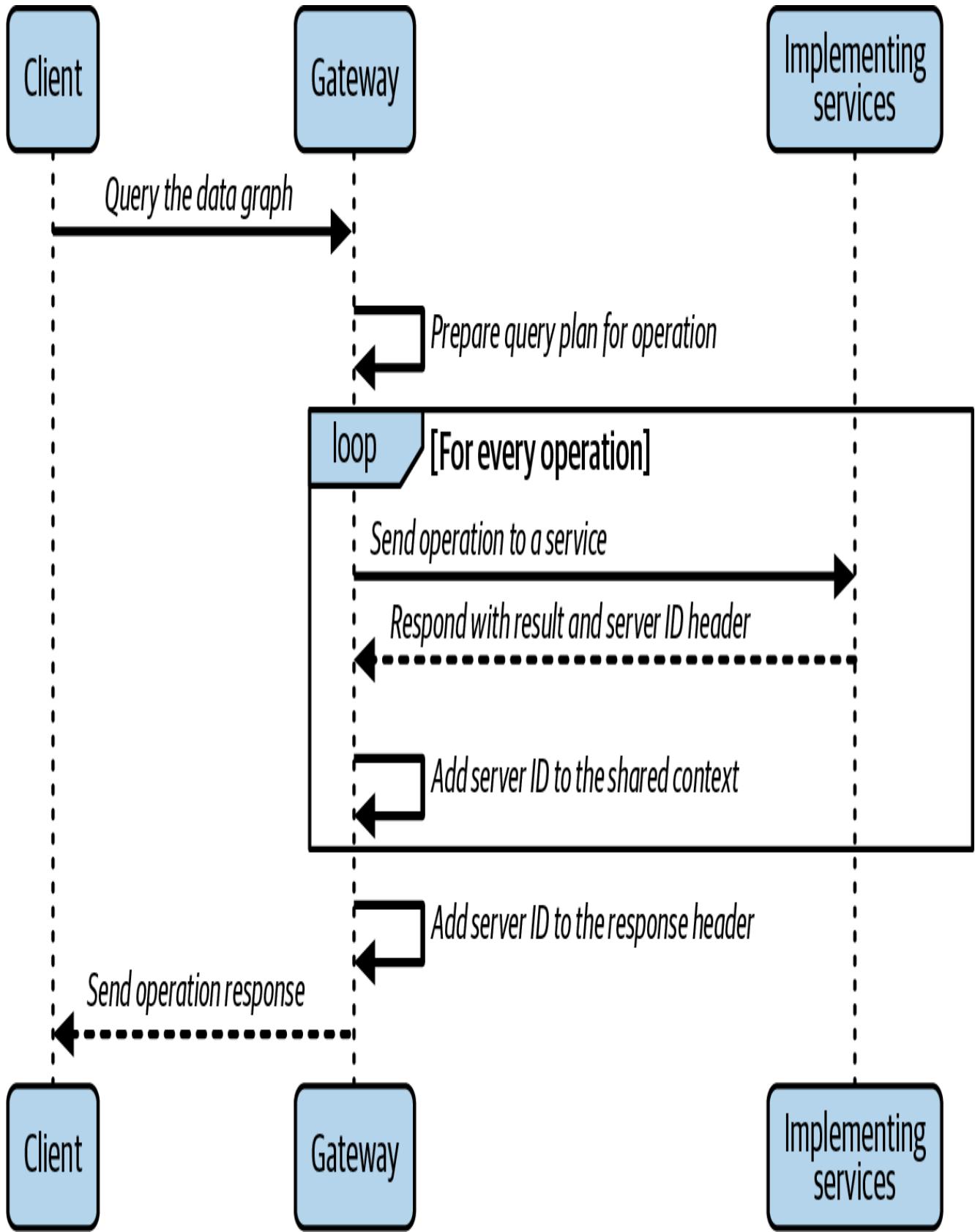


Figure 9-17. How schema federation exposes all the schemas from multiple services

Schema federation represents the evolution of **schema stitching**, which has been used by many large organizations for similar purposes. However, it wasn't designed well,

which led Apollo to deprecate schema stitching in favor of schema federation. More information regarding the schema federation is available on [Apollo's documentation website](#).

Using GraphQL with Micro-Frontends and a Client-Side Composition

Integrating GraphQL with micro-frontends is a trivial task, especially after reviewing the implementation of the API gateway and BFF. With schema federations, the teams who are responsible for a specific domain's APIs can create and maintain the schema for their domain and then merge all the schemas into a unique data graph for client applications. This approach allows the team to be independent, maintaining their schemas and exposing what the clients would need to consume.

When we integrate GraphQL with a vertical split and a client-side composition, the integration resembles the others described earlier in this chapter—the micro-frontend is responsible for consuming the GraphQL endpoint and rendering the content inside every component present in a view. Applying such scenarios with microservices becomes easier due to schema federation, as shown in [Figure 9-18](#).

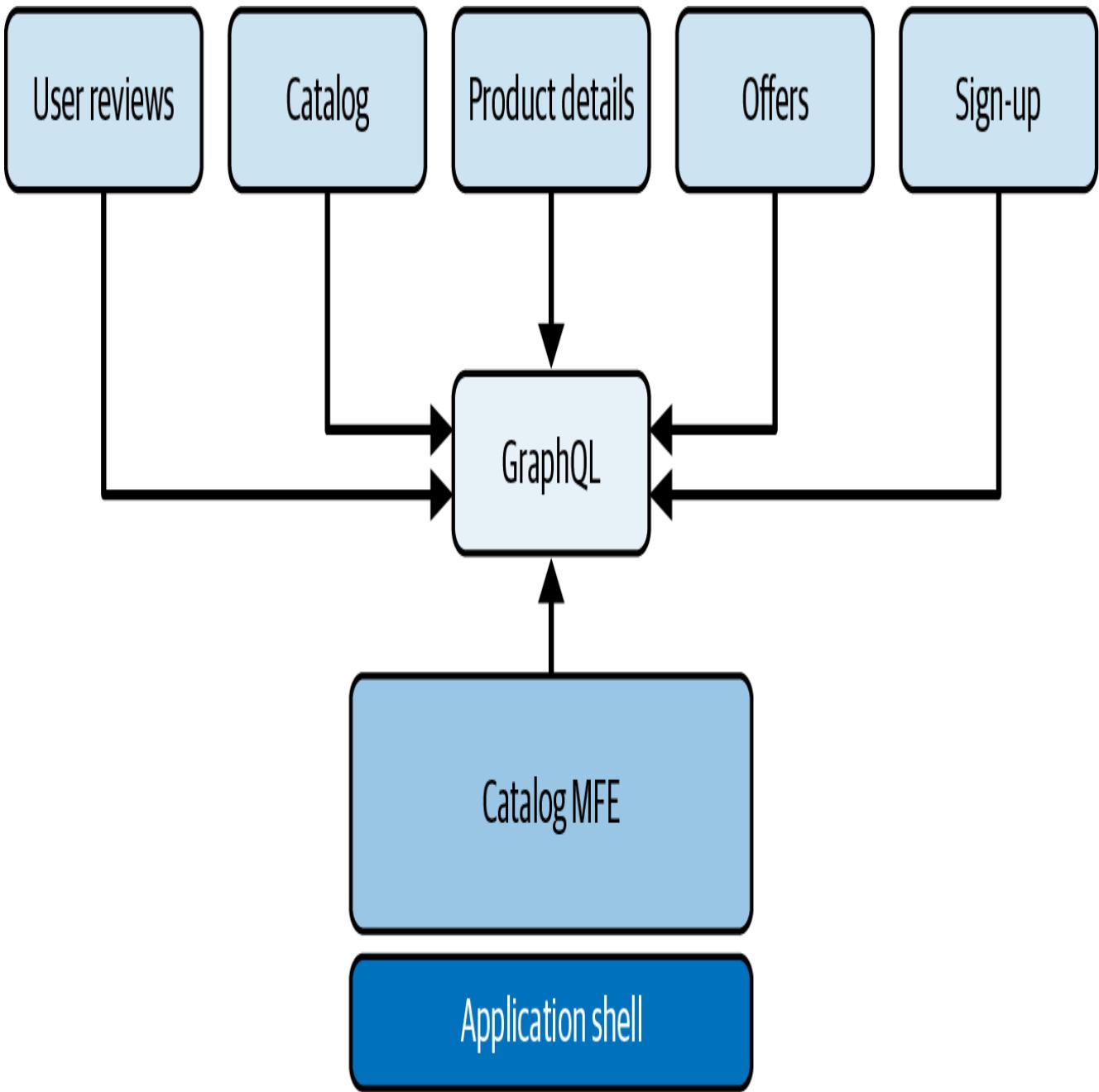


Figure 9-18. High-level architecture composing a microservice backend with schema federation, with the catalog micro-frontend consuming the graph of all schemas inside the GraphQL server

In this case, thanks to the schema federation, we can compose the graph with all the schemas needed and expose a supergraph for a micro-frontend to consume. Interestingly, with this approach, every micro-frontend will be responsible for consuming the same endpoint. Optionally, we may want to split the BFF into different domains, creating a one-to-one relation with the micro-frontend. This would reduce the scope of work and make our application easier to manage, considering the domain scope is smaller than having a unique data graph for all the applications.

Applying a similar backend architecture to horizontal-split micro-frontends with a client-side composition isn't that different from other implementations we have

discussed in this chapter.

As we can see in [Figure 9-19](#), the application shell exposes or injects the GraphQL endpoint to all the micro-frontends, and all the queries related to a micro-frontend will be performed by every micro-frontend.

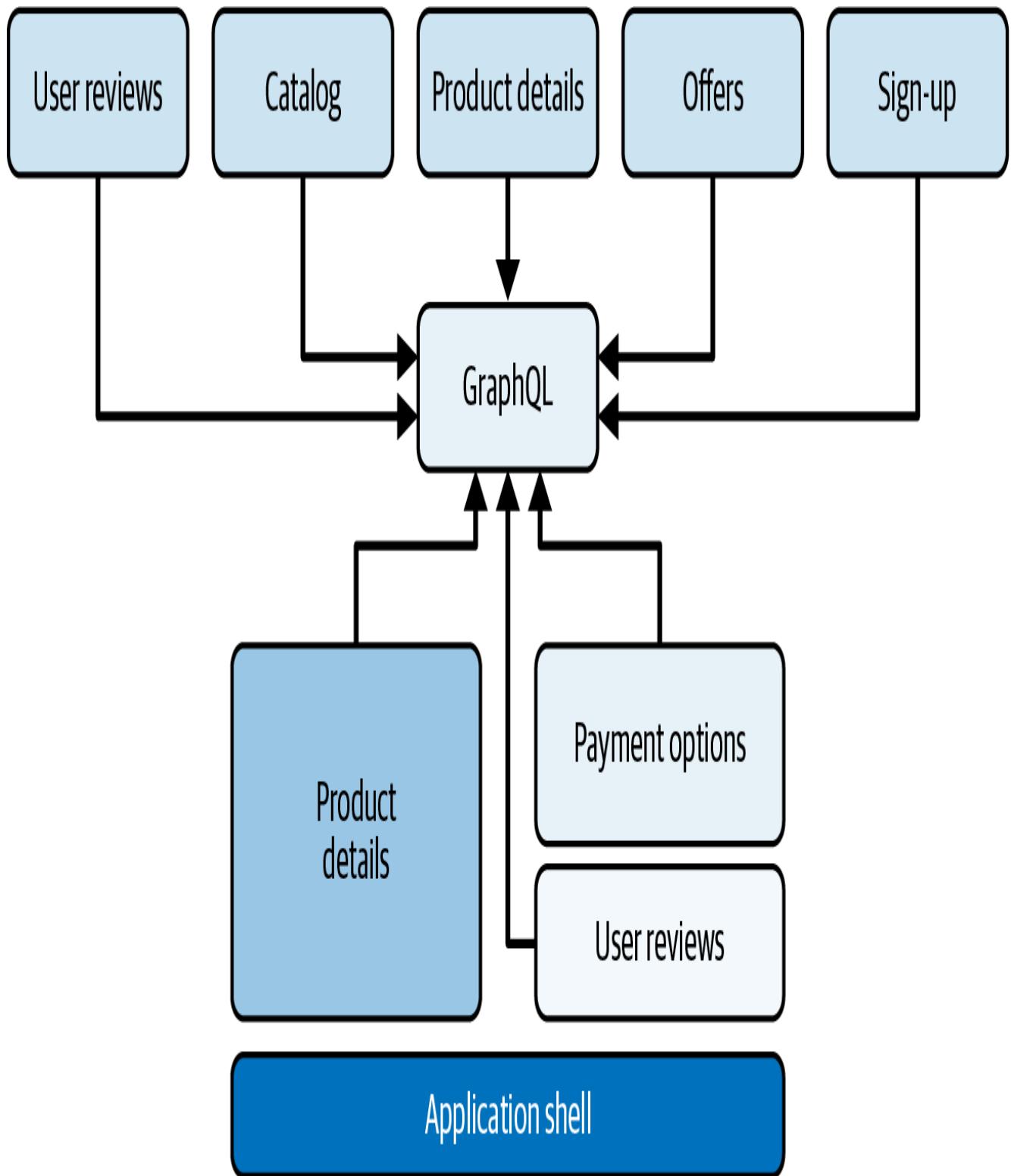


Figure 9-19. High-level GraphQL architecture with schema federation, where horizontal-split micro-frontends query the graph layer in client-side composition

When we have multiple micro-frontends in the same or different views performing the same query, it's wise to consider the query and response cacheability at different levels—such as the [CDN used](#)—and otherwise leverage the GraphQL server-client cache.

Caching is a very important concept that must be leveraged properly; doing so can protect your origin from burst traffic, so invest the time. Even when we have dynamic data, caching for tens of seconds or a few minutes helps reduce the strain on the origin and lowers the risk of failures.

Using GraphQL with Micro-Frontends and a Server-Side Composition

The final approach involves using a GraphQL server with a micro-frontend architecture, featuring a horizontal split and server-side composition.

When the UI composition requests multiple micro-frontends to their relative microservices, every microservice queries the graph and prepares the view for the final page composition, as shown in [Figure 9-20](#).

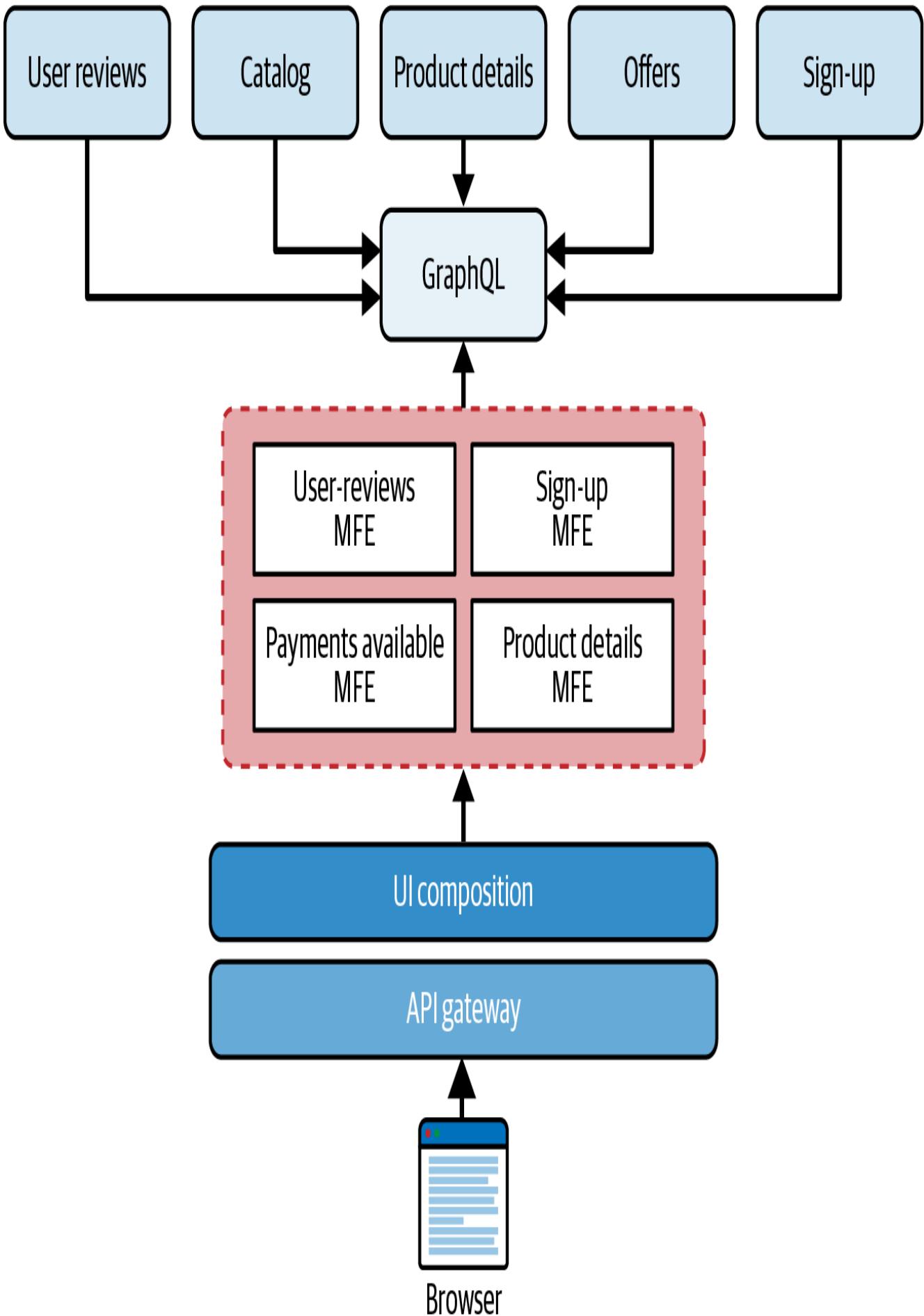


Figure 9-20. A high-level micro-frontend architecture with a server-side composition, where every micro-frontend consumes the graph exposed by the GraphQL server

In this scenario, every microservice that queries the GraphQL server requires access to the unique entry point, self-authentication, and retrieval of the data needed for rendering the micro-frontend requested by the UI composition layer. This implementation overlaps quite nicely with the API gateway and BFF patterns.

Best Practices

After discussing how micro-frontends can fit with multiple backend architectures, we must address some topics that are architecture-agnostic but could help with the successful integration of a micro-frontend architecture.

Multiple Micro-Frontends Consuming the Same API

When working with horizontal-split architecture, we might encounter situations where similar micro-frontends exist within the same view, consuming identical APIs with the same payload. This scenario could lead to an increase in backend traffic, necessitating more complex solutions for managing the traffic and its associated costs.

In such instances, it's crucial to question whether maintaining separate micro-frontends truly adds value to our system. Is grouping them into a single, unified micro-frontend a more effective approach?

A possible solution is to transform the micro-frontends into components and consolidate them within a single micro-frontend, as depicted in [Figure 9-21](#).

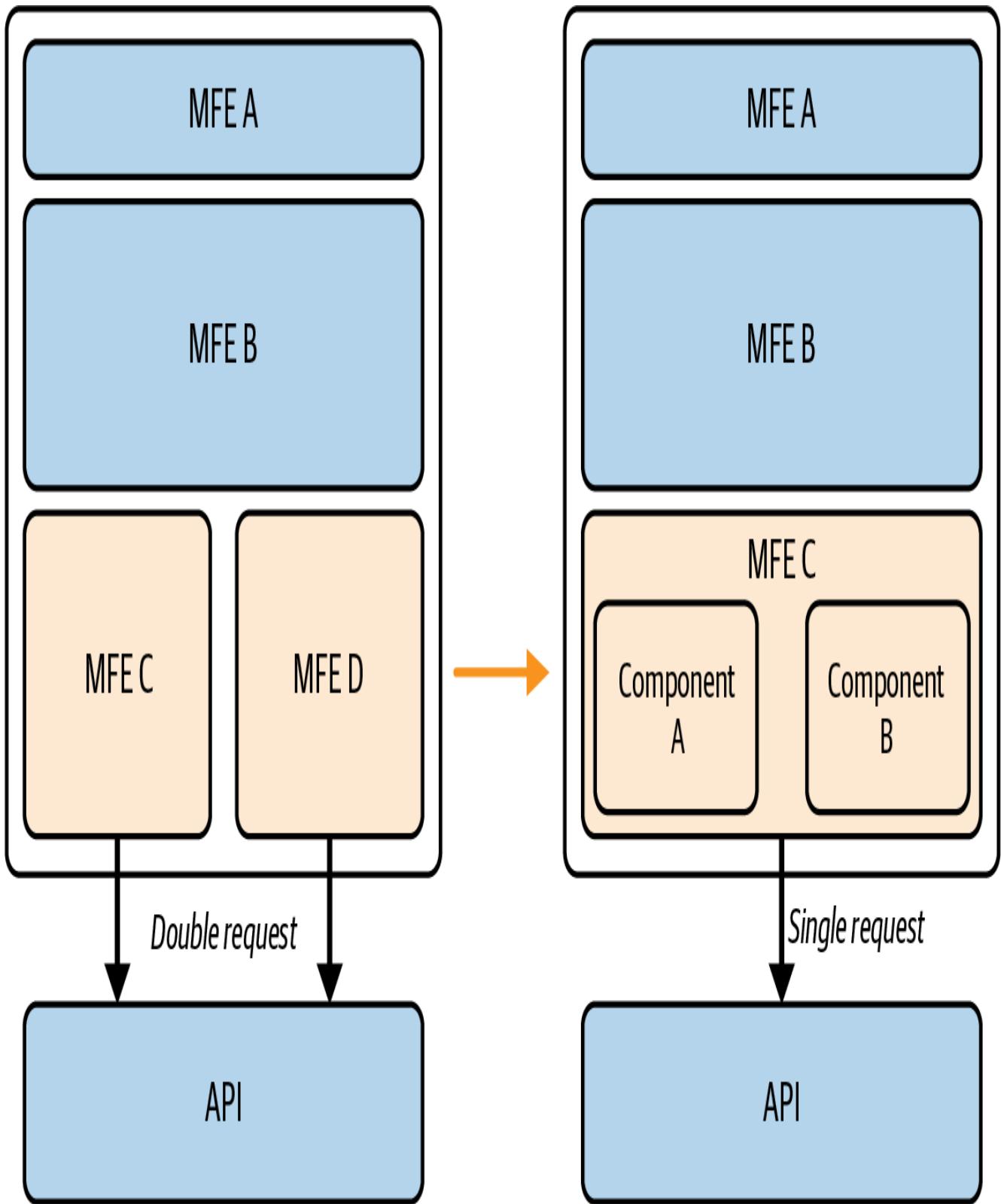


Figure 9-21. Multiple micro-frontends consuming the same API with identical payloads, with a single micro-frontend injecting the response into new components to reduce the server chattiness

In this scenario, the micro-frontends will execute a single request to the API, which will then inject the response into the components within the page, as we are accustomed to implementing with other architectures. These components can be easily imported by the

micro-frontends as an npm library, maintaining clear boundaries and reducing redundant API calls.

Additionally, consider reassessing team ownership. Implementing this solution may increase the team's cognitive load because of the new micro-frontends containing additional components and handling more business requirements.

Typically, such situations should prompt consideration for architectural enhancement. Do not overlook this signal; instead, reassess the decisions made at the project's outset with the available information and context, ensuring that it's acceptable to make duplicate API requests within the same view. If not, be prepared to reevaluate the boundaries of the micro-frontends.

APIs Come First, Then Implementation

Independent of the architecture that we implement in our projects, we should apply API-first principles to ensure all teams are working with the same understanding of the desired result.

An API-first approach means that for any given development project, your APIs are treated as “first-class citizens.”

As discussed at the beginning of this book, we need to make sure the APIs identified for communication between micro-frontends or for client-server communication are defined up front to enable our teams to work in parallel and generate more value in less time. In fact, investing time early on to analyze the API contract with different teams will reduce the risk of developing a solution that is not suitable for achieving the business goals or integrating smoothly within the system. Gathering all the teams involved in the creation and consumption of new APIs can save a lot of time further down the line when the integration starts.

At the end of these meetings, producing an API spec with mock data enables teams to work in parallel.

The team responsible for developing the business logic will have clarity on what to produce and can create tests to ensure the expected result, while the teams consuming the API can begin integration, evolving or developing the business logic using the mocks defined during the initial meeting.

Moreover, when a breaking change must be introduced in an API, sharing a [request for comments \(RFC\)](#) with the teams consuming the API may help to update the contract in a collaborative way. This will provide visibility on the business requirements, and it

allows everyone to share their thoughts and collaborate on the solution using a standard document for gathering comments.

RFCs are very popular in the software industry. Using them to document API changes allow us to scale knowledge and communicate the reasoning behind certain decisions, especially with distributed teams where it is not always possible to schedule a face-to-face meeting in front of a whiteboard.

RFCs are also used when we want to change part of the architecture, introduce new patterns, or modify the infrastructure. Beyond facilitating alignment, they create a valuable paper trail of architectural decisions—an essential resource for onboarding new engineers and preserving organizational context over time.

API Consistency

Another challenge we need to overcome when we work with multiple teams on the same project is creating consistent APIs by standardizing several aspects, such as error handling. API standardization allows developers to easily grasp the core concepts of new APIs, minimizing the learning curve and making the integration of APIs from other domains easier. A clear example would be standardizing error handling so that every API returns a similar error code and description for common issues like incorrect body requests, unavailable services, or API throttling.

This is true not only for client-server communication but also for micro-frontends. Let's consider the communication between a component and a micro-frontend, or between multiple micro-frontends in the same view. Identifying the event schema and the possibilities we allow inside our system is fundamental for ensuring application consistency and for speeding up the development of new features.

There are very interesting insights available online for client-server communication, some of which may also be applicable to micro-frontends. [Google](#) and [Microsoft](#) API guidelines share a well-documented section on this topic, with many details on how to structure a consistent API inside their ecosystems.

WebSocket and Micro-Frontends

In some projects, we need to implement a WebSocket connection to notify the frontend that something is happening, such as in a video chat application or an online game.

Using WebSockets with micro-frontends requires a bit of attention because we may be tempted to create multiple socket connections—one per micro-frontend. Instead, we

should create a unique connection for the entire application and inject or make available the WebSocket instance to all the micro-frontends loaded during a user session.

When working with horizontal-split architecture, create the socket connection in the application shell and communicate any messages or status changes (error, exit, and so on) to the micro-frontends in the same view via an event emitter or custom events for managing their visual update. In this way, the socket connection is managed once instead of multiple times during a user session.

However, there are some challenges to take into consideration. Imagine that some messages are communicated to the client while a micro-frontend is loaded inside the application shell. In this case, creating a message buffer may help to replay the last few messages and allow the micro-frontend to catch up once fully loaded.

Finally, if only one micro-frontend needs to listen to a WebSocket connection, encapsulating this logic inside that micro-frontend would not cause any harm because the connection will live naturally inside its subdomain.

For vertical-split architectures, the approach is less definitive. We may want to load the socket connection inside each micro-frontend instead of at the application shell, simplifying the life cycle management of the connection.

The Right Approach for the Right Subdomain

Working with micro-frontends and microservices provides a level of flexibility that we didn't have before.

To leverage this new capability within our architecture, we need to identify the right approach for the job.

For instance, in some parts of an application, we may want micro-frontends communicating with a BFF instead of a regular service dictionary, because that specific domain requires an aggregation of data retrievable from existing microservices, but the data should be aggregated in a completely different way.

With microarchitectures, these decisions are easier to embrace due to the architecture's intrinsic characteristics. To achieve this flexibility, we must invest time at the beginning of the project to analyze the boundaries of every business domain and then refine whenever we see complications arise in API implementation.

In this way, every team will be empowered to use the right approach for the job instead of following a standard approach that may not apply to the solution they are developing.

This is not a one-off decision; it should evolve and be revised with a regular cadence to support business evolution.

Summary

In this chapter, we have covered how micro-frontends can be integrated with multiple API layers.

Micro-frontends are suitable for working not only with microservices but also with monolithic architecture.

There may be strong reasons why we cannot change the monolithic architecture on the backend, but we want to create a new interface with multiple teams. Micro-frontends can provide the solution to this challenge.

We discussed the service dictionary approach, which can help with cross-platform applications and reduce the need for a shared client-side library that gathers all the endpoints. We also discussed how BBF can be implemented with micro-frontends, along with a different twist on BFF that uses API gateways.

In the last part of this chapter, we reviewed how to implement GraphQL with micro-frontends, discovering that the implementation overlaps quite nicely with the API gateway and BFF patterns.

Finally, we closed the chapter with some best practices, such as approaching API design with an API-first approach and leveraging DDD at the infrastructure level to apply the right technical approach for each subdomain.

As we have seen, micro-frontends offer different implementation models based on the backend architecture we choose. The quickest approach to start integrating a new micro-frontend project is the service dictionary, which can evolve over time into more sophisticated solutions like BFF or GraphQL.

Remember that every solution shared in this chapter brings a fair amount of complexity if not carefully analyzed and contextualized within the organization's structure and communication flow. Don't focus your attention only on the technical implementation—take a step further and consider the governance for future API integrations or breaking changes.

Chapter 10. Common Antipatterns in Micro-Frontend Implementations

Over the past 10 years, I've had the privilege of guiding hundreds of teams worldwide through their micro-frontend journeys, witnessing both triumphs and pitfalls firsthand. These experiences, coupled with insights from discussions with industry experts, have shed light on crucial aspects often overlooked: organizational capabilities, platform considerations, and the sociotechnical aspects of distributed systems in general.

While many teams eagerly dive into micro-frontend implementations—looking to the newest framework or the latest library to incorporate into their projects—a hard truth emerges: the technical implementation is merely the tip of the iceberg (see [Figure 10-1](#)). The real challenges lie beneath the surface, in the realms of organizational dynamics, communication patterns, and architectural decision making.

Let's delve into the most common micro-frontend antipatterns, focusing on the technical pitfalls that can derail even the most promising projects. These antipatterns are the ones I've encountered most frequently with teams embracing this approach for the first time or struggling to progress with implementation due to the constant friction they create. You may have encountered some of these antipatterns in previous chapters, but the collection in this chapter will offer a more in-depth exploration of each one.

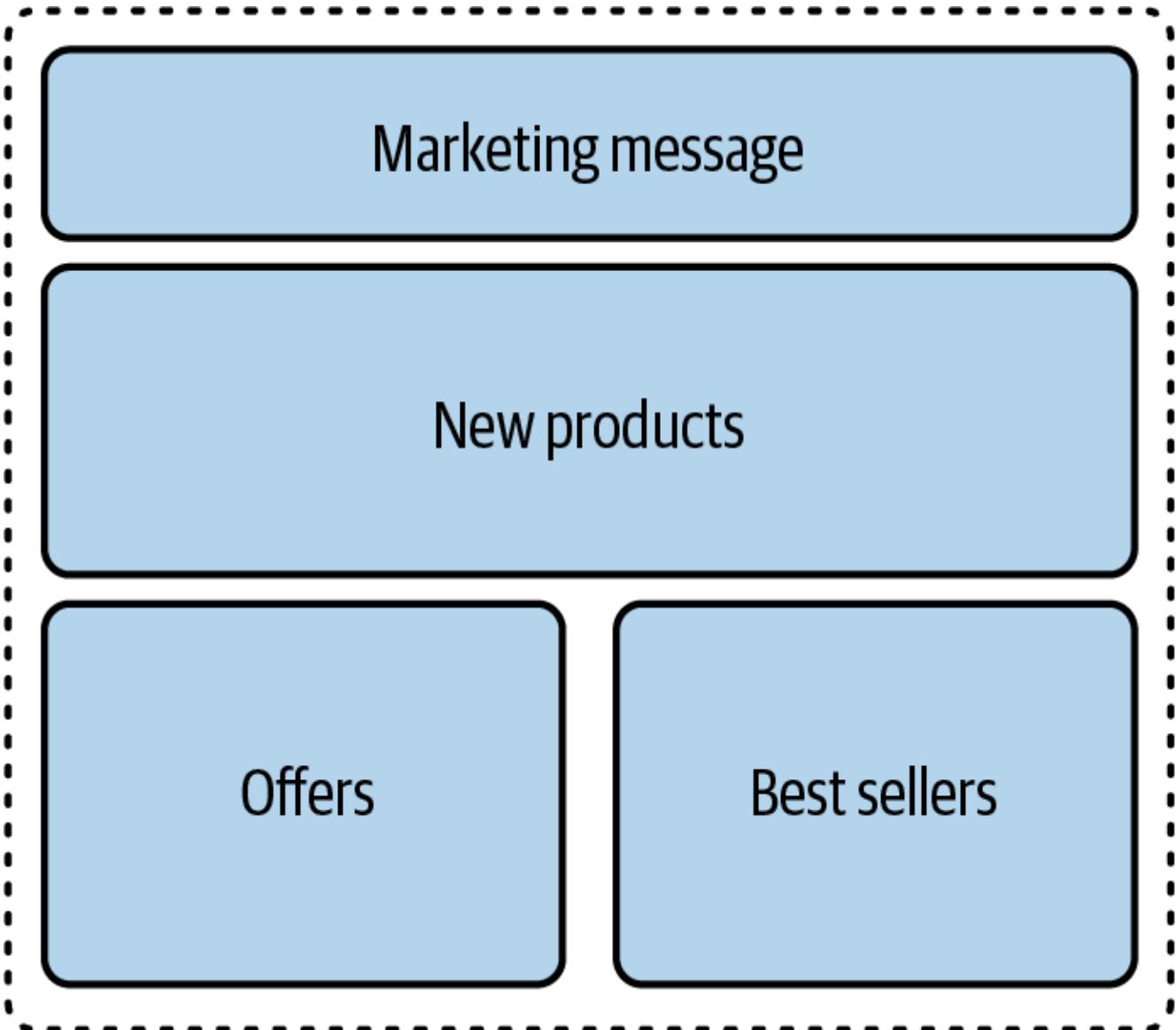


Figure 10-1. Technical implementation as just one aspect to consider when embracing micro-frontend architectures

We'll examine common mistakes that have emerged across different projects and industries. Understanding these pitfalls will help you to navigate the complexities of micro-frontend architectures more effectively. While we'll focus mainly on technical aspects here, remember that organizational and strategic factors—which we'll discuss in later chapters—are just as crucial for success.

Please also remember that culture, organizational structure, and software architecture are closely related. When one of these dimensions shifts in a different direction, the other two will inevitably be impacted. The effects might not be immediate, but they will eventually surface. Therefore, always consider these three dimensions before making any foundational decision.

In this chapter, we'll explore specific antipatterns as well as guidance on how to avoid them to ensure a more successful micro-frontend implementation.

Micro-Frontend or Component?

Picture this: you've decided to embrace micro-frontends for your next project, and now you're faced with the crucial task of defining the boundaries of your independent units. Where do you begin? How "micro" should a micro-frontend really be?

This is a common challenge encountered by every organization transitioning to a micro-frontend architecture. I completely understand the complexity of this problem, but rest assured that there are effective strategies to identify and test appropriate boundaries for micro-frontends without confusing them with mere components.

We explored some helpful heuristics in [Chapter 2](#), but I'd like to discuss a more concrete example here. Determining the right granularity for micro-frontends is one of the most frequent and costly pitfalls in their design and implementation. When boundaries are too fine-grained, teams often face death by governance, endless coordination, pipeline sprawl, and operational overhead—all outweighing the benefits of modularity.

Take a look at the home page of an ecommerce platform shown in [Figure 10-2](#).

Header

Hero product

Product A

Product B

Product C

Experience carousel

90-days-return-policy banner

Where to buy

Support

Footer

Figure 10-2. An ecommerce home page that wants to move from a monolithic approach to micro-frontends

An ecommerce home page is a quintessential example of a user interface that typically encompasses multiple domains within an organization. The complexity of these domains can vary, but let's explore how we might divide this interface into micro-frontends.

In this scenario, the team has opted for a horizontal-split approach for the home page. This decision was driven by the need to involve multiple teams in the delivery of this page, allowing for parallel development and domain-specific expertise.

To begin, let's identify the distinct business domains that are present within this view. This exercise will not only help us define our micro-frontend boundaries but also clarify which teams will be responsible for each domain ([Figure 10-3](#)).

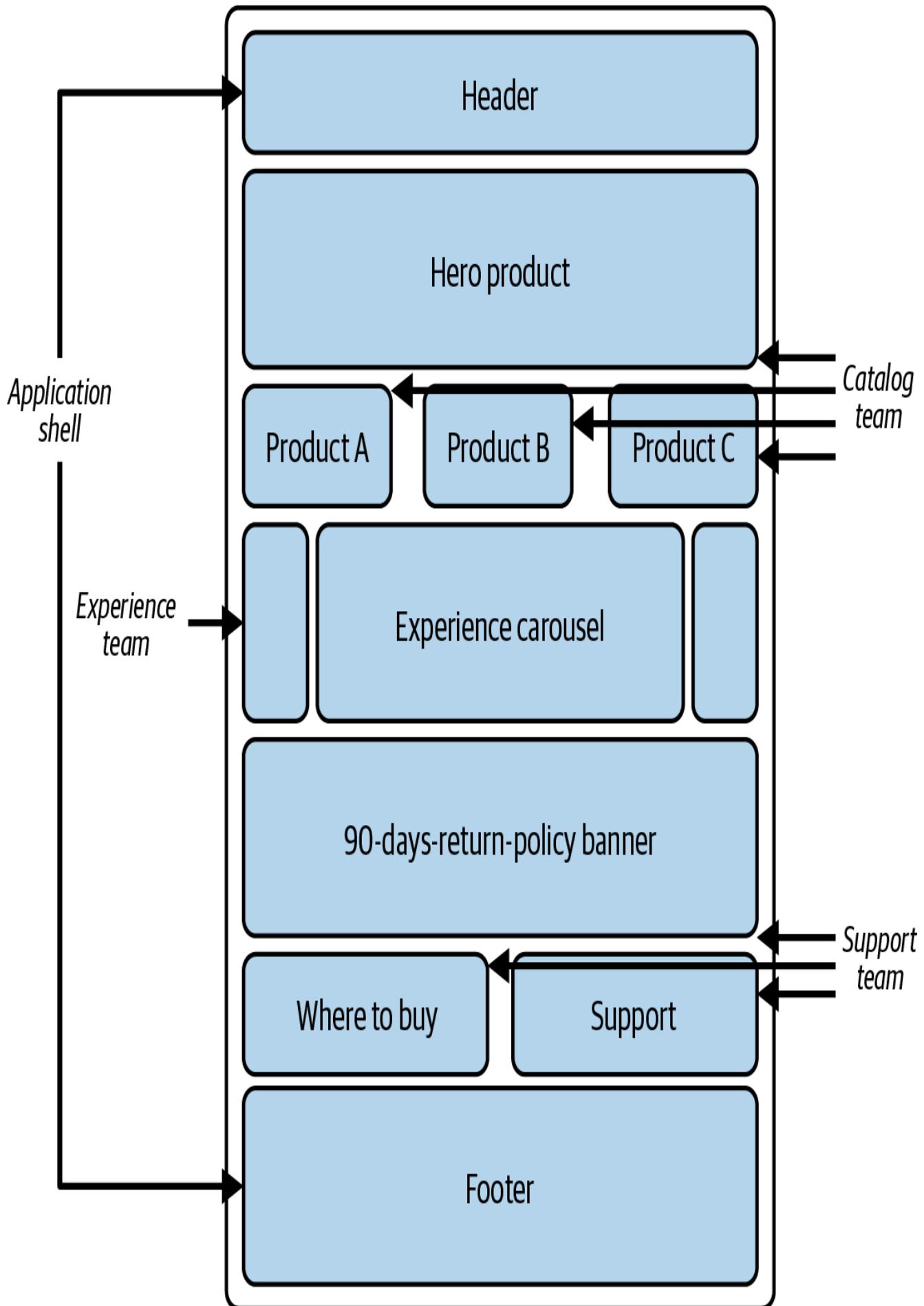


Figure 10-3. Three domains, plus the application shell for composing the final view

The application shell contains the header and footer—elements that remain consistent across the entire website. After analyzing the GitHub history, the team discovers that these components have a low rate of change—typically once or twice a year. The changes are generally straightforward to implement, and extracting them as a shared library or loading them at runtime would introduce unnecessary complexity. Given their stability, the team decides to keep these components within the application shell, avoiding the need for additional governance and automation pipelines for rarely changing elements.

Beyond the shell, three primary domains emerge in this view: catalog, experience, and support. The catalog domain is represented by components showcasing the latest available products. While these could technically be encapsulated as individual micro-frontends, doing so might lead to oversegmentation without clear benefits. Consider the three product displays beneath the hero-product component. Although they're replicas with consistent behavior and aesthetics, separating them into independent, runtime-deployed micro-frontends would likely introduce more complexity than advantages—due to needing different automation pipelines, cross-team coordination, and so on.

These catalog components are best assigned to a single team responsible for catalog pages. This approach leverages domain expertise, allowing the team to align closely with product and business requirements for optimal product display.

A similar approach applies to the experience and support domains. As these components typically redirect users to their respective domain areas, it's logical to have domain experts create and maintain the associated micro-frontends that contain the components. This example illustrates how identifying domains within a view can aid in understanding ownership and distinguishing between micro-frontends and components. The decision-making process is informed by business requirements, change frequency, and domain ownership.

It should be clear that a page with multiple components often translates to fewer micro-frontends than one might initially assume. When using a horizontal-split approach, having more than five to seven micro-frontends on a single page often indicates overly fine-grained division. This can lead to domain leakage between micro-frontends, creating deployment coupling and external dependencies across teams.

Remember, these boundaries should evolve with your application. When friction arises, review whether the identified boundaries still suit your application's needs, or if

aggregation or further splitting is necessary. Distributed systems are living entities that require nurturing and attention, much like children learning to manage their emotions.

Sharing State Between Micro-Frontends

This issue frequently arises when new teams adopt micro-frontends. While we're used to designing single-page applications (SPAs) by first selecting a UI library and then choosing a suitable state management library, this approach may not be optimal for distributed systems. This antipattern typically emerges in horizontal splits, where micro-frontends within the same view need to communicate in response to user actions or external API signals via sockets or server-sent events.

Sharing, in this context, is a form of coupling akin to design-time coupling in microservices—i.e., the extent to which one service must change due to alterations in another service. This coupling occurs when one service directly or indirectly depends on concepts owned by another service.

Similarly, in micro-frontends, we may encounter situations where modifying a data type in the shared state necessitates retesting multiple micro-frontends on the same page to ensure functionality. This scenario also introduces coordination challenges between teams when implementing breaking changes in the shared state. Consider a deployment scenario where other teams rely on the data structure of the state manager. A breaking change by one team would require coordination with all affected teams to ensure compatibility, both with current production versions and future iterations of their micro-frontends. If incompatibility arises with the current production version, it may necessitate adding tasks to other teams' backlogs and waiting for them to incorporate the changes, which could potentially cause delays and complications.

A more efficient approach would be to implement a mechanism that keeps micro-frontends aligned, allowing them to react to user interactions or state changes in other micro-frontends. This method, shown in [Figure 10-4](#), could potentially mitigate the challenges associated with shared state management in micro-frontend architectures.

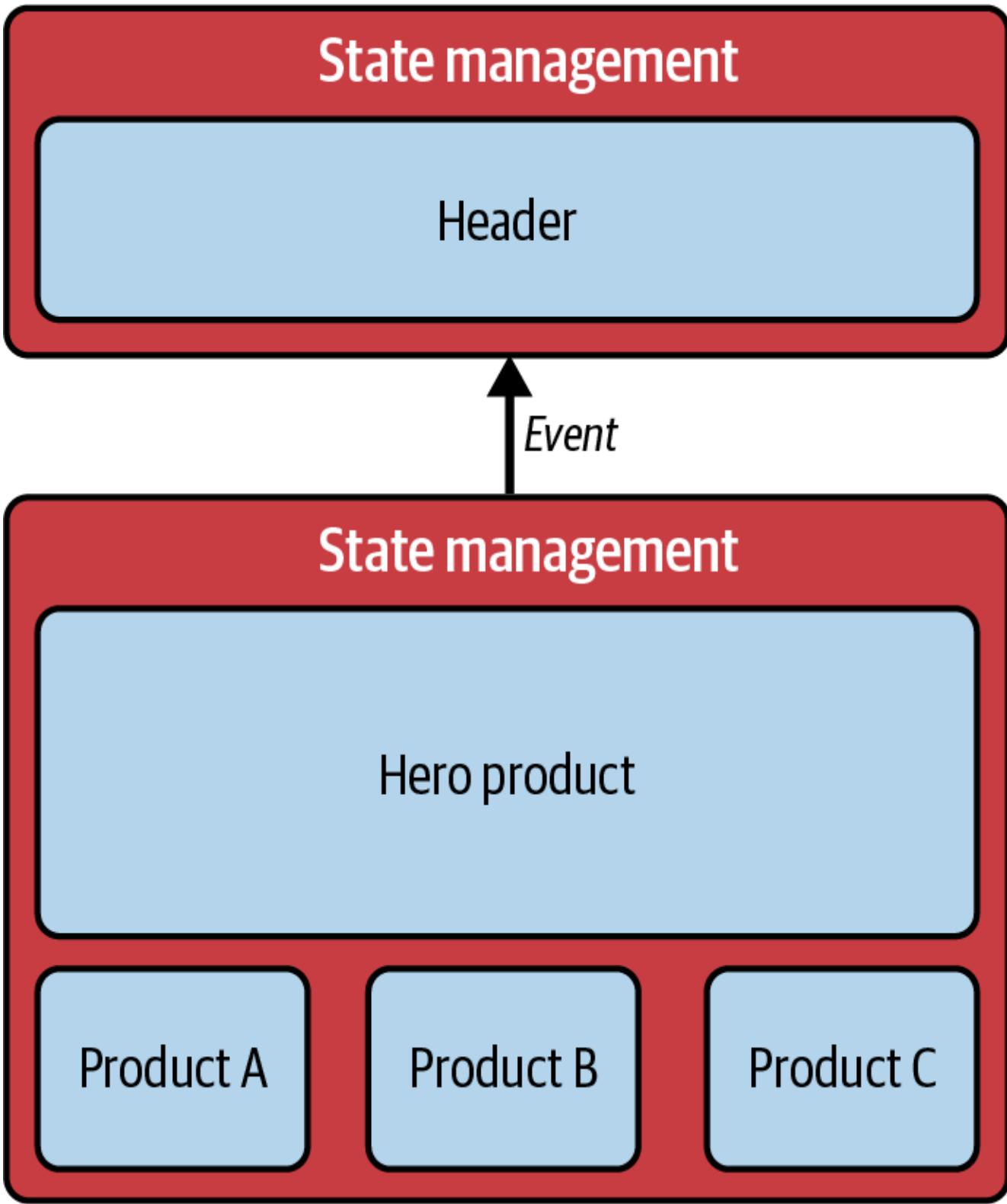


Figure 10-4. Two micro-frontends communicating via events and maintaining their internal state to reach low coupling and independence

The pub/sub pattern embodies this concept. This messaging pattern is widely implemented in software architecture to facilitate asynchronous communication between various system components.

To maintain the independence of micro-frontends within the same view, it's highly recommended to encapsulate the state manager within each micro-frontend. In this way, the micro-frontend can be evolved independently without the need to coordinate changes across multiple teams. Communication between these micro-frontends can then be established using mechanisms such as event emitters or custom events. Event emitters are generally preferred over custom events due to their DOM-agnostic nature, which offers significant advantages in micro-frontend architectures. Custom events, by design, bubble through DOM elements up to the `window` object. This behavior can lead to complications if event propagation is inadvertently prevented at any point in the DOM tree. In such cases, identifying and debugging the issue can become a challenging and time-consuming task.

In contrast, event emitters operate on a subscription model, where components directly subscribe to and notify subscribers. This approach is independent of the DOM structure, providing greater flexibility and reliability. As a result, you can freely move your micro-frontend within the DOM tree without risking event loss or necessitating code refactoring.

This DOM-agnostic quality of event emitters enhances the modularity and maintainability of your micro-frontend architecture. It allows for more robust communication between components, reducing the likelihood of unforeseen issues related to DOM structure changes. Ultimately, using event emitters can lead to a more resilient and easier-to-maintain system, particularly in complex micro-frontend environments where component placement and interaction are critical considerations. This approach preserves the autonomy of individual micro-frontends while allowing for necessary interactions.

However, it's crucial to exercise caution regarding the frequency and volume of communication across micro-frontends. Excessive inter-micro-frontend communication often indicates improperly defined boundaries within the view. Striking the right balance is essential for maintaining a clean and efficient architecture. When implementing this pattern, carefully consider the appropriate level of interaction to ensure optimal system design and functionality.

Micro-Frontend Anarchy

This antipattern typically manifests when organizations embrace the idea of independent development without establishing proper guidelines, standards, or governance

structures. Sometimes, organizations want to optimize their architecture for a multi-framework approach without realizing the long-term impact of this decision.

Let me ask you this question: would you use a multi-framework approach for an SPA? Your answer is probably no—but technically, you could!

In a micro-frontend “anarchy” scenario, different teams within an organization start developing their parts of the application with complete autonomy. While this autonomy can foster innovation and enable teams to move quickly, it can also lead to a fragmented and inconsistent application architecture.

In the absence of clear guidelines, teams may adopt a wide array of technologies, frameworks, and libraries. While this diversity can be beneficial in some cases, it often leads to a chaotic tech stack. You might find one team using React, another using Vue.js, and yet another opting for Angular—all within the same application. This proliferation of technologies can result in an increased bundle size due to multiple framework libraries being loaded and an inconsistent user experience across different parts of the application.

As the application grows and evolves, the maintenance challenges posed by an anarchic micro-frontend approach become increasingly apparent. The fragmented nature of the architecture introduces significant hurdles in managing and updating the system as a whole. Implementing application-wide changes or upgrades becomes a daunting task, requiring coordination across multiple teams and potentially incompatible technologies. Debugging issues that span multiple micro-frontends grows in complexity, often requiring developers to navigate through disparate codebases and technologies to identify and resolve problems.

Maintaining consistent security practices across all micro-frontends becomes a concern, as each team may implement security measures differently, potentially creating vulnerabilities in the overall application. The integration and testing of micro-frontends developed with different technologies introduce additional complications, requiring sophisticated testing strategies and potentially custom integration solutions.

One of the often-overlooked consequences of the micro-frontend anarchy antipattern is its impact on knowledge sharing and developer mobility within an organization. As teams become deeply entrenched in their chosen technologies, knowledge silos begin to form. This fragmentation leads to several significant challenges in the development process. Sharing best practices across teams becomes increasingly difficult due to the differing technological contexts, limiting the spread of valuable insights and innovations.

The ability to move developers between teams is greatly reduced, as each micro-frontend requires a unique skill set, hampering organizational flexibility and career growth opportunities. This fragmentation of knowledge not only impacts the efficiency of the development process but can also lead to a sense of isolation among teams. The resulting silos can potentially affect morale and hinder collaboration, as developers may feel disconnected from the broader organizational goals and their peers working on other parts of the application. Finally, this knowledge fragmentation can undermine the very benefits that micro-frontends were intended to provide, such as increased agility and improved team autonomy.

However, a multi-framework approach in micro-frontends can be a valuable strategy in specific scenarios, particularly during transitional phases in an organization's technical evolution. This approach, while not ideal for long-term implementation, offers significant benefits in certain contexts. One prime scenario for leveraging a multi-framework approach is during the migration of a monolithic system to a micro-frontend architecture. In this case, organizations can gradually transition different parts of their application to micro-frontends while maintaining the existing system. This incremental approach allows for continuous delivery of value to customers without the need for a complete system overhaul, which could potentially take months or even years.

Similarly, when an organization decides to migrate from one UI framework to another—such as from an old version of Angular to a new one—a multi-framework approach can be instrumental. It enables teams to begin developing new features or refactoring existing ones using the new framework while keeping the rest of the application functional in the original framework. This strategy allows for a smoother transition, minimizing disruption to the user experience and maintaining business continuity.

Moreover, this approach provides a safety net during the transition period. If critical issues arise in the newly developed micro-frontends, teams can quickly revert to the original implementation. This fallback option reduces risk and provides peace of mind during the migration process, allowing teams to be more adventurous in their approach to new technologies and architectures.

However, it's crucial to view the multi-framework approach as a temporary solution rather than a long-term architectural strategy. Organizations should have a clear plan to eventually converge on a more standardized approach to avoid the pitfalls associated with long-term maintenance of diverse technologies within the same application.

In conclusion, while a multi-framework approach in micro-frontends comes with its challenges, it can be a powerful tool for organizations undergoing significant

technological transitions. By enabling gradual migration, continuous delivery, skills development, and risk mitigation, this approach can bridge the gap between legacy systems and modern architectures, ensuring that businesses can evolve their technology stacks without compromising their ability to deliver value to customers.

Anti-Corruption Layer to the Rescue

Imagine that you are tasked with integrating an existing application into your brand-new micro-frontend architecture. The main issue is that you have to integrate it in just a month.

The application shell is framework-agnostic; however, the decision was to use an event emitter for communication across micro-frontends, and the existing web application was not designed with micro-frontends in mind. Because of time constraints and limited understanding of the impact of loading the application from the shell, you decide to use an iframe to isolate the code and avoid potential runtime errors when adopting the micro-frontend system.

In this case, the challenge is deciding whether the application shell should support iframe communication via the `postMessage` API or continue working with the event emitter while refactoring the existing application—a task whose complexity the team does not yet fully understand.

Another factor to consider is that, at some point in the future, the existing application will be refactored into micro-frontends, meaning the code to support the iframe in the application shell will eventually be thrown away.

This approach might open the door to multiple methods of composing micro-frontends, creating long-term complexity. The application shell—which should be the most stable part of the system—would instead be subject to constant changes and new implementations, requiring every integration point to be tested on each update. Moreover, multiple teams could request to introduce other ways to compose micro-frontends, given that no single approach is established. As you can imagine, this seemingly simple decision could trigger difficult discussions in the future about how the architecture should evolve.

This problem is more common than you think. It happens when one company acquires another, or when the C-suite wants to create a more consistent experience for their customers or for internal systems. Teams usually jump straight away to the solution, without evaluating the trade-offs they will have to live with in the long run.

The question is, do we have any alternative? Luckily, the answer is: yes, we do. The alternative is called the anti-corruption layer (ACL).

The ACL pattern is an effective architectural strategy that helps keep your systems clean and manageable, especially when integrating different technologies or legacy systems. Traditionally associated with backend architectures, the ACL acts as a translator and mediator, ensuring that the complexities and potential inconsistencies of one system do not “corrupt” the integrity of another. This pattern is especially valuable when dealing with legacy systems that cannot be easily modified or replaced, allowing new systems to evolve independently while maintaining necessary interactions.

In micro-frontends, we can use the ACL by creating a wrapper around the iframe that contains the existing application, leveraging `postMessage` communication between the anti-corruption layer and the iframe, while using the event emitter for communication between micro-frontends and the shell (see [Figure 10-5](#)).

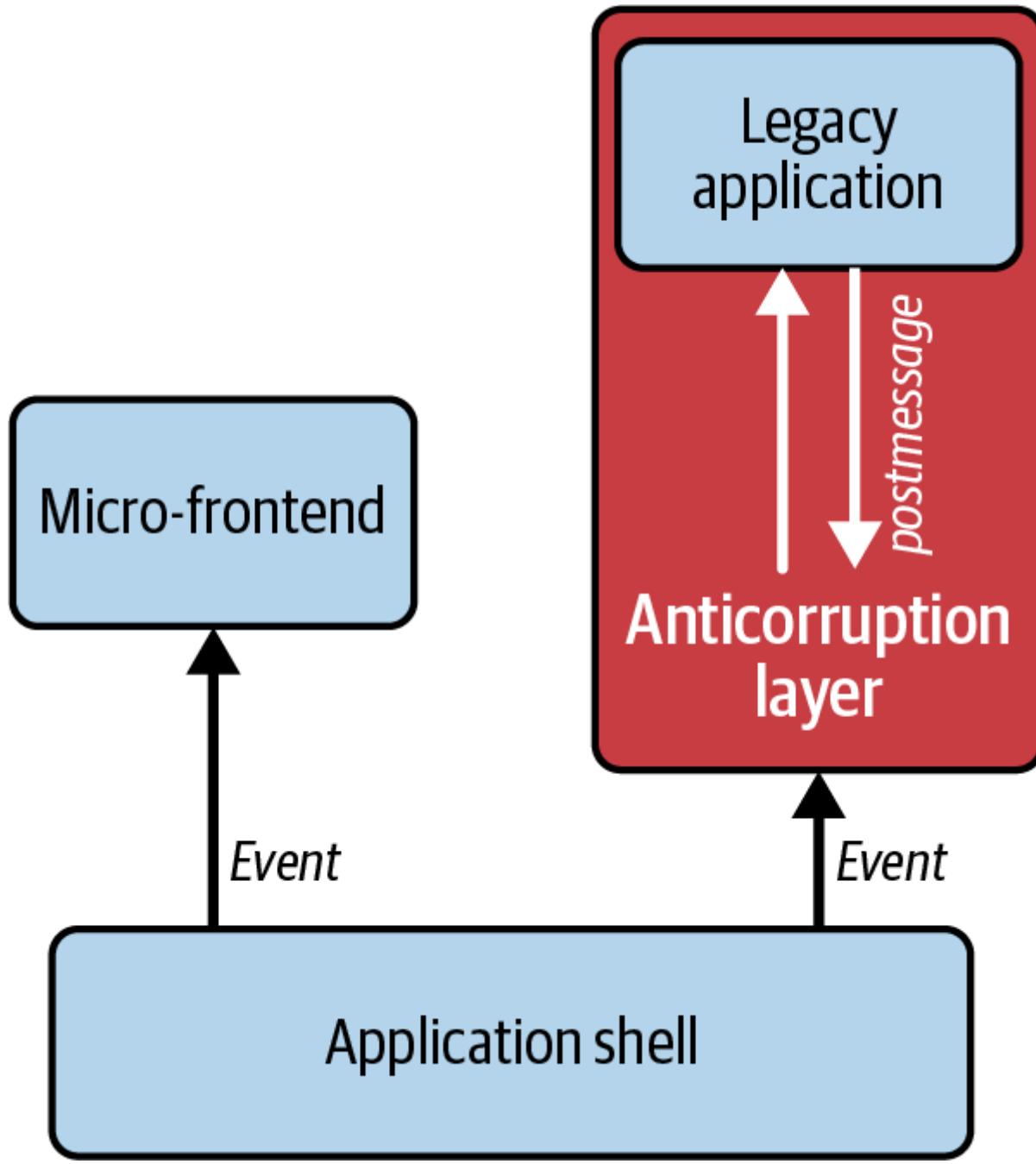


Figure 10-5. Application shell codebase remains the same across all micro-frontends; legacy system isolated in an iframe, communicating with its wrapper via postmessage API

In this way, the application shell codebase will remain the same for all micro-frontends. The wrapper around the legacy system will sanitize communication between the application shell, other micro-frontends, and the iframe without leaking implementation details. Finally, when the legacy application is refactored, the only change needed will be to load a new micro-frontend endpoint instead of the anti-corruption layer. This is a very neat approach that is evolutionary by design.

One of the biggest perks of adopting an ACL in your frontend architecture is the consistency it brings to your application. By establishing a clear interface for all

external interactions, you create a stable foundation that makes it easier to maintain and evolve your code over time. Plus, if you ever need to update or replace an external service, you can do so without disrupting the rest of your application. This flexibility is invaluable in today's fast-paced development environment, where change is the only constant.

Unidirectional Sharing

The next antipattern occurs when we treat micro-frontends like components that can be freely shared across the system. Even when we want to reuse components in real time and share them across other micro-frontends, we always have to bear in mind how we are sharing our data and resources.

Bidirectional or omnidirectional sharing across micro-frontends can quickly become a double-edged sword. While it might seem convenient at first, it often leads to a tangled web of dependencies that can stifle development speed and introduce unexpected bugs. The core issue lies in the increased communication and coordination burden placed on development teams. When multiple micro-frontends share data and functionality in multiple directions, changes in one component can have far-reaching and often unforeseen consequences on others. This scenario necessitates constant communication between teams, potentially leading to bottlenecks in the development process.

The challenges of bidirectional sharing extend beyond mere inconvenience; they can fundamentally undermine the very benefits that micro-frontend architecture aims to provide. One of the primary goals of adopting micro-frontends is to enable teams to work independently, allowing for faster development cycles and easier maintenance. However, when components are tightly coupled through bidirectional dependencies, this independence is compromised.

Moreover, consider the impact on testing and deployment. In a system with extensive bidirectional sharing, a change in one micro-frontend might require comprehensive testing across all interconnected components. This not only slows down the development process but also increases the risk of introducing bugs that are difficult to isolate and fix. Furthermore, it complicates the deployment process, as teams must coordinate their release schedules to ensure compatibility across all shared interfaces. The diagram in [Figure 10-6](#) shows how omnidirectional sharing can harm your deployment.

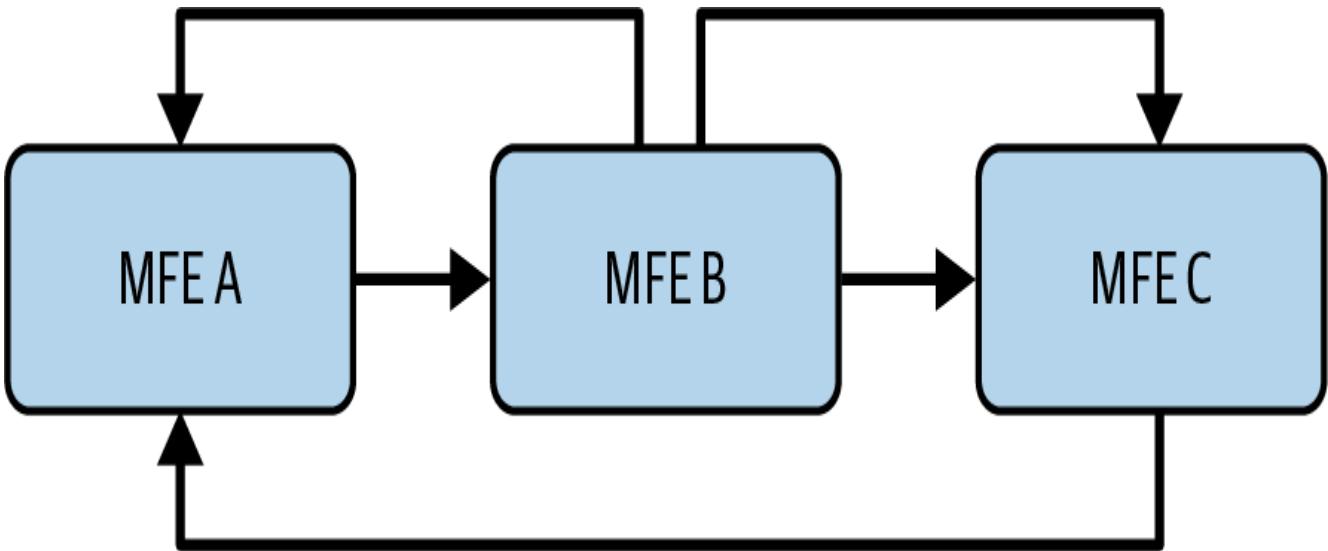


Figure 10-6. Omnidirectional sharing harming your deployment and rollback strategies due to shared dependencies

Another significant drawback is the potential for circular dependencies. When micro-frontends share data and functionality in both directions, it's easy to inadvertently create loops where component A depends on B, which in turn depends on C, which then depends back on A. Such circular dependencies are notoriously difficult to manage and can lead to runtime errors, performance issues, and debugging nightmares.

In contrast, a unidirectional flow of data from a parent component to its children would localize the impact of every change. Through dependency injection, the application shell will pass only the necessary data to each micro-frontend.

Moreover, unidirectional data flow promotes a clearer mental model of the application's structure. Developers can more easily reason about data changes and their effects when they know that information flows in only one direction. This clarity not only aids in development but also simplifies debugging and maintenance tasks.

Bear in mind that events with a pub/sub approach, like an event emitter, won't follow this rule because they are hierarchy-unaware. The pattern removes the idea of relations and lets micro-frontends subscribe to and be notified of events.

It's worth noting that adopting a unidirectional sharing approach doesn't mean completely isolating micro-frontends from each other. Instead, it encourages thoughtful design of component interfaces and promotes the use of well-defined APIs for necessary inter-component communication. This strategy strikes a balance between component independence and system cohesion, allowing teams to work autonomously while still creating a unified user experience.

By embracing unidirectional sharing from parent (the application shell) to child components (every micro-frontend), development teams can maintain the independence and flexibility that make micro-frontends so powerful, while still creating cohesive and feature-rich applications.

Premature Abstraction

Sharing or not sharing—this is the dilemma. When developing micro-frontends, it's crucial to approach abstraction with caution, particularly when considering whether to encapsulate specific functionality or utilities in a shared library. While abstraction can promote code reuse and maintainability, premature or restless abstraction can lead to unforeseen challenges. The real test isn't in the initial implementation, but in the long-term maintenance and evolution of these shared dependencies.

Kent C. Dodds—a full-time educator very well known for his contributions to React and Remix—captures this concept with his AHA programming principle, which stands for “avoid hasty abstractions.” This approach encourages developers to resist the urge to abstract code too quickly, instead allowing patterns to emerge naturally over time. By doing so, teams can create more robust and flexible abstractions that truly serve their needs.

In the context of micro-frontends, hasty abstractions in shared libraries can create a ripple effect of compatibility issues across multiple teams and components. For instance, consider a scenario where a team creates a shared date-formatting utility. Initially, it might seem like a straightforward abstraction that could benefit multiple micro-frontends. However, as different teams begin to rely on this utility, they may discover edge cases or require slight variations in formatting. Suddenly, what started as a simple utility becomes a complex, overengineered solution trying to accommodate every possible use case. When this library is used in multiple micro-frontends, you risk forcing other teams to update it quickly due to critical bugs or enhancements, without knowing how busy their backlogs are or if they can accommodate the update.

By embracing the AHA principle, teams can avoid these pitfalls. Instead of rushing to create shared libraries, they might start by duplicating code across micro-frontends where necessary. This approach allows each team to tailor the functionality to their specific needs while providing the opportunity to observe common patterns over time. Once clear and repeated use cases emerge across multiple micro-frontends, teams can create thoughtful and well-designed abstractions that truly serve the project's needs.

Remember, when making these decisions, the trade-offs and the intention you want to express when sharing the library must be taken into consideration. The goal is to strike a balance between code reuse and flexibility. By being mindful of when and how to abstract shared functionality in micro-frontends, teams can create more maintainable, evolvable systems that stand the test of time within changing requirements.

Summary

When embarking on a micro-frontend journey, it's essential to ensure that your architectural choices align seamlessly with your business objectives and system requirements.

To achieve this alignment, evaluate several critical factors with intent. You'll want to consider your application's scalability needs, assessing whether the projected growth justifies the added complexity that micro-frontends introduce. Equally important is your organization's ability to support autonomous teams for each micro-frontend, as this structure is fundamental to realizing the full benefits of this architecture.

By carefully weighing these aspects and understanding the associated trade-offs, you'll be well-equipped to make an informed decision about implementing micro-frontends in your system. Remember, there's no universal solution in architecture—the key lies in tailoring your approach to your specific business needs and technical constraints.

Chapter 11. Migrating to Micro-Frontends

Let's be honest: the idea of migrating a large, established application to micro-frontends can feel overwhelming. Maybe you've heard stories of migrations stretching on for months (or even years), sapping team energy and putting business goals on hold. It's no wonder so many teams hesitate to get started.

But it doesn't have to be that way!

One of the best things about distributed systems and micro-frontends in particular is that you don't have to do everything at once. In fact, the most successful migrations are usually *iterative*. You can start small, deliver value quickly, and keep your users happy along the way. With the right approach, you can see real business impact in just a few weeks.

For this chapter, I want to try something a little different. I've collected the most common questions I've heard from teams over the past five years. Questions about what to migrate first, how to avoid breaking things, and how to keep everyone on the same page. If you're wondering about something, chances are that someone else has wondered and asked about it before.

Most people don't read a technical book cover-to-cover (despite the fact that you should with this book). You might be flipping straight to the section you need right now. For this reason, I've organized this chapter around real questions, with clear answers and practical tips that you can use right away.

Why Go Iterative?

One of the strengths of micro-frontends is making change easier. By breaking your app into smaller, independent pieces, you give your teams more freedom to experiment, adopt new tech, and scale up smoothly. And best of all, you can migrate one piece at a time. No need for a risky, all-at-once "big bang."

Here's why an iterative migration is usually the way to go:

Faster wins

You can start delivering improvements to your users and stakeholders right away while creating momentum and a winning mindset within your teams.

Lower risk

Small, focused changes are easier to test and roll back if something goes wrong.

Continuous learning

Each step is a chance to learn what works (and what doesn't) before moving on. This will help to build or configure the different tools and practices we have been discussing in this book. Not everything has to be there on day one.

Maintaining business continuity

You don't have to freeze all new development while you migrate. You can keep building features and fixing bugs as you go.

To make this chapter as practical and actionable as possible, I've structured each question using a consistent format. Here's what you can expect for every topic:

Best practices and patterns

Actionable advice and proven strategies you can use right away, including real-world techniques that have worked for other teams

Trade-offs and pitfalls

An honest look at what could go wrong, common mistakes to watch out for, and the trade-offs you'll need to consider as you make decisions

Checklist

A quick set of questions or steps to help you apply the guidance to your own situation, so you can move from theory to action

A quick example

A scenario or mini case study to ground the advice in a real-world context and help you visualize how it might work in practice

Personal experience

Sharing stories from my own journey, where relevant—what worked, what didn’t, and what changed for me or my teams along the way

This structure is designed to help you quickly find the information you need, understand the reasoning behind each recommendation, and feel confident applying these lessons to your own migration.

The following is a list of questions we are going to cover in this chapter:

- Which problem(s) am I trying to solve with micro-frontends?
- How can I get the management on board with this migration?
- How should I actually plan the migration?
- How do I decide which modules or features to migrate first?
- How do I maintain user experience and consistency during migration?
- How do I handle shared dependencies and version management between legacy and micro-frontend code?
- How do I manage cross-cutting concerns like auth, routing, or state during migration?
- Do I need microservices to migrate to micro-frontends?

Which Problem(s) Am I Trying to Solve with Micro-Frontends?

Migrating is a significant investment, so make sure you’re solving real and pressing problems, not just following a trend. The main drivers often include team autonomy, faster release cycles, scalability, tech stack flexibility, and improved maintainability. If your current monolithic frontend is slowing down releases, making it hard for teams to work independently, or struggling to scale with your business, these are strong signals that micro-frontends could help in your context.

Best Practices and Patterns

Some best practices are as follows:

Team autonomy and parallel development

Micro-frontends let multiple teams own and release parts of your system independently, removing bottlenecks where one team's delay holds up everyone else.

Faster, safer releases

Independent deployment means you can release updates to one part of your app without risking the whole system. This isolation reduces the blast radius of bugs and makes rollbacks easier.

Scalability

As your organization grows, micro-frontends scale with you. Teams can be added or reorganized around business domains without stepping on each other's toes.

Improved maintainability

Smaller, focused codebases are easier to maintain, test, and refactor. Teams can specialize and take full ownership of their domains.

Faster onboarding

Because the system is split, the cognitive load for new team members is lower compared to a monolithic application. I've seen new teams contribute production-ready features within weeks instead of months.

Trade-Offs and Pitfalls

You also need to consider some pitfalls:

Increased complexity

Micro-frontends introduce new challenges in dependency management, debugging, and environment configuration. Without strong governance, you risk creating a "Frankenstein" user experience that doesn't benefit your users.

Overhead for small teams/apps

If your team or app is small, the overhead of micro-frontends may outweigh the benefits. Sometimes, simply splitting the app by routes or page groups can offer most of the advantages without the added complexity.

Shared code and dependencies

Managing shared libraries across micro-frontends can lead to larger bundles or version conflicts. Solutions like Module Federation or import maps can definitely help because they employ mechanisms to simplify dependency sharing with minimal configuration. Remember, in distributed systems, the goal is to reduce external dependencies and maintain a fast development flow. Starting with some duplication where it makes sense enables you to make more informed decisions about what to abstract later. Don't rush into premature abstractions—gather data before introducing shared dependencies.

Organizational readiness

Micro-frontends work best when your organization is ready for distributed ownership and has mature coordination and communication processes.

Checklist

Here are a few questions you should ask yourself:

- Is your frontend codebase large and growing?
- Do multiple teams need to work independently on different features?
- Are release cycles slowed down by dependencies between teams?
- Do you want to adopt new technologies without a full rewrite?
- Is scaling your engineering organization a priority?
- Are you struggling with performance due to large bundle sizes or slow builds?
- Is your business domain complex enough to justify splitting it into separate modules?

A Quick Example

Let's say you're working at a fast-growing ecommerce platform. Every new feature requires coordination across several teams, causing delays and frustration. Releases are infrequent and risky. The codebase is increasingly hard to maintain. Sound familiar?

By moving to micro-frontends, each product area—like the product catalog, check out, and user profile—becomes a separate module owned by an autonomous team. Releases speed up, teams can experiment with new technologies, and the platform scales more effectively as the business grows.

Personal Experience

I've personally experienced a 10-times increase in deployments after moving to micro-frontends. Implementing tools like canary releases, blue-green deployments, and feature flags enabled developers to deploy safely to production, leading to greater confidence and more testing with real users. We moved from a few releases per month to tens per micro-frontend per month—sometimes even per day!

Migrating to micro-frontends is best suited for organizations struggling with team autonomy, release speed, scalability, or tech stack flexibility. While the benefits can be significant, it's important to weigh them against the added complexity and ensure your organization is ready for the transition.

How Can I Get the Management On Board with This Migration?

From experience, selling an iterative migration to the business is far easier than pitching a risky “big bang” rewrite. Why? Because business stakeholders care about outcomes—faster time-to-market, lower risk, and delivering value to customers—more than the technical elegance of a fresh start.

When you tie your migration strategy directly to business goals, you build trust and buy-in, which is absolutely critical for success.

Best Practices and Patterns

Some best practices are as follows:

Speak the business language.

When pitching this approach, focus on outcomes that matter to stakeholders—such as faster releases that enable quicker response to market changes and customer needs, or improved ROI by spreading investment and value over time rather than waiting months or years for a payoff.

Start with a proof of concept.

Start with a small, high-impact feature or module as a pilot. This demonstrates value early, builds confidence, and gives you a template for the rest of the migration.

Trade-Offs and Pitfalls

Remember that with an iterative migration, you will gain immediate value, safer releases, shorter feedback cycles, and easier alignment with business milestones. However, you will face more complex integration (the legacy and new micro-frontends coexist for a while), which requires careful planning and communication. You need to avoid “split-brain” scenarios where two systems do the same thing for too long.

Remember to avoid focusing only on technical wins. Always connect migration steps to business goals, set clear milestones, celebrate progress to keep up the morale internally and, finally, make sure teams are aligned and understand the shared vision.

Checklist

Here are a few questions you should ask yourself:

- Have you mapped business goals to migration milestones?
- Can you identify a pilot area that will deliver visible value quickly?
- Do you have stakeholder buy-in for an incremental approach?
- Are you set up to measure and communicate progress regularly?
- Is there a clear plan for routing, integration, and retiring legacy code?

A Quick Example

Let's say you're working at a fast-growing streaming platform. Every new feature requires coordination across several teams, causing delays and frustration. Releases are

infrequent and risky. The codebase is increasingly hard to maintain.

Instead of pitching a full rewrite, you propose migrating incrementally. You start with the video catalog. With just three developers and three weeks, you build a micro-frontend version of the catalog that runs on web, PlayStation, and Tizen. Not only does it match the old functionality, but it also performs better than the monolith. This quick win demonstrates value, builds trust, and sets the stage for further migration.

Personal Experience

The “quick example” in the previous subsection is in fact my own story. Back in 2015, when I pitched this architecture to the chief technology officer of DAZN, I asked for three developers and three weeks. In that time, we recreated the video catalog as a micro-frontend, running on web, PlayStation, and Tizen.

The result? We showed a clear improvement in performance compared to the monolith, and (crucially) we provided tangible business value in less than a month. This early success made it much easier to get the business on board for the rest of the migration.

Iterative migration is almost always the best path, both technically and from a business perspective. It’s easier to sell, delivers value sooner, and reduces risk. The key is to speak the language of your stakeholders, tie migration steps to business outcomes, and start with a pilot that proves the value of your approach.

How Should I Actually Plan the Migration?

Once you have an agreement to move forward, planning the migration is where your choices will have the biggest impact on both the speed of delivery and the safety of your rollout. Business stakeholders want to see progress and value quickly, without risking the stability of your current system.

Best Practices and Patterns

Some best practices are as follows:

Favoring entire pages or groups of pages

Whether you choose a horizontal or vertical split, the most effective approach is to migrate and deploy entire pages or logical groups of pages at a time. This allows you to intercept user requests and route

them to the new micro-frontend, minimizing integration complexity and making it easier to roll back if needed.

Using the strangler fig pattern

Leveraging a router at the edge helps you to implement a strangler fig pattern—a proven method for incremental migration. The strangler fig pattern introduces a façade or proxy that routes requests to either the legacy system or the new micro-frontends. Over time, more functionality is handled by the new system, and the old monolith is gradually phased out. This approach minimizes risk, allows for continuous delivery, and keeps the user experience seamless.

Progressive rollouts and release strategies

Take advantage of advanced release strategies like canary releases, feature flags, or targeting specific user segments (such as a country or percentage of users) to further de-risk your rollout. This way, you can validate new micro-frontends in production with real users, catch issues early, and build business confidence.

Faster rollbacks and safer releases

If you need to roll back a migration, you can simply update the routing rule at the edge to point requests back to the monolith, with no redeployments or code changes required.

Single source of truth for routing

By centralizing routing at the edge, you avoid having to keep routing logic in sync across multiple codebases. Any changes to routing can be made in one place, reducing the risk of inconsistencies and simplifying maintenance (see [Figure 11-1](#)).

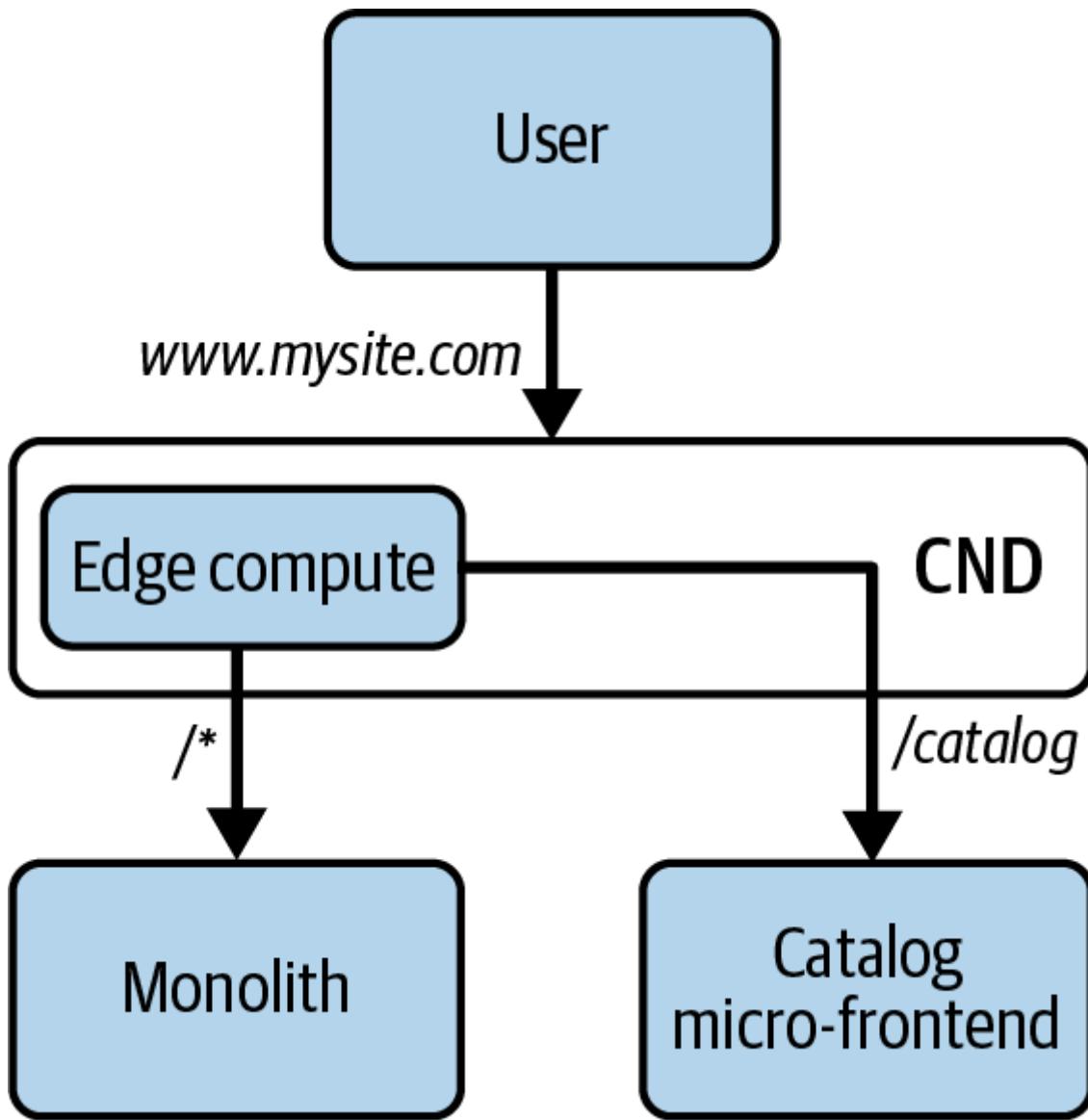


Figure 11-1. Routing configuration at the edge instead of inside the code to avoid maintaining throwaway code in two codebases

Trade-Offs and Pitfalls

At all costs, avoid rendering micro-frontends and legacy code together on the same page or UI view. This increases integration complexity, can lead to inconsistent user experiences, and often slows down the migration process.

There is a clear possibility of having dependency clashes or unexpected UI behaviors that may occur at runtime. Despite it being tempting, it usually requires way more work to design, build, and maintain.

Checklist

Here are a few questions you should ask yourself:

- Have you mapped your application into logical pages or groups of pages that can be migrated independently?
- Do you have the ability to route user requests at the CDN, edge, or server level?
- Do you have a rollback plan that can be executed quickly (ideally by just changing a routing rule)?

A Quick Example

Suppose you're migrating a large ecommerce platform. Instead of rewriting the entire frontend or mixing micro-frontends into existing pages, you start with the check-out flow. You configure your edge compute service to route all `/check-out` requests to the new micro-frontend, while the rest of the site still runs on the monolith. If you encounter issues, rolling back is as simple as changing the routing rule to point back to the old system. You can even release the new check out to just 5% of users or only to users in a specific country, giving you time to observe real-world performance and user feedback before a full rollout.

Personal Experience

I have implemented several strategies for applying a strangler pattern on the frontend. So far, the easiest to build and maintain has been on the edge, where you can store a configuration on an edge storage service from your CDN provider and apply the logic to the edge compute service for routing your system correctly. I've seen many organizations go down this safe route, creating a lot of confidence thanks to the possibility of rolling back quickly without the need for a new redeployment. I spoke at the AWS re:Invent 2019 event about a [basic implementation](#), if you're interested in engaging further.

How Do I Decide Which Modules or Features to Migrate First?

Prioritizing the right areas can make your migration smoother, deliver value faster, and build confidence, both within your team and the business.

Best Practices and Patterns

Some best practices are as follows:

Start with new features or major refactors.

If your product team is planning new functionality or a significant redesign, that's often the best place to introduce micro-frontends. You're already investing effort, so you can modernize without duplicating work. This also avoids the challenge of needing to justify the migration of a module that will "look the same" after the work is done.

Pick less risky, self-contained modules.

For your first migration, consider starting with a module that's relatively isolated and low risk—something that won't break the whole application if issues arise. This allows you to test out the migration process and integration patterns before tackling more complex areas. Moreover, all the learnings during the first development will be helpful for the next ones.

Go end-to-end with your first micro-frontend.

Choose a module you can take through the entire life cycle: development, testing, observability, and deployment. This approach will not only accelerate your understanding of how micro-frontends behave in your environment, but also surface hidden integration issues early—those issues that often go unnoticed until multiple parts come together. Don't be afraid to challenge your initial findings and adapt as you learn more. Embrace a lean, learning-focused mindset to refine your architecture as you go.

Establish a proof of concept and pilot.

Develop a low-level design and a proof of concept for your first micro-frontend. Use this as a "point of no return" milestone—if it works and integrates well, you can confidently proceed; if not, you can adjust your approach before scaling up.

Trade-Offs and Pitfalls

You also need to consider some pitfalls:

Migrating high-traffic modules first can be risky.

While you'll see business impact quickly, any issues will be highly visible. Make sure you have strong monitoring and rollback strategies in place.

Migrating low-traffic or less critical modules first is safer but slower to show value.

This approach is great for learning and de-risking, but might not excite stakeholders or justify the migration investment early on.

Mixing too many approaches can create confusion.

For example, mixing micro-frontends and monolith code on the same page can add complexity and slow down progress. Aim for clear, page-level or route-level boundaries when possible.

Technical debt and duplicated effort can slow progress.

If you migrate modules that are about to be retired or redesigned, you risk wasting effort. Always align migration priorities with the product roadmap.

Checklist

Here are a few questions you should ask yourself:

- Are there any new features or major refactors planned?
- Which modules are most critical for business value or user experience?
- Which modules are self-contained and at low risk for a first migration?
- Do you have a clear design and proof of concept for your first micro-frontend?
- Have you aligned your migration plan with the product and business roadmap?
- Is your team ready to monitor, measure, and roll back if needed?

A Quick Example

Suppose your product team is about to launch a new user dashboard. Instead of building it in the monolith, you develop it as a micro-frontend. This lets you test your integration

approach, validate performance, and show business value early. Once that's live, you can tackle the next module—maybe the account settings or check-out flow—using the lessons you've learned.

Personal Experience

While I was consulting a financial unicorn startup based in Singapore, we chose the main dashboard as a good starting point. It was a high-visibility area with some interesting challenges to solve, but also had clear boundaries, making it a good candidate for migration. Going end to end with this module taught them a huge amount about development, deployment, and monitoring in the new architecture. The early success there made it much easier to get buy-in for the rest of the journey. In only a month, they deployed the first version in production with success.

Deciding what to migrate first is about balancing risk, value, and learning. Start where you can deliver impact and gain confidence, and use those wins to fuel the rest of your migration. Embrace a lean mindset: treat every migration as a learning opportunity, and don't be afraid to adapt as you go.

How Do I Maintain User Experience and Consistency During Migration?

Before we get into the details, let's be honest: some discrepancies in look and feel are likely to happen as you migrate to micro-frontends, especially if different teams are moving at different speeds or using different technologies. But here's the good news: this isn't necessarily a bad thing if you're delivering real value to your users, such as new features or a better experience. Most customers will be forgiving during the transition.

Best Practices and Patterns

Some best practices are as follows:

Invest in a design system.

A robust design system is an essential tool for maintaining consistency across micro-frontends. It provides shared styles, components, and guidelines so that every team can build features

that look and feel like they belong together, even if they're using different frameworks or tech stacks.

- If possible, use web components for your design system. Web components are framework-agnostic and encapsulated, so they can be reused across all your micro-frontends, no matter what technology each team picks.
- If you don't have a design system yet, start with design tokens (colors, spacing, typography) and build up a new system using web components as you go.

Avoid reusing monolith component libraries.

It may be tempting to pull components from your old monolith, but this can create unwanted dependencies and slow you down. Instead, focus on building only the components you need for the first micro-frontends, and expand your library as new modules are built.

Build only what you need, when you need it.

Don't feel pressured to recreate every single component up front. Start with the essentials for your first micro-frontend, and let your design system grow organically as you migrate more features.

Route easily between the old and new world.

If you have centralized the routing on the edge as explained earlier, the way you will handle the hard navigation between your previous system and the new one is via an absolute URL. A simple change will trigger the router on the edge and will load the next system without creating too much complexity. This is shown in the sequence diagram in [Figure 11-2](#).

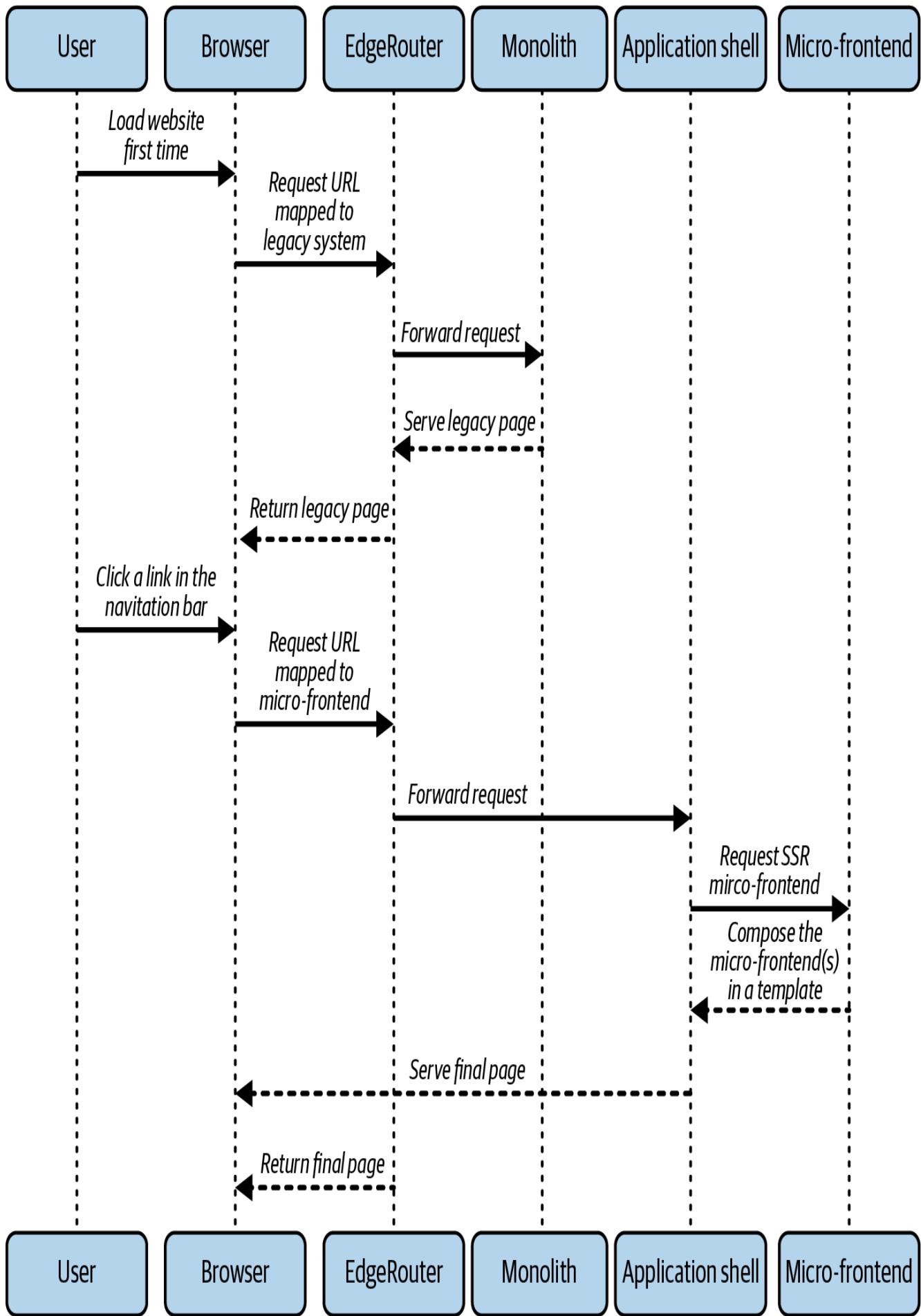


Figure 11-2. Hard navigation between micro-frontends and the legacy system, leveraging absolute URLs

Trade-Offs and Pitfalls

You also need to consider some pitfalls:

Inevitability of some inconsistency

Early in the migration, you may see minor visual or behavioral differences. As long as you’re delivering value, most users will accept these temporary quirks.

Overengineering the design system

Trying to build a “perfect” design system before you start can slow you down. Focus on the basics, and let your system evolve as you learn what works.

Tight coupling to legacy code

Reusing monolith components can create dependencies that are hard to untangle later. Build new, modular components when possible.

Lack of communication between teams

Without regular check-ins and shared guidelines, teams may drift apart in their implementation. Keep everyone aligned with clear documentation and open channels.

Checklist

Here are a few questions you should ask yourself:

- Do you have a design system (or at least design tokens) that all teams can use?
- Are your shared components framework-agnostic (e.g., web components)?
- Are you building only the components needed for the current migration phase?
- Is navigation seamless between the old and new world?
- Are you running regular design reviews and automated UI tests?
- Are teams communicating and sharing feedback regularly?

A Quick Example

Imagine you're migrating the account dashboard to a micro-frontend. Instead of reusing the old monolith's component library, you build a few core web components—like buttons and form fields—based on your new design tokens. The result: the new dashboard looks and feels consistent with the rest of the app, even though it's built with different technology. As you migrate more features, you add new components to your shared library, keeping the experience cohesive for users.

Personal Experience

In my experience across many teams, I've found that starting with a small, focused set of web components and a clear design system made a huge difference. We built only what we needed for the first micro-frontend, then expanded as we went. This approach lets us deliver value quickly, avoid unnecessary dependencies, and keep the user experience consistent, even as we move fast and learn along the way.

Consistency is important, but don't let the pursuit of perfection slow you down. Focus on delivering value, use a design system and web components to align teams, and let your user experience improve as your migration progresses.

How Do I Handle Shared Dependencies and Version Management Between Legacy and Micro-Frontend Code?

When migrating to micro-frontends, one of the most common concerns is how to manage shared dependencies and versions between your legacy monolith and the new micro-frontends. It's easy to fall into the trap of trying to avoid all duplication at any cost. But in distributed systems, the priorities are different—we optimize for speed and autonomy, not just for reusability.

Best Practices and Patterns

Some best practices are as follows:

Don't fear duplication.

In distributed systems, some duplication is not only acceptable, but often desirable. It allows teams to move faster, make independent

decisions, and avoid unnecessary coupling. Use the migration as an opportunity to rebuild or rethink parts of your system that have become bloated or outdated.

Balance reuse and independence.

If you have a well-designed, framework-agnostic design system or utility library that genuinely fits both worlds, feel free to reuse it. But don't obsess over sharing everything. The goal is to enable fast flow and independent deployment, not to create a tangled web of shared dependencies.

Create clear rules to follow.

Define straightforward guidelines for every engineer who encounters roadblocks during the migration. Don't stop or overthink every obstacle; there will be many, and moving fast is important. Remember, many of the decisions you make can easily be changed later. Don't worry if a choice leads to some friction; you'll learn and adapt as you go.

Decouple version management.

Set clear rules so that versioning for micro-frontends is completely independent from the legacy monolith. This avoids "dependency hell," where changes in one world break the other. For example, if you need to update a library, do it in the micro-frontend first and only touch the monolith if absolutely necessary.

Domain-driven splits help.

If you've identified your domains well and are splitting your app by first-level URL (e.g., /catalog, /check_out), it's much easier to keep dependencies and versions isolated. Each domain can evolve at its own pace.

Trade-Offs and Pitfalls

You also need to consider some pitfalls:

Too much sharing creates coupling.

Oversharing libraries or components between legacy and micro-frontends can slow everyone down and make upgrades risky. It also makes it harder to refactor or modernize your new codebase.

Unmanaged duplication can lead to bloat.

If you duplicate everything without discipline, you may end up with larger bundles or inconsistent user experiences. Be intentional about what you duplicate and why.

Avoid premature abstraction.

Don't rush to abstract or share code before you have enough duplication to justify it. Let patterns emerge naturally; then extract shared libraries when there's a proven need.

Checklist

Here are a few questions you should ask yourself:

- Are you comfortable with some duplication if it improves team autonomy and speed?
- Did you define some common rules to follow to avoid slowing down the migration?
- Are versioning and release processes for micro-frontends completely independent from the monolith?
- Have you split your domains clearly, preferably by first-level URL?
- Do you regularly review duplicated code to identify when (and if) it should be abstracted?

A Quick Example

Suppose your legacy app and your new micro-frontend both need a date picker. Instead of forcing both to use the same shared component, you build a new, lightweight date picker for the micro-frontend that fits your new design system. If, over time, you find that several micro-frontends need the same version, you can extract it into a shared package for the new world only, leaving the monolith untouched.

Personal Experience

When we started the migration in 2016, the only thing we reused between the old system and the new one was the design tokens. We didn't have a design system, so we took this opportunity to revisit our framework and library choices and to plan a design system that would serve the micro-frontend architecture well. We also involved a large portion of the developers in the initial phase to determine collectively how to approach specific problems that were affecting multiple teams.

It's important to create a collaborative environment where people feel included and empowered. This kind of culture encourages everyone to push the boundaries and continuously improve the system—whether you're in the room or not.

How Do I Manage Cross-Cutting Concerns Like Auth, Routing, or State During Migration?

Managing cross-cutting concerns—such as authentication, routing, and state—is one of the trickiest parts of migrating to micro-frontends. These concerns span multiple modules and can easily become sources of bugs or user frustration if not handled thoughtfully. The good news is that with the right approach, you can keep things simple and robust while maintaining a smooth user experience.

Best Practices and Patterns

Some best practices are as follows:

Routing

Where possible, centralize routing at the edge (CDN or edge compute) and use absolute URLs for navigation between the legacy app and micro-frontends. This approach drastically simplifies navigation logic, avoids coupling between systems, and provides a fast and reliable rollback strategy if something goes wrong.

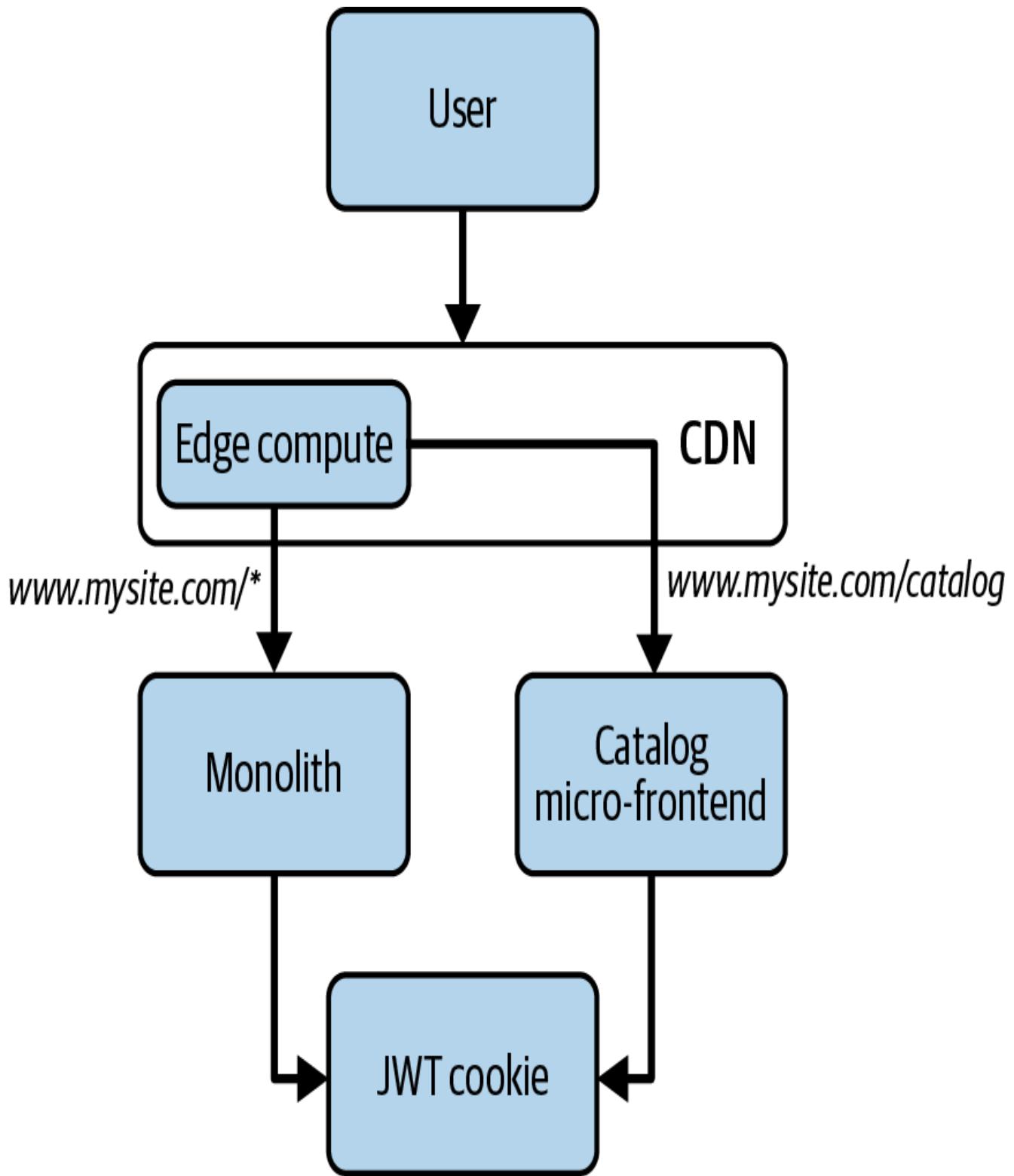
State management

If you need to share state between micro-frontends and the legacy app (or vice versa), use query strings for ephemeral data that only needs to persist for a single navigation event. For more persistent or configuration data, rely on backend APIs to pass information between

systems. Avoid using local or session storage for critical data, as this can lead to awkward edge cases (like browser refreshes) and complicate your migration in the long run.

Authentication

As long as your micro-frontends are served from the same subdomain (`www.` or similar) as your legacy application, they'll have access to the same cookies and session/local storage. This makes it straightforward for each micro-frontend to access session information. Just remember, you'll need to implement refresh token logic in both the legacy system and the new micro-frontends to keep users logged in seamlessly (see [Figure 11-3](#)).



Because both systems are served by the same subdomain

Figure 11-3. Cookies, session, and local storage accessible by both the legacy system and micro-frontends—shared via the same subdomain for local and session storage, or with the right configuration for cookies

Trade-Offs and Pitfalls

You also need to consider some pitfalls:

Auth drift

If your authentication logic diverges between the legacy system and micro-frontends, you risk inconsistent user experiences or security issues. Keep the core logic aligned, and test thoroughly across both systems. Handle changes to the identity provider after the migration is done, or create a backend proxy to route requests to the correct identity provider.

Overcomplicating routing

Trying to handle routing logic inside each app can lead to confusion and bugs. Centralizing at the edge keeps things simple and makes rollbacks much easier.

State synchronization headaches

Relying on browser storage for key data can create hard-to-debug issues, especially with refreshes or multiple tabs. Stick to query parameters for transient data and APIs for anything more substantial.

Checklist

Here are a few questions you should ask yourself:

- Are all micro-frontends and the legacy app served from the same subdomain for seamless cookie and storage access?
- Is refresh token logic implemented in both systems?
- Are you using edge routing and absolute URLs for navigation between systems?
- Do you use query strings for passing ephemeral state between apps?
- Are APIs in place for sharing configuration or persistent data?
- Have you avoided relying on local/session storage for critical state?
- Are you regularly testing authentication, navigation, and state flows across both systems?

A Quick Example

Suppose you're migrating the user profile section to a micro-frontend. The user logs in through the legacy app, and the session cookie is available to the new micro-frontend because both are on the same subdomain. When navigating from the legacy dashboard to the new profile page, you use an absolute URL, which triggers edge routing and loads the correct system. If you need to pass a temporary filter or view state, add it to the query string. For persistent preferences or user data, both systems read from a shared backend API.

Personal Experience

In my own migrations, keeping authentication and session management consistent across both worlds was crucial. At DAZN, we made sure all micro-frontends were served from the same subdomain, so cookies and session data were always accessible. We also centralized routing at the edge, which made navigation seamless and rollbacks trivial. This approach kept the migration smooth, and it minimized user disruption—even as we moved quickly.

Cross-cutting concerns can make or break your migration. Keep things simple: centralize where you can, avoid unnecessary coupling, and let real needs drive your abstractions. This will help you maintain a consistent, reliable experience for your users throughout the transition.

Do I Need Microservices to Migrate to Micro-Frontends?

This is a very common question, and the answer is simple: no. You do not need microservices to migrate to micro-frontends. Any frontend—including micro-frontends—just needs a clear contract to interact with APIs. Don't overcomplicate things by thinking you need to modernize your backend first or in parallel.

Best Practices and Patterns

Some best practices are as follows:

The frontend and backend are independent.

Micro-frontends are about how you structure and deliver your frontend code. As long as your frontend can call APIs (REST, GraphQL,

etc.), it doesn't matter if those APIs are powered by a monolith or microservices. Remember that every software is a living system, so there will be changes, optimizations, and throwaway code every single sprint. Just accept the nature of software, and you will be fine.

Start from the frontend.

You can begin your modernization journey by breaking up your frontend into micro-frontends and connecting them to your existing backend. There's no technical requirement to migrate your backend to microservices first. This approach is usually faster due to the stateless nature of frontends.

Data has gravity.

In my experience, the frontend is stateless and much quicker to modernize. The backend—especially when it involves data replication or redesigning database schemas—takes much longer. Focus on delivering value quickly through frontend improvements while planning for backend changes at your own pace.

Iteratively modernize the backend.

Once your frontend is modular and delivering value, you can gradually modernize your backend if and when it makes sense for your business.

Trade-Offs and Pitfalls

You also need to consider some pitfalls:

Don't block backend modernization.

Waiting for a full backend migration before starting on the frontend can delay value for users and the business. Frontend and backend can evolve independently.

API contracts matter.

As long as your APIs are well-defined, your micro-frontends can work with any backend architecture. Just ensure the contracts are stable and documented.

Backend migration is harder and slower

Migrating backend systems, especially with complex data, is often much more time-consuming and riskier than frontend work. Don't underestimate this difference.

Partial modernization still delivers value.

Even if some backend APIs remain in the monolith, you can still achieve significant improvements in user experience, team autonomy, and release speed through micro-frontends.

Checklist

Here are a few questions you should ask yourself:

- Can your frontend teams deliver value independently of backend modernization?
- Are you able to iterate on the frontend without waiting for backend changes?
- Have you planned for backend modernization as a separate, future project if needed?

A Quick Example

Let's say you have a monolithic backend serving all your APIs. You start migrating your frontend to micro-frontends, each one calling the same set of APIs as before. Users see new features and improved performance quickly, while the backend remains untouched for now. Later, you can refactor backend services one by one, but you're already delivering value.

Personal Experience

In dozens of modernization projects, I've seen that micro-frontends can be delivered in as little as 14 months, while backend migrations often take much longer due to data complexity and schema changes. The frontend is stateless and much easier to move quickly. Even when backend APIs are still running on the old monolith, the value delivered by a modern, modular frontend is huge. Don't let backend modernization block your progress; start where you can make the biggest impact fastest.

You do not need microservices to migrate to micro-frontends. Focus on decoupling and improving your frontend first; the backend can follow when you're ready.

Summary

Migrating to micro-frontends is a meaningful journey—challenging at times, but far from impossible. The most important lessons? Start small, move iteratively, keep feedback loops tight, and always tie your work to real business value. You don't need to wait for a perfect backend or a flawless design system to begin.

Remember:

- Iterative beats big bang. Break your migration into manageable pieces. Use edge routing and patterns like the strangler fig to keep changes safe and reversible.
- Prioritize for impact and learning. Start with modules that let you deliver value and learn quickly. Don't be afraid of a little duplication or some temporary inconsistency.
- Keep things simple. Centralize cross-cutting concerns, avoid overengineering, and let your design system and shared code evolve as you go.
- Microservices aren't required. Micro-frontends can thrive with any backend, as long as you have clear API contracts.

With clear goals, teamwork, and a willingness to adapt, you'll unlock new speed and flexibility for your teams and your business.

And if you're curious about how all these ideas come together in practice, stay tuned: toward the end of the book, I'll share a real migration story, including the reasoning and steps that took us all the way to production with a micro-frontends system.

Chapter 12. From Monolith to Micro-Frontends: A Case Study

Let's imagine that in the last few weeks, you've researched and reviewed articles, books, and case studies, and you've completed several proofs of concept. You've spoken with your managers to find the best people for the project, and you've even prepared a presentation for the chief technology officer explaining the benefits the business and team can get from introducing micro-frontends in your platform.

At last, you've received confirmation that you have been granted the resources to prepare a plan and start migrating your legacy platform to micro-frontends. Great job!

It's been a long few weeks, and you've done an amazing job, but this is only the start of a large project. Next, you will need to prepare an overall strategy—one that's not too detailed but not too loose. Too detailed, and you'll spend months just trying to nail everything down. Too loose, and you won't have enough guidance. You need enough of a strategy to get started and to use as a North Star to follow during the journey whenever you discover new challenges and details that you didn't think about until that point—and, trust me, you will!

Meanwhile, you also have a platform to maintain in production, which the product team would like to evolve because the re-platforming to micro-frontends shouldn't block the business. The situation is not the simplest, but you can mitigate these challenges and find the right trade-off to make everyone happy and ensure the business is successful while the tech teams are migrating to the new architecture.

We have learned a lot about how to design and implement micro-frontends, but I feel this book would not be complete without looking at a migration from a monolithic application to micro-frontends—by far the most common use case of this architecture. I believe any project should start simply. Then, over the course of the months or years—when the business and the organization are growing and they are reaching maturity with this new way to work—the architecture should evolve accordingly to support the evolving business needs. There may be some scenarios where starting a new application with micro-frontends may help the business move in the right direction, such as when you have an application that is composed of several modules that you can ship all together, along with some customization for every customer. But the classic use case

of micro-frontends is the migration from a legacy frontend application to this new approach.

In this chapter, I will share a fictional case study that stitches together all the information we have discussed in this book.

The Context

ACME Inc. is a fairly new organization that, in only a few years, has gained popularity for its video-streaming service across several countries in the world. The company is growing fast. In the last couple of years, it has moved from hundreds of employees to thousands, all across the globe, and the tech department is no exception.

The streaming platform is currently available on desktop and mobile browsers, as native applications, and on some living-room devices, like smart TVs and consoles.

Currently, the company is onboarding many developers in different locations across Europe. Having all the developers in Europe was a strategic decision to avoid slowing down the development across distributed teams while having some hours of overlap for meetings and coordination.

Due to the tech department's incredible growth from tens to hundreds of people, tech leadership reviewed and analyzed the work done so far, finally embracing a plan to adapt their architecture to the new phase of the business. Leadership acknowledged that maintaining the current architecture would slow down the entire department and wouldn't enable the agility required for the current expansion the business is going through.

Technology Stack

The current platform uses a three-tier application deployed in the cloud. It is composed of a single database with read replicas (they have more reads than writes in their platform); a monolithic API layer with autoscaling for the backend, which scales horizontally when traffic increases; and a single-page application (SPA) for the frontend. [Figure 12-1](#) illustrates this platform.

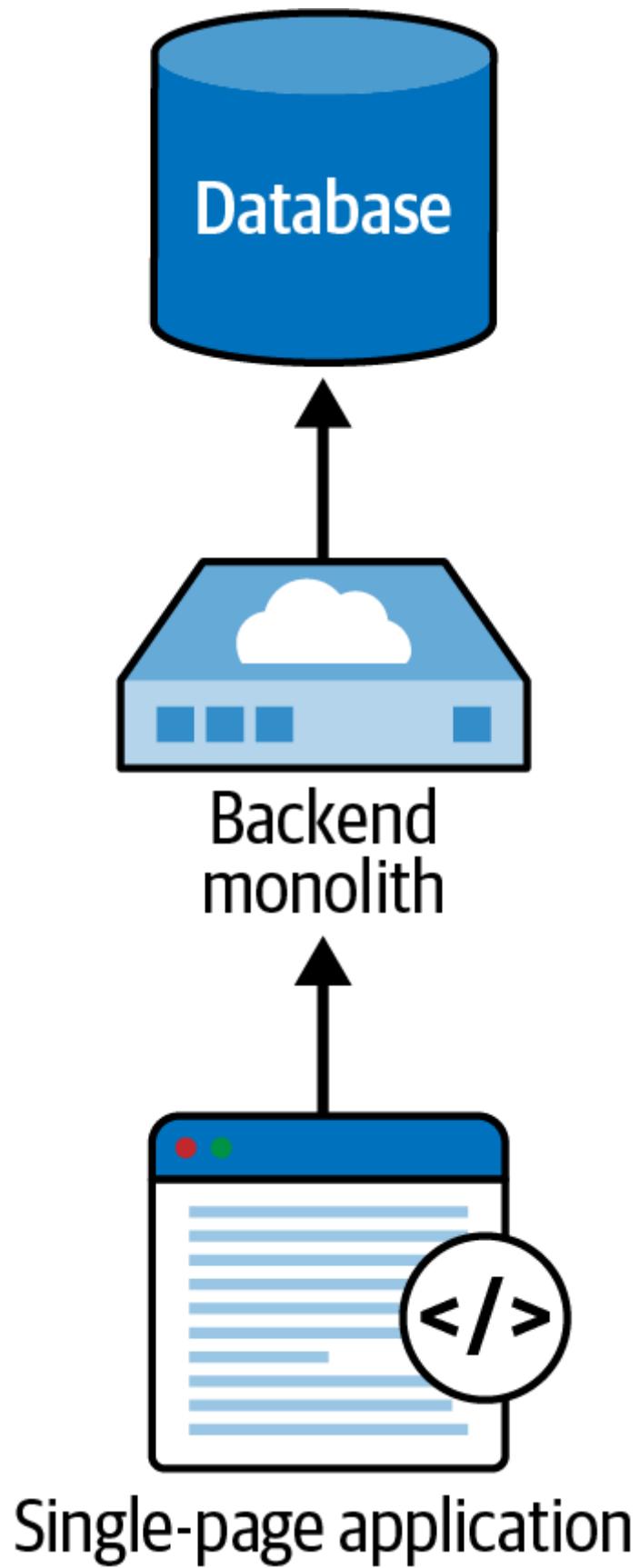


Figure 12-1. ACME Inc.'s three-tier web application platform

A three-tier application allows the layers to be independently developed; however, ACME Inc. is scaling and increasing in complexity, as well as increasing the teams

working on the same project. This architecture is now impacting the day-to-day throughput and generating communications overhead across teams, which may lead to more complexity and coordination being required despite not being necessary in other solutions.

As the tech leadership team rightly points out, in this new phase of the business, the tech department needs to scale with more developers and features than before. A task force with different skill sets reviews how the architectures—frontend and backend—should evolve in order to unblock the teams and enable the company to scale in relation to business needs.

After several weeks, the task force proposed migrating the backend layer to microservices and the frontend to micro-frontends. This decision was based on the capabilities and principles of these architectures. They will allow teams to be independent, moving at their own speed, scaling the organization as requested by the business, drifting toward the direction the business needs, choosing the right solution for each domain, and scaling the platform according to the traffic on a service-by-service basis, leveraging the power of cloud vendors.

From here, we'll focus our discussion on the frontend part with some references to the backend.

Platform and Main User Flows

The frontend is composed of the following views:

- Landing page
- Sign-in
- Sign-up
- Payment
- Forgot email
- Forgot password
- Redeem gift code
- Catalog (with video player)
- Schedule

- Search
- Help
- My account

To have enough information to understand how the migration to micro-frontends will work, we will analyze the authentication flow for existing customers, the creation of a subscription flow for new customers, and the experience within the platform for authenticated customers. Many of these suggestions can be replicated for other areas of the application or applied with small tweaks.

When a new user wants to subscribe to the video-streaming platform, they follow these steps, as outlined in **Figure 12-2**:

1. The user arrives on the landing page, which explains the value proposition.
2. The user then moves to the sign-up page, where they create an account.
3. On the next page, the user adds their payment information.
4. The user can then access the video platform.

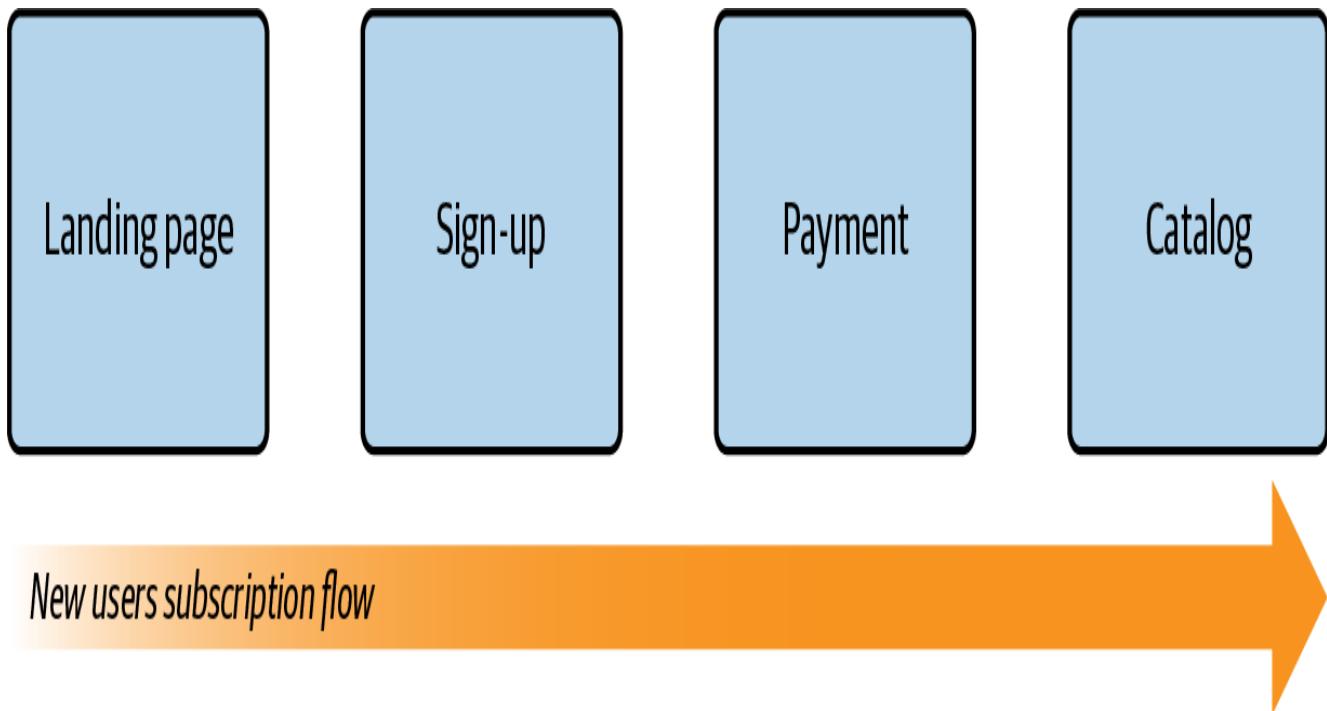


Figure 12-2. New user subscription flow

When an existing user wants to sign in on a new platform (browser or mobile device, for instance) to watch some content, they will follow these steps (see **Figure 12-3**):

1. Access the platform in the landing page view.
2. Select the sign-in button, which redirects them to the sign-in view.
3. Insert their credentials.
4. Access the authenticated area and explore the catalog.

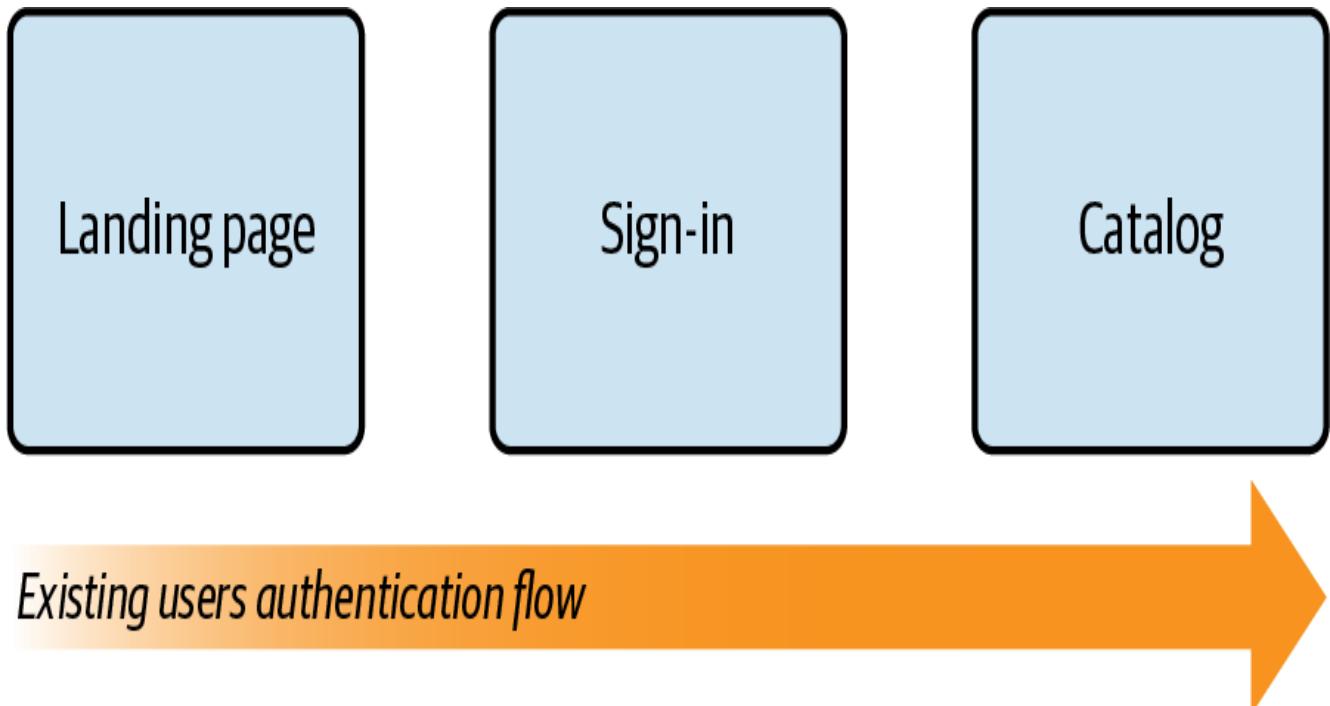


Figure 12-3. Existing users authenticating in a new platform (e.g., browser or mobile devices)

Once a user is authenticated, they can watch video content and explore the catalog following these steps (see **Figure 12-4**):

1. They start at the catalog to choose the content to view.
2. When content is selected, the user sees more details related to the content and the possibility to either search for similar content or just play the content.
3. When the user chooses to play the content, they are redirected to a view with only the video player.
4. When the user wants to search for specific content not available in the catalog view, they can choose to use the search functionality.

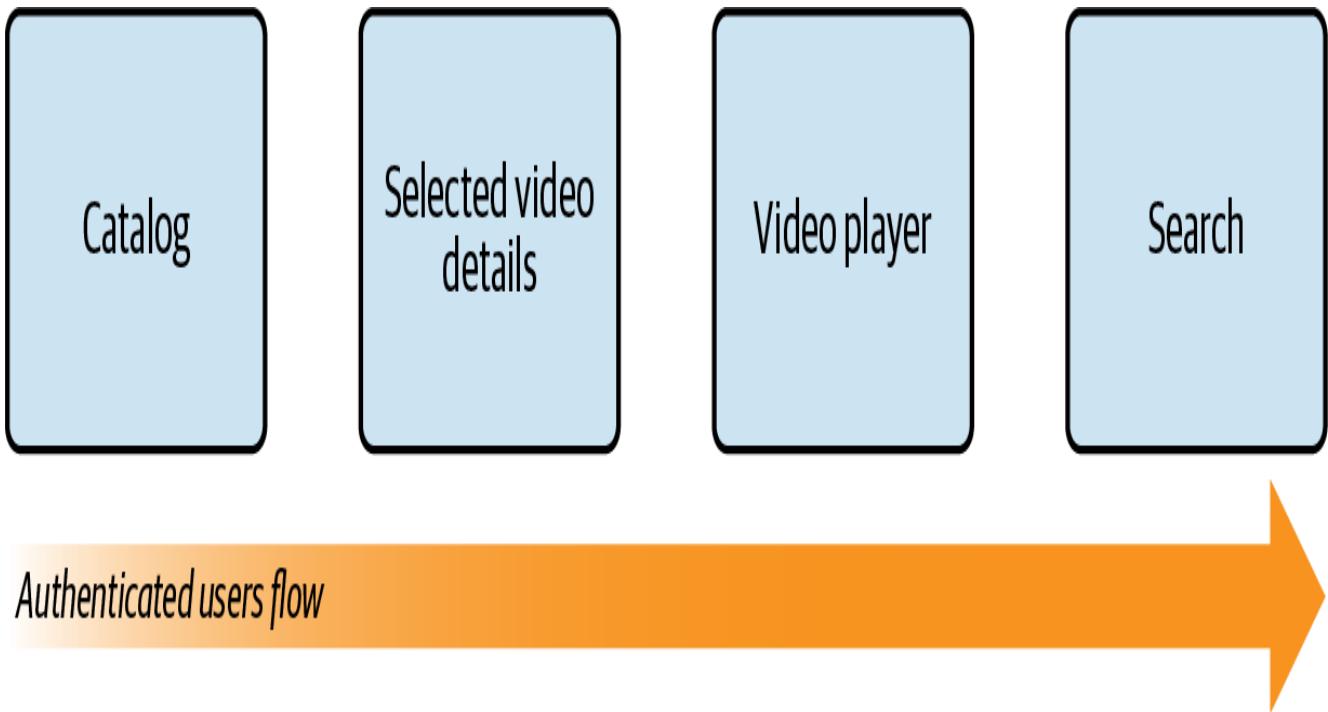


Figure 12-4. Option for existing users to navigate content via the catalog or search functionality and then play any content after discovering the details of what they are about to watch

These are the main flows, which should be enough to explore how to migrate to micro-frontends. Obviously, there are always more edge cases to cover, especially when we implement error management, but those won't be covered in this chapter.

The application is written with Angular, with a continuous integration (CI) pipeline and a deployment that happens twice a month because it is strictly coupled with the backend layer. In fact, the static files are served by the application servers where the APIs live. Therefore, every time there is a new frontend version, the teams have to wait for the release of a new application server version. The release doesn't happen very often due to the organization's slow release-cycle process.

The final artifact produced by the automation pipeline is a series of JavaScript, Cascading Style Sheet (CSS) files with an HTML entry point. In the CI process, the application has some unit testing, but the code coverage is fairly low (roughly 30%), and the automation process takes about 15 minutes to execute end to end to create an artifact ready to be deployed in production.

The organization is using a three-environment strategy: testing, staging, and production. As a result, the final manual testing happens in the staging environment before being pushed into production—another reason why deployments can't happen too often. The user acceptance testing (UAT) department does not have enough resources, compared to the developers who handle platform enhancement. Due to the simple automation process put in place, some developers on different teams are responsible for maintaining the

automation pipelines; however, it's more of an additional task to shoehorn into their busy schedules than an official role assigned to them. This sometimes causes problems because resolving issues or adding new functionalities in the CI process may require weeks instead of days or hours.

Finally, the platform was developed with observability in mind—not only on the backend but also on the frontend. In fact, both the product team and the developers have access to different metrics to understand how users interact with the platform so they can make better decisions for enhancing the platform's capabilities. They are also using an observability tool for tracking JavaScript runtime errors inside their frontend stack.

Technical Goals

After deciding to move their frontend platform to micro-frontends, the tech leadership identified the key goals they should aim for with this investment.

The first goal is maintaining a seamless experience for developers despite the architectural changes. Degrading the frictionless developer experience (DX) that was available with the SPA could lead to a slower feedback loop and lower software quality. Moreover, the leadership decided that it doesn't want to reinvent the wheel, so it will be acceptable to create some tools for filling specific gaps, but not a completely custom DX that might prevent new tools from being embraced in the future. It's important to strengthen the automation strategy to reduce the feedback loop, which currently takes too long.

Another key project goal is to decouple the micro-frontends and enable independent evolution and deployment. Every micro-frontend should be an independent artifact deployable in any environment. Teams need to optimize for fast flow, reducing the external dependencies for each team.

An additional goal is to generate value as quickly as possible to demonstrate to the business the return on their investment. Therefore, a strategy for transitioning the SPA to micro-frontends must be defined so that when a micro-frontend is ready, it can initially work alongside the monolith.

Finally, the tech leadership has also requested tracking onboarding time for new team members to understand whether this approach extends developer onboarding time. The team will need to find ways to reduce this period, perhaps by creating additional documentation or exploring different approaches.

The last goal for this project is finding the right organizational setup for reducing external dependencies between teams and reducing the communication overhead that could increase due to the company's massive growth.

Migration Strategy

Based on tech leadership's requirements and goals, the teams started to work on a plan for migrating the entire platform to micro-frontends.

The first step was embracing the micro-frontend decision framework as a guideline to define the foundation of the new architecture. The first four decisions—defining what a micro-frontend is in the architecture, composing micro-frontends, routing micro-frontends, and communicating between micro-frontends—will lead the entire migration toward the right architecture for the context.

Micro-Frontend Decision Framework Applied

The first decision of the framework is how a micro-frontend will look. The ACME Inc. teams decided on a *vertical split*, where micro-frontends represent a subdomain of the entire application, as shown in [Figure 12-5](#).

A vertical split is the approach chosen by several other organizations worldwide for kicking off their new micro-frontend architecture. It definitely has fewer sharp edges to take into account, and it can always be mixed with a horizontal split when needed.

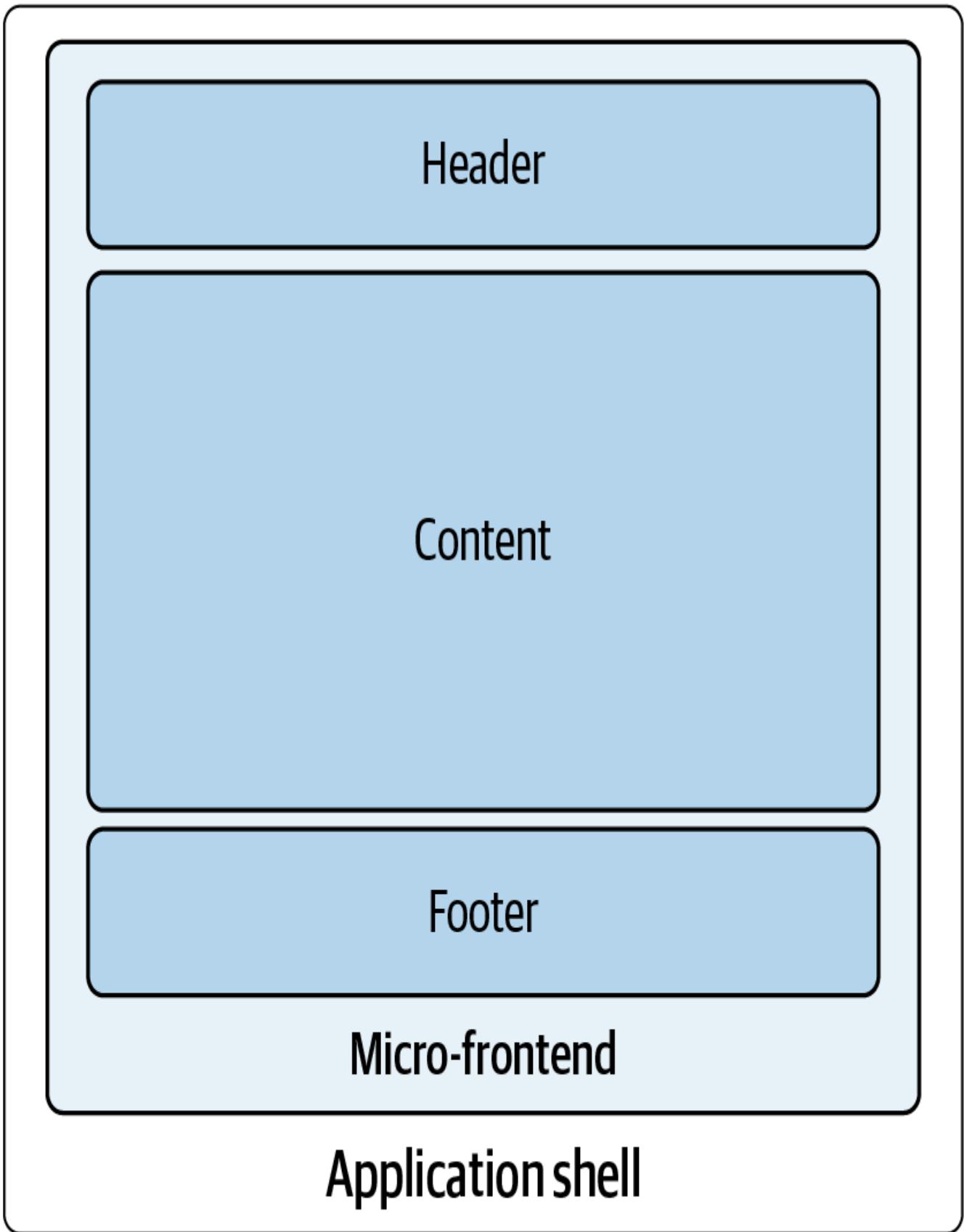


Figure 12-5. A vertical-split micro-frontend, where the application shell loads only one micro-frontend at a time

The teams considered the following characteristics for their context before deciding to use a vertical split:

Similar DX

Because the current platform is an SPA, a vertical split allows developers to work similarly to how they have worked so far, but with a smaller context and less code to be responsible for.

Better integration with current automation strategy

The vertical split fits very well with the current automation strategy, given that ACME Inc. is currently building an SPA. The teams have enhanced their automation pipelines for building multiple SPAs without the need to create custom tools for embracing this architecture style. They will need to use infrastructure as code for automating the process of building their pipelines and replicating them without human intervention.

No risk of dependency clashes

In a vertical split, we always load one micro-frontend at a time, due to the nature of the vertical split. As a result, teams won't have to deal with dependency clashes (like different versions of the same library), because only the dependencies of the currently loaded micro-frontend are present, reducing the possibility of runtime errors and bugs in production. There also won't be any CSS style clashes, because only one stylesheet per micro-frontend is loaded.

A consistent user experience (UX)

Creating a consistent UX is easier with a vertical split because the same team is working on one or multiple views within the same application. Obviously, a level of coordination is required for maintaining consistency across micro-frontends. The introduction of a design system will help mitigate the potential UX fragmentation. However, it's definitely less prone to errors than having multiple micro-frontends in the same view developed by multiple teams.

Reduction of cognitive load

For ACME Inc., a vertical split will decrease its developers' cognitive load because they'll only have to master and maintain a part of the platform. This choice also won't dilute the decisions made by

developers within their business subdomain. However, every developer should have an overall understanding of the platform architecture so that when they're on call, they can understand the touchpoints of their business domain and recognize where a bug may appear despite not being inside their domain.

Faster onboarding process

As the tech leadership requested, using this approach will lead to a faster onboarding process because the teams can use well-known, standard tools and won't need to create their own to build, test, and deploy micro-frontends. Also, because teams will be responsible for only a part of the platform, less coordination with other teams will be required. New members can get up to speed faster, with less information needed to start. Finally, every team will be encouraged to create a starter kit and induction process for new joiners to speed up the learning process and help them contribute to the base code as quickly as possible.

The second decision of the framework is related to the composition of the micro-frontends. In this case, the best approach is composing them on the client side, considering they are using a chosen approach.

This means that the teams will have to create an application shell that is responsible for mounting and unmounting micro-frontends, and ensuring it will always be available during the user session (see [Figure 12-6](#)).

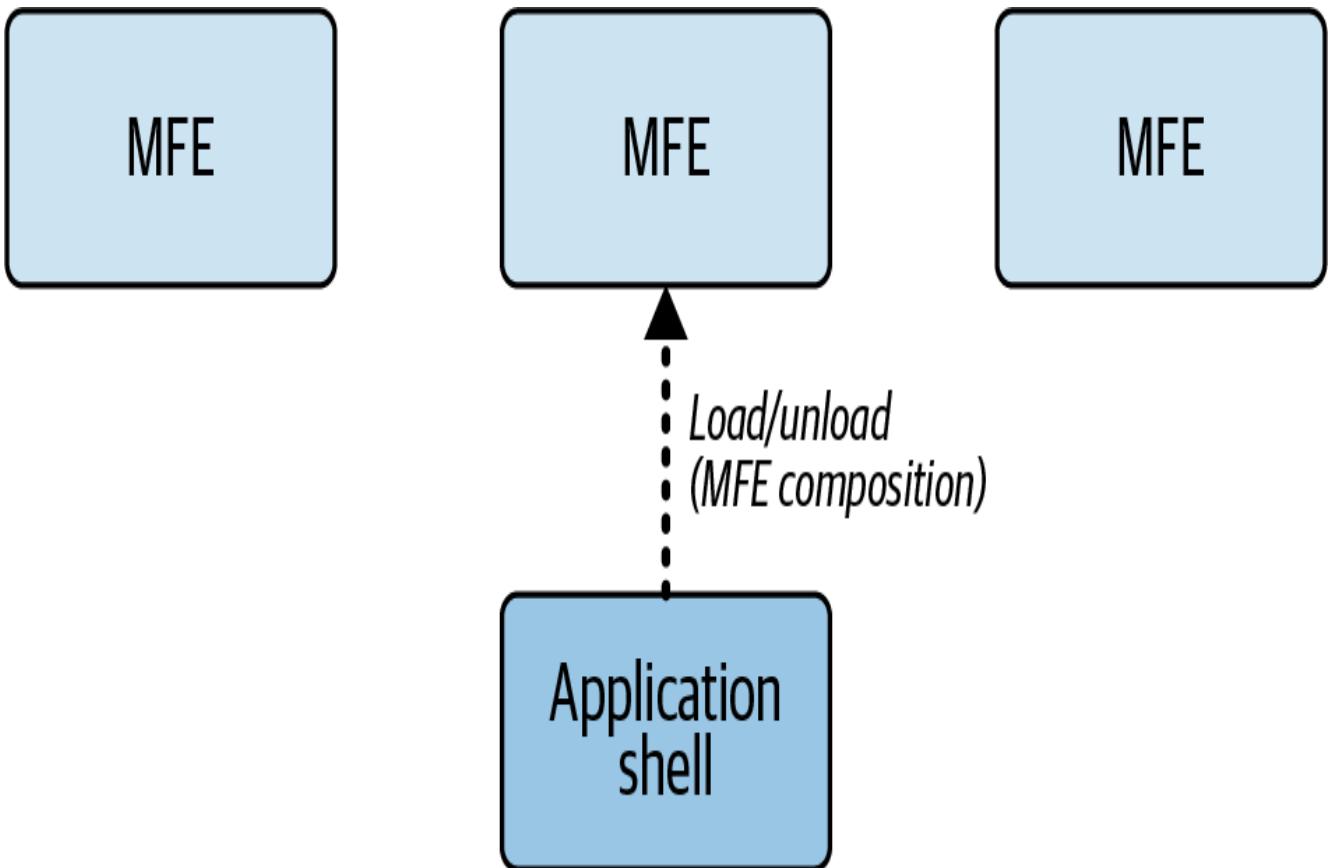


Figure 12-6. Client-side composition, where the application shell is responsible for loading and unloading one micro-frontend at a time.

A server-side composition was rejected immediately due to the traffic spikes, which required more effort to support and maintain than the simple infrastructure they would like to use for this project. Moreover, the majority of the application lives behind authentication. Therefore, they couldn't even benefit from the organic SEO offered by the server-side rendering architecture.

The third decision is the routing of micro-frontends—that is, how to map the different application paths to micro-frontends. Because ACME Inc. will use a vertical split and is composing on the client side, the routing must happen on the client side too—where the application shell knows which micro-frontend to load based on the path selected by the user. This mechanism also has to handle the deep-linking functionality; if a user shares a movie's URL with someone else, the application shell should load the application in exactly that state (see [Figure 12-7](#)).

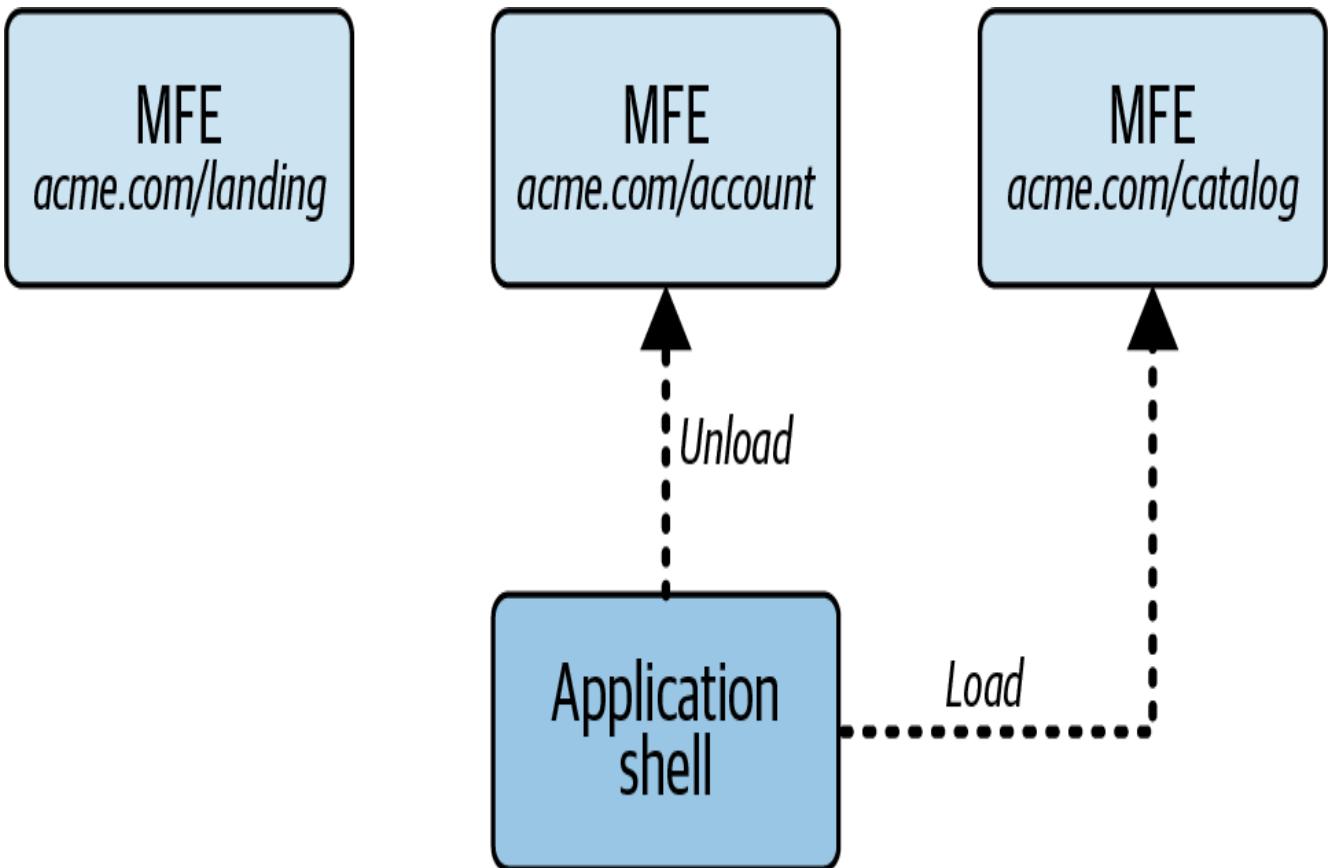


Figure 12-7. User signs in from the /account path, redirected to the authenticated area (/catalog), with the application shell handling unloading and loading of micro-frontends based on the URL

When an unauthorized user tries to access an authenticated part of the system via deep linking, the application shell should validate *only* if the user has a valid token. If the user doesn't have a valid token, it should load the landing page so the user can decide to sign in or subscribe to the service.

Last but not least, ACME Inc. teams must decide how micro-frontends communicate with each other. With a vertical split, communication can happen only via query string or using local storage. There is no need to use other techniques like event emitters or custom events. ACME Inc. decided to leverage the local storage primarily and to use the application shell as a proxy for storing the data. In this way, the application shell can verify the space available and make sure data won't be overridden by other micro-frontends (see [Figure 12-8](#)).

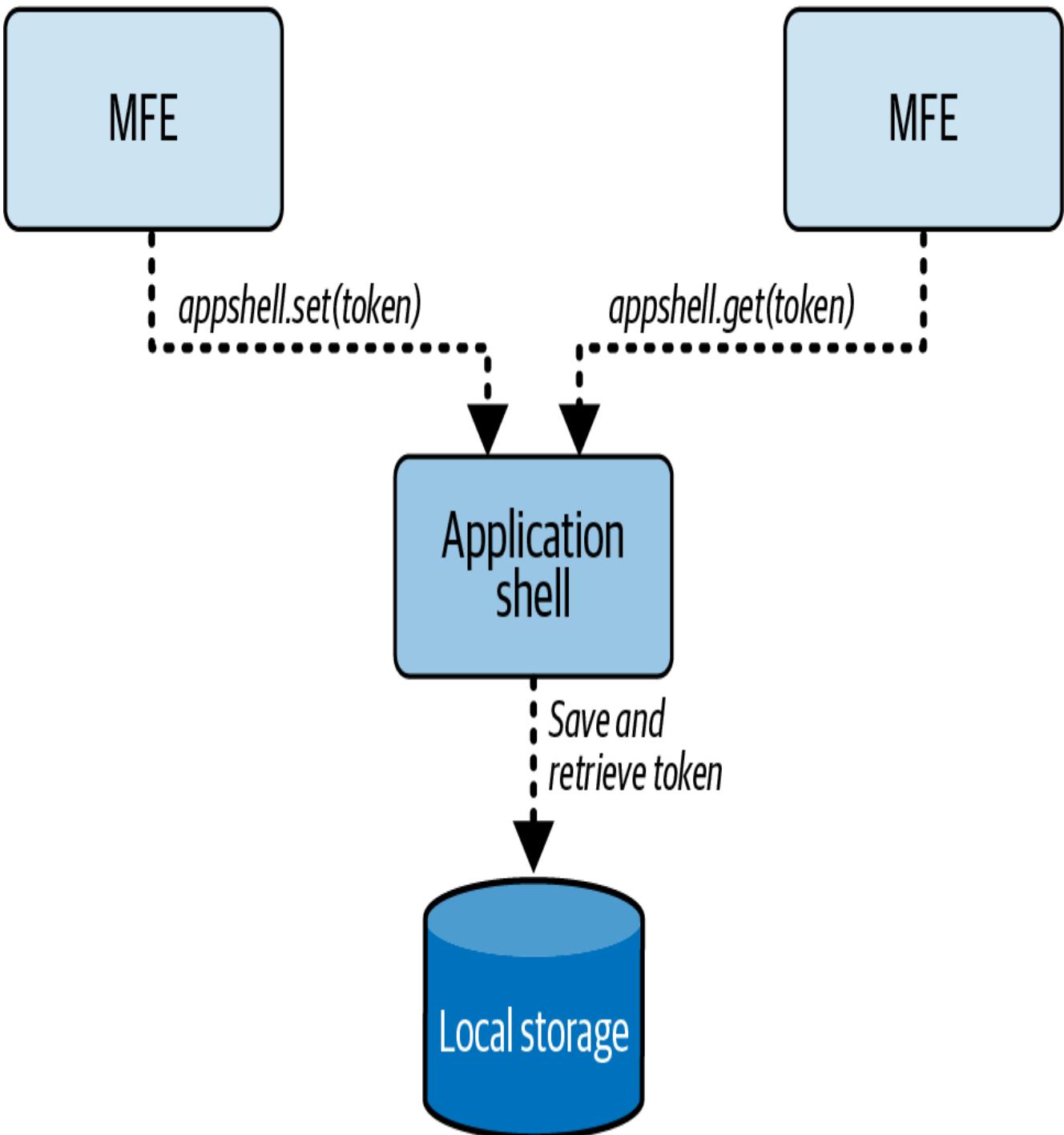


Figure 12-8. Application shell responsible for storing data in the local storage and exposing several APIs to the micro-frontends for storing and retrieving data

Table 12-1 summarizes the decisions made by the teams.

Table 12-1. Summary of ACME Inc. architectural decisions

Micro-frontend decision framework	
Defining micro-frontends	Vertical split
Composing micro-frontends	Client side via application shell
Routing micro-frontends	Client side via application shell
Communication between micro-frontends	Using web storage via application shell

Splitting the SPA into Multiple Subdomains

After creating their micro-frontend framework, the ACME Inc. tech teams analyzed the current application's user data to understand how the users were interacting with the platform. This is another fundamental step that provides a reality check for the teams. Often, what tech and product people envision for platform usage is very different from what users actually do.

The SPA was released with a Google Analytics integration, and the teams were able to gather several custom data points on user behavior for developing or tweaking features within the platform. This data is extremely valuable in the context that ACME Inc. operates, because it helps identify how to slice the monolith into micro-frontends.

Looking at user behaviors, the teams discover the following:

New users

Users who are discovering the platform for the first time follow the sign-up journey as expected. However, there are significant drops in visualization from one view to the next. As we can see in [Table 12-2](#), all the new users access the landing page, but only 70% of that traffic moves to the next step, where the account is created. At the third step (payment), there is a drop of an additional 10%. At the last step, only 30% of the initial traffic has been converted to customers.

Table 12-2. New user traffic per view in the ACME Inc. platform

View	Traffic
Landing page	100%
Sign-up	70%
Payment	60%
Catalog	30%

Unauthenticated existing users

Existing users who want to authenticate on a new browser or another platform (such as a mobile device) usually skip the landing page, going straight to the sign-in URL. After signing in, they have full access to the video catalog, as seen in **Table 12-3**.

Table 12-3. Unauthenticated existing user traffic per view for accessing the ACME Inc. platform

View	Traffic
Landing page (as entry point)	25%
Sign-in (as entry point)	70%

Authenticated existing users

Probably the most interesting result is that authenticated users are not signing out for days. As a result, they won't see the landing page

or sign-in/sign-up flows anymore. They occasionally explore their account page or the help page. But a vast majority of the time, authenticated users are staying in the authenticated area and not navigating outside of it (see **Table 12-4**).

Table 12-4. Authenticated existing user traffic per view for accessing the ACME Inc. platform

View	Traffic
Landing page	0%
Sign-in	1%
Sign-up	0%
Catalog	92%
My account	4%
Help	2%

This is extremely valuable information for identifying micro-frontends. In fact, ACME Inc. developers can assert the following:

- The landing page should immediately load for new users, giving them the opportunity to understand the value proposition.
- Landing page, sign-in, and sign-up flows should be decoupled from the catalog, because authenticated users only occasionally navigated to other parts of the application.
- The “my account” and “help” pages don’t receive much traffic.

- There is a considerable drop in new users between the landing page and sign-up flows, so we can expect the product team would like to make multiple changes to reduce this drop.

Another important aspect is understanding how the current architecture can be split into multiple subdomains following domain-driven design (DDD) practices. Taking into consideration the whole platform (not only the client-side part), the teams identified some subdomains and a relative bounded context.

For the frontend part, the subdomains that the teams took into consideration for their final decisions are as follows:

Value proposition

A subdomain for sharing all the information needed to decide on subscribing to the platform

Onboarding

A subdomain focused on subscribing new users and granting access to the platform for existing users, which may be split into smaller subdomains in the future (e.g., payment methods, user creation, user authentication) if complexities arise

Catalog

A core subdomain where ACME Inc. gathers the essential part of its business proposition, such as catalog, video player, and all controls for allowing users to consume content while respecting right holders' agreements

User management

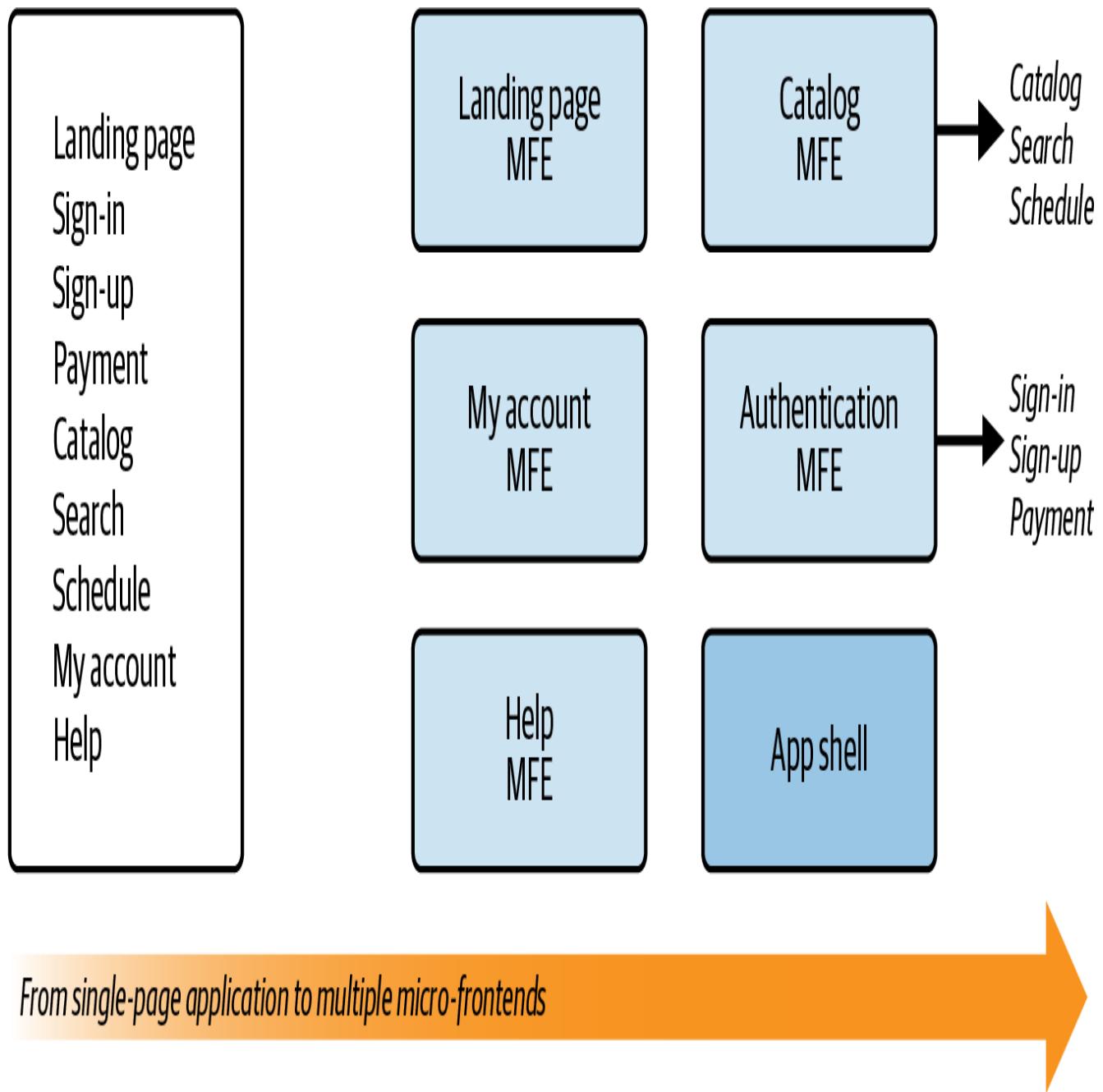
A subdomain where users can change account preferences, payment methods, and other personal information

Customer support

A subdomain for helping new and existing users to solve problems in any part of the platform

With this information in mind and the decisions made for how to approach this project using the micro-frontend decision framework, the teams identified the migration path

with the micro-frontends outlined in [Figure 12-9](#).



The micro-frontends are as follows:

Landing page

Considering that the landing page is viewed by all new users, the teams want to create a super-fast experience where the page is rendered in the blink of an eye. It needs its own micro-frontend, so all the technical best practices for a highly cacheable micro-frontend with a small size to download can be applied.

Figure 12-9. Migration path from an SPA to micro-frontends

Authentication

This micro-frontend is composed of all the actions an unauthenticated user should perform before accessing the catalog, such as moving from sign-in to sign-up view, retrieving their credentials, and so on.

Catalog

This is an authenticated area frequently viewed by authenticated users. The teams want to expedite the experience for these users when they return to the platform, so they encapsulate it in a single micro-frontend.

My account

This micro-frontend is a combination of information available in different domains of the backend, allowing users to manage their account preferences. It's available only for authenticated users. Considering the small traffic and the cross-cutting nature of this domain, ACME Inc. decided to encapsulate it in a micro-frontend.

Help

Like the “my account” micro-frontends, “help” has low traffic, a different use case from other micro-frontends, and highly cacheable content (because “help” pages are not updated very often). Encapsulating this subdomain in a micro-frontend allows ACME Inc. to use the right infrastructure for optimizing this part of the platform.

Application shell

This is the micro-frontend orchestrator. Because ACME Inc. decided to use a vertical split with a client composition, this element is mandatory to build. The main caveat is trying to keep it light and as decoupled as possible from the rest of the application so that all the other micro-frontends can be independent and evolve without any dependency on the application shell.

Technology Choice

Because the Angular SPA was developed some years ago, with patterns and assumptions that were best practices at that time, the ACME Inc. tech teams investigated their relevance and assessed new practices that might make the developers' lives easier and more productive. The teams agreed to use React, and they have discovered a development boost during their proof of concepts in the reactive programming paradigm.

Although Redux allows them to embrace this paradigm using libraries such as [redux-observable](#), they found in [MobX state tree](#) an opinionated and well-documented reactive state management that works perfectly with React and allows state composition, so they can reuse states across multiple views of the same micro-frontend. This will enhance the reusability of their code inside the same bounded context.

Thanks to the nature of the vertical-split micro-frontends—which load only one micro-frontend at a time—there is no need to coordinate naming conventions or similar agreements across multiple teams. The teams will mainly share best development practices and approaches to make the micro-frontends similar and to enable team members to understand the codebase of other micro-frontends or even join a different team.

The micro-frontends will be static artifacts, and therefore highly cacheable through a content delivery network (CDN), so there's no need for runtime composition on the server side. However, because of this aspect, the delivery strategy will need to change. Currently, ACME Inc. is serving all the static assets directly from the application server layer. Because the API integrations are happening on the client side, there will be no need to continue maintaining the application servers for serving static content—only for exposing the backend API.

ACME Inc. decided to use object storage (e.g., AWS S3), storing all the artifacts to serve in production in a regional bucket and enhancing the distribution across all the countries they need to serve using a CDN (e.g., Amazon CloudFront). This will simplify the infrastructure layer, reducing possible issues in production due to misconfiguration or scalability. Additionally, the frontend has a different infrastructure from the API layer, which enables the frontend developers to evolve their infrastructure as needed. This new infrastructure allows every team to independently deploy their micro-frontend artifacts (HTML, CSS, JavaScript files) in an AWS S3 bucket so they are automatically available for loading by the application shell.

Another goal for this migration is to reduce the risk of bugs in production when a new micro-frontend version is deployed, while still creating value for users and the

company without waiting for the entire application to be rewritten with the new architecture. Considering the simple frontend infrastructure chosen for the project, the ACME Inc. teams decided to leverage AWS Lambda@Edge—a serverless computation layer (see [Chapter 7](#) and [Chapter 8](#))—to analyze incoming traffic and serve a specific artifact to the application shell. This implements, de facto, a frontend canary release mechanism at the infrastructure level that doesn’t impact the application code but instead runs in the cloud—as shown in [Figure 12-10](#).

Thanks to this implementation, ACME Inc. can also apply the strangler fig pattern (see [Chapter 7](#)) for gradually moving to micro-frontends while maintaining the legacy application. In fact, they can use the application URL for triggering the Lambda@Edge that will serve the legacy or application shell to the user, as in [Figure 12-11](#).

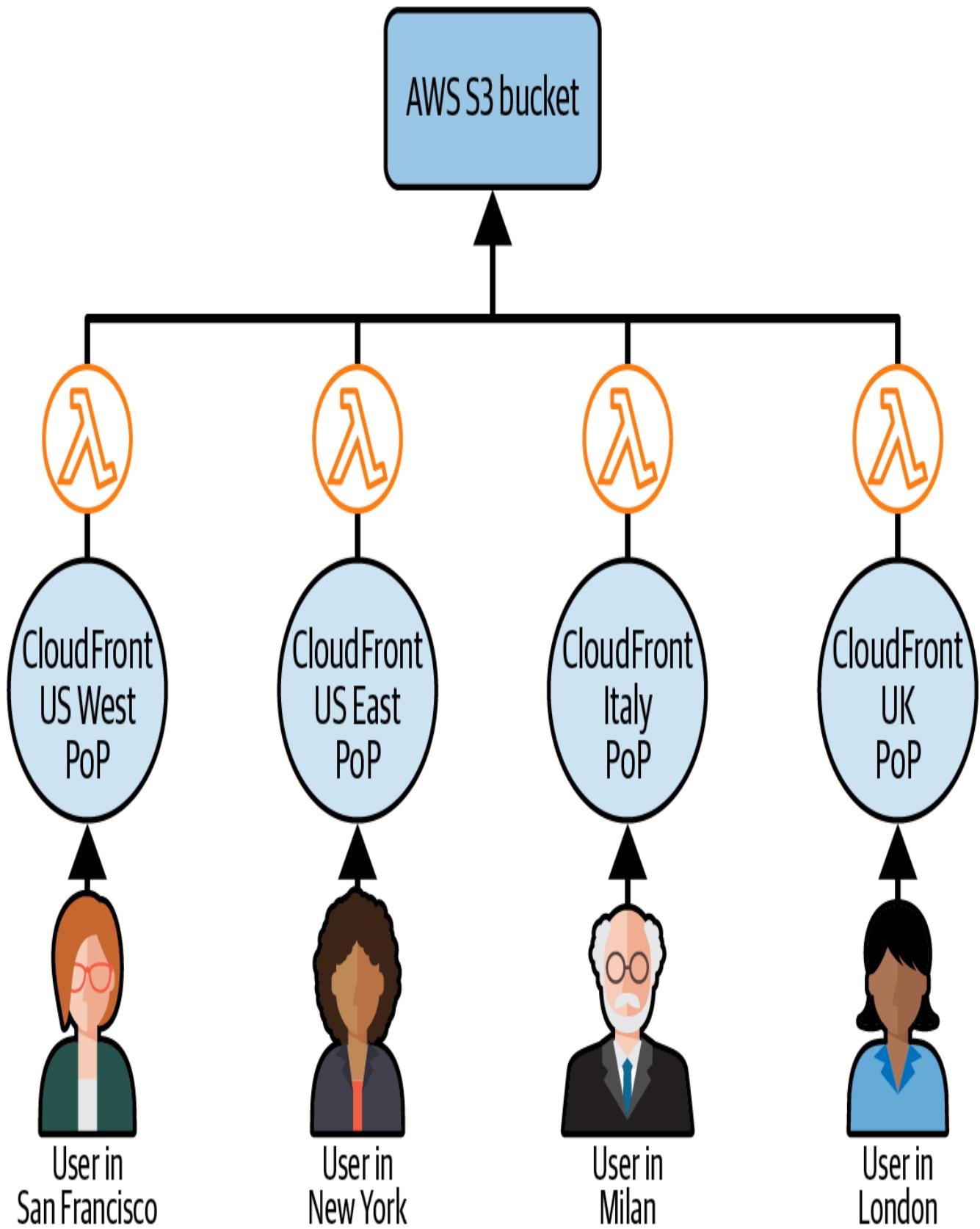


Figure 12-10. Simple infrastructure using an S3 bucket, Lambda@Edge, and a CDN (like Amazon CloudFront) with a point of presence (PoP) available all over the world

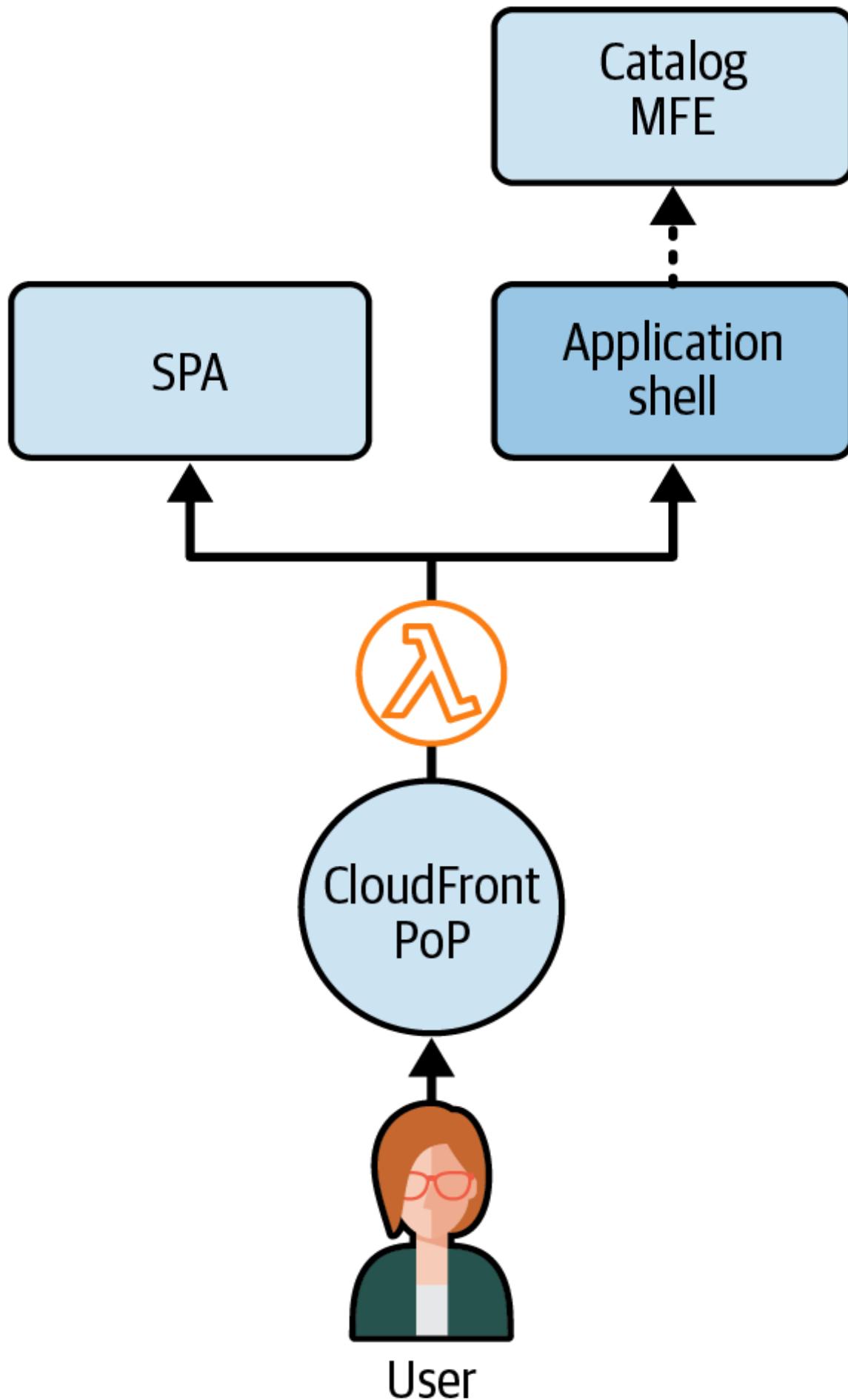


Figure 12-11. Strangler fig pattern applied at the infrastructure level using Lambda@Edge for funneling the traffic toward either the legacy or micro-frontend app

In the configuration file loaded by Lambda@Edge at the initialization phase, developers mapped the URLs belonging to the legacy application and to the micro-frontend application. Let's clarify this with an example.

Imagine that the catalog micro-frontend is released first—because, at this stage, you want all or part of the traffic going toward the micro-frontend branch (see [Figure 12-11](#)). The authentication remains inside the legacy application, so after the user signs in or signs up, the SPA will load the absolute URL for the catalog (e.g., www.acme.com/catalog). This request will be picked up by Amazon CloudFront, which will trigger the Lambda@Edge and serve the application shell instead of the SPA artifact.

This plan acknowledges that during the transition phase, a user will download more library code than before, because they're downloading two applications at the end. However, this won't happen for existing users; they will always download the micro-frontend implementation, not the legacy one.

As you can see, there is always a trade-off to make. Because ACME Inc. set the goal of finding a way to mitigate bugs in production and generate value immediately, these were the points they had to optimize for—especially if this is just a temporary phase until the entire application is switched to the new architecture. At this stage, ACME Inc. teams have made enough decisions to start the project effectively. They opt to create a new team to take care of the catalog micro-frontend, which will be the first to be deployed into production when ready.

The teams know that the first micro-frontend will take longer to be ready because, in addition to migrating the business logic toward this new architecture style, the new team has to define the best practices for developing a micro-frontend, which other teams will follow. For this reason, the catalog team begins with some proofs of concept to nail down a few details—such as how to share the authentication token between the SPA and the micro-frontend initially and then between micro-frontends once the application is fully ported to this pattern, or how to integrate with the backend APIs while considering the migration on that side as well as the potential impact to API endpoints, contracts, and so on.

Initially, the team splits the work into two parts. Half of the team works on the automation pipeline for the application shell and the catalog micro-frontends. The other half focuses on building the application shell. The shell should be a simple layer that

initializes the application, retrieves the configuration for a specific country, and orchestrates the micro-frontend life cycle—such as loading and unloading micro-frontends or exposing functions for notifying when a micro-frontend is fully loaded or about to be unloaded.

The first iteration of this process will be reviewed and optimized when more teams join the project. The automation and application shell will be enhanced as new requirements arise or new ideas to improve the application are applied.

Implementation Details

After identifying the next steps for the architecture migration, ACME Inc. has to solve a few additional challenges along the migration journey. These challenges—such as authentication and dependency management—are common in any frontend project. Implementing the features discussed in this subsection within a micro-frontend architecture may have some caveats that differ from other architectures. The topics we'll cover are:

- Application shell responsibilities
- Integration with APIs, taking into account the migration to microservices that is happening in parallel
- Implementation of an authentication mechanism
- Dependency management between micro-frontends
- Components sharing across multiple micro-frontends
- Introducing design consistency in the UX
- Canary releases for frontend
- Localization

In this way, we will cover the most critical aspects of a migration—from SPA to micro-frontends. This doesn't mean there aren't other important considerations, but these topics are usually the most common ones for a frontend application, and applying them at scale is not always as easy as we may think.

Application Shell Responsibilities

The application shell is a key part of this architecture. It's responsible for mounting and unmounting micro-frontends, initializing the application. It's also responsible for retrieving data from web storage and triggering life cycle methods. Finally, the application shell knows how to route between micro-frontends based on a given URL.

Application Initialization

The first task of the application shell is to consume an API that retrieves the platform configuration stored in the cloud. This configuration includes a services dictionary with a few endpoints used for validating tokens before granting access to authenticated areas, as well as a list of micro-frontends available to mount.

After retrieving the configuration from the backend, the application shell performs several actions:

- Exposes the relevant part of the configuration to any micro-frontends by appending it to a `window` object so that every mounted micro-frontend can access it
- Validates any token in web storage through the API layer, routing the user to the authenticated area (if they're authorized) or to the landing page (if they're not)
- Mounts the right micro-frontend depending on the user's state (authenticated or not)

Communication Bridge

The application shell offers a small set of APIs that every micro-frontend can use for storing or retrieving data or for dealing with life cycle methods. There are three important goals addressed by the application shell exposing these APIs:

Exposing the life cycle methods

It ensures memory is freed before a micro-frontend is unmounted, removes listeners, and initializes the micro-frontend once all resources are loaded

Managing access to web storage

It acts as the gatekeeper, so the underlying storage mechanism doesn't matter to individual micro-frontends. The application shell chooses the best way to store data based on the device or browser it is

running on. Remember that this application runs on web, mobile, and living room devices, so there is a huge fragmentation of storage to take care of. It can also perform checks on memory availability and return consistent messages to the user in case not all the permissions are available in a browser.

Enabling data sharing

It allows micro-frontends to share tokens or other data through in-memory or web storage APIs.

All APIs exposed to micro-frontends will be available at the `window` object in conjunction with the configurations retrieved by consuming the related API.

Mounting and Unmounting Micro-Frontends

Since ACME Inc.'s micro-frontends will have HTML files as an entry point. The application shell needs to parse the HTML file and append itself the related tags. For instance, any tag available in the body element of the HTML file will be appended inside the application shell body. In this way, the moment an external file tag is appended within the application shell DOM—such as JavaScript or a CSS file—the browser fetches it in the background. There is no need to create custom code for handling something that is already available at the browser level.

To facilitate this mechanism, the teams decided to add an attribute in the HTML elements of every micro-frontend for signaling which tags should be appended and which should be ignored by the application shell.

Sharing State

A key decision made at ACME Inc. is to keep state sharing between micro-frontends as lightweight as possible. Thus, no domain logic should be shared with the application shell—it should only be used to store and retrieve data from web storage. Because the vertical-split architecture means only one micro-frontend can load at a time in the application shell, the state is very well encapsulated inside the micro-frontend. Only a few things are shared with other micro-frontends, such as access tokens and temporary settings that should expire after a user ends the session. Some components will be shared across multiple micro-frontends, but in this case, there won't be any shared

states—just well-defined APIs for the integration and a strong encapsulation for hiding the implementation details behind the contract.

Global and Local Routing

Last but not least, the application shell knows which micro-frontend to load based on the configuration loaded at runtime, where a list of micro-frontends and their associated paths is available. In this configuration, every micro-frontend has a global path that should be linked to it. For example, the authentication micro-frontend is associated with *acme.com/account*, which will load when a user types the exact URL or selects a link to that URL.

When a micro-frontend is an SPA, it can manage a local route for navigating through different views. For instance, in the authentication micro-frontend, the user can retrieve a password or sign up for the service. These actions have different URLs available, so that the logic will be handled at the micro-frontend level. The application shell is completely unaware, then, of how many URLs are handled inside the micro-frontend logic. The only important URL part handled by the application shell is the first-level URL (e.g., *acme.com/first-level-url*).

In fact, the micro-frontend appends a parameter belonging to a view to the path. The sign-up view, for instance, will have the following URL: *acme.com/account/sign-up*. The first part of the URL is owned by the application shell (global routing), while the sign-up part is owned by the micro-frontend. In this way, the application shell will have a high-level logic for handling global routing for the application, and the micro-frontend will be responsible for knowing how to manage the local routing and evolving. This avoids the need to change anything in the application shell codebase.

Migration Strategy

During the migration period, the application shell will live alongside the SPA. This means that ACME Inc. can deliver incremental value to its users, testing that everything works as expected and redirecting traffic to the SPA if it finds some bugs or unexpected behavior in the new codebase. The trade-off will be in the platform's performance, because the user will download more code than previously. However, this method will enable one of the key business requirements: reducing the risk of introducing the micro-frontend architecture. In combination with the canary release, this will make the migration bulletproof to massive issues, thanks to several levers the teams can pull if any inconveniences are found during the migration journey.

Backend Integration

Because ACME Inc. is migrating the backend layer from a monolith to microservices, the first step will be a lift and shift, in which they will migrate endpoint after endpoint from the monolith to microservices. Using a [strangler fig pattern](#), they will redirect traffic to either the monolith or a new microservice. This means the API contract between frontend and backend will remain the same in the first release. There may be some changes, but they will be the exception rather than the rule.

This approach allows ACME Inc. to work in parallel at different speeds between the two layers. However, it may also create a suboptimal solution for data modeling. The drastic changes required to accommodate the distributed nature of microservices mean that some services may not be designed as well as they can be. For the teams, though, this is an acceptable trade-off, considering there are a lot of moving parts to define and learn about on this journey. The tech teams agreed to revisit their decisions and design after the first releases to improve the data modeling and API contracts.

Based on this context, the development and platform teams agreed to use load balancers to funnel the traffic to the monolithic or microservices layer, so that the client won't need to change much. Every change will remain at the infrastructure level. Deciding the best way to roll out a new API version can be done without the client even being aware of all these changes.

The client will fetch the list of endpoints at runtime via the configuration retrieved initially by the application shell, eliminating the need to hardcode the endpoints in the JavaScript codebase.

Integrating Authentication in Micro-Frontends

One of the main challenges of implementing micro-frontend architecture is dealing with authentication, especially with a shared state across multiple micro-frontends. The ACME Inc. teams decided to ensure that the application shell is not involved in the domain logic, keeping every micro-frontend as independent as possible. Thanks to the vertical-split approach, sharing data between micro-frontends is a simple action because they can use web storage or a query string for passing persistent data (e.g., simple user preferences) or volatile data (e.g., product ID).

ACME Inc. uses the local storage in its monolithic SPA for storing basic user preferences that don't require synchronization across devices, such as the video player's volume level and the JSON web token (JWT) used for authenticating the user. Because the developers want to generate immediate value for their users and company,

they decided to stick with this model and deliver the authenticated area of the catalog alongside the SPA. When a user signs up or signs in within the SPA, the JWT will be placed in the local storage. When the application shell loads the catalog micro-frontend, the micro-frontend will then request the token through the application shell and validate it against the backend (see [Figure 12-12](#)).

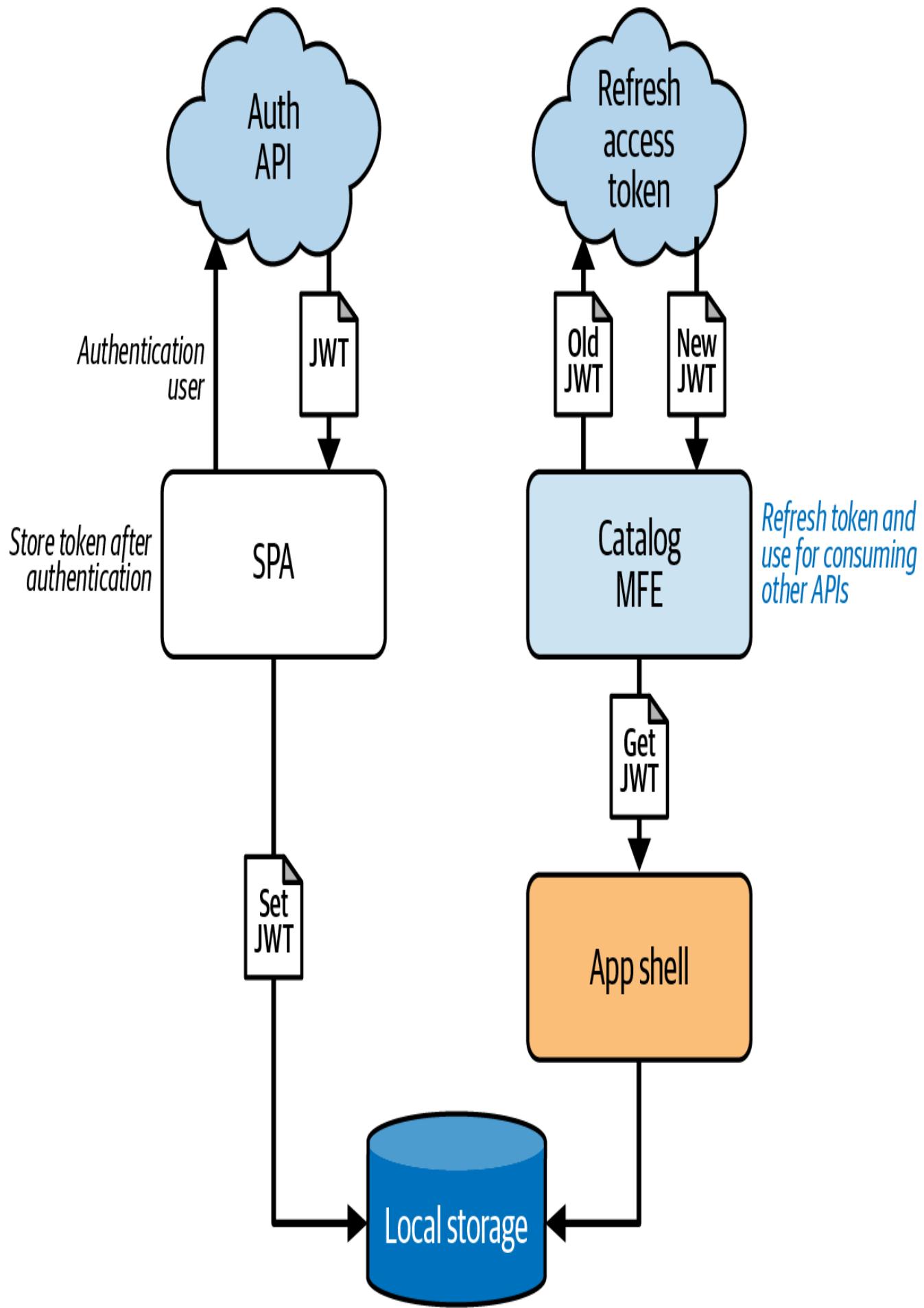


Figure 12-12. The SPA authenticating a user and storing the token in the local storage during migration, which the authenticated micro-frontend retrieves once loaded

Due to the local storage security model, the SPA, application shell, and all micro-frontends have to live in the same subdomain because the local storage is accessible only from the same subdomain. Therefore, the SPA will have to be moved from being served by an application server to the AWS S3 bucket, where the new architecture will be served from.

LOCAL STORAGE SECURITY MODEL

The data processed using the `localStorage` object persists through browser shutdowns, while data created using the `sessionStorage` object will be cleared after the current browsing session.

It's important to note that this storage is origin-specific. This means that a site from a different origin cannot access the data stored in an application's local database. For instance, if we store some website data in the local storage on the main domain `www.mysite.com`, the data stored won't be accessible by any other subdomain of `mysite.com` (e.g., `auth.mysite.com`).

Thanks to this approach, ACME Inc. can treat the SPA as another micro-frontend with some caveats. When it finally replaces the authentication part and finishes porting to this new architecture, every micro-frontend will have its own responsibility to store or fetch from the local storage via the application shell (as in [Figure 12-13](#)).

After the architecture migration, ACME Inc. will revisit where to store the JWT. The usage of local storage exposes the application to cross-site scripting (XSS) attacks, which may become a risk in the long run when the business becomes more popular—when more hackers might be interested in attacking the platform.

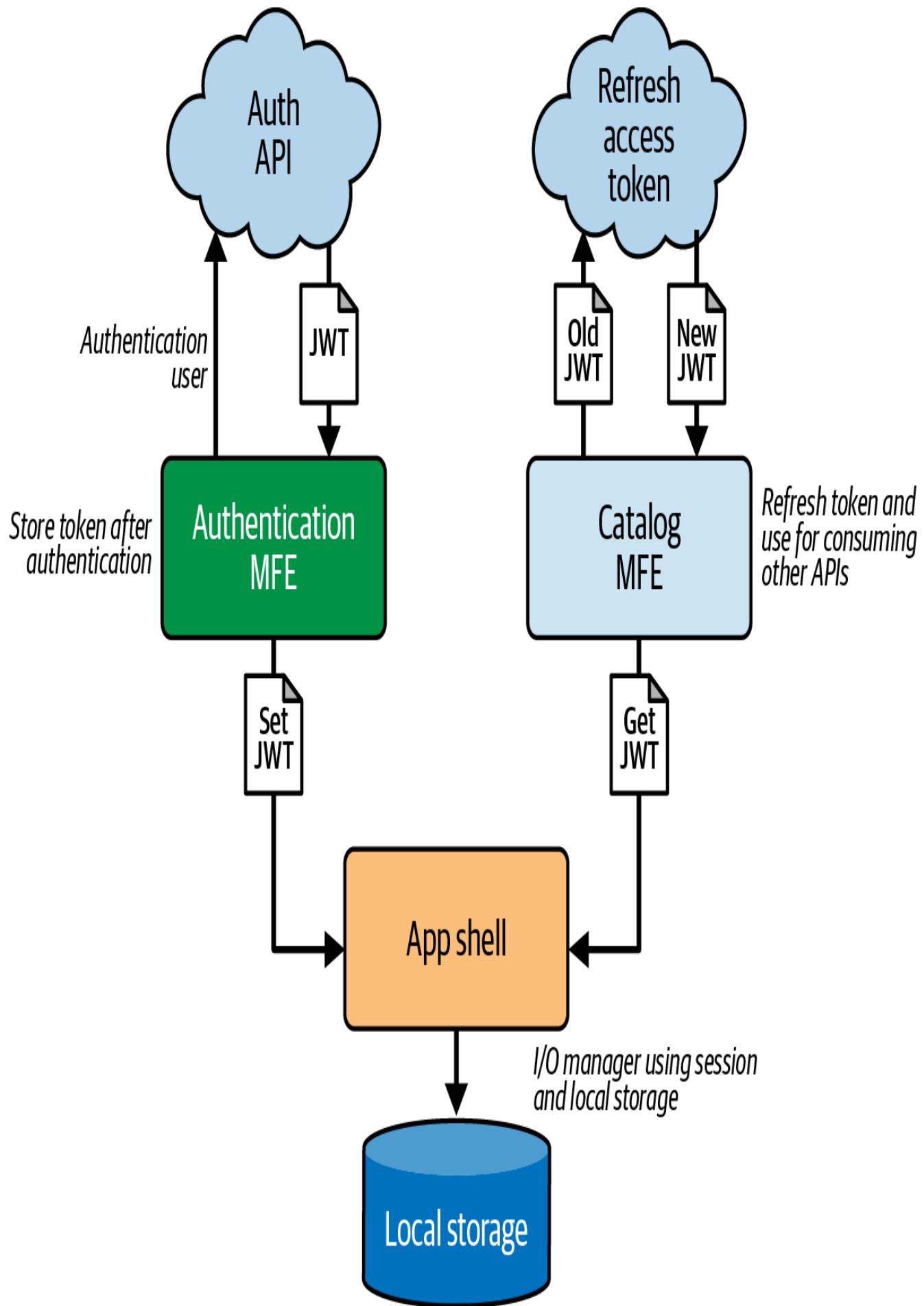


Figure 12-13. Every micro-frontend being responsible for managing part of the user authentication when the frontend platform is fully migrated to micro-frontends

CROSS-SITE SCRIPTING (XSS)

XSS attacks are a type of injection in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser-side script, to a different end user. Flaws that allow these attacks to succeed are widespread, and they occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way of knowing that the script should not be trusted and will execute it. Because the browser thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information that the browser retains and uses with that site. These scripts can even rewrite the content of the HTML page.

Dependency Management

ACME Inc. decided to share the same versions of React and MobX with all the micro-frontends, reducing the code the user has to download. However, the teams want to be able to test new versions on limited areas of the application so that they can test new functionalities before applying them to the entire project. They decided to bundle the common libraries and deploy to the AWS S3 bucket used for all the artifacts. This bundle doesn't change often and is therefore highly cacheable at the CDN level, as shown in [Figure 12-14](#).

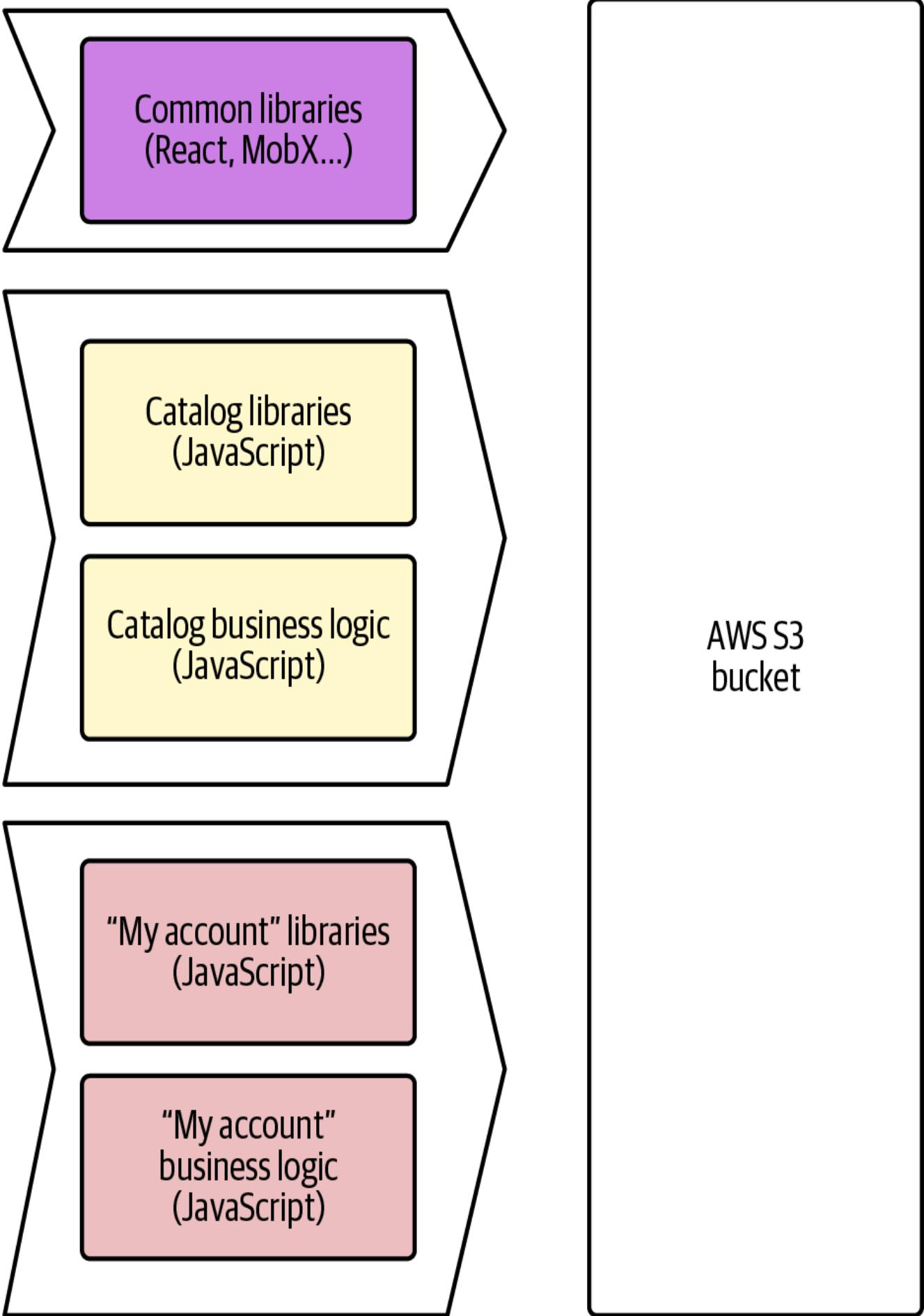


Figure 12-14. Every micro-frontend building and deploying its own JavaScript dependencies, apart from the common libraries, which have a separate automation strategy

Other teams that want to experiment with new common library versions can easily deploy a custom bundle for their micro-frontend alongside the other final artifact files and use that version instead of the common one.

In the future, ACME Inc.'s teams are planning to enforce bundle size budgets in the automation pipelines for every micro-frontend to ensure there won't be an exploit of libraries bundled together, which increases the bundle size and the time to render the whole application. This way, ACME Inc. aims to keep the application size under control while keeping an eye on the platform evolution, allowing the tech teams to innovate in a frictionless manner.

Integrating a Design System

To maintain user interface (UI) consistency across all micro-frontends, the tech teams and the UX department decided to revamp the design system available for the SPA using Web Components instead of Angular. Migrating to Web Components allows ACME Inc. to use the design system during the transition from monolith to distributed architecture, maintaining the same look and feel for the users.

The first iteration would just migrate the components from Angular to Web Components, maintaining the same UI. Once the transition is complete, there will be a second iteration where the web components will evolve with the new guidelines chosen by the UX department.

The initial design system was extremely modular, so developers could pick basic components to create a more complex one. The modularity also means the design system library will not be a huge effort to migrate, and the implementation will be as quick as it was before.

Due to the distributed nature of the new architecture, ACME Inc. decided to enforce, at the automation-pipeline level—using a fitness function—a validation that requires every micro-frontend to use the latest version of the design system library. This helps them avoid potential discrepancies across micro-frontends and ensure all teams are kept up to date with the latest version of the design system. The fitness function will control the existence of the design system in every micro-frontend's `package.json` and validates the version against the most up-to-date release. The build automation will be blocked and return a message in the logs, so the team responsible for the micro-frontend will know the reason why their artifact wasn't created.

Sharing Components

ACME Inc. wants a fast turnaround on new features and technical improvements to reduce external dependencies between teams. At the same time, it wants to maintain design consistency and application performance, so it will share some components across micro-frontends. The guidelines for deciding whether a component may be shared are based on complexity and the evolution, or enhancement, of a component.

For example, the footer and header used to be changed once a year. Now, however, these components will change based on user status and the area a user is navigating. The solution applied to the header and footer will be created with the different modular elements exposed by the design system library. These two elements won't be abstracted inside a component, because the effort to maintain this duplication is negligible, and there are only a few micro-frontends to deal with. These decisions may be reversed quickly, however, if the context changes and there are strong reasons for abstracting duplicated parts into a component library.

To avoid external dependencies for releasing a new version or bug fix within a component, the teams decided to load components owned by a single specialized team—like the video player components—at runtime. As a key component of this platform, the video player evolves and improves constantly, so it's assigned to a single team that specializes in video players for different platforms. The team optimizes the end-to-end solution—from encoders and packagers to the playback experience. Because the header and footer will load at runtime, they won't need to wait until every micro-frontend updates the video player library. The video player team will be responsible for avoiding contract-breaking changes without needing to notify all the teams consuming the component.

ACME Inc. will make an exception for the design system. Although it's built by a team focused only on the consistency of the UX, the design system will be integrated at development time to allow developers to control the use of different basic components and to create something more sophisticated within their micro-frontends. As shown in [Figure 12-15](#), all the other components will be embedded inside a micro-frontend at development time, like any other library, such as React or MobX.

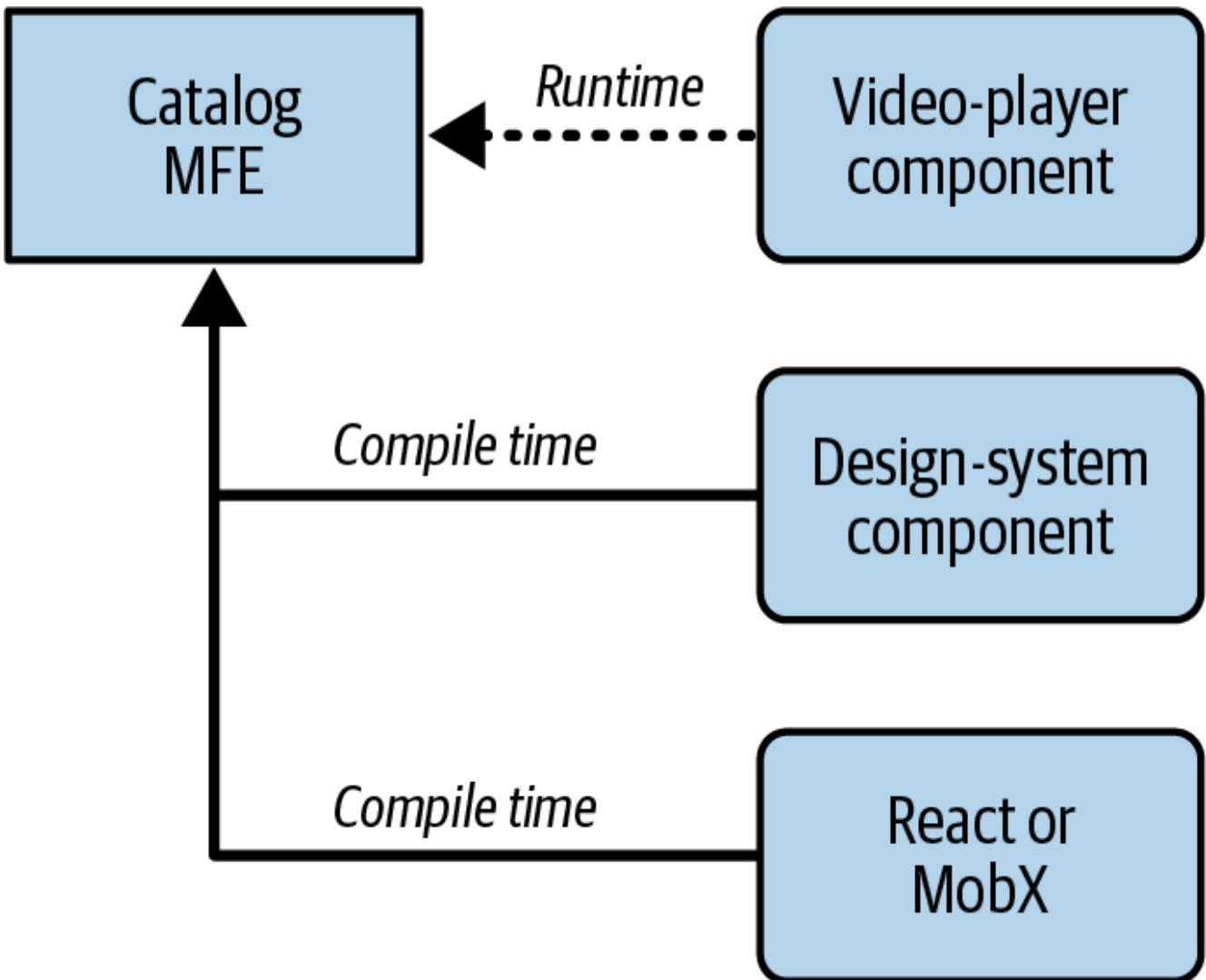


Figure 12-15. Complex components owned by a single team loaded at runtime, and all others embedded at compile time, with the design system being the only exception due to its modular nature

None of the components created within each team will be shared among multiple micro-frontends. If there are components that might simplify multiple teams' work if shared, a committee of senior developers, tech leads, and architects will review the request and challenge the proposal according to the principle defined at the beginning of the project. These principles will be reviewed every quarter to make sure they are still aligned with the platform evolution and business roadmap.

Implementing Canary Releases

Another goal of this project is to be able to release often in production and gather real data directly from the users. It's a great target to aim for, but it's not as easy to reach as we may think.

Based on its infrastructure for serving frontend artifacts, ACME Inc. decided to implement a canary release mechanism at the edge, so that it can extend the logic of its

Lambda@Edge once the migration is completed, adding logic to manage the micro-frontend releases.

ACME Inc. will also need to modify the application shell to request specific micro-frontend versions and delegate the retrieval of the exact artifact version to Lambda@Edge. The tech teams decided to identify every micro-frontend release using **semantic versioning (semver)**. This enables them to create unique artifacts by appending semver to the filename, and easily avoid caching problems when they release new versions.

SEMANTIC VERSIONING

Given a version number MAJOR.MINOR.PATCH, like 1.1.0, increment as follows:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backward-compatible manner
- PATCH version when you make backward-compatible bug fixes

Additional labels for prerelease and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

First, the application shell retrieves a configuration from the APIs. The configuration contains a map of available micro-frontend versions where only the major version is specified (e.g., 1.x.x). This allows the teams to upgrade the application while maintaining backward compatibility. They also only need to upgrade the major version when an API breaking change updates the configuration file served by the backend.

When the artifact request hits Amazon CloudFront, a Lambda@Edge that retrieves a list of versions available for the micro-frontends is triggered; the traffic should then be redirected to a specific version. The logic inside the lambda will associate a random number—from 1 to 100—to every user. If a user is associated with 20%, and 30% of the traffic should be redirected to a new version of the requested micro-frontend, that user will see the new version. All the users with a value higher than 30 will see the previous version. The sequence diagram in [Figure 12-16](#) illustrates this process.

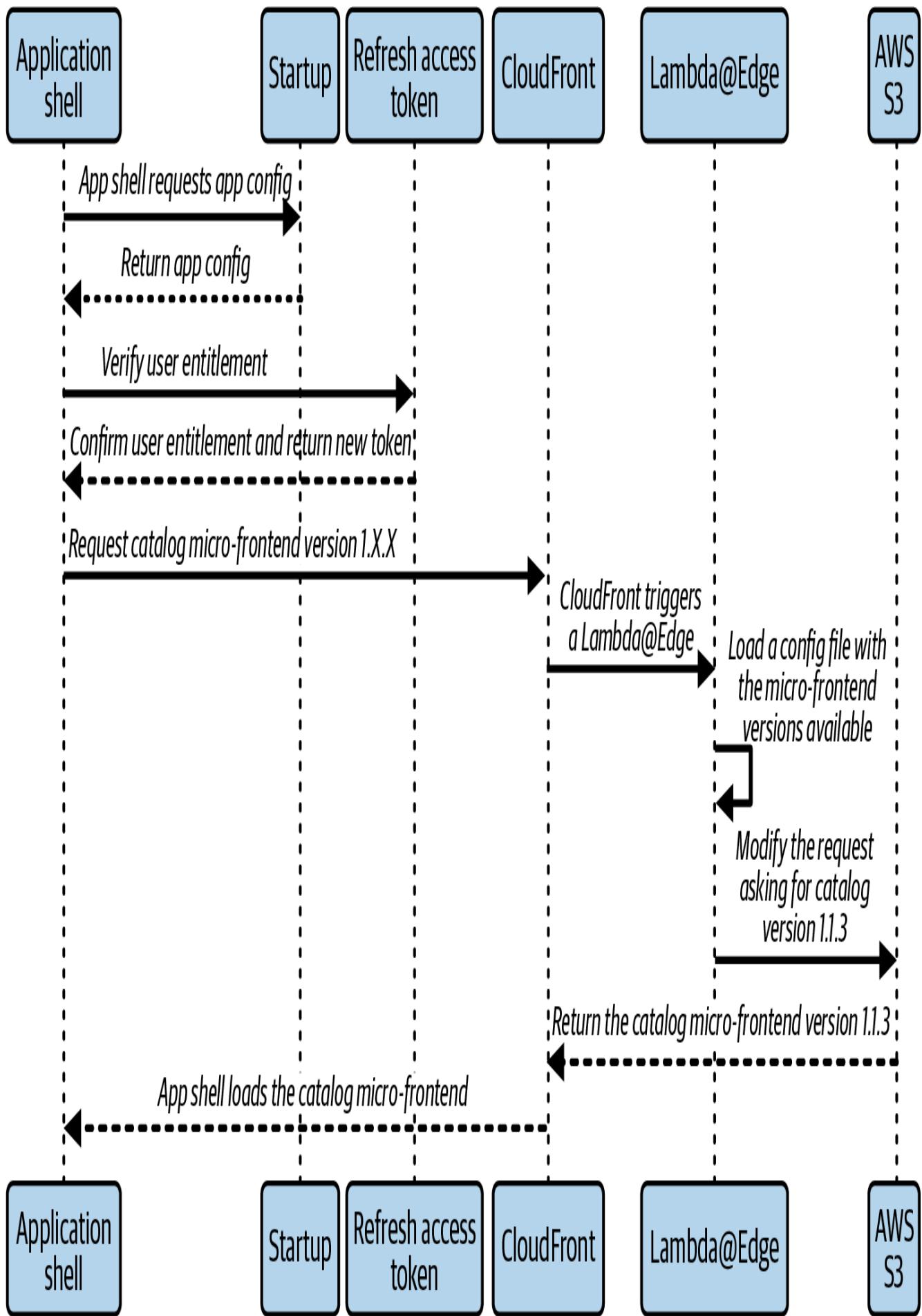


Figure 12-16. How ACME Inc. implements canary releases for micro-frontends

The lambda returns the selected artifact and generates a cookie where the random value associated with the user is stored. If the user comes back to the platform, the logic running in the lambda will validate just the rule applied to the micro-frontend requested and evaluate whether the user should be served the same version or a different one based on the traffic patterns defined in the configuration. As a result, both authenticated and unauthenticated traffic will have a seamless experience during the canary exploration of an artifact.

Using this mechanism, ACME Inc. can reduce the risk of new releases without compromising fast deployment, because it can easily move users from newer versions to an older one simply by modifying the configuration retrieved by the Lambda@Edge.

The team plans to introduce a **Frontend Service Discovery** to facilitate canary releases once the system has fully migrated to micro-frontends (as discussed in [Chapter 9](#)). This approach will allow seamless integration with the initial call made by the application shell, ensuring that the number of API requests made at the start of the application remains consistent, while also supporting the canary release process.

Localization

The ACME Inc. application needs to render in different languages based on the user's country. By default, the application will render in English, but the product team wants the user to be able to change the language in the application and have the option to retain this choice for authenticated users within their profile settings, creating a seamless experience for the user across all their devices.

In this new architecture, ACME Inc. tech teams must consider two forces:

- Every micro-frontend has a set of labels to display in the UI, some of which may overlap with other micro-frontends, such as common error messages.
- Every micro-frontend represents a business subdomain, so the service must return just enough labels to display for that specific subdomain and not much more, or resources will be wasted.

ACME Inc. tech teams decided to modify the dictionary API available in the monolith to return only the labels needed within a micro-frontend. In this way, the SPA can still receive all the labels available for a given language, and the micro-frontend will only receive the label needed for its subdomain during the transition (see [Figure 12-17](#)). At

migration completion, all the micro-frontends will consume the microservices API instead of the legacy backend, and there won't be a way to retrieve all the labels available in the application through the legacy backend.

When a micro-frontend consumes the dictionary API, it has to pass the subdomain as well as the language and country related to the labels in the request body in order to display them in the UI. When it receives the request, the microservice will fetch the labels from a database based on the user's country and preferred language and the micro-frontend subdomain.

Because micro-frontends are not infinite, and the platform supports fewer than a dozen languages, having a CDN distribution in front of the microservice will allow it to cache the response and absorb the requests coming from the same geographical area.

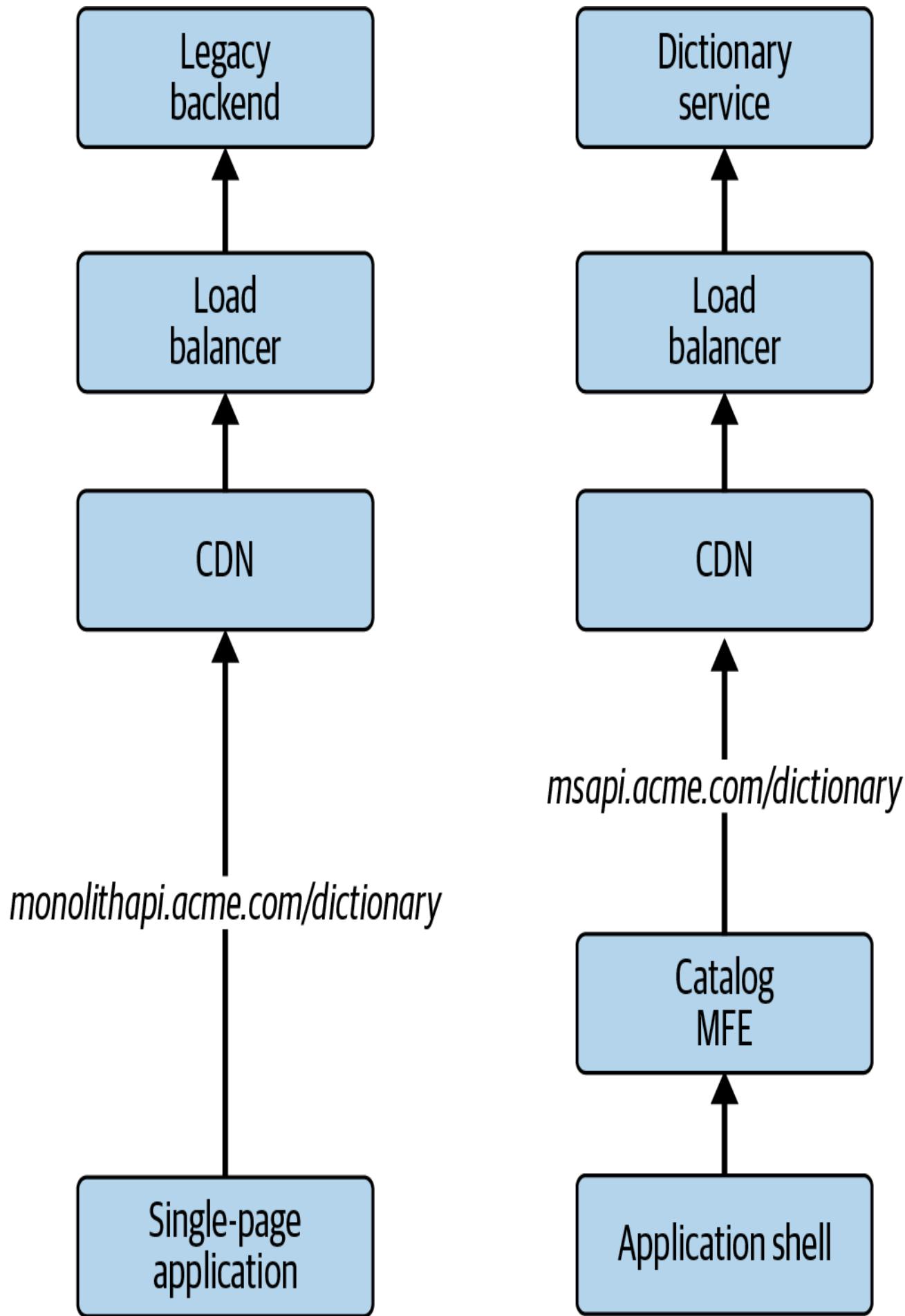


Figure 12-17. The micro-frontend consuming a new API for fetching the labels to display in the interface through a new microservice, and the SPA consuming the API from the legacy backend

Being able to rely on the monolith via a different endpoint during micro-frontend development creates a potential fallback, if needed. It allows older versions of native applications on mobile devices to continue working without any hiccups.

Summary

In this chapter, we have gathered all the insights and suggestions shared across the book and demonstrated how they play out in a real-world example. Sharing the reasoning behind certain decisions—the *why*—is fundamental for finding the right trade-off in architecture and, really, in any software project. When you don’t know the reasons for certain decisions within your organization, I encourage you to find someone who can explain them to you. You will be amazed to discover how much effort is spent before finding the right trade-offs between architecture, business outcomes, and timing.

In your career, you will see that what works in one context won’t necessarily work in another because there are so many factors stitching the success of a project together—such as people skills, environment, and culture. Common obstacles include the seniority of engineers, company culture, communication flows that don’t map team interactions, dysfunctional teams, and so on.

When we develop any software project at scale, there are several aspects we need to take into consideration as architects and tech leaders. In this chapter, I wanted to highlight the thought processes that transitioned ACME Inc. from using an SPA to adopting micro-frontends, because migration is very likely going to be your first step toward a micro-frontend architecture. Some of the reasoning shared in these pages may help you steer your project in the right direction toward success.

One thing that I deeply like about micro-frontends is that we finally have a strong say in how to architect our frontend applications. With SPAs, we followed well-known frameworks that provided speed of development and delivery because they solved many architectural decisions for us. Now, we can leverage these frameworks and contextualize them, using their strengths in relevant parts of our projects.

Now is our chance to shape this space with new tools, practices, and patterns. Imagine a world where micro-frontends empower teams to work independently, yet seamlessly integrate their pieces into a cohesive whole. Think about the agility and speed we can achieve by breaking down monolithic structures and embracing a modular approach.

With micro-frontends, we're not just building applications; we're crafting experiences that can evolve and scale with ease.

The only thing holding us back is our imagination. So, let's dive in, experiment, and push the boundaries of what's possible.

Chapter 13. Introducing Micro-Frontends in Your Organization

You've arrived at the second-to-last chapter of this book and have learned a lot. We have explored how to create micro-frontends, the best architectural approaches for your project, and all the best practices to follow to make your project successful. It's time to start your project and write a few lines of code, right?

Not quite. There are still some key topics related to the human aspect that we must take into consideration when we introduce this architecture, just as we do whenever we revisit our architecture or introduce a new one. When making significant changes to architecture, we need to think about how to organize the communication flows, how to avoid siloed groups, and how to empower the developers to make the right decisions inside a business domain. These are just some of the many important considerations related to the human side of the project we need to consider at the beginning and during the entire life cycle.

Micro-frontends may help you mitigate some of these considerations, but they can make others more complex if not approached properly. Therefore, it's crucial for you to invest the time needed to analyze your current organization structure and see how it would fit inside your new architecture.

Why Should We Use Micro-Frontends?

Tech leaders and chief technology officers often ask this question when someone introduces the idea of micro-frontends within an organization. It's a valid question, and the best way to answer it is to use a common language to evaluate the benefits of this architecture paradigm.

Micro-frontends bring several benefits to the table, such as team independence, riskless deployment, reduction of cognitive load for developers, fast iterations, and innovation. Despite all of that, they also bring challenges, such as the risk of creating silos within the organization, higher investment in automation pipelines, and the risk of UI discrepancies.

When introducing the idea of micro-frontends to your organization, focus your presentation on not only the technical benefits but also the organizational benefits. Let me offer some food for thought to help you prepare an impactful presentation for your stakeholders:

- Point out that micro-frontends enable faster feature iterations and reduce the risk of introducing bugs into the entire application.
- Research and describe the context you operate in daily, and why micro-frontends may help you to achieve business goals.
- List the problems you are trying to solve with this paradigm.
- Ponder the best way to implement micro-frontends in your context.
- Analyze the impact this architecture may have on team communication.
- Identify the ideal governance for managing such an architecture.
- Retrieve metrics from your automation pipeline—such as time to deployment and testing—and think about how you would be able to improve them.

These are just some topics that are relevant to your organization's tech leaders or clients.

Remember to present not just a technical solution—which can leave many organizational challenges to overcome—and instead think about an end-to-end transformation that brings value to the company as well as to your customers. To discover the best technical solution for your context, I strongly encourage you to first run a proof of concept to understand the challenges and benefits of this approach better; what works for one team or organization doesn't always work for another. Be mindful, and share the insights that will work best for your organization with your peers and tech leaders. Try to involve the right people up front, because it can be difficult to understand the context in which you operate—especially in mid- to large-sized organizations, where you may be dealing with distributed teams whose culture and context vary from office to office. In the following sections, you'll discover insights on managing governance, documentation, organizational setup, and communication flows for a micro-frontend project.

Data to the Rescue

In recent years, I've spoken with numerous companies in the industry, from tech leaders to vice presidents worldwide. A common strategy for raising awareness about the necessity of a distributed system in frontend development involves establishing a baseline and setting goals to facilitate significant improvements.

Let me explain with an anecdote. In 2024, I started a video podcast where I interviewed engineers, architects, and micro-frontend practitioners. During an interview with Warren Fitzpatrick, principal engineer at Dunelm, he shared how he convinced management to embrace this model. I had a similar experience at my previous company, using the same approach to gain development approval. In both cases, the strategy involved identifying existing problems that hindered innovation and new feature development, establishing a baseline, and presenting potential improvements with data relevant to the business.

Warren shared that their key problem was the slow deployment of features to production due to convoluted automation pipelines and a high number of tests. Interestingly, Warren didn't emphasize technical improvements to his stakeholders. Instead, he focused on the slow pace of deploying new features and how a more modular strategy, like micro-frontends, could increase deployment frequency. Although their first attempt didn't achieve the desired results, they adjusted practices and revisited decisions. Now, they are fully committed to a server-side rendering micro-frontend architecture powered by serverless services in the cloud.

The main takeaway from this story is that identifying a problem that is slowing down the business can be an effective strategy for encouraging investment in a new approach. This method is not limited to micro-frontends but applies to any organizational improvement. Gathering data and proving, with a proof of concept, that the company could improve its current situation is a technique that I've used extensively in the past and continue to use. It might not always work, but it will certainly spark discussions within your tech leadership—guaranteed!

NOTE

If you are interested in learning more about [Dunelm's journey](#) and the insights from other companies that have embraced micro-frontends, I highly encourage you to listen to my “Micro-frontends in the Trenches” show on [Spotify](#), [Apple Podcast](#), or [YouTube](#).

Creating a Trade-Off Analysis

A methodical way to work with data when making architectural and design decisions is to conduct a trade-off analysis. This typically involves considering three key dimensions:

- Business requirements
- Architecture characteristics
- Organizational capabilities

NOTE

I strongly recommend documenting these findings to allow for revalidation of your decisions at any time. This practice will help your team make informed decisions and provide future team members with a clear understanding of the rationale behind the chosen architectural approach.

Business Requirements

Always start with what the business aims to achieve. Engage with key stakeholders to understand why a particular characteristic is important, what they want to attain, and why it matters. It is even better if you can gather some baseline metrics to work backward from.

For example, if the business wants to achieve a faster time to market for new features, investigate the current time frame from ideation to successful deployment of your workload and identify the bottlenecks. Visualizing these findings will help communicate the potential solutions more effectively and improve the current process.

Architecture Characteristics

The next step is to gather the architectural characteristics defined in collaboration with your tech leadership for implementation within the system. For instance, if you need to build a latency-sensitive solution in the cloud, a multi-region server-side approach could help reduce response times and improve the core web vitals of your application.

If you need to integrate a design system into a micro-frontend implementation, consider automating this process to accelerate the feedback loop for developers, such as when they open a pull request.

Organization Capabilities

Finally, you need to understand your specific context. This is crucial for selecting the right approach. Too often, companies adopt a mechanism or practice simply because a large company did, but this will not necessarily lead to success for all organizations.

I recommend identifying the bottlenecks that your developers face, reviewing the current feedback loop, and planning improvements through automation. Additionally, assess the skills you have in-house and consider providing training to enhance your team's capabilities. Every context is different, and it is essential to take this into account.

This is a valuable exercise to invest time in before starting your journey with micro-frontends. It will provide clarity of intent, set expectations, and ensure alignment across teams.

The Link Between Organizations and Software Architecture

What sort of software architecture should you be implementing? You'd be forgiven for wanting to copy others' success. But there is no such thing as the "best architecture." Your job is to find the best trade-off for your context. Perfection is unavailable, unfortunately. What we need to create is an architecture that fits our organization's needs and, especially, the context we operate in.

We often hear conference talks that explain a specific use case. The ideas and solutions the speaker brings up feel like a perfect fit for what we are trying to solve in our organization. Unfortunately, it's not always the case. In any talk or book, the solution to a problem is given from the perspective of just one person, who may represent only part of the organization. Often, the speaker or writer focuses more on the *how* and less on the context where their specific solution was successful. There's little on *why* this solution worked for that context. Yet that context provides the information we need to make the right trade-offs for our architectures.

What software architecture should you be implementing? Which business and architectural characteristics should you take into account? How would you express them in your design decisions? These are the initial questions everyone should start with. There is no single architecture that works well in all cases and at all times. You have to customize your architecture for your needs, applying the patterns that solve your problems and fit your situation best. Bear in mind that businesses evolve over time and, therefore, a good trade-off today may not be a good one tomorrow.

Modularity is the key to moving in the direction the business wants to go, allowing it to arrive faster and with minimal complexity. At the same time, modularity is far from a trivial task. It requires discipline, analysis, and a lot of work from everyone involved in the project. Micro-frontends are no exception to this. They’re not suitable for all projects. But they may be useful when you work in a mid- to large-sized environment, with three or more teams working exclusively on the frontend side of a project. If you have cross-functional teams already working with microservices, it’s very likely that micro-frontends are a suitable approach for your client-side application. Even with mobile applications, many organizations started to use micro-frontends in conjunction with React Native, for instance. They may also be really helpful when the organization needs to be scaled; when an application’s success depends on time to market; when we are transitioning from a legacy application to a new one and want to generate immediate value for our users instead of waiting several months before the application is finished; and in many other scenarios.

You may want to embrace a micro-frontend implementation that is different from the approaches described in this book, or even try new approaches for solving specific problems within your organization. This is absolutely fine—as long as there are strong reasons for doing so and the new trade-off will benefit a team, the entire organization, or a process. Remember, we are writing code for our customers or users, and this should be at the forefront of every developer’s mind when working on a software project.

How Do Committees Invent?

In [Chapter 2](#), I briefly introduced Conway’s Law: “Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization’s communication structure.” This law is from the 1968 paper [“How Do Committees Invent?”](#) by Melvin Conway. In his paper, Conway explains how software architecture is usually designed alongside the company structure. But this is not always the case. Sometimes we want to focus on a high-level architecture that is the best trade-off for designing a platform and then restructure our teams around that architecture. In these situations, we are applying the opposite technique, called the Inverse Conway Maneuver. In the “Stages of Design” section of the paper, where he describes the steps for designing software architectures, Conway recommends the following:

- An understanding of the boundaries, both on the design activity and on the system to be designed, set by the sponsor and by the world's realities
- Achievement of a preliminary notion of the system's organization so that design task groups can be meaningfully assigned

Though these principles are half a century old, they feel more relevant than ever. In the book *Accelerate*, authors Nicole Forsgren, Jez Humble, and Gene Kim share incredible research on the best practices of high-performance organizations from across multiple industries. The Inverse Conway Maneuver plays an important part in the sociotechnical aspect of every high-performance organization that the authors study.

Architecture and team communication are strongly linked. It's crucial to understand and internalize this, because it will greatly influence which micro-frontend architecture we decide to use. Ideally, we would design the best architecture possible for a given context and then assign teams to fulfill the design, but that's not always possible. In fact, in my experience, it's rarely possible, but it could happen sometimes. In cases where we need to respect the current organizational structure, we must consider the teams' current communication flows and daily interactions during our process in order to design an architecture suitable for our teams. Realizing that communication flow and architecture are linked enables you to aim for the best architecture for the context you operate within.

When considering communication flows, we need to distinguish between co-located and distributed teams. Sam Newman shared a very valid point in an [article about Conway's law](#) (with my emphasis in italics):

"The communication pathways that Conway refers to are in contrast to the code itself, where a single codebase requires fine-grained communication, but a distributed team is only capable of coarse-grained communication. Where such tensions emerge, looking for opportunities to split monolithic systems up around organizational boundaries will often yield significant advantages."

The communication type—coarse or fine—is another essential consideration when we design our architecture. In a distributed company, the best way to achieve fine-grained communication across teams is to have a fully remote organization so that there isn't any difference between teams. However, the moment an organization has multiple developer centers in multiple locations, the communication flow changes again, and having multiple teams working on the same area of the codebase in different offices may be more of an issue than a benefit. A good way to mitigate this problem is by assigning all the subdomains that intersect and share similarities to a co-located team instead of

distributing them. For example, imagine a video-streaming platform composed of the following areas:

- Landing page
- Movie catalog
- Playback
- Search
- Personalization
- Sign-in
- Sign-up
- Payment
- Remember email
- Remember password
- Help
- My account

When we group subdomains that intersect and share similarities, we can group subdomains related to new-user onboarding:

- Landing page
- Sign-up
- Payment

Then, we can group subdomains related to existing users who may or may not already have been authenticated within our platform:

- Sign-in
- Remember email
- Remember password
- My account

Finally, we can group subdomains related to existing users who have been authenticated:

- Movie catalog
- Playback
- Search
- Personalization

What we've done is group subdomains by user journey—that is, subdomains that intersect. The “playback” experience, for instance, certainly has more in common with the “movie catalog” than with the “help” pages. Did you notice that the “help” domain wasn't in any of the previous groups? That's because the help section may be useful for authenticated and unauthenticated users alike.

It's very hard to have a perfect split between the user journeys. This is also true when we identify the different subdomains available in these user journeys. In this example, we ended up with several buckets of user journeys, with one or more subdomains within each bucket. However, the moment we can identify these subdomains, we can determine how to map the development of our micro-frontends within a company. This exercise may force us to swap some domains from one office to another. This can provide great long-term benefits, such as reducing external dependencies across offices and keeping them in the same one; and maintaining fine-grained communication where subdomains work together, and coarse-grained communication across offices where the need for synchronization happens less often and with fewer touchpoints. As stated before, however, it's very unlikely to have this perfect split, so don't be surprised if you end up with some subdomains developed by different offices. Just be sure to constantly review the performance and bottlenecks created inside the organization and adjust your decisions accordingly. When done right, microarchitectures are great because they can follow a business's evolution and—thanks to their modular nature—provide the tech department a great degree of flexibility.

Features Versus Components Teams

Nowadays, many companies are debating which team structure they should use to enable developers to work on their tasks without impediments or external dependencies. Usually, agile methodologies suggest one of two structures: feature teams and component teams. Feature teams, also known as cross-functional teams, are

organized with all the skills needed for delivering a specific feature. When we are developing a web application, for instance, a cross-functional team is organized to deliver user value around a specific feature. Let's imagine that we have a cross-functional team that will create the credit card payment feature inside an ecommerce store. The team will have both frontend and backend (a.k.a. full stack) developers who will develop, test, and deploy the feature end to end. **Figure 13-1** depicts this example with Team Burritos, which is responsible for delivering the “product details” micro-frontend.

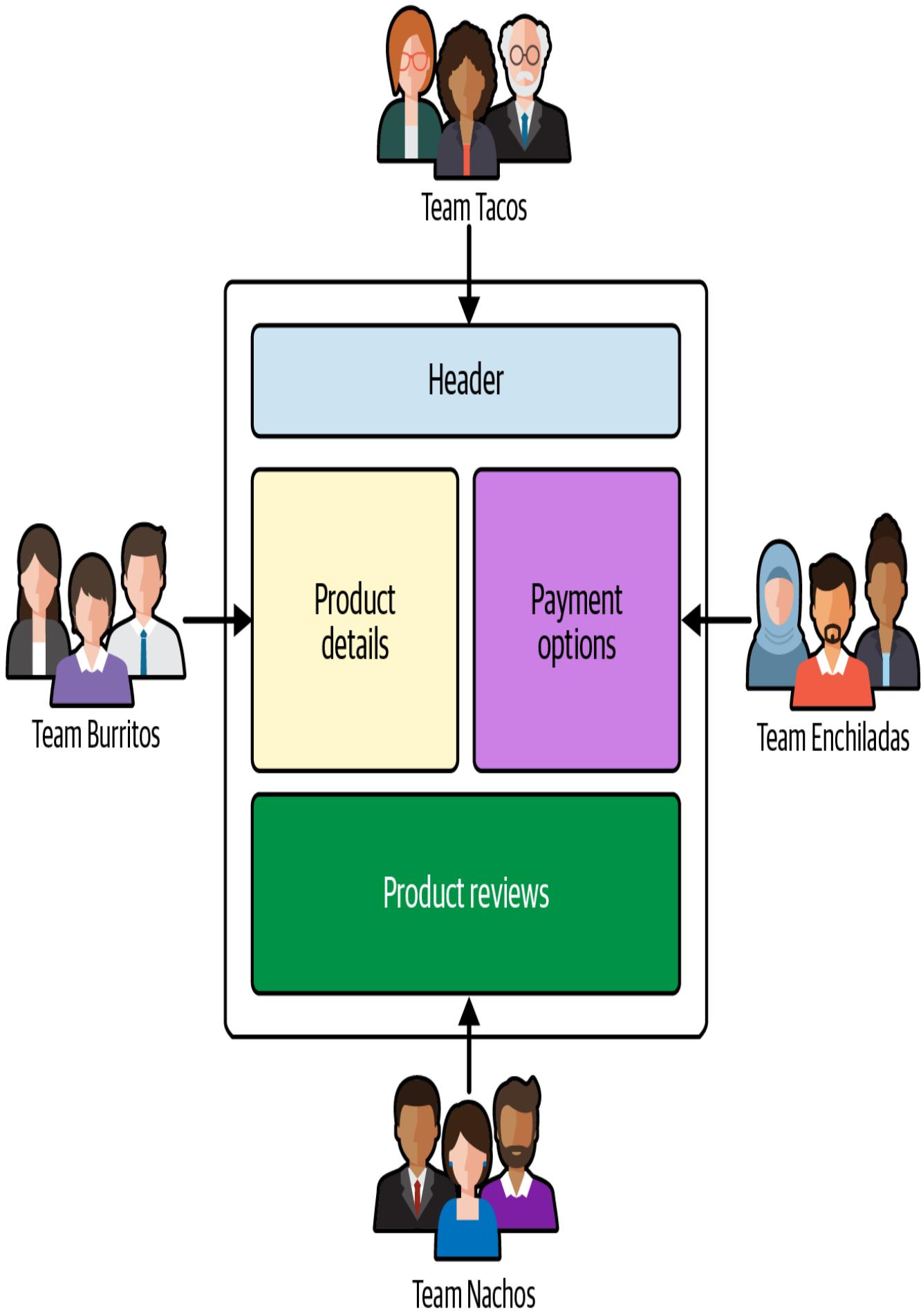


Figure 13-1. The responsibilities of the feature teams working on a horizontal-split architecture, where every team has end-to-end responsibility for a micro-frontend and the APIs that it consumes

In this case, Team Burritos will be composed of full-stack developers working on the APIs, as well as the frontend that will consume these APIs.

Features teams are recommended when you can organize the architecture using a horizontal split, and every team is responsible for one or more micro-frontends. With this approach, the teams can focus solely on their features with an end-to-end approach, taking care of the entire feature life cycle. The cognitive load of a feature team is more manageable than any other team's structure, because every person responsible for generating value for the user is part of the same team. Usually, feature teams are highly focused on the user, iterating constantly to enhance the user experience and the value created by their development effort. This approach allows feature teams to become domain experts in specific parts of the system, contributing not only to the technical aspects but also to product decisions. Providing valuable trade-offs accelerates the deployment of micro-frontends and reduces complexity upfront.

One challenge with this approach is that we will need to assign page composition to an external team or, more likely, to one of the teams developing a feature. The team responsible for composing the page must ensure the final result for the user is the one expected, without any logical or cosmetic bugs. We can mitigate this challenge by standardizing the page composition with templates and conventions. In this case, it would be easier to manage, but would offer a lot less flexibility across page layouts. Another option would be to work with component teams, where every team is responsible for a specific component of a platform (a vertical split), as we can see in [Figure 13-2](#).

With a vertical-split architecture, Teams Burritos and Enchiladas are responsible for the backend services, and Teams Nachos and Tacos are responsible for the frontend. This way of organizing teams works better when we are dealing with cross-platform applications in mid- to large-sized organizations, where we usually develop at least a web application in conjunction with a native mobile application, each for Android and iOS. In this instance, when the APIs are consumed by several client applications (mobile, web, and possibly other devices), a backend team responsible for creating an API takes into consideration all the needs of the consumer (frontend teams) instead of feature teams optimizing the APIs for just one platform and treating the other clients' teams as second-class citizens. When we are working with cross-platform applications and hybrid technologies like Flutter, Ionic, or React Native, cross-platform teams are a more viable option than component teams. The codebase between frontend projects may

be shared across targets, so organizing teams around features becomes the better choice. If you want to pursue the native option, however, think twice about how to organize your teams—switching from one native language to another and finally to web and backend languages is challenging and increases the cognitive load.

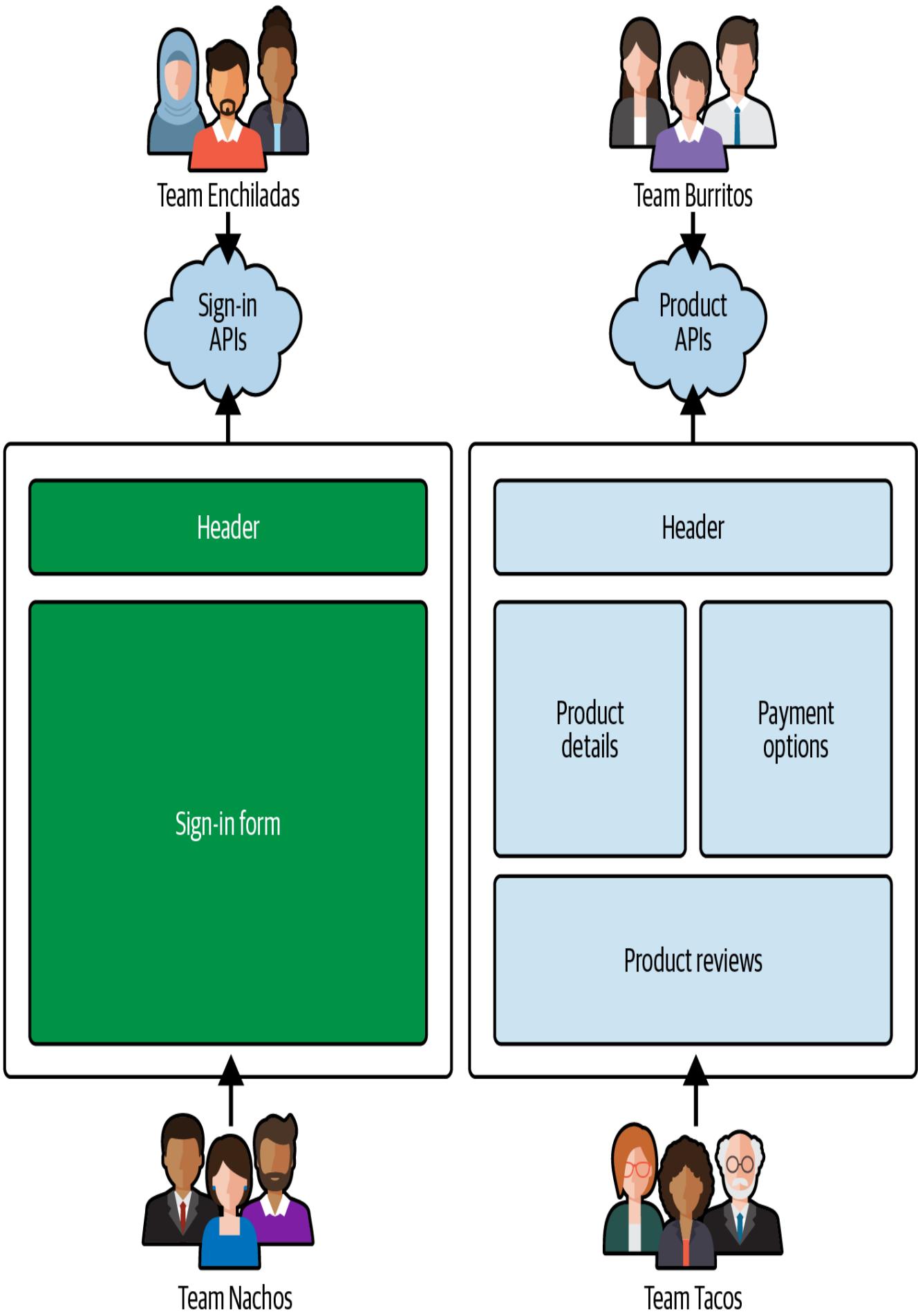


Figure 13-2. In this example, component teams are each responsible for a specific part of a platform

It's important to recognize that different stages of the business life cycle require different team structures. In the case of growth, you will encounter a moment where the organization requires some structural changes. These changes will lead you to reassess the teams around the architecture in order to reduce the friction for delivering a feature or any other stream of work. Invest time in analyzing the communication flows and potential patterns established across teams, like constant external dependencies or slow story throughput due to distributed teams in multiple time zones.

Domain-driven design (DDD) is helpful when we are organizing our team structure, because it helps to consider the direct connection between architecture and team structure. Not only does DDD help identify the boundaries between subdomains, but we can follow these boundaries for structuring our teams as well. For instance, with a small company, it's very likely that a team would be responsible for a specific business subdomain end to end; however, within a larger organization, a multitude of teams create a subdomain, working together due to the work's inherent complexity and scope. It's not always possible to create the perfect structure, and often we need to make some trade-offs to create a model that almost fits everywhere. This is not necessarily a roadblock for your strategy, but do understand that trade-offs could happen. However, when trade-offs become a constant across the entire organization, we need to step back and review our team structure and architecture.

The structure of our teams—whether organized around features or components—and how we structure our micro-frontend applications will impact the communication flow within the organization. There are certain practices that may help us achieve an efficient spread of information across teams, enhancing the governance for developing new features and capabilities inside our platform. Let's analyze some of them in the next section.

Implementing Governance for Easing the Communication Flows

Working for mid- to large-sized organizations means defining communication flows that work; otherwise, we risk slowing down development or creating too many external dependencies across teams. An investment worth making is governance. This is not just an upfront investment; it has to be a constant review and optimization of the practices and documents needed for scaling an organization. There are some simple wins that help our developers scale their communication, especially in the future.

We should remember that there will always be new employees at our company; the best way for them to fill their knowledge gaps would be to understand the context in which certain decisions were made. Without that context, they won't have enough information to fully understand the situation. Architecturally speaking, there are two practices that can spread the information and track why certain decisions were or were not made: the requests for comments (RFCs) and the architecture decision record (ADR). Remember that asynchronous communication can greatly benefit introverts or those who don't perform well in meetings, allowing their voices and ideas to be heard. Don't underestimate the power of this approach—you might be surprised by the valuable insights from less vocal members of your organization.

Requests for Comments

RFCs are an established way to gauge the interest in a change in a technical approach or a new technology or practice within an organization. Usually, RFCs are kicked off by developers or tech leads who see some gaps or potential improvement opportunities in the organization and want to understand if there is room for a change. RFCs are often available in the version control system (GitHub, for example), so every technical person has access to them. RFCs are timeboxed and are a short markdown document composed of the following sections:¹

Feature name

The name of the feature or practice to introduce or change

Summary

One-paragraph explanation of the feature or practice

Motivation

Usually the *why* of making a change

Description of the change

Detailed analysis of the change or new feature

Drawbacks

All the potential issues identified by the proposal submitter

Alternatives

Potential alternatives to achieve the goals with pros and cons

Unresolved questions

Any blind spots in the proposal

Additional resources

A list of resources related to the RFC

After filing an RFC, the submitter shares the link with the interested parties in the organization. Then the dialogue starts to flow, with people sharing ideas, asking questions, and trying to understand whether the proposal has any potential for being introduced inside the company. Despite its simplicity, this document is important because it may improve current software development or practices. Moreover, the RFC has a fundamental benefit in that it tracks the discussion happening among all the developers, architects, and tech leads, recording the full discussion and approaches. This history will give new employees a clear context that describes the reasons behind certain decisions. RFCs are great for proposing not only new features but also changes. For instance, when we need to update an API contract, using an RFC enables us to gather our consumers' thoughts, ideas, and concerns, allowing us to shape the best way to achieve the goal. This scenario often happens when we work with component teams, and the backend team comprises multiple teams that consume their APIs.

With an RFC, the team that owns the API contract can propose changes and collect feedback from the other teams, gathering the evolution of the API and the reasons behind it. This practice becomes even more important when we work with distributed teams across multiple time zones, because we can share all the information needed without remote meetings, closing the feedback loop in a reasonable time.

Architecture Decision Records

Another useful document for sharing decision context for current and future developers is the ADR, in which architects or tech leads gather the decisions behind a specific architecture implementation. ADRs are focused solely on architecture, but they are still useful for providing future readers with a context and a snapshot in time of your organization. In fact, an ADR specifically describes the company context when the ADR is first written. ADRs also differ from RFCs in that they provide the context of why an architecture change is needed and explain why a specific decision was made.

Architecture is always about finding the balance between long-term and short-term wins. These trade-offs are defined in the company context we operate within.

With ADR, we want to create a snapshot of the company context and provide a description of why we pick one direction over another. An ADR structure is composed of the following sections:

Status

The status of an ADR (e.g., draft, agreed)

Stakeholders

People behind the ADR, usually architects and tech leads

Outcome

The final decision made

Due date

The date for when the decision has to be made

Owners

Document's owners

Introduction

A paragraph describing the company context and the problem the ADR is trying to solve

Forces

The parallel or overlapping streams of work that are pushing toward an architecture change

Options

List of potential solutions with business and technical details, and the pros and cons for every proposal

Final decision and rationale

A summary of the final decision explaining the reasons behind choosing a proposal listed in the options paragraph

Appendix

Additional resources needed for providing more context to the readers

As with RFCs, not all these parts are mandatory, but they are highly recommended. Remember, ADRs have to provide the context for everyone interested in why an architectural decision was made, so the reader needs to have clarity on the technical and business context when the ADR was created. When we design our micro-frontends, we may change the framework or design patterns implemented inside the architecture. By using ADRs, we can provide the context that existed before the architecture decision and why we now want to change it. This way, everyone will be on the same page, despite not being physically present in the meetings.

Techniques for Enhancing the Communication Flow

When first approaching micro-frontends, many people think this architectural pattern may result in organizational silos due to its intrinsic characteristics, such as independence and decentralization. Although micro-frontends enable teams to work in parallel and release artifacts independently, that's no excuse to neglect collaboration. In fact, decentralization increases the need for intentional cross-team rituals—such as architecture syncs, shared RFCs, and design reviews—to avoid hidden silos and ensure coherence across the system. We cannot embrace distributed systems without establishing mechanisms for teams to come together regularly for sharing knowledge, solutions, and challenges. It's essential to curate the technical as well as the social aspects for guaranteeing the right flow of information inside an organization. Everything should start from the feature's specifications.

Working Backward

Famous for its customer-centric approach, Amazon often works backward when considering product ideas. Simply put, it starts with the customer and works backward to the product rather than starting with a product idea and bolting customers onto it. It's a method for creating a customer-focused vision of your product. A working-backward document, called a PR/FAQ, is up to six pages long: a one-page press release (PR) and up to five pages of frequently asked questions (FAQs). An appendix section is also

included. While working backward can be applied to any specific product decision, using this approach is especially important when developing new products or features.

Because it starts with where you want to be 12 months in the future, the working-backward method forces you to think big, focusing on big goals and the changes you need to achieve. A well-crafted PR is a great use of storytelling. It gets the team excited and focused before any lines of code are written.

The FAQ section is composed of two subsections. The first one is based on the public questions that a customer might have about the product or feature, written as if it were a public product documentation released at the same time as the PR. The second subsection consists of questions that internal stakeholders might have asked during the product development process.

The PR/FAQs focus effort on how a specific feature benefits the customer and why the company should invest in a product or feature like that in its system. After a PR/FAQs is written, a meeting is scheduled with the main stakeholders, including developers, QAs, tech leads, architects, and other product people. In general, though, any stakeholder who may help improve the decision process is invited. This may seem excessive, but one hour of socializing requirements can allow techies to raise questions and become familiar with the feature. It's a first step for having multiple teams understand the initial requirements of a new feature and aligning it with the business goals to be reached. When we work with micro-frontends, PR/FAQs can bridge the teams that will collaborate in the implementation phase.

Two extremely valuable benefits of the PR/FAQs process are the resulting concise documentation and the initial collaboration phase before the implementation phase. Usually, a PR/FAQs document is a good starting point for architects to think about the high-level design, including the challenges and the architecture characteristics needed for implementing a feature.

This is also true for micro-frontends when a new requirement arises and it has to be implemented across multiple domains. Having this kind of document can facilitate the discussion between teams via the requirements socialization between engineers and product teams.

If you are interested in knowing more, I recommend reading Chris Vander Mey's *Shipping Greatness*, a book that provides more information on how to write PR/FAQs, following Amazon learnings and suggestions. If you prefer a short document about PR/FAQs instead, check out "["PR FAQs for Product Documents"](#)", a blog post by Robert (Munro) Monarch that offers a great summary of this topic.

Community of Practice and Town Halls

The community of practice and town halls are two more important practices for facilitating the communication flows across the organization. With both cross-functional and component teams, there is a need to spread knowledge among developers of the same discipline (frontend developers in the micro-frontend world). Usually, communities of practice are biweekly or monthly meetings scheduled by engineer managers or tech leads to facilitate discussions across team members responsible for the same discipline. In these meetings, the developers share best practices, how they have solved specific problems, new findings, or topics they've recently been exposed to within their domain. Communities of practice are useful for introducing new practices across the organization, discussing automation pipeline improvements, or even hosting **mob programming events**, which see engineers collaboratively implementing new features or discussing specific programming approaches all together. While usually restricted to a team, I've experienced some mob programming sessions during a community of practice that have worked well.

MOB PROGRAMMING

Mob programming is a software development approach in which a whole team works on the same project as a group, working on the same computer at the same time. This is similar to pair programming, in which two people work on the same code together at one computer. Mob programming just extends the collaboration to everyone on the team. This technique is typically used when a team is implementing an important but complex feature or during a community of practice where a vertical within an organization wants to introduce new practices across the tech department.

Town halls are events organized across the tech department that provide a general knowledge of what's happening across teams, such as a team's recent achievements or new practices to introduce inside the organization. Town halls work especially well when an organization works with distributed teams, and developers cannot engage with all the teams on a daily basis. During these events, the tech leadership facilitates the knowledge to be shared through short presentations covering the key initiatives brought up by different teams. Considering the large audience attending these events, any questions or deep dives should be taken up at a separate time by the interested people and the team or person involved in a given initiative.

Town halls are very useful when a team would like to share a new library that they have developed that may be used by different teams, or new practices introduced by a team

and the results after embracing them. They are also useful for more general topics like new joiners or shared goals across the department.

Depending on the company's size, town halls may not be the right choice. A good alternative for spreading these initiatives and communications could be an internal newsletter for the tech department. We may decide to split the newsletter by topics and allow developers to pick their favorite, but this will depend on the organization's size and structure.

Managing External Dependencies

Sometimes during a **sprint**, external dependencies may impede the delivery of a task or story. While this is not usually a problem, when distributed teams work on microarchitectures, it can slow down feature delivery. This can create frustration and friction across teams. When a team is hampered by too many external dependencies, it may be time to revisit our decisions and review the boundaries of a micro-frontend. Having frequently occurring external dependencies is one of the strongest signs that something is not working as expected. But fear not: it's a fixable problem! As long as the information bubbles up from the teams to the tech leadership, leadership may decide to rearrange the organization, reducing communication friction and improving the throughput.

With micro-frontends, this situation can occur when we share libraries across micro-frontends or when we compose multiple micro-frontends in the same view, if we don't pay enough attention to how to decouple them.

NOTE

"Reusability represents a form of coupling!", as Neal Ford says, director of architecture at Thoughtworks.

With a horizontal-split architecture, we need to invest time reviewing the communication flows, especially at the beginning of the project. A classic example is when we are porting a frontend project from a monolith SPA or from a server-side-rendering one to micro-frontends. When we embrace the horizontal-split architecture, we need to assign ownership of the view composition process. In fact, despite having multiple teams contributing with their micro-frontends to the final result, we need to identify the owner of the composition stage (either client or server side). This team should be responsible for not only composing the view but also understanding potential

scenarios where a micro-frontend could cause other micro-frontends problems due to CSS style issues or events dispatched but not properly handled by other micro-frontends.

It's true that this architecture style provides more flexibility in reusing micro-frontends across a project, but at the same time, we need to make sure the final result is what the user expects to have. Another challenge in the communication flow for the horizontal-split architecture may happen when a micro-frontend is reused in different views of one or more applications. In this case, the team responsible for the micro-frontend should create strong relationships with all the other teams that may asynchronously interact with the micro-frontend, and have regular catch-ups to make sure the touchpoints within a view are respected.

The problem of too many external dependencies may happen in very limited cases with a vertical-split architecture, especially if we are using an application shell that orchestrates the micro-frontend life cycle. For example, let's say we want to add a new route inside our application. The application shell will need to be aware that a new micro-frontend needs to be loaded and that it will have to manage the new route. When designed well, the application shell loads an external configuration retrieved from a static file served by a content delivery network (CDN) or as a response of an endpoint. In this case, the effort to coordinate with the team that owns the application shell would be minimal because all that's required is changing the configuration, adding new automation tests, and following the testing life cycle implemented inside the organization.

Another potential activity slowdown can occur when you need to make changes across multiple micro-frontends. This usually happens once or twice a year; if it occurs more often, that's another sign we should review the division of our micro-frontends. However, with a vertical split, teams have more autonomy. If we are able to review the communication flows iteratively, we shouldn't have many surprises or external issues.

For all these situations, reviewing the communication flows with the right cadence (every quarter, for instance), and making sure the assumptions made during the architecture design process are still valid, are good practices. Remember that friction between teams should be seen as a signal of a problem within the organization, not necessarily as an architectural issue. Often, the root cause lies elsewhere, such as in organizational structure, micro-frontend boundaries, or a lack of practices for encouraging collaboration within the developer community.

Another way to ensure the communication flows across teams is by having ad hoc meetings for the teams that have to work together for the final-view result, especially for a horizontal-split architecture. Using agile ceremonies like Scrum of Scrums or less informal catch-ups on a regular cadence can result in a better understanding of the overall system, as well as better bonding between team members.

Finally, agile practices provide some tools that may be used either ad hoc or on a case-by-case basis to solve specific challenges we face with our teams. One tool that I have personally experienced in multiple companies I've worked for is big room planning, where we gather an entire department for one or two days inside a room, and we map the activities for the next few months. In this way, we can immediately spot external dependencies and potential bottlenecks due to an incorrect sequence of deliverables. There are many techniques available for solving specific challenges. I recommend first gaining an understanding of the problem you need to solve and then finding the right approach for that problem.

A Decentralized Organization

A key advantage of working with micro-frontends specifically, and with microarchitectures in general, is the possibility of empowering the teams to own a business domain end to end. As we have seen throughout this book, micro-frontends are not for all organizations. They work well for mid- to large-sized ones, where insight is needed into the intrinsic complexity that the company is working on. Complexity is not necessarily a negative attribute; it can allow us to move from a centralized approach to a decentralized one.

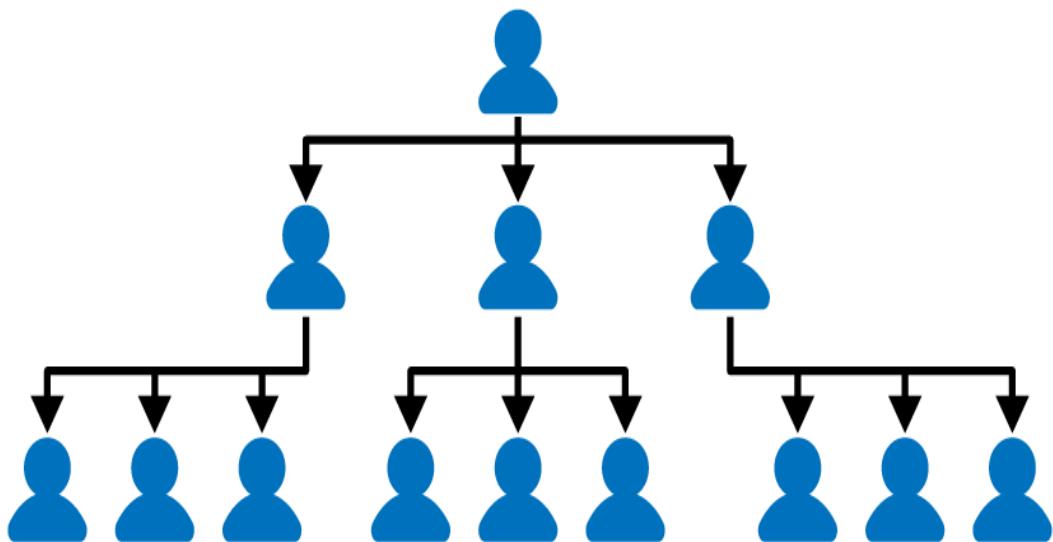
Every company moves through different phases. In the startup phase, a company usually has a small tech team capable of working on a project end to end. The communication flow in the startup phase is straightforward because all the developers are aware of the goals to achieve, and the number of connections needed for a correct communication flow is manageable. When the startup grows larger, it's usually structured around business function hierarchies, introducing roles like head of engineering, engineer manager, tech leader, and many other well-known titles in the tech ecosystem. Usually, these organizational layers provide directions to coordinate teams, defining the communication flows. In an ideal scenario, the teams may have the level of autonomy needed to do their jobs and to experiment with new practices and technologies at the same time, but this doesn't always happen. The reality will depend on the company culture, as well as the leadership style of every individual.

When the organization moves from hundreds to thousands of employees, we need to look again at how to organize our teams. A natural evolution from the hierarchical structure is to align teams around value streams instead of following a centralized hierarchy. Decentralizing decision making and allowing an independent path for teams within a value stream empowers technical teams to express themselves in the best way within the large and complex context where they operate. These three types of structure are visually represented in [Figure 13-3](#).

Startup structure



Hierarchical structure



Decentralized structure

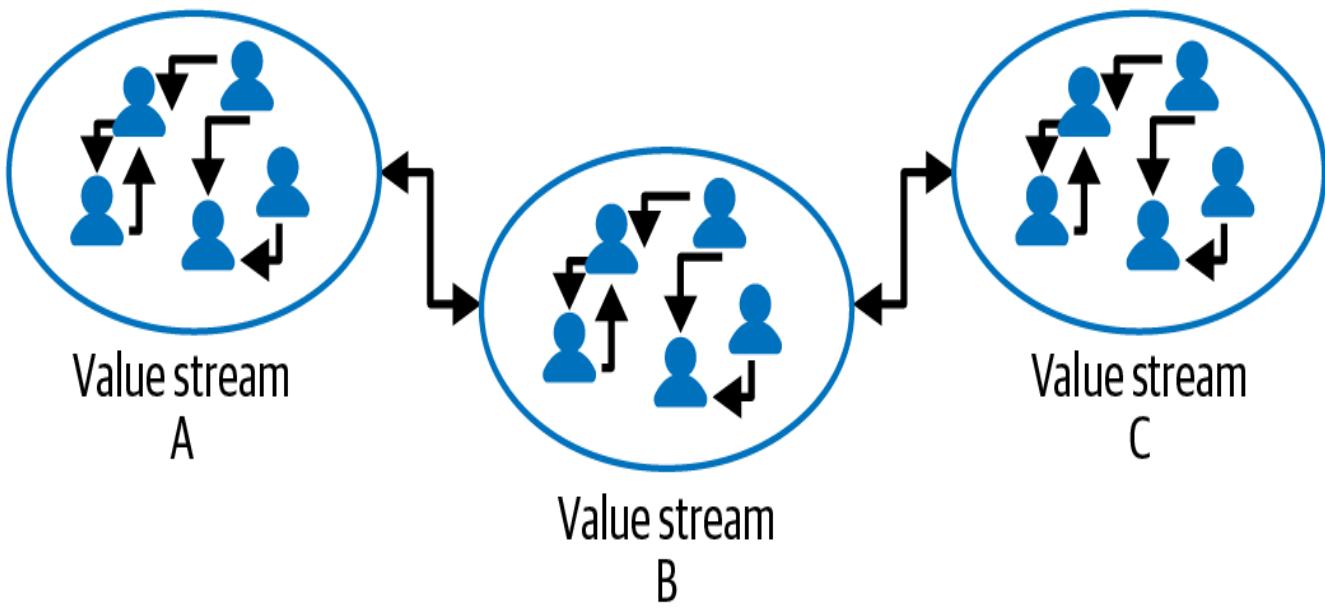


Figure 13-3. Different organizational structures, usually implemented in different stages of a company's life cycle

An interesting point highlighted in [Figure 13-3](#) is that, with the decentralized structure, teams must coordinate among themselves when needed. Empowering these teams means providing not only technical freedom but also organizational responsibilities. Technical leaders then become a support function that facilitates the streams of work within the teams, providing context or technical direction when requested, and driving alignment of the technical boundaries with business results so that teams understand how to achieve these goals.

Another great achievement of decentralization is error mitigation. In this structure, it's unlikely that only one or a few people are capable of making every decision for every team, especially because the leadership is not always involved in the day-to-day conversations. Therefore, the role of an architect or a tech lead would include posing the right questions and acting as a supportive leader for the team, creating solutions hand in hand with the team rather than in isolation.

ORGANIZE FOR COMPLEXITY

Many of the concepts of decentralization that I've described are part of a great book called *Organize for Complexity* by Niels Pflaeging. This short, straightforward book provides many insights into how to decentralize an organization that is handling complexity. I was lucky enough to meet Niels during an agile retreat and received a complimentary copy of his book. It changed the way I thought about tech organizations, opening several doors in my mind. The book doesn't focus on tech organizations but more generally on any organization. That's the reason I shared these insights, contextualizing the core concepts in the tech context. I found these concepts extremely valid for microarchitectures.

Since 2019, the DDD community has increasingly emphasized the sociotechnical aspect of software architecture, drawing numerous insights from the remarkable book *Team Topologies* by Matthew Skelton and Manuel Pais. This book is eye-opening and a must-read for any tech leader seeking to organize their organization for fast flow and distributed systems.

Decentralization Implications with Micro-Frontends

The first step in decentralizing decision making and empowering the teams that are closer to the business domain is to identify the subdomains in an application. As we have seen so far, DDD helps us identify the business subdomains where we create a common language (ubiquitous language), introduce patterns for communicating across subdomains to decouple them, and allow them to evolve at their own pace. Another important aspect to consider is user behavior, especially when porting an existing application to micro-frontends. These two factors help us identify the different pieces of a complex puzzle and assign each piece to a specific team.

I highly recommend basing the micro-frontends split on data, because this can really save you a lot of aggravation in the long run. Starting with incorrect assumptions creates friction across teams and release cycles. Data can help prevent those incorrect assumptions.

Another important thing to consider is balancing complexity when we assign a team to a subdomain. When a team is assigned to multiple complex subdomains, there is a high risk of resource burnout and intrinsic maintenance complexity. There are several situations we need to be aware of, which are outlined in the following subsections.

High-Complexity Subdomain

This type of subdomain usually doesn't manifest at the beginning of the project. They're created when we underestimate a subdomain's complexity in terms of the business logic and permutations that a micro-frontend needs. Usually, a micro-frontend becomes complex over time because new features are added to it. A good practice, therefore, is to understand the cognitive load of every team member working on that subdomain and determine whether they can handle the situation properly. Are there enough team members equipped to handle the demands of the subdomain? Do they possess the requisite knowledge to implement all requested features effectively? Should we further decompose or consolidate micro-frontends? Are there any sources of friction that need addressing? These are excellent starting questions.

The main struggle is often in maintenance and support, especially when the team deals with projects that have to be live 24/7 and where bugs have to be fixed as quickly as possible. High complexity is difficult to handle in general, and it's even more difficult when a developer is under pressure because a live bug is found in the middle of the night and requires a quick fix.

In these cases, remember to review the boundaries of your subdomain and see if it can be split in a more sensible manner, especially when you work with a vertical-split

architecture. Consider, for instance, when you have an authentication micro-frontend containing the sign-in and sign-up flows in a vertical-split architecture. If you are working on a global platform, you may have to support multiple payment methods, which adds a lot of information to remember and own. Splitting the authentication micro-frontends into sign-in and sign-up micro-frontends would maintain a frictionless user experience while reducing the cognitive load on the team, as described in Figure 13-4.

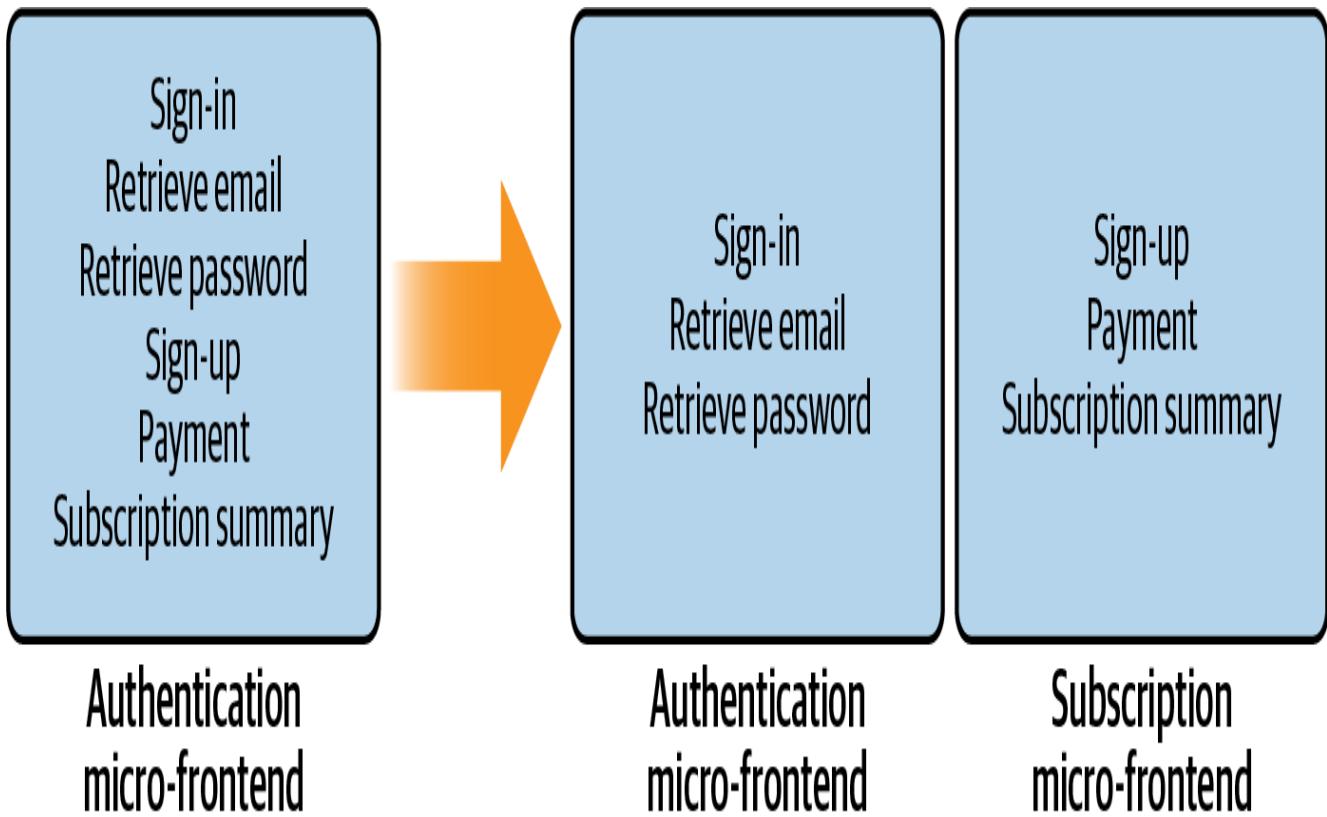


Figure 13-4. Splitting the micro-frontend to reduce the team's cognitive load without impacting the user's experience

Now we have a team dedicated to new users who want to sign up and another dedicated to existing users who have to sign in or retrieve their email or password. In this way, we simplify the logic and code and can have quicker fixes when bugs are discovered.

High Initial-Effort Subdomain

Sometimes we may have micro-frontends that require a high effort to create, but then they don't evolve very often in the application life cycle. In this case, we can afford to have a team with multiple micro-frontends, bearing in mind we need to balance the micro-frontends' complexity to avoid drowning the team in work.

Normal-Complexity Subdomain

When considering a single team, these are the subdomains we should aim for.

Sometimes, when we have high-complexity subdomains, we may decide that splitting the micro-frontend representing that subdomain would not help much. However, we can componentize a specific complex part of the micro-frontend and assign the component to another team, as we can see in [Figure 13-5](#).

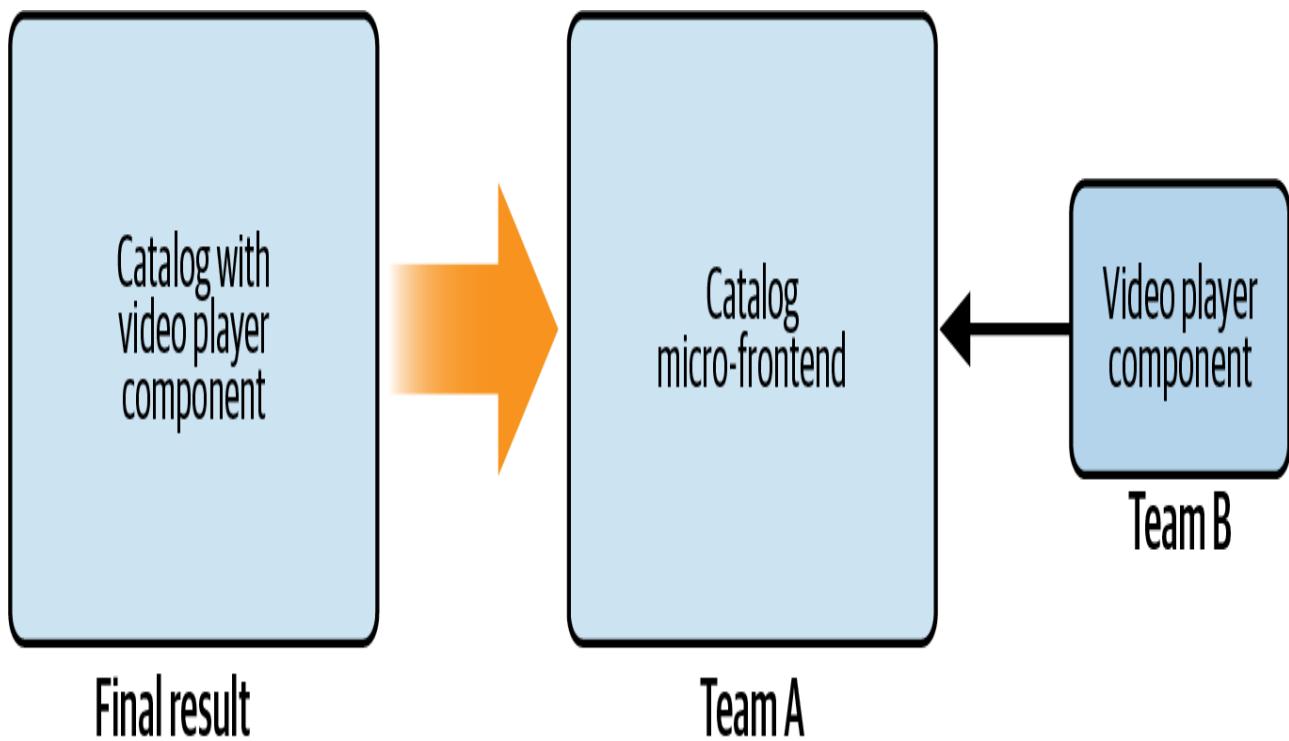


Figure 13-5. A project with high complexity but with the possibility of extracting a component to another team to spread the cognitive load

In this case, we have a vertical-split architecture with a micro-frontend with multiple views and a complex business logic to implement for the video player that should play video, advertising, and so on. The video player can be peeled off from the micro-frontend. This self-contained component may be reused in other micro-frontends and has an intrinsic complexity, making it a good choice to be handled by a different team, reducing the cognitive load of both the micro-frontend and the video player. The teams can collaborate when new versions of the video players are available. In this case, working with an API contract and scheduling regular touchpoints between teams is sufficient to coordinate the integration, new releases, and breaking changes.

Low-Complexity Subdomain

Finally, some micro-frontends are easy to build and maintain, so one team can own a multitude of them. As with the previous cases, make sure to regularly rebalance the complexity assigned to these teams, because while the complexity may be low in this case, having dozens of low-complexity micro-frontends may lead to high context-switching, reducing the team's productivity.

Decentralizing decision making and empowering teams doesn't mean we need to create chaos within the organization. In fact, some decisions should remain centralized and be made by teams that span all of an organization's domains, such as a platform or developer experience team, or by tech leadership (like the head or vice president of engineering) providing a framework for teams to operate within. We mentioned such decisions in previous chapters, including the platform for running the automation strategy; programming languages or frameworks available to teams; guidelines on when to abstract and when to duplicate code; architecture characteristics, such as performance metrics, code coverage, and complexity; setting up observability for the entire platform (frontend and backend); and supporting governance when the application fails in production. All these decisions provide a concrete framework for the teams to operate within. They don't affect the teams' freedom, and they align the company behind guidelines that should help your technical teams achieve business goals.

Summary

In this chapter, we learned that we cannot design a software architecture without taking into consideration the human factor. Architecture, company culture, and organizational structure are interdependent factors that are crucial for the success of any project. We need to be aware that these two forces are part of a project's success, and they cannot be decoupled. They must be looked at together and revised often. Any business can evolve over time, and the same is true for software architecture and communication flows inside a company. The communication flow can be enhanced by spreading information across the teams, but it has to be thought through and designed carefully, and we need to iterate to find the right balance. What works in one company will not necessarily work in others, so carefully analyze your context and apply the best practices for your organization.

Finally, we looked at how decentralization helps the implementation of any microarchitecture—whether microservices or micro-frontends. It's important to highlight that the micro-frontend architecture we chose should influence the way we

structure our teams. It's very unlikely that when we move from a monolith architecture to a microarchitecture, the organization can remain the same. The communication flow changes with the new architecture, and we need to at least review the flow so that we don't create bottlenecks inside the teams due to the wrong setup.

¹ This list is just a suggestion. Not all of the items may be present in your RFC template, but it's a good starting point.

Chapter 14. AI and Micro-Frontends: Augmenting, Not Replacing

We are at a moment in software development where AI feels both magical and overwhelming. The pace of innovation is staggering, with new tools appearing weekly, promising to generate code, design interfaces, migrate legacy systems, and even reason about our architecture. And yet, if you have built micro-frontends—or any serious software system—you know something fundamental: *AI will not replace developers anytime soon, but it will improve the outcomes when used properly.*

That is not a motivational statement. It is a practical one.

Yes, AI can assist. It can generate boilerplate, suggest optimizations, or automate repetitive tasks. But the hard work of architecture—especially splitting a monolith into independently deployable frontends—still requires deep understanding, context, and intentional design. AI can suggest *how* to code, but it still struggles to decide *what* should be coded and *why*.

This chapter is not a manifesto against AI—quite the opposite. I will show you how to integrate AI into your workflow today to test, prototype, document, refactor, and optimize your micro-frontends. But here is the spoiler up front:

AI is powerful, but not omniscient. Developers who understand the foundations—design principles, boundaries, system behavior—will remain essential. Especially when things break. And they will.

So, how do we harness AI without outsourcing our judgment? How do we integrate it into a micro-frontend architecture responsibly, without losing control or introducing chaos? And what does this mean for our role as developers, tech leads, and architects?

That is what this chapter is about.

As a disclaimer: I won't be sharing specific prompts or tool recommendations, as those change too quickly. Instead, to keep this chapter valuable for as long as possible, I'll focus on real activities you can perform with AI—the kind that stay relevant no matter how fast the tools evolve.

Architecture Is Not Yet Automatable

As of today, AI is not a tool you can rely on to design your systems; frontend or backend, it makes no difference. Architecture is not a code generation problem, and that is where current tools fall short.

The core issue is that architecture cannot be designed in a vacuum. Throughout this book, we have taken a holistic view of software systems, and that lens is even more important when thinking about splitting applications into micro-frontends. It is not just about code; it is about people, teams, and long-term decision making—basically, *your context*.

You cannot detect architectural boundaries purely by analyzing source code. There are many invisible forces at play—your organization’s structure, the skills of your teams, communication patterns, deployment constraints, and more. These sociotechnical elements shape the architecture just as much as the codebase does.

Even within your own organization, much of this context is fragmented or undocumented. The knowledge needed to make informed architectural decisions often does not live in Jira tickets or internal wikis. And even if it once did, let’s be honest—most documentation becomes outdated within a few months of the first release. What remains accurate tends to be partial, overly simplified, or locked behind institutional knowledge.

Many critical details live in people’s heads—things like how the system actually behaves in production, which features are fragile, who to talk to when something breaks, or why certain trade-offs were made. Information is scattered across tools, Slack threads, dashboards, and hallway conversations. Reconstructing this puzzle is hard for human beings—and without access to this nuanced and often implicit context, AI tools will default to generic “best practices.”

But best practices are not always *your* best solution.

That is why architectural decisions still require human judgment. Not because AI cannot reason, but because it lacks access to the messy, imperfect, and highly specific reality of your system.

From a technical perspective, architectural boundaries are also shaped by how the system evolves over time. One key factor is code volatility, which refers to the rate and frequency of change in different areas of the codebase. Volatile areas often indicate parts of the system where requirements shift frequently, experiments are ongoing, or

business logic is still maturing. In these cases, isolating such code into its own micro-frontend can help prevent constant changes from impacting unrelated features and teams.

But volatility alone is not enough. Another useful lens is the distinction between integrators and disintegrators. Integrators are components or services that bring together multiple capabilities. They often act as orchestration layers, UI composition hosts, or coordination points across domains. While they are necessary, placing too much logic in integrators can lead to bottlenecks and increased coupling.

Disintegrators, by contrast, help reveal natural seams in a system. They indicate where responsibilities start to diverge—whether in logic, data, or team ownership. These are typically good candidates for separation because they reduce cognitive load, limit the blast radius of changes, and support independent delivery.

In the context of micro-frontends, recognizing these patterns becomes essential. For example, a product detail page in an ecommerce site may behave as an integrator—combining pricing, inventory, reviews, and recommendations. Meanwhile, the customer-reviews widget could function as a disintegrator, owned by a different team, evolving independently, and suited to live as a standalone micro-frontend. Splitting it out can improve autonomy and speed without fragmenting the overall experience.

The concepts of volatility, integrators, and disintegrators will not give you a definitive answer, but they equip you with better architectural questions. And that is something AI tools are not yet capable of doing today.

Let's consider a concrete example. Imagine you are working on the product listing page of an ecommerce website. Your company may choose to separate the product filter sidebar into its own micro-frontend because it changes frequently and is maintained by a separate team focused on search and merchandising. Another company, facing the same problem domain, might opt for a vertical split—combining product listing, filtering, and sorting into a single micro-frontend—to optimize for delivery speed and reduce cross-team overhead. Both choices are valid in their context, and neither can be derived purely from code inspection.

Beyond the technical and organizational context, we also need to factor in new feature pipelines, business requirements, and sometimes regulatory or cultural considerations. These elements all shape the way we decompose and evolve our frontend systems—and they do so in ways that AI currently cannot fully grasp.

Take new feature pipelines as an example. If your team is about to introduce a new subscription model with dynamic pricing and limited-time offers, you might decide to carve out a dedicated micro-frontend to support rapid experimentation and A/B testing,

isolated from the rest of the check-out experience. Another team working on the same domain, but with a more stable product catalog, might not feel the need for that separation at all.

Performance considerations are another crucial driver of architectural decisions, including in the frontend. For example, a real-time dashboard showing live data updates might be architected as a dedicated micro-frontend to optimize WebSocket connections and rendering efficiency. Meanwhile, less time-sensitive sections like settings or user profiles could be separated to avoid impacting the performance of the critical real-time features. Designing your micro-frontends with these performance needs in mind will help you to deliver a smoother, more responsive user experience—but the entire application should not necessarily have the same pattern.

Business requirements also vary drastically across organizations. A marketplace that prioritizes vendor onboarding may need to expose admin tools for sellers directly within the user-facing product area, leading to architectural choices that favor modular, access-controlled micro-frontends. By contrast, a direct-to-consumer (DTC) brand might centralize everything to maintain visual consistency and a tightly curated UX, choosing to avoid splitting unless absolutely necessary.

Then there are regulatory factors, which can have a profound architectural impact. If your application handles personal data of European users, you might isolate everything related to user profiles, consent management, and cookie handling into a dedicated micro-frontend to simplify compliance with General Data Protection Regulation (GDPR).

And don't overlook cultural considerations, not only within your organization, but also in how users from different regions interact with your interfaces. In languages written from right to left—such as Arabic or Hebrew—your UI layout needs to be mirrored to feel natural, otherwise users may find the interface unintuitive or disorienting.

Cultural sensitivity in layout and symbolism is equally important. In some contexts, specific colors, icons, or visual cues may carry meanings that differ dramatically across regions. These differences might not be visible in code, but they deeply influence how an interface is perceived and used.

These are not edge cases. They are everyday realities. Architecture is always a reflection of context. And context is rarely visible in the code alone.

Now, let's assume for a moment that AI magically advances and becomes able to account for all these dimensions—business, tech, people, regulations—and returns a perfect architectural layout. Even then, the work is not done.

This is because you are the one responsible for running that system in production. You are the one who gets paged at 3:00 a.m. You are the one who has to trace bugs, interpret the logs, and deploy a fix with confidence. You are the one mentoring new team members, sharing domain knowledge, and keeping the system maintainable as the product evolves.

AI might not yet be ready to fully design your architecture, but it should be seen as a powerful companion to augment your capabilities. For example, when I'm exploring how to approach a specific architectural challenge, I often evaluate multiple patterns with AI, asking it to highlight the pros and cons in context and even suggest alternative options I might have missed. The real value comes from first developing a deep understanding of your system's unique needs, constraints, and context. Once you've built that foundation, AI can help accelerate decision making, improve code quality, and unlock new levels of productivity. Embracing this journey allows you to leverage AI not as a replacement, but as a tool that enhances your skills and empowers you to build better micro-frontends.

Context Engineering to the Rescue

If architecture isn't automatable, then what is possible today? The answer lies in context engineering, a practice designed to work within the limitations of today's AI systems. Language models are capable of reasoning, suggesting alternatives, and accelerating workflows, but they operate within a limited token window, and they lack persistent memory of your system's architecture, constraints, or goals. That's not because the models are unintelligent, but because they don't retain context between interactions unless you explicitly provide it. This is where context engineering comes into play. By supplying architectural knowledge like team boundaries, ownership rules, shared contracts, and domain constraints, you help AI assistants act more like informed collaborators and less like autocomplete engines. Rather than relying on repeated one-off prompts, you create a persistent context layer that guides the AI's reasoning. Modern tools like Cursor (as shown in [Figure 14-1](#)) or Amazon Kiro are beginning to support this idea natively, allowing developers to embed and reuse architectural context directly in the development environment.

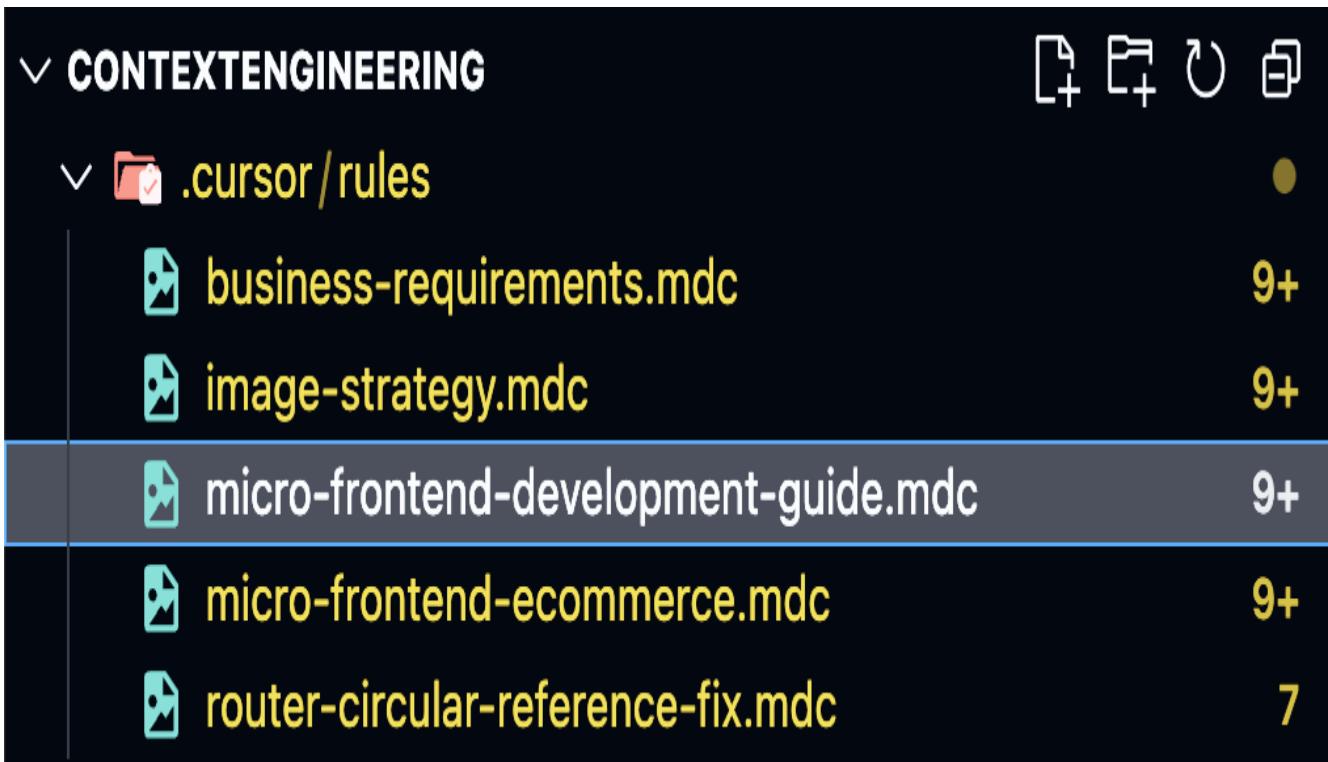


Figure 14-1. How to provide context in Cursor using Cursor Rules

For instance, in one of my projects, I created a markdown-based context file that describes the architecture of a micro-frontend ecommerce platform. This document defines each team's scope, their deployment pipelines, performance budgets, and the shared modules they interact with. It maps folder structure to ownership and outlines boundaries that should not be crossed. This simple, human-readable file acts as a persistent source of truth for both humans and AI assistants. An example of a possible structure is given in [Figure 14-2](#).

```
1  # Micro-Frontend E-commerce Architecture
2
3  > ## Project Structure \(Monorepo\)...
15
16  > ## Micro-Frontend Teams and Responsibilities...
82  | ⌘L to chat, ⌘K to generate
83  > ## Technology Stack...
96
97  > ## Module Federation Configuration...
146
147 > ## JSON-Driven Dynamic Orchestration...
406
407 > ## Benefits of JSON-Driven Dynamic Orchestration...
431
432 > ## Image Strategy...
460
461 > ## Error Handling and Loading States...
543
544 > ## Routing Architecture \(Hard Rule\)...
549
550 > ## State Management with Preact Signals...
584
585 > ## Bundle Size Enforcement...
607
608 > ## Testing Strategy...
653
654 > ## Architecture Validation with TS-Arch...
669
670 > ## Development Workflow...
677
678 > ## Best Practices \(Luca Mezzalira\)...
686
687 > ## Code Organization...
693
694 > ## Build and Deployment...
700
701 > ## Communication Patterns...
835
836 > ## Security Considerations...
846
```

Figure 14-2. Context engineering applied to micro-frontends, where all the architectural and design decisions are aligned per paragraph so that Cursor can consider them when implementing code for the application

Another practical example of context engineering comes from a reusable micro-frontend development guide that I maintain across projects. This markdown file serves as both a template and a rulebook. It describes the tech stack (e.g., Preact, WebPack 5, Module Federation 2.0), prescribes async boundary patterns, provides standard `webpack.config.js` and `package.json` templates, and enforces rules like “no shared state between micro-frontends” or “maximum bundle size: 300 KB.” It also includes configuration for routing, event bus messaging, and CI considerations.

The goal isn’t just documentation; it’s to externalize institutional knowledge in a form that AI assistants can use. When integrated with tools like Cursor or Amazon Kiro, this guide becomes a living boundary layer shaping AI-generated code, prompting the right abstractions, and avoiding common pitfalls. It’s a textbook case of context engineering as an operational discipline.

Effective context engineering requires more than just documenting your system—it demands clarity of intention. This means investing time upfront, at the beginning of a feature or project, to think through *what* you want to build and *why*. It’s about explicitly expressing architectural choices, team boundaries, performance budgets, and shared practices. But context isn’t static. As your system evolves, your context must evolve too. Keeping these documents up to date is critical—otherwise, you risk feeding stale or misleading information to your AI assistants. Tools like Cursor support scoped context, meaning you can define high-level architectural rules at the project root and override or extend them within specific micro-frontends or service folders. This is especially valuable in a monorepo, where you might want one context for a shared platform component and another for a backend for frontend (BFF) service owned by a different team.

Still, even with a well-curated context, AI is not infallible. It can hallucinate, generate incorrect or overly generic code, or miss subtle domain constraints. The goal isn’t to get perfect code on the first try; it’s to accelerate your thinking and reduce grunt work. You remain responsible for reviewing, guiding, and refactoring the output to align with your standards and team’s expectations. Think of context engineering not as automation, but as a collaborative setup.

Feasible AI Use Cases Today

While AI isn't yet ready to fully design or refactor your micro-frontend architecture, it already offers practical support across many development activities. From testing and automation to prototyping and debugging, AI tools can help you save time, reduce errors, and improve code quality. This section offers some realistic use cases where AI is proving valuable today. These are some of the use cases I personally use AI for and where I think it enhances the quality of my systems.

Testing

AI-powered testing tools can automatically generate test cases, identify edge cases you might miss, and even maintain test suites as your code evolves. For example, using AI to generate unit tests for your micro-frontend components can speed up coverage and reduce manual effort. Tools like `ts-arch` leverage AI to help architect and test TypeScript projects more effectively. Even if you don't know how to use a testing library, AI can drastically accelerate the way to test your code.

Automation

One of the most impactful uses of AI today is accelerating the setup of development workflows and platform capabilities. Tasks that used to take hours or even days—like configuring dependency updates with Dependabot, setting up Git hooks, or implementing fitness functions—can now be scaffolded in minutes using natural language prompts. Instead of digging through documentation, you can describe what you want, and AI tools will generate working configurations or code snippets instantly. This not only speeds up delivery but also helps teams implement useful safeguards and optimizations they may have previously skipped due to time constraints.

Debugging with AI

Modern browsers are beginning to integrate AI directly into their developer tools, and Chrome is leading the way. In recent releases, Chrome DevTools introduced an AI assistant that helps developers debug performance, inspect styles, and understand network behavior—all through natural language conversations. You can select specific performance events and ask follow-up questions in the same chat, making the debugging process feel more interactive and contextual. The assistant even retains your chat history locally, allowing you to revisit past sessions. While this doesn't eliminate the need to understand what's happening under the hood, it enhances your problem-solving capabilities by surfacing insights faster and reducing the cognitive load of scanning.

flame charts or digging through logs. For frontend developers working on complex systems like micro-frontends, this can translate into quicker diagnostics, fewer context switches, and more productive dev loops.

Diagram Generation

System diagrams have always been essential for communicating architecture, onboarding new team members, and aligning stakeholders. But creating them has often been a manual, time-consuming task—especially when the system evolves frequently. With AI, you can now generate architectural diagrams from natural language descriptions, markdown files, or even source code. Tools like Mermaid.js can be integrated into your workflow and enhanced by AI to produce flowcharts, sequence diagrams, or component maps with minimal effort. Imagine describing your micro-frontend setup in a few sentences and getting a diagram that maps out the integration points, team ownership boundaries, or deployment flows. It's not always perfect, but it's a starting point—and starting points matter. Even automating the update of diagrams in the README file becomes a breeze in this way.

Keeping the README Alive

A well-maintained README is often the first casualty of a fast-moving project. We update the code, add new features, fix bugs—but forget to reflect those changes in our documentation. The result? A growing disconnect between what the project *does* and what the README *says*.

AI can close that gap. Modern tools can analyze your codebase, extract relevant commands, configurations, and usage examples, and either generate or update your README accordingly. With a simple prompt like “Summarize how to run and build this project,” an AI assistant embedded in your IDE can extract scripts from your `package.json`, infer build steps, and format them in markdown.

What once took an hour of piecing together scattered notes now takes minutes.

Beyond the basics, you can also ask AI to generate onboarding instructions, environment setup guides, or explain architectural decisions—especially helpful for new contributors. And because AI understands `diffs`, it can also suggest README updates as part of your pull request workflow, catching changes that deserve documentation in real time.

Documentation becomes less of a burden and more of a living asset, integrated into your development flow. But remember that AI can hallucinate, so always review what it writes in your documentation.

Prototyping

The early stage of a project is critical for exploring ideas, reducing risk, and aligning on direction. It's also where AI can truly accelerate momentum. Whether you need to scaffold a new micro-frontend, simulate service interactions, or validate UI flows, AI assistants help teams move from idea to prototype in minutes—not days. This unlocks more bandwidth to test assumptions and iterate on value, rather than spending time on boilerplate.

What's most transformative is how AI lowers the barrier to exploration. Tools like V0 and Frontend AI by Webcrumbs now allow even non-developers to sketch functional interfaces or workflows using plain language. By the time a developer picks it up, the intent is clearer, more concrete, and often already validated.

Project Scaffolding

AI can become a new tool for scaffolding systems that reflect architectural intent. Imagine a team embracing a serverless-first approach across their backend landscape. For most use cases, assets are served from AWS S3 buckets, but a new performance-critical feature reveals unacceptable latency with that model. Instead of compromising the architecture or wasting hours on research, the team decides to mount a shared EFS volume directly to AWS Lambda to serve micro-frontend assets faster.

Now imagine describing this reasoning to an AI assistant: “We want a serverless-first setup, but instead of using AWS S3 to fetch micro-frontend modules, we want to load them from EFS for better performance. Please scaffold a lambda that mounts EFS, retrieves files from `/mnt/efs/dashboard`, and serves them via an API gateway.” With the right assistant, you’ll get a complete working setup tailored to your constraints, not just boilerplate from the open web.

Code Optimization

Performance and maintainability often hinge on the quality of the code you ship. AI can act as a pair of fresh eyes—spotting inefficiencies, suggesting faster alternatives, or even identifying dead code paths you might have missed. From frontend render loops to backend API calls, modern AI models can reason about complexity and recommend

optimizations that align with best practices. For example, it might propose lazy-loading strategies for noncritical components in a micro-frontend, or point out redundant re-renders in a React tree. It's not a replacement for performance profiling or benchmarks, but it adds an additional layer of insight—especially helpful during refactoring or when onboarding legacy code. When time is short and priorities are shifting, these nudges can mean the difference between shipping something that's “good enough” and shipping something great.

Bouncing Ideas and Discovering Alternatives with Agentic AI

One of the most promising applications of AI in modern development is its ability to help you explore design and implementation options—not just finish lines of code. As tools like Cursor, Amazon Q, or even ChatGPT become more deeply integrated into the IDE, they shift from simple autocomplete engines to collaborative agents that help you think through decisions.

Imagine you’re building a product-listing page and wondering whether to use infinite scroll, pagination, or SSR fallback. Instead of googling around or digging through documentation, you can prompt your AI assistant with the problem and constraints: “I need good SEO, smooth UX, and support for dynamic content—make a plan for that.” The assistant may suggest trade-offs, offer reasoning, and propose a few valid approaches, often referencing relevant tools or libraries. You can start by bouncing ideas and plan revisions off the assistant, using it as a thinking partner to accelerate exploration. But remember, while AI can generate drafts and surface options, human judgment should always drive the final implementation.

Accessibility

Let’s be honest: accessibility often ends up as a “nice to have” until the very last minute. This is not because we don’t care, but because deadlines, scope creep, and shifting priorities get in the way. This is precisely where AI can play a valuable supporting role.

Today’s AI assistants can already help identify basic accessibility issues in your code, such as missing ARIA labels, improper heading structures, or poor contrast ratios. But what’s more useful is how they can act as on-the-fly consultants—suggesting alternative implementations, offering semantic HTML fixes, or explaining why a certain pattern may exclude users with screen readers or keyboard-only navigation.

Even in the flow of coding, a quick “Is this component accessible?” can trigger a checklist and practical suggestions. And as accessibility tooling continues to evolve, you’ll likely see deeper integration in IDEs and design tools—surfacing issues as you go rather than after the fact.

AI won’t make your application inclusive on its own, but it lowers the barrier to doing the right thing earlier in the process. It turns best practices into a copilot, gently nudging you toward better, more inclusive outcomes.

Code Migration

One of the most impressive applications of AI in my experience has been code migration. I recently took on the challenge of porting a dashboard template from Angular to React—without prior knowledge of the original codebase. In the past, this kind of task would have taken me days, if not weeks, just to untangle the layers of templates, services, and component logic.

With the help of AI, I was able to complete the initial migration and even wire up some basic capabilities *in half a day*.

But let’s be clear: it wasn’t magic. AI didn’t automatically refactor the entire app. Instead, it became a thinking partner—breaking the problem into smaller steps, helping translate idioms between frameworks, and flagging areas that needed manual refactoring. This enabled me to move forward incrementally, controlling the architectural direction while avoiding the overwhelm of doing everything at once.

That’s the key: AI accelerates the porting process, but *you* remain in control of how the migration unfolds. It’s not about pressing a button; it’s about having a companion that reduces friction so you can focus on the strategic choices that matter. Remember, AI-assisted migrations still require thorough testing to validate correctness and ensure nothing breaks after the migration.

Discovering Alternatives and Challenging Assumptions with AI

Using AI as a sounding board to bounce ideas and explore alternative approaches can be very useful. For instance, imagine you’re building a complex micro-frontend feature, and you ask the AI whether your current communication pattern between micro-frontends is optimal. The AI might suggest alternative approaches, such as event-driven messaging instead of direct API calls, or highlight potential pitfalls like tight coupling

that you hadn't considered. This helps you uncover what you might be missing in your codebase and challenge assumptions. Such insights encourage you to rethink and refine your architecture early on, improving both quality and resilience. However, AI suggestions should always be balanced with your team's domain knowledge and roadmap priorities to ensure practical and aligned decisions.

My Playbook to Work with AI

As micro-frontend architectures mature and AI tooling evolves, the intersection of the two presents new opportunities but also introduces new complexities. This playbook captures a set of practical strategies for integrating AI assistants into the workflows of distributed frontend teams. Rather than focusing on tools or one-off prompts, it emphasizes how to reason, structure, and collaborate with AI in a way that respects autonomy, reinforces architectural boundaries, and accelerates delivery without compromising design integrity. Everything shared in these steps comes from months of prompting and recording results to understand what works and what doesn't.

Given the rapid pace of change in AI tooling, some of these practices may evolve or become obsolete, but the principles behind them should remain a useful foundation.

Step 1: Define What You Want to Achieve

Too often, developers use AI as if it were an oracle: “Generate the code for x ” or “Build me a login page.” But without a clear definition of the problem, you’re more likely to get generic, bloated, or even misleading results. In distributed frontend architectures, this becomes even more important: you’re dealing with boundaries, shared contracts, deployment concerns, and coordination across teams.

Why it matters

AI models are extremely sensitive to ambiguity. If your intent isn’t clear, the model will guess—and that guess may be based on open source defaults that don’t align with your architecture, coding conventions, or business logic. In a micro-frontend setup, this can lead to outputs that violate autonomy, overstep ownership boundaries, or create hidden coupling.

How to approach it

Start by framing your goal in terms of the following:

Functional intent

What should the code *do*?

Architectural constraints

What’s the context in which it lives?

Team boundaries

What should be shared versus private?

This not only improves the model's output but also mirrors how experienced architects approach decomposition. Ideally, write what you need the code to express in a markdown file. This allows you to point the code assistant to that document whenever it loses context or produces results that don't meet your needs. Writing these things down will help you to have clarity of intentions when you are writing your code.

Effective prompting

Consider the following prompt:

Goal: Shared auth module for MFEs in a monorepo

I'm working in a monorepo with multiple React-based micro-frontends using Module Federation. I need to implement a shared authentication module that:

- *Uses AWS Cognito*
- *Exposes hooks for login/logout*
- *Automatically refreshes tokens*
- *Lives in a shared workspace (/packages/auth)*
- *Avoids hard dependencies between MFEs*

Design the module structure before writing any code.

This prompt works because it:

- Sets clear architectural constraints
- Specifies behavioral expectations
- Asks for *design first*, not implementation

The assistant may propose the following:

- A package layout with `auth-provider.tsx`, `useAuth.ts`, and `tokenService.ts`
- Context-based integration strategies
- Suggestions for handling Cognito's refresh tokens in a composable way

You can then iterate from that structure, staying in control while letting AI help you move faster.

Pitfalls to avoid

When prompting, avoid the following pitfalls:

- Don't ask for "the code" immediately, or you'll end up rewriting most of it.
- Don't assume the model knows your stack. Instead, always provide key architectural and technical context.
- Avoid vague requests like "build an auth module." Be specific about inputs, outputs, constraints, and integration points.
- Bounce ideas with AI until you reach the quality level you intend to express in your code before writing a single line of code.
- Remember, AI cannot read your mind yet.

Key takeaways

Be specific with your prompts and don't worry if, at the beginning, it will output incorrect solutions. You will get better over time. Prompt engineering is a new skill we need to master, and it requires a shift in mindset compared to how we are used to thinking about development.

Step 2: Engineer the Context; Don't Just Prompt

Most AI-generated code fails not because the model is flawed, but because it operates without the necessary architectural awareness. This problem is amplified in micro-frontend environments, where boundaries, ownership, contracts, and deployment rules deeply influence how code should be structured. Simply giving a prompt like "add logout to this app" is like asking a new teammate to contribute without onboarding. This is where context engineering comes in.

Why it matters

Language models operate within a limited token window and have no built-in memory of your system's intent, rules, or constraints. Without this context, AI will make unsafe assumptions, usually based on open source defaults or popular boilerplate patterns.

This can lead to:

- Recreating utility functions instead of reusing shared libraries
- Violating ownership boundaries between micro-frontends
- Breaking deployment independence by tightly coupling modules
- Skipping architectural constraints like bundle budgets or event-driven communication

In distributed frontend systems, these aren't trivial errors; they introduce long-term maintenance debt and architectural erosion.

How to approach it

Before expecting it to contribute meaningfully, you need to provide structured onboarding. In the context of micro-frontends, this means offering more than just a one-off prompt; it means consistently embedding your architectural intentions, conventions, and constraints into the development workflow.

This begins by investing upfront time to clarify your intent. Too often, we jump into implementation without pausing to articulate what a feature or module is supposed to do, how it fits into the larger system, and what constraints should shape its design.

A practical way to capture this intent is by documenting it directly in your codebase. Tools like Cursor or Amazon Kiro enable developers to define scoped AI contexts—essentially, giving you the ability to attach project-level and folder-specific metadata that guides the assistant's behavior. For example, you might define general rules for the entire monorepo at the root (such as deployment boundaries or naming conventions), while more granular guidance can live within specific micro-frontend directories. This gives the assistant a sense of proximity and relevance, helping it reason within the right context at the right level.

Just like any form of documentation, these context files must evolve with your system. As implementations change or architectural decisions are revisited, your contextual guidance should be updated accordingly. Otherwise, AI tools will act based on outdated assumptions, leading to incorrect or brittle code.

When structured well, this approach enables you to reference documentation dynamically in your prompts. Instead of rewriting context every time, you can simply say, “Refer to `/shared/auth/.cursor/rules/.ai-context.md` before making changes,” and let the assistant align its understanding to that persistent source of truth.

Ultimately, context engineering is not just about helping machines understand—it's about reinforcing your own architectural discipline. It transforms AI collaboration from ad hoc prompting into a deliberate, repeatable process rooted in the same principles that support long-term software maintainability

Effective prompting

Consider the following prompt:

Create a `useAuth()` hook that reads the current user from the cookie and exposes an `isAuthenticated` boolean and a `logout()` function. The hook should reuse token logic from `/shared/auth/token.ts`.

With this structured prompt, AI is far more likely to:

- Reuse `token.ts` correctly.
- Import from the right shared package.
- Use `useNavigate()` from React Router v6 for redirection.
- Avoid redundant or misaligned logic.
- Stay within architectural boundaries.

When using tools like Cursor or Windsurf, you can set this context file as a system prompt that persists across sessions. This becomes your assistant's architectural brain.

Strategy: Embed context in the codebase

For sustainable velocity, context should live in the repo and not just your memory. I recommend the following files:

`/shared/auth/.cursor/rules/.ai-context.md`

Describes token logic, logout expectations, and reuse rules

`/app-shell/.cursor/rules/.ai-context.md`

Lists global event bus contracts and app shell responsibilities

`.cursor/rules/conventions.md`

Acts as a catch-all architectural playbook (ports, budgets, routing rules, etc.)

This gives you a context reset switch when things go wrong. You don't have to re-explain everything, but just reference the file: "Refer to the authentication rules before implementing logout."

Prompt refinement example

Consider an initial prompt of: "Add logout to this React app." The result might be AI creating local state logic, missing shared libraries, and hardcoding routing.

Now, consider this refined prompt with context engineering:

Each MFE in our monorepo uses /shared/auth for Cognito token management. Implement a logout() function that clears the token, resets state, and redirects using useNavigate() from React Router v6. Follow the shared logic in /shared/auth/token.ts.

The result? It is properly scoped, aligned with architectural intent, and reusable. Now, the model understands *where* to pull logic from and *how* to wire it in without overstepping boundaries.

Key takeaways

As explained throughout this book, architecture is just an expression of intentions. Business and architecture characteristics must be factored into the design of any solution. With the advent of AI, this step cannot be skipped because you will risk having unexpected outcomes, especially when the AI context starts to increase over time due to different implementations you may apply within your system. When you have a handy document to reframe the context for your code assistant, you will save a lot of time and headaches.

Step 3: Reason Like You Normally Would as a Developer

It's tempting to treat AI as a shortcut, a way to avoid thinking through the problem by simply offloading it. But the reality is that AI is most effective when it enhances your thinking, not replaces it. The best results come when you reason through the problem *first*, and then bring the assistant in to validate, refine, or expand your approach.

Think of AI as a pair programmer who's seen a lot of code but doesn't know your domain, your constraints, or your goals unless you share them.

Why it matters

Generative AI excels at filling in blanks, not deciding what the blanks *should* be. If you jump straight to asking for a solution, you're effectively asking the model to make design decisions without any understanding of your architectural intent, trade-offs, or long-term goals. That's a recipe for long-term technical debt.

In micro-frontend architectures, this is especially dangerous. The whole point of a micro-frontend is to allow teams to make local decisions within global constraints. Offloading reasoning removes the very autonomy the architecture is designed to protect.

How to approach it

Before you prompt the assistant, pause and ask yourself:

- What are my constraints?
- What trade-offs am I willing to accept?
- What would I do manually if I weren't using AI?

Then, *explain that reasoning* to the model. You don't need to be verbose—just clear. This gives the assistant the opportunity to challenge, complete, or optimize your thinking, instead of generating code in a vacuum.

Effective prompting

Consider the following prompt:

I'm building a dashboard composed of multiple micro-frontends, each exposed via Module Federation. I want to design an Application Shell component that can dynamically load and render remote widgets at runtime, based on a configuration object.

My current plan:

- *Each micro-frontend is a federated module exposed as a React component.*
- *The Application Shell uses React.lazy() + Suspense to load widgets.*
- *The config object includes the widget key, remote URL, and mount point.*
- *Shared libraries (e.g., React, UI components) are provided as singletons via Webpack configuration.*

Prepare a plan for how you would implement it.

This kind of prompt does two things:

1. It communicates your thought process.
2. It keeps you in the driver's seat.

You're not asking the assistant to invent a solution—you're inviting it into your design review.

When to use this pattern

Use this pattern in the following scenarios:

- You're designing shared libraries or utilities.
- You're deciding between competing patterns.
- You're considering performance trade-offs.

Example: AI as design reviewer

You might be debating how to handle global error reporting in a distributed setup. Your reasoning might be as follows:

Each MFE should log client-side errors to a central service, but I want to avoid coupling every app directly to the logging library. My idea is to expose a shared logging interface in /shared/logging, and inject the actual logger at runtime via dependency injection.

You can then ask the assistant to:

- Validate the pattern.
- Suggest improvements for resilience.
- Help scaffold the initial interface and implementation.

This mirrors how you'd approach the problem with a senior engineer, but now you're just getting that input faster.

Pitfalls to avoid

When prompting, avoid the following pitfalls:

Skipping the thinking

Asking the model to “design a solution for logging across MFEs” is too vague and abdicates responsibility.

Over-specifying the output

You’re not asking for code yet, you’re co-designing. Let the model respond with alternatives, not just confirmations.

Ignoring pushback

If the assistant challenges your assumptions, don’t dismiss it. Evaluate the counterpoints and revise if needed.

Encouraging alternative solutions

Ask if there are better ways to handle the problem that you are not aware of.

Key takeaways

Integrate AI as a strategic partner in your brainstorming processes. Don’t only rely on your own knowledge, but ask AI to check online for other resources or repositories to challenge your assumptions or ideas.

Having this capability inside our favorite IDE or CLI can only accelerate the decision making and increase the robustness of our code. You might be surprised to see how many things you can learn with this approach.

Step 4: Iterate in Small, Focused Steps

When working with AI, there’s a strong temptation to “go big”—to ask for an entire module, feature, or integration in a single prompt. While modern models are capable of handling complex tasks, this approach almost always leads to bloated, brittle, or contextually incorrect output.

Micro-frontends demand precision, because even small boundary violations can cause runtime and deployment issues across teams. Small mistakes in how boundaries are drawn, how state is shared, or how modules are initialized can introduce long-term complexity and coupling. The best way to mitigate this risk is to work incrementally—one focused step at a time.

Why it matters

Language models are strongest when asked to solve *narrow*, well-defined problems. Each response gives you the opportunity to:

- Validate assumptions.
- Spot incorrect logic early.
- Maintain architectural integrity.
- Gradually build up a correct, well-factored solution.

This mirrors the best practices of manual development: test-driven development, frequent commits, and small PRs. The same mindset applies when working with AI.

How to approach it

Instead of asking the model to generate “a loader for all micro-frontends,” start by asking it to write a utility that loads a single remote module by name. Once that’s working, add error handling. Then integrate it into a dynamic host component. Each step builds on the previous one with validated behavior.

Effective prompting

Consider the following prompt:

Write a TypeScript utility function that loads a remote federated module at runtime. Inputs:

- `remoteUrl: string` (e.g.,
`https://www.mysite.com/errorsdomain/remoteEntry.js`)
- `scope: string` (e.g., `errors_widget`)
- `module: string` (e.g., `'./MFE'`)

Return the loaded module as a promise of a React component.

This is intentionally scoped:

- One task (loading the module)
- No side concerns (e.g., rendering, state)
- Typed inputs and expected output
- Hints at the next step (rendering) without mixing concerns

Now, consider this follow-up prompt:

Now write a `RemoteComponent` React wrapper that uses the utility above, calls it with props, and renders the result using `React.lazy()` and `Suspense`. Show an error boundary and a fallback spinner.

By separating module loading from rendering, you preserve single responsibility and gain clearer control over error handling and fallback UI—all while keeping the AI focused.

Why this works

This works because:

- Smaller prompts reduce the chance of hallucination.
- Feedback cycles are tighter, so you can test and validate before moving on.
- Errors are easier to isolate and fix.
- You maintain full control over architecture and composition.

Antipattern: The “one prompt to rule them all”

Let’s compare. Here’s a bad prompt:

Create a widget host that loads any micro-frontend from a config, handles failures, shows a spinner, injects state, and renders layout dynamically.

This will likely result in:

- Poor error boundaries
- Assumptions about how micro-frontends are exposed
- Missing types
- An overwhelming amount of hardcoded logic you’ll need to untangle

By contrast, when you prompt incrementally, you:

- Keep code composable.
- Align with your project’s design.
- Reduce refactoring overhead.

Key takeaways

As long as you think like a developer and you work iteratively, you will be in great shape to leverage the power of any code assistant. Remember to refactor often because AI works better with co-located code than with sparse code across multiple contexts (a.k.a. files and folders).

Step 5: Review and Refactor the Output

When working with AI-generated code, one critical misconception is to treat the output as final or production-ready. In reality, the AI acts like a junior developer: capable of generating scaffolding and boilerplate, but requiring thorough review, adjustment, and integration to meet your project's standards and constraints. It's important to remember that AI often misses edge cases, so human validation remains essential to ensure robustness and correctness.

This step is where your expertise is indispensable. It ensures the generated code fits your architecture, coding standards, and handles edge cases gracefully.

Why it matters

AI models generate plausible code based on patterns learned from vast data sets, but they don't understand your specific domain or project nuances straightaway. Without a rigorous review, you risk introducing:

- Subtle bugs (e.g., incorrect async handling or state mutations)
- Security issues (e.g., improper token management)
- Performance regressions (e.g., unnecessary re-renders or large bundle sizes)
- Architectural antipatterns (e.g., breaking encapsulation between micro-frontends)

By treating AI output as a first draft rather than a final product, you safeguard code quality and maintain your system's integrity.

How to approach it

You can approach this in the following ways:

Run static analysis and type checking.

Integrate the AI-generated code into your IDE and run your linters, type checkers, and unit tests immediately. You can also ask AI to run your test and fix the problems that arise.

Verify domain logic.

Compare the generated logic against your business rules and domain models. This is especially critical for shared services like authentication, authorization, or data validation.

Refactor for clarity and maintainability.

Improve variable naming, extract reusable functions, and simplify complex expressions. This step often reduces technical debt created by verbose or redundant AI code.

Add missing error handling and edge cases.

AI often omits handling network errors, retries, or fallback UI. Add these to align with your system's resilience strategy.

Ensure consistency with existing codebase.

Adapt styles, naming conventions, and architectural patterns to seamlessly integrate the new code.

Example scenario

Suppose the AI generates a React hook to refresh JSON Web Tokens (JWTs):

```
function useTokenRefresh() {
  const [token, setToken] = useState(null);

  useEffect(() => {
    async function refresh() {
      const newToken = await fetchNewToken();
      setToken(newToken);
    }
    refresh();
  }, []);

  return token;
}
```

Our review observations could be:

- The hook does not handle token expiration timing or errors.
- It uses `useState` for token storage, which may duplicate state already managed in a global context.
- No cleanup or retry logic is implemented.

The refactored version could look as follows:

```
function useTokenRefresh() {
  const { token, setToken } = useAuthContext();

  useEffect(() => {
    let isMounted = true;

    async function refresh() {
      try {
        const newToken = await fetchNewToken();
        if (isMounted) setToken(newToken);
      } catch (error) {
        // handle error, e.g., log out user or retry
      }
    }

    // Refresh token 1 minute before expiration
    const timerId = setTimeout(refresh, calculateRefreshTime(token));

    return () => {
      isMounted = false;
      clearTimeout(timerId);
    };
  }, [token, setToken]);

  return token;
}
```

This refactor improves robustness, integrates with the existing context, and handles life cycle events properly.

Prompt pattern for review

When asking AI for help with review or refactoring, frame your prompt to highlight your concerns:

Here is a React hook generated by AI that refreshes auth tokens. Please review it for:

- *Proper error handling*
- *Integration with a global auth context*
- *Memory leaks or life cycle issues*
- *Performance optimizations*

Suggest improvements or rewrite as needed.

Key takeaways

When reviewing and refactoring the output, remember:

- Never accept AI code blindly; treat it as a starting point.
- Apply your domain knowledge and coding standards rigorously.
- Use review as an opportunity to teach the AI by iterating your prompts with feedback.

Step 6: Automate Through AI

One of the most underestimated advantages of integrating AI into a micro-frontend architecture is how effortlessly it can help automate development workflows. From dependency management to performance budgets, AI isn't just about generating code; it's a force multiplier when embedded in the right parts of your engineering pipeline.

Automation is a core tenet of scalable systems. In micro-frontends, where multiple teams contribute to isolated yet interconnected pieces of the frontend, keeping alignment across boundaries is often tedious and prone to errors. But now, with AI-powered tools and assistants, common setup tasks and architectural enforcement can be delegated efficiently and confidently.

Why it matters

Micro-frontends multiply the surface area of architectural decisions. Without automation:

- Dependency drift can go unnoticed across apps.
- Teams might break encapsulation rules without realizing.

- Performance regressions sneak into production through bloated bundles.

By automating routine checks and configuration tasks with AI, you reduce manual oversight and reinforce architectural discipline—even across large, fast-moving codebases.

How to approach it

Start by identifying repetitive or governance-critical tasks that AI can help you scaffold and maintain. For example:

Keep dependencies updated

Instead of manually configuring Dependabot, ask your assistant: “Generate a Dependabot configuration that checks all `package.json` files in this monorepo weekly.” AI will scaffold the YAML file and explain how to scope updates per micro-frontend if needed.

Enforce architectural boundaries

Use tools like `ts-arch` to write natural language constraints that are translated into TypeScript rules: “Prevent any imports from `/shared/auth` inside `/apps/public-site`; only `/apps/dashboard` is allowed.” AI will turn this into an architectural test and optionally integrate it into your CI pipeline.

Guard bundle size

You can request the assistant to scaffold a pre-push Git hook that runs `webpack-bundle-analyzer` and fails if the micro-frontend exceeds a defined size budget. Example prompt: “Write a Git hook that checks if the built JavaScript bundle for `/apps/profile` exceeds 200 KB. If it does, fail the push.”

CI integrations

Describe the automation you want (e.g., linting, testing, visual diffs) and AI can scaffold the GitHub Actions workflow from scratch: “Create a GitHub workflow that runs lint, tests, and checks bundle size for all micro-frontends in parallel.”

In all of these, AI acts as your DevOps copilot. What once took hours of reading docs and wiring configs can now be bootstrapped in minutes—and reviewed, customized, and expanded with ease.

Effective prompting

When aiming for automation, provide:

- The tool (e.g., `ts-arch`, `webpack-bundle-analyzer`)
- The scope (e.g., a specific micro-frontend or shared package)
- The policy (e.g., max size, allowed import paths)
- The trigger (e.g., pre-commit, CI job, nightly check)

Here is an example prompt:

Create a test using `ts-arch` to block any import from `/packages/analytics` inside `/apps/public-site`.

Then ask follow-ups, like “Integrate that test in a GitHub Actions CI pipeline and fail the job if the rule breaks.”

Key takeaways

AI lets you shift from reactive fixes to proactive governance in hours—not days or weeks. Automating routine and repetitive enforcement in micro-frontends is no longer a manual burden; it’s a strategic use of AI as a policy enforcer. This accelerates development velocity while reducing the likelihood of architectural drift. By leaning into automation with AI, you create space for teams to focus on product quality and innovation, without sacrificing cohesion or control.

Where Code Assistants and AI Fall Short

Even with a carefully crafted prompting strategy, there are moments when AI hits a wall, and that’s when your experience truly shines.

I remember one case clearly. I was working on a micro-frontend setup using Module Federation and needed to integrate `react-router-dom`. The AI kept trying to help by generating new abstractions and patterns, churning out code after code, each more complex and tangled than the last. It was stuck in a loop, endlessly iterating, but the code it produced was becoming a mess—hard to maintain, harder to understand.

Something in my gut told me this wasn't the right path. I paused the AI, stepped back, and dug into the problem myself. After a bit of searching on GitHub, I found the smoking gun: a known incompatibility between `react-router-dom` version 7 and Module Federation version 2. This wasn't something the AI could have known on its own. It doesn't have the lived experience or the intuition born from wrestling with tricky bugs and subtle library clashes over time.

This moment was a wake-up call. The AI was powerful, sure. It could speed up routine work and suggest solutions. But it couldn't replace the hard-earned wisdom that comes from real-world experience. This reinforces why architectural discipline and review remain nonnegotiable, even with AI. It didn't know when to stop chasing patterns and start looking for documented issues or proven workarounds.

That day, I realized that AI is a remarkable tool, but it's still a junior partner. It's up to you—the experienced developer—to recognize those edge cases where AI falls short, to bring your judgment, and to guide the process forward. In complex architectures like micro-frontends, this balance between AI assistance and human insight isn't just valuable; it's essential.

Don't just take my word for it! Take some industry analysis that reviews the impact of AI on coding. [The GitClear “AI Copilot Code Quality” 2025 report](#) reveals a dramatic shift in code authorship patterns since 2020. Let's unpack the key metrics and their real-world impact through the lens of continuous AI iteration.

Understanding Code Change Operations

To contextualize the findings, we first define the core operations tracked in the study:

Added code

Newly authored lines distinct from existing logic

Moved code

Lines relocated to new files/functions (indicating refactoring)

Copy and pasted code

Duplicated blocks committed simultaneously across files

Churn

Lines revised within two weeks of authorship (suggesting instability)

The data in [Table 14-1](#) illustrates how AI-assisted development has reshaped codebase dynamics.

Table 14-1. A four-year analysis of the impact of AI on code

Metric	2020	2024	Change
Added code	39.2%	46.2%	+17.9%
Moved code	24.1%	9.5%	-60.6%
Copy and pasted	8.3%	12.3%	+48.2%
Churn (revised within two weeks)	3.1%	5.7%	+83.8%

Let's translate these key metrics to reality:

Refactoring collapse

Moved code plummeted from nearly a quarter of changes to under 10%. This suggests developers are prioritizing rapid feature delivery over structural optimization.

Duplication surge

Copy-and-pasted operations now exceed moved code for the first time in recorded history. Concurrently, commits containing duplicated blocks (≥ 5 lines) rose 8.5 times, from 0.70% (2020) to 6.66% (2024).

Instability spike

Code revised within two weeks increased by 26% year-over-year, with 79.2% of modified code in 2024 being under one month old (versus 70.0% in 2020). This “disposable code” pattern accelerates technical debt.

A quote in the [GitClear “AI Copilot Code Quality” research report](#) that grabbed my attention immediately was:

The decline of moved code and rise of copy/paste suggest developers are trading consolidation for expediency—a tradeoff that accrues technical debt exponentially.

A marked decline in refactoring activity, a surge in code duplication, and rising code churn all signal a trend toward short-term gains over long-term maintainability.

Micro-frontends rely on well-defined interfaces, encapsulated logic, and stable shared contracts between teams. When AI-assisted workflows prioritize speed and local optimization—such as copy and pasting logic across micro-frontends or skipping structural refactors—it erodes the very principles that make the architecture effective.

To mitigate these risks, teams should introduce architectural safeguards into their AI workflows, as discussed throughout this book. Establish metrics to monitor churn, duplication, and interface volatility across micro-frontends—not just within isolated repos. Treat AI-generated code as a first draft, not a final product. Regular, intentional refactoring must become a scheduled practice, not an afterthought. Reviewing pull requests and investing time in code quality and clarity of intent is no longer optional—it's a core skill for every team.

When guided by discipline, sound engineering practices, and architectural intent, AI can indeed accelerate development without compromising modularity. But if we prioritize speed over cohesion, we risk repeating well-known mistakes at an unprecedented pace. We've seen what happens when systems evolve without direction. You should remember your time well with spaghetti code, brittle dependencies, and costly rewrites. AI doesn't change that outcome; it simply gets us there faster. And when that tipping point arrives, the challenge will be even greater: we'll have less human context, more fragmented logic, and an overwhelming volume of machine-generated code to make sense of.

The risks aren't just technical; they're operational. When something fails in production, how do we recover quickly if we no longer understand the architecture, the intent behind the code, or even what the AI was trying to do? The path forward isn't to reject AI, but to embrace it with structure, intention, and the architectural maturity that these systems demand. All the practices we have learned in the past won't be thrown away, but they will become the guardrails to properly govern our codebase.

What the Future Holds for AI and Micro-Frontends

As AI becomes more embedded in our workflows, one clear shift is emerging: developers are moving from writing every single line of code to reviewing, steering, and refining what AI generates. That doesn't mean we stop coding—far from it. The best engineers will still think like developers, but they'll also design like architects and review like editors. In architectures like micro-frontends—where composition, boundaries, and autonomy are essential—that level of oversight isn't optional: it's a requirement.

This oversight is more than just manual review; it demands context engineering. Feeding AI models with precise, well-structured context about architectural boundaries, shared contracts, design constraints, and team ownership ensures the generated code is relevant, cohesive, and resilient. Without thoughtful context, AI risks becoming disconnected fragments that introduce complexity rather than reduce it.

In a micro-frontend environment, context engineering is even more critical. Because each team independently develops and deploys its own frontend part, AI must understand not only the technical interfaces but also the organizational ownership and UX boundaries that define each micro-frontend. Properly engineered context enables AI to generate micro-frontends that seamlessly integrate with an application shell and other micro-frontends, respecting state encapsulation, routing, and styling conventions. It also enables teams to evolve their parts autonomously without breaking the global experience—a foundational promise of the micro-frontend approach. Neglecting this leads to brittle, tightly coupled outputs that undermine the benefits of modularity and team autonomy.

We're also seeing a new kind of acceleration: idea-to-execution tools that enable developers, and even non-developers, to express product intent and see working interfaces materialize in real time. Platforms like V0, Lovable, and similar AI-first UI generators are reducing the friction between concept and code. When paired with clearly defined design systems and architecture constraints, this shift can dramatically improve the speed and accuracy with which teams ship usable features into production. In a micro-frontend world, where every team owns a slice of the experience, this reduces coordination overhead while preserving autonomy.

Meanwhile, foundational shifts in browser technology like the WebGPU API are unlocking new possibilities. Today, it means GPU-accelerated rendering and local compute workflows. Tomorrow, it could enable lightweight, in-browser AI models that can enhance the UX or DX entirely offline. That opens the door to deeply personalized, responsive developer environments and more secure local inference.

At the operational level, automation is becoming more accessible than ever. If you have an idea for automating a repetitive task or enhancing guardrails to get a better system, act on it. What once took hours of scripting can now be scaffolded with a few well-structured prompts. AI won't eliminate complexity, but it will make it cheaper to manage.

But with all this acceleration comes a crucial reminder: don't trade short-term speed for long-term chaos. It's still your system. AI can generate code, but it can't entirely own the architecture. It can scaffold logic, but it can't guarantee resilience. The responsibility for designing a system that's coherent, testable, and adaptable still belongs to you.

And perhaps most importantly, I hope that this new wave of productivity will give senior engineers more time to do something that has always mattered: train the next generation. Because the only sustainable future in tech is one where solid technical skills are broadly distributed across the community. Without that, we risk concentrating power and knowledge into the hands of a few. But over the past 40 years, it was the open sharing of ideas, mentorship, and a culture of helping others without gatekeeping that made this community thrive. That collective spirit didn't just shape software; it shaped economies, connected the world, and gave rise to entirely new industries. We owe it to the next wave of engineers to carry that spirit forward.

The future is fast, but it should also be fair, intentional, and shared. When AI is paired with care, clarity, and community, we don't just build better systems—we build a better industry!

Index

A

- abstraction of code, [Introducing Micro-Frontends](#)
 - favoring code duplication over incorrect abstractions, [Architecture Evolution](#)
 - premature abstraction, [Introducing Micro-Frontends](#), [Premature Abstraction](#), [Trade-Offs and Pitfalls](#)
 - into shared library, [Architecture Evolution](#)
- access control, fine-grained, in polyrepo environments, [Polyrepo](#)
- access tokens, [How to Manage Data](#)
- accessibility, using AI in, [Accessibility](#)
- account management in internal ecommerce site, [The Project](#)
- Account Management micro-frontend, [Account Management Micro-Frontend](#)
- ACL (see anti-corruption layer)
- acquisitions, using multi-framework approach to integrate acquired software, [Multi-framework approach](#)
- adapter pattern, [Embedding a Legacy Application](#)
- ADRs (architecture decision records), [Architecture Decision Records](#)
- AHA (avoid hasty abstractions) programming principle, [Premature Abstraction](#)
- AI and micro-frontends, [AI and Micro-Frontends: Augmenting, Not Replacing-What the Future Holds for AI and Micro-Frontends](#)
 - architecture not yet automatable, [Architecture Is Not Yet Automatable-Architecture Is Not Yet Automatable](#)
 - capabilities and limitations of AI, [AI and Micro-Frontends: Augmenting, Not Replacing](#)

code assistants and AI, falling short, [Where Code Assistants and AI Fall Short-Understanding Code Change Operations](#)

context engineering, [Context Engineering to the Rescue-Context Engineering to the Rescue](#)

feasible AI use cases today, [Feasible AI Use Cases Today-Discovering Alternatives and Challenging Assumptions with AI](#)

accessibility, [Accessibility](#)

automation, [Automation](#)

bouncing ideas and discovering alternatives with agentic AI, [Bouncing Ideas and Discovering Alternatives with Agentic AI](#)

code migration, [Code Migration](#)

code optimization, [Code Optimization](#)

diagram generation, [Diagram Generation](#)

discovering alternatives and challenging assumptions, [Discovering Alternatives and Challenging Assumptions with AI](#)

keeping README alive, [Keeping the README Alive](#)

project scaffolding, [Project Scaffolding](#)

prototyping, [Prototyping](#)

testing, [Testing](#)

future of AI and micro-frontends, [What the Future Holds for AI and Micro-Frontends](#)

playbook for working with AI, [My Playbook to Work with AI-Key takeaways](#)

automating through AI, [Step 6: Automate Through AI-Key takeaways](#)

defining what you want to achieve, [Step 1: Define What You Want to Achieve-Key takeaways](#)

engineering the context, not just prompting, [Step 2: Engineer the Context; Don't Just Prompt-Key takeaways](#)

iterating in small, focused steps, **Step 4: Iterate in Small, Focused Steps-Key takeaways**

reasoning normally as a developer, **Step 3: Reason Like You Normally Would as a Developer-Key takeaways**

reviewing and refactoring output, **Step 5: Review and Refactor the Output-Key takeaways**

“AI Copilot Code Quality” 2025 report (GitClear), **Where Code Assistants and AI Fall Short**

alternatives, discovering using AI, **Discovering Alternatives and Challenging Assumptions with AI**

Amazon, **Highly Observable**

(see also AWS)

Amazon Q, **Bouncing Ideas and Discovering Alternatives with Agentic AI**

CloudFront, **Hosting a Client-Side Rendering Micro-Frontend Project**

two-pizza team rule, **Working with a Service Dictionary**

working backward when considering product ideas, **Working Backward**

ambiguity, AI models' sensitivity to, **Why it matters**

Angular

legacy application developed using, **Embedding a Legacy Application**

Native Federation library adopted by community, **Composing micro-frontends**

anti-corruption layer (ACL), **Anti-Corruption Layer to the Rescue**

using in micro-frontends, **Anti-Corruption Layer to the Rescue**

using when integrating different technologies or legacy systems, **Anti-Corruption Layer to the Rescue**

antipatterns

common, in micro-frontend implementations, **Common Antipatterns in Micro-Frontend Implementations-Summary**

anti-corruption layer to the rescue, [Anti-Corruption Layer to the Rescue](#)-[Anti-Corruption Layer to the Rescue](#)

micro-frontend anarchy, [Micro-Frontend Anarchy](#)-[Micro-Frontend Anarchy](#)

micro-frontend or component, [Micro-Frontend or Component?](#)-[Micro-Frontend or Component?](#)

premature abstraction, [Premature Abstraction](#)

sharing between micro-frontends, [Sharing State Between Micro-Frontends](#)-[Sharing State Between Micro-Frontends](#)

unidirectional sharing, [Unidirectional Sharing](#)-[Unidirectional Sharing](#)

one prompt to rule them all, [Antipattern: The “one prompt to rule them all”](#)

API gateway, [API Integration and Micro-Frontends](#), [Working with an API Gateway](#)-[Server-Side Composition with an API Gateway](#)

capabilities of, [Working with an API Gateway](#)

client-side composition with service dictionary and, [Client-Side Composition with an API Gateway and a Service Dictionary](#)

downsides of, [Working with an API Gateway](#)

one API entry point per business domain, [One API Entry Point per Business Domain](#)

scenarios where BFF is more useful, [Working with the BFF Pattern](#)

server-side composition with API gateway, [Client-Side Composition with an API Gateway and a Service Dictionary](#), [Server-Side Composition with an API Gateway](#)

simplifying client/server communication and centralizing functionalities via edge functions, [Working with an API Gateway](#)

API layer

advantages of independent deployment of microservices for, [Independent Deployment](#)

designed as monolith in new projects, [Monolith to Distributed Systems](#)
GraphQL acting as unique entry point for, [Using GraphQL with Micro-Frontends](#)
monolithic, in monolith to micro-frontends case study, [Technology Stack](#)
service dictionary, list of endpoints available for a micro-frontend, [Working with a Service Dictionary](#)

API-first development, [Hidden Implementation Details](#)

APIs

aggregating by domain, in combination of BFF and micro-frontends, [Working with the BFF Pattern](#)

API integration and micro-frontends, [Backend Patterns For Micro-Frontends-API Integration and Micro-Frontends](#)

API strategies for SSR micro-frontends, [API Strategies](#)

APIs come first, then implementation, [APIs Come First, Then Implementation](#)

cacheable and non-cacheable, [Moving to Microservices](#)

communication across projects defined via APIs in polyrepo, [Polyrepo](#)

in components versus micro-frontends, [Modeled Around Business Domains](#)

consistency of, [API Consistency](#)

designing API schema with GraphQL, [Using GraphQL with Micro-Frontends](#)

GraphQL query language for, [Using GraphQL with Micro-Frontends](#)

multiple micro-frontends consuming same API, best practices, [Multiple Micro-Frontends Consuming the Same API-Multiple Micro-Frontends Consuming the Same API](#)

reducing API surface exposed to micro-frontend containers, [Testing Your Micro-Frontend Boundaries](#)

service dictionary provided via, [Working with a Service Dictionary](#)

serving raw data, clear separation from services performing frontend rendering, [HTML Fragments](#)

well-defined, for necessary inter-component communication, [Unidirectional Sharing](#)

application initialization, [Application Initialization](#)

application shell

cart component present in for internal ecommerce site project, [Developing the Check-Out Experience](#)

consuming service dictionary API as first step, [Working with a Service Dictionary](#)

exposing or injecting GraphQL endpoint to all the micro-frontends, [Using GraphQL with Micro-Frontends and a Client-Side Composition](#)

implementation in internal ecommerce site project, [Application Shell-Application Shell](#)

loading micro-frontend acting as adapter for legacy application, [Embedding a Legacy Application](#)

loading micro-frontend on at runtime, [Composition](#)

main reasons to load micro-frontends in, [Application Shell](#)

maintaining separation between My account micro-frontend and, [Account Management Micro-Frontend](#)

parent, unidirectional data flow to children (micro-frontends), [Unidirectional Sharing](#)

responsibilities in monolith to micro-frontends migration, [Application Shell Responsibilities](#)

in vertical-split micro-frontends

client-side composition and routing using application shell, [Application Shell](#)

in vertical-split micro-frontends, [Vertical Split](#), [Application Shell-Application Shell](#)

[App_Shell](#), [Real-World Implementation](#)

architecture decision records (ADRs), [Architecture Decision Records](#)

architectures for micro-frontends, [Discovering Micro-Frontend Architectures-Summary](#)

analysis of architecture characteristics, [Architecture Characteristics](#)

analysis of technical implementations, [Architecture Analysis](#)

applying the micro-frontend decision framework, [Discovering Micro-Frontend Architectures-Horizontal Split](#)

horizontal split, [Horizontal Split](#)

vertical split, [Vertical Split](#)

architectural decisions in monolith to micro-frontends case study, [Micro-Frontend Decision Framework Applied](#)

architecture and trade-offs, [Architecture and Trade-Offs](#)

architecture not yet automatable, [Architecture Is Not Yet Automatable-Architecture Is Not Yet Automatable](#)

architecture team in automation pipeline case study, [Setting the Scene](#)

architecture testing tools, [Code-Quality Review](#)

edge-side, [Edge Side](#)

horizontal split, [Horizontal-Split Architecture-iframes](#)

horizontal-split with server-side composition, [Server Side-Modern Server-Side Rendering Frameworks](#)

importance of CI in, [Continuous Integration Strategies](#)

micro-frontend decision framework and, [Micro-Frontend Decision Framework](#)

server-side rendering frameworks, modern, [Modern Server-Side Rendering Frameworks-Edge Side](#)

testing of architecture characteristics, [Fitness Functions](#)

using iframes, [iframes-Web Components](#)

using web components, [Web Components-Web Fragments](#)

vertical split, [Vertical-Split Architectures-Horizontal-Split Architecture](#)

Web Fragments, Web Fragments

artifacts

storage in artifact repository or AWS S3 buckets, [Post-Build Review](#)

storage in monolith to micro-frontends case study, [Technology Choice](#)

assumptions, challenging, using AI assistants, [Discovering Alternatives and Challenging Assumptions with AI](#)

Astro.js framework, [Horizontal Split](#), [Composing Micro-Frontends](#), [Modern Server-Side Rendering Frameworks](#)

server islands, use in composition of SSR micro-frontends, [Composition Approaches](#)

use cases for, [Use Cases](#)

authentication

centralizing via edge functions in API gateway, [Working with an API Gateway](#)

challenges in horizontal-split architectures, [Authentication](#)

handling in vertical-split micro-frontends, [Application Shell](#)

integrating in micro-frontends in monolith to micro-frontends case study, [Integrating Authentication in Micro-Frontends](#)

managing during migration to micro-frontends, [Best Practices and Patterns](#)

managing tokens in multi-zone micro-frontend architecture, [How to Manage Data](#)

streamlined, in micro-frontend composed by splitting URL, [Splitting by URL](#)

authentication tokens, [Challenges Unique to Micro-Frontends](#), [Micro-Frontend Communication](#), [How to Manage Data](#)

sharing across micro-frontends, [Sharing state](#)

authorization, [Working with an API Gateway](#)

handling in vertical-split micro-frontends, [Application Shell](#)

Authorization headers containing JWTs, [Application Shell](#)

automation

in API layer of new project designed as monolith, **Monolith to Distributed Systems**
automating with AI, **Step 6: Automate Through AI**-Key takeaways

automation pipelines scaling alongside codebase in monorepos, **Monorepo**
better integration with current automation strategy, **Micro-Frontend Decision Framework Applied**

constant investment in tools, monorepos requiring, **Monorepo**

culture of, in micro-frontends, **Culture of Automation**

culture of, in microservices, **Culture of Automation**

current use cases for AI in, **Automation**

of design system version validation, **Implementing a Design System**

microservices and, **Micro-Frontend Principles**

review of zones using shared components and automatic updates of dependencies,
How to Handle Shared Components

automation of micro-frontends, **Micro-Frontend Automation**

automation principles, **Automation Principles-Define Your Test Strategy**

defining guardrails, **Define Your Guardrails**

defining test strategy, **Define Your Test Strategy**

empowering teams, **Empower Your Teams**

iterating often, **Iterate Often**

keeping the feedback loop fast, **Keep the Feedback Loop Fast**

continuous integration strategies, **Continuous Integration Strategies-Continuous Integration Strategies**

developer experience, **Developer Experience-Environments Strategies**

micro-frontend-specific operations for automation pipelines, **Micro-Frontend-Specific Operations**

observability, **Observability**

using fitness functions to test architecture characteristics in CI, **Fitness Functions**-
Fitness Functions

version control, **Version Control-A Possible Future for a Version Control System**

automation pipeline for micro-frontends, case study, **Automation Pipeline for Micro-Frontends: A Case Study-Summary**

ACME Inc., video-streaming service, building micro-frontends, **Setting the Scene-Setting the Scene**

automation strategy summary, **Deployment**

automation strategy, key areas in, **Setting the Scene**

builds, **Build**

initialization of pipeline, **Pipeline Initialization**

post-build review, **Build**

automation strategy sample with key steps for building micro-frontends, **Frictionless Micro-Frontend Blueprints**

AWS (Amazon Web Services), **Micro-Frontend Principles**

AWS Lambda@Edge, **Technology Choice**

Frontend Service Discovery on AWS, **Availability in the Market**

Lambda, **Scalability Challenges, Deployment**

S3 (Simple Storage Service), **Hosting a Client-Side Rendering Micro-Frontend Project**

S3 bucket for micro-frontend, **Deployment, Technology Choice**

S3 buckets, **Post-Build Review, Dependency Management**

serverless services, use of HTML fragments, **HTML Fragments**

B2B applications, [Vertical-Split Architectures](#)

no need for SSR, [When to Use Server-Side Rendering](#)

use of horizontal-split architecture with server-side composition, [Use Cases](#)

use of horizontal-split architectures, [Use Cases](#)

using iframes to implement micro-frontends in, [iframes](#)

B2C applications, [Vertical-Split Architectures](#), [Server-Side Rendering Micro-Frontends](#)

B2C websites

monolithic backend architecture for internal ecommerce site project, [The Project](#)

use of horizontal-split architecture with server-side composition, [Use Cases](#)

backend for frontend pattern (see BFF pattern)

backend patterns for micro-frontends, [Backend Patterns For Micro-Frontends-Summary](#)

backends

API exposed by, adding product to cart, [Developing the Check-Out Experience](#)

backend integration in monolith to micro-frontends migration, [Backend Integration](#)

BFF pattern developing niche backends for each user experience, [Working with the BFF Pattern](#)

including APIs supported by backend in service dictionary, [Working with a Service Dictionary](#)

independence from frontends, [Best Practices and Patterns](#)

iteratively modernizing the backend, [Best Practices and Patterns](#)

migrating backend layer to microservices in monolith to micro-frontends case study, [Technology Stack](#)

migrations harder and slower than frontend, [Trade-Offs and Pitfalls](#)

BBC website architecture, caching in action, [The BBC Architecture: Caching in Action](#)

BEM (Block Element Modifier), [Clashes with CSS class names and how to avoid them](#)

best practices, **Best Practices-The Right Approach for the Right Subdomain**

API consistency, **API Consistency**

APIs come first, then implementation, **APIs Come First, Then Implementation**

composing micro-frontends in horizontal split with iframes, **Best Practices and Drawbacks**

for deciding which problems are solved with migration to micro-frontends, **Best Practices and Patterns**

in deciding which modules or features to migrate first, **Best Practices and Patterns**

in getting management on board with migrating to micro-frontends, **Best Practices and Patterns**

in handling shared dependencies and version management between legacy and micro-frontend code, **Best Practices and Patterns**

of high-performance organizations across industries, **How Do Committees Invent?**

in maintaining user experience and consistency during migration, **Best Practices and Patterns**

maintenance of, difficulties using polyrepo, **Polyrepo**

in managing cross-cutting concerns during migration, **Best Practices and Patterns**

multiple micro-frontends consuming same API, **Multiple Micro-Frontends Consuming the Same API-Multiple Micro-Frontends Consuming the Same API**

in planning migration to micro-frontends, **Best Practices and Patterns**

right approach for the right solution, **The Right Approach for the Right Subdomain**

WebSocket and micro-frontends, **WebSocket and Micro-Frontends**

BFF (backend for frontend) pattern, **API Strategies, Backend Patterns For Micro-Frontends, API Integration and Micro-Frontends, Working with the BFF Pattern-Server-Side Composition with a BFF and a Service Dictionary**

client-side composition with BFF and service dictionary, **Client-Side Composition with a BFF and a Service Dictionary-Client-Side Composition with a BFF and a Service Dictionary**

concerns due to some inherent pitfalls, [Working with the BFF Pattern](#)
scenario for use by micro-frontend instead of API gateway, [One API Entry Point per Business Domain](#)

server-side composition with BFF and service dictionary, [Server-Side Composition with a BFF and a Service Dictionary](#)

bidirectional sharing across micro-frontends, [Shared code](#), [Module Federation 101](#)
challenges resulting from, [Unidirectional Sharing](#)

Block Element Modifier (BEM), [Clashes with CSS class names and how to avoid them](#)
blue-green deployments, [Independent Deployment](#), [Welcome to the Frontend Discovery Schema](#), [Real-World Implementation](#)

versus canary releases, [Blue-Green Deployment Versus Canary Releases](#)-[Blue-Green Deployment Versus Canary Releases](#)

blueprints, frictionless, for micro-frontends, [Frictionless Micro-Frontend Blueprints](#)

BMW, use of micro-frontends, [BMW](#)

bootstrap.js file, [Application Shell](#)

boundaries, micro-frontend

identifying boundaries for micro-frontends without confusing them with components, [Micro-Frontend or Component?-Micro-Frontend or Component?](#)

reviewing for high-complexity subdomains, [High-Complexity Subdomain testing](#), [Testing Your Micro-Frontend Boundaries](#)

bounded contexts, [DDD with Micro-Frontends](#)

best API entry point strategy per bounded context, choosing, [One API Entry Point per Business Domain](#)

defining, [How to Define a Bounded Context](#)

mapping frontend and backend together in, [DDD with Micro-Frontends](#)

subdomains overlapping with, [DDD with Micro-Frontends](#)

branching strategies

- different per project in polyrepos, [Polyrepo](#)

- trunk-based, in monorepos, [Monorepo](#)

browsers

- foundational shifts in technology such as WebGPU API, [What the Future Holds for AI and Micro-Frontends](#)

- integrating AI into developer tools, [Debugging with AI](#)

- prefetching and prerendering features to reduce perceived latency, [Next.js Multi-Zones](#)

bugs

- finding and fixing, costs rising over time, [Setting the Scene](#)

- fix-forward strategy for, [Setting the Scene](#)

- reducing risk in production in monolith to micro-frontends case study, [Technology Choice](#)

- unexpected, introduced by bidirectional sharing, [Unidirectional Sharing](#)

build-time sharing of components in multi-zone architectures, [How to Handle Shared Components](#)

builds

- build stage in micro-frontend automation pipeline case study, [Build](#)

- post-build review, [Build](#)

bundle size, [Fitness Functions](#)

bundlers, [Module Federation 101](#)

- (see also [Webpack](#))

business domains, [Micro-Frontend Principles, API Integration and Micro-Frontends](#)

- (see also domains)

- identifying for home page of ecommerce platform, [Micro-Frontend or Component?](#)

modeling around, in micro-frontends, **Modeled Around Business Domains**

modeling around, in microservices, **Modeled Around Business Domains**

one API entry point per business domain in API gateways, **One API Entry Point per Business Domain**

perfectly clean API separation across, rarely practical, **One API Entry Point per Business Domain**

subdomain presented across views in horizontal-split micro-frontends, **Horizontal Split**

business goals, tying migration strategy directly to, **How Can I Get the Management On Board with This Migration?**

business point of view, micro-frontend design based on, **DDD with Micro-Frontends**

business requirements, **Business Requirements**, **Architecture Is Not Yet Automatable**

drastic variations across organizations, **Architecture Is Not Yet Automatable**

ByteDance, use of Module Federation, **Module Federation**

C

caches

GraphQL server-client cache, **Using GraphQL with Micro-Frontends and a Client-Side Composition**

types to consider when building SSR micro-frontends, **Types of Caches Every Developer Should Know**

content delivery networks, **Content Delivery Networks**

in-memory database cache, **In-Memory Database Cache**

warm cache, **Warm Cache**

caching

by API gateway, **Working with an API Gateway**

in BBC website's architecture, **The BBC Architecture: Caching in Action**

cacheable and non-cacheable APIs, [Moving to Microservices](#)

content served inside iframes, [Architecture Characteristics](#)

flexibility of strategies in splitting micro-frontends by URL, [Splitting by URL](#)

highly cacheable pages, [Server-Side Composition](#)

in horizontal-split micro-frontends with server-side composition, [Scalability and Response Time](#)

in hosting of client-side rendering micro-frontends, [Caching](#)

in monolith to micro-frontends case study, [Technology Choice](#)

pivotal role in effective SSR scaling, [Scalability Challenges](#)

scaling SSR with smart caching strategies, [Don't Fear the Cache: Scaling SSR with Smart Caching Strategies](#)

in server-side composition, [Server Side](#)

when multiple micro-frontends are performing same query, [Using GraphQL with Micro-Frontends and a Client-Side Composition](#)

canary releases, [Independent Deployment](#), [Welcome to the Frontend Discovery Schema](#), [Real-World Implementation](#)

blue-green deployment versus, [Blue-Green Deployment Versus Canary Releases](#)-
[Blue-Green Deployment Versus Canary Releases](#)

implementing in monolith to micro-frontends case study, [Implementing Canary Releases](#)

cart component (check-out micro-frontend), [Developing the Check-Out Experience](#)

case studies on companies embracing specific architectures, [Architecture and Trade-Offs](#)

catalogs

Catalog micro-frontend in internal ecommerce site, [The Project](#), [Catalog Micro-Frontend](#)

catalog view in client-side implementation of horizontal-split architecture, [Client Side](#)

implementing for horizontal-split architecture video-streaming website, [Client Side](#)

CD (continuous delivery), [Automation Principles](#)

CD (continuous deployment), [Automation Principles](#)

(see also CI/CD pipelines)

CDNs (content delivery networks), [Micro-Frontend Composition](#)

aiding scalability in vertical-split micro-frontend, [Architecture Characteristics](#)

caching and scaling SSR micro-frontends, [Content Delivery Networks](#)

caching in BBC website architecture, [The BBC Architecture: Caching in Action](#)

CDN layer in server-side composition, [Composing Micro-Frontends](#)

choice between single storage with a CDN and multiple storages with a unified CDN, [Hosting a Client-Side Rendering Micro-Frontend Project](#)

CloudFront CDN in automation pipeline case study, [Deployment](#)

Edge Side Includes markup language, [Edge-Side Composition](#)

ESI revolutionizing dynamic content delivery, [Edge Side](#)

hosting of client-side rendering micro-frontends, different TTL values for CDN cache, [Caching](#)

product pages prerendered and cached by, [Scalability Challenges](#)

routing and scalability of micro-frontends and, [Routing Micro-Frontends](#)

routing requests by first-level URL, [Splitting by URL](#)

using single or multiple storages with, [Hosting a Client-Side Rendering Micro-Frontend Project](#)

using to improve response time in server-side composition, [Scalability and Response Time](#)

centralized portal for developers to retrieve blueprints, **Frictionless Micro-Frontend Blueprints**

ChatGPT, **Bouncing Ideas and Discovering Alternatives with Agentic AI**

check-out micro-frontend, developing for internal ecommerce site project, **Developing the Check-Out Experience**

Chrome DevTools, AI assistant introduced in, **Debugging with AI**

CI (continuous integration), **Automation Principles**

CI/CD (continuous integration and continuous deployment) pipelines, **Micro-Frontend Automation**

continuous integration pipeline in monolith to micro-frontends case study, **Platform and Main User Flows**

continuous integration strategies, **Continuous Integration Strategies-Continuous Integration Strategies**

using Webpack DevServer proxy for end-to-end testing during CI pipeline, **Testing Technical Recommendations**

CI/CD (continuous integration or continuous delivery) pipelines, **Monolith to Distributed Systems**

circular dependencies, **Unidirectional Sharing**

client-side composition, **Client-Side Composition**

with API gateway and service dictionary, **Client-Side Composition with an API Gateway and a Service Dictionary-Client-Side Composition with an API Gateway and a Service Dictionary**

with BFF and service dictionary, **Client-Side Composition with a BFF and a Service Dictionary-Client-Side Composition with a BFF and a Service Dictionary**

in horizontal-split micro-frontends, **Horizontal Split**

implementation of horizontal-split architecture, **Client Side-Client Side**

new view with catalog and marketing message, **Client Side**

teams developing landing page and catalog, **Client Side**

for internal ecommerce site, [The Project](#)

in monolith to micro-frontends migration, [Micro-Frontend Decision Framework Applied](#)

routing, [Routing Micro-Frontends](#)

service dictionary with micro-frontends, [Working with a Service Dictionary](#)

techniques for composing vertical-split micro-frontends, [Composing micro-frontends](#)

use of Module Federation with, [The Project](#)

using GraphQL with micro-frontends and, [Using GraphQL with Micro-Frontends and a Client-Side Composition-Using GraphQL with Micro-Frontends and a Client-Side Composition](#)

client-side includes, [Client-Side Composition](#)

client-side rendering, [Vertical-Split Architectures](#), [Client-Side Rendering Micro-Frontends-Summary](#)

example project, internal ecommerce t-shirt website, [The Project-The Project](#)

internal ecommerce t-shirt site, project evolution, [Project Evolution-Caching](#)

caching, [Caching](#)

developing check-out experience, [Developing the Check-Out Experience](#)

embedding a legacy application, [Embedding a Legacy Application](#)

hosting, [Hosting a Client-Side Rendering Micro-Frontend Project](#)

Module Federation basic concepts, [Module Federation 101](#)

technical implementation of internal ecommerce site project, [Technical Implementation-Account Management Micro-Frontend](#)

Account Management micro-frontend, [Account Management Micro-Frontend-Account Management Micro-Frontend](#)

application shell, [Application Shell-Application Shell](#)

Catalog micro-frontend, [Catalog Micro-Frontend](#)

Home micro-frontend, [Home Micro-Frontend](#)-[Home Micro-Frontend](#)
project structure, [Project Structure](#)

using HTML fragments for composition, [HTML Fragments](#)

using OpenFrameworks for, [Available Frameworks](#)

client-side routing, [The Project](#)

use with vertical-split micro-frontends, [Vertical Split](#)

cloud

custom cloud automation pipeline with Docker containers, [Setting the Scene](#)

infrastructure flexibility provided by, [Server Side](#)

providers offering spot instances, [Environments Strategies](#)

public cloud providers, monoliths and, [Micro-Frontend Principles](#)

CloudFlare, [Web Fragments](#), [Web Fragments](#)

CloudFront, [Hosting a Client-Side Rendering Micro-Frontend Project](#), [Deployment](#)

CLS (cumulative layout shift), [Server-Side Rendering Micro-Frontends](#)

code abstraction (see abstraction of code)

code assistants and AI, where they fall short, [Where Code Assistants and AI Fall Short](#)-[Understanding Code Change Operations](#)

code change operations tracked in study of code assistants and AI, [Understanding Code Change Operations](#)

code coverage, [Fitness Functions](#)

code duplication, [Architecture Evolution](#)

arising from use of polyrepos, [Polyrepo](#)

BFF pattern and, [Working with the BFF Pattern](#)

favoring over incorrect abstractions, [Architecture Evolution](#)

code encapsulation, [Architecture Evolution](#)

code optimization, using AI in, [Code Optimization](#)

code refactoring

large-scale, monorepos facilitating, [Monorepo](#)

code reusability (see [reusability](#))

code sharing

strength of monorepos for, [Monorepo](#)

code volatility, [Architecture Is Not Yet Automatable](#)

code-quality review, [Code-Quality Review](#)-[Code-Quality Review](#)

codebase

cohesive, with less technical debt in monorepos, [Monorepo](#)

embedding context in, [Strategy: Embed context in the codebase](#)

scaling tools when codebase increases in monorepos, [Monorepo](#)

cognitive load

reducing by splitting micro-frontend without impacting user experience, [High-Complexity Subdomain](#)

reducing in monolith to micro-frontends migration, [Micro-Frontend Decision Framework Applied](#)

collaboration across teams

monorepos contributing to, [Monorepo](#)

command-line interfaces (CLIs)

creating command-line tool for scaffolding micro-frontends, [Frictionless Micro-Frontend Blueprints](#)

command-line interfaces (CLIs)

CLI tools in performance checks in automation pipelines, [Post-Build Review](#)

companies implementing tools to scaffold micro-frontends, [Developer's Experience](#)

tools for improving development teams' communication, [Developer Experience](#) communication

implementing governance to ease communication flows, [Implementing Governance for Easing the Communication Flows](#)-[Architecture Decision Records](#)

link between architecture and team communication, [How Do Committees Invent?](#)

team communication and maintaining control of final outcome, [Team Communication and Best Practices for Maintaining Control of the Final Outcome](#)

techniques for enhancing flow of, [Techniques for Enhancing the Communication Flow](#)-[Managing External Dependencies](#)

community of practice and town halls, [Community of Practice and Town Halls](#)

managing external dependencies, [Managing External Dependencies](#)-[Managing External Dependencies](#)

working backward, [Working Backward](#)

communication between micro-frontends, [Micro-Frontend Decision Framework](#), [Micro-Frontend Communication](#)-[Micro-Frontends in Practice](#)

excessive, indicating improperly defined boundaries in the view, [Sharing State Between Micro-Frontends](#)

in horizontal-split architecture with server-side composition, [Micro-Frontend Communication](#)

in horizontal-split architectures, [Micro-frontend communication](#)-[Micro-frontend communication](#)

in horizontal-split micro-frontends, [Horizontal Split iframes](#) and, [Use Cases](#)

in internal ecommerce site project, [The Project](#)

in monolith to micro-frontends migration, [Micro-Frontend Decision Framework Applied](#)

via query strings or URLs, [Micro-Frontend Communication](#)

in the same view, [Sharing State Between Micro-Frontends](#)

shared event emitter allowing communication across domain boundaries and maintaining their independence, [Account Management Micro-Frontend](#)

sharing data in different views, [Micro-Frontend Communication](#)

summary of in micro-frontend decision framework, [Micro-Frontend Communication](#)

communication bridge in monolith to micro-frontends case study, [Communication Bridge](#)

community of practice and town halls, [Community of Practice and Town Halls](#)

complexity

decentralizing organizations that handle, [A Decentralized Organization](#)

high-complexity subdomain, [High-Complexity Subdomain](#)

increased, with micro-frontends, [Trade-Offs and Pitfalls](#)

low-complexity subdomain, [Low-Complexity Subdomain](#)

modern AI models reasoning about, [Code Optimization](#)

normal-complexity subdomain, [Normal-Complexity Subdomain](#)

components

distinguishing between micro-frontends and components, [Micro-Frontend or Component?-Micro-Frontend or Component?](#)

features versus components teams, [Features Versus Components Teams-Features Versus Components Teams](#)

managing shared components in multi-zone micro-frontends, [How to Handle Shared Components](#)

versus micro-frontends, [Modeled Around Business Domains, Testing Your Micro-Frontend Boundaries](#)

sharing in monolith to micro-frontends case study, [Sharing Components](#)

subdomains in DDD, [Defining Micro-Frontends](#)

transforming multiple micro-frontends consuming same API into components,
Multiple Micro-Frontends Consuming the Same API

in vertical-split micro-frontend design system, **Implementing a Design System**

composer layer in server-side composition, **Composing Micro-Frontends**

composition layer, **The Formula 1 Website is Powered by Micro-Frontends**

infrastructure ownership in, **Infrastructure Ownership**

UI, Server-Side Composition with an API Gateway, Server-Side Composition with a BFF and a Service Dictionary, Using GraphQL with Micro-Frontends and a Server-Side Composition

composition of micro-frontends, **Testing Your Micro-Frontend Boundaries-Server-Side Composition**

approaches to composing SSR micro-frontends, **Composition Approaches**

choosing composition strategy for SSR micro-frontends, **The Main Challenge**

choosing in horizontal-split micro-frontends, **Horizontal Split**

client-side, **Client-Side Composition**

with application shell loading/unloading one micro-frontend at a time, **Micro-Frontend Decision Framework Applied**

composing vertical-split micro-frontends in application shell, **Composing micro-frontends**

edge-side, **Edge-Side Composition**

server-side, **Server-Side Composition**

summary of in micro-frontend decision framework, **Micro-Frontend Communication**

using Module Federation for micro-frontend architecture, **Composition**

compute scalability, handling, **Scalability Challenges**

containers

custom cloud automation pipeline with Docker containers, **Setting the Scene**

discouraging use to serve static files, [Hosting a Client-Side Rendering Micro-Frontend Project](#)

micro-frontend, reducing API surface exposed to, [Testing Your Micro-Frontend Boundaries](#)

micro-frontends hosted in within secure networks, [Hosting a Client-Side Rendering Micro-Frontend Project](#)

Web Fragments as, [Web Fragments](#)

content delivery networks (see CDNs)

contentWindow object (iframes), [Best Practices and Drawbacks](#)

context APIs (React), [Implementing a Service Dictionary in a Horizontal-Split Architecture](#)

context awareness of micro-frontends, [Testing Your Micro-Frontend Boundaries](#)

context engineering, [Context Engineering to the Rescue-Context Engineering to the Rescue](#)

using with AI assistants, not just prompting, [Step 2: Engineer the Context; Don't Just Prompt-Key takeaways](#)

contexts, [Architecture Is Not Yet Automatable](#)

link between organizations and software architecture, [The Link Between Organizations and Software Architecture](#)

optimizing for context in architecture decisions, [Architecture and Trade-Offs](#)

organization capabilities, understanding in introducing micro-frontends, [Organization Capabilities](#)

researching and describing context you operate in, [Why Should We Use Micro-Frontends?](#)

continuous delivery (CD), [Automation Principles](#)

(see also CI/CD pipelines)

continuous deployment (CD), [Automation Principles](#)

(see also CI/CD pipelines)

contracts

analyzing API contracts with different teams, [APIs Come First, Then Implementation](#)

API contracts in migration to micro-frontends, [Trade-Offs and Pitfalls](#)

polyrepos encouraging thinking about, [Polyrepo](#)

Conway's law, [How Do Committees Invent?](#)

cookies

JWTs set as cookies in browser, security settings, [How to Manage Data](#)

using to store authentication tokens, [Authentication](#)

coordination

analysis for architectures, [Architecture Analysis](#)

in horizontal-split micro-frontends, [Architecture Characteristics](#)

of horizontal-split architecture with server-side composition, [Architecture Characteristics](#)

micro-frontends composed using iframes, [Architecture Characteristics](#)

micro-frontends implemented as web components, [Architecture Characteristics](#)

in SSR architecture, [Architecture Characteristics](#)

syncing work between teams, [Team Communication and Best Practices for Maintaining Control of the Final Outcome](#)

in vertical-split micro-frontends, [Architecture Characteristics](#)

core subdomains, [Defining Micro-Frontends](#)

costs

of finding and fixing defects, rising over time, [Setting the Scene](#)

coupling

components tightly coupled through bidirectional dependencies, [Unidirectional Sharing](#)

of projects, risks in monorepos, [Monorepo](#)

reusability representing form of, [Managing External Dependencies](#)

sharing between micro-frontends in same view in horizontal splits, [Sharing State Between Micro-Frontends](#)

tight coupling to legacy code in reuse of monolith components, [Trade-Offs and Pitfalls](#)

CPU

strict allocation limits for edge computing, [Edge Side](#)

cross-boundary APIs, [Working with the BFF Pattern](#)

cross-cutting concerns such as authentication, routing, and state, managing during migration, [How Do I Manage Cross-Cutting Concerns Like Auth, Routing, or State During Migration?-Personal Experience](#)

best practices and patterns, [Best Practices and Patterns](#)

checklist of questions to ask yourself, [Checklist](#)

personal experience of author, [Personal Experience](#)

quick example, [A Quick Example](#)

trade-offs and pitfalls to consider, [Trade-Offs and Pitfalls](#)

cross-functional teams, [Working with a Service Dictionary](#)

cross-origin resource sharing (CORS) authentication, [Composing micro-frontends](#)

cross-platform applications requiring different user experiences per device used, [Working with the BFF Pattern](#)

cross-site scripting (XSS), [Integrating Authentication in Micro-Frontends](#)

CSS

clashes with class names, avoiding in horizontal-split architectures, [Clashes with CSS class names and how to avoid them](#)

culture

close relationship with organizational structure and software architecture,
Common Antipatterns in Micro-Frontend Implementations

impacts on architecture, **Architecture Is Not Yet Automatable**

of sharing and innovation, encouraging, **Empower Your Teams**

cumulative layout shift (CLS), **Server-Side Rendering Micro-Frontends**

Cursor, **Bouncing Ideas and Discovering Alternatives with Agentic AI**

custom elements in web components, **Web Components Technologies**

custom events, **Micro-Frontend Communication**

cyclomatic complexity (CYC), **Fitness Functions, Code-Quality Review**

scores on, **Code-Quality Review**

D

dashboards

performance considerations impacting architecture, **Architecture Is Not Yet Automatable**

using Web Fragments to modernize incrementally, **Web Fragments**

data

how micro-frontends will share data, **Micro-Frontend Decision Framework**

managing in multi-zone micro-frontend architecture, **How to Manage Data**

using in defining bounded contexts, **How to Define a Bounded Context**

data gravity, **Edge Side**

databases, **Monolith to Distributed Systems**

database configuration store in discovery mechanism, **Real-World Implementation**

database in monolith to micro-frontends case study, **Technology Stack**

in move from monolith to microservices, **Moving to Microservices**

shared between microservices, disadvantages of, **Hidden Implementation Details**
DAZN, use of micro-frontends, **DAZN**

DDD (domain-driven design), **Modeled Around Business Domains**, API Integration and
Micro-Frontends

applied to frontends, **Modeled Around Business Domains**

applying to micro-frontend vertical split architecture, **Defining Micro-Frontends**

helpful when organizing team structure, **Features Versus Components Teams**

hiding implementation details in, **Hidden Implementation Details**

separating different DDD subdomains, **Working with the BFF Pattern**

sociotechnical aspect of software architecture, **A Decentralized Organization**

subdomains, types of, **Defining Micro-Frontends**

using with micro-frontends, **DDD with Micro-Frontends-DDD with Micro-Frontends**

debugging, **Developer Experience, Trade-Offs and Pitfalls**

easier when code origin is known, **Shared code**

reliance on browser storage for key data, creating hard-to-debug issues, **Trade-Offs and Pitfalls**

using AI, **Debugging with AI**

decentralization

decentralized organization, **A Decentralized Organization-A Decentralized Organization**

implications with micro-frontends, **Decentralization Implications with Micro-Frontends-Low-Complexity Subdomain**

high initial-effort subdomains, **High Initial-Effort Subdomain**

high-complexity subdomains, **High-Complexity Subdomain**

low-complexity subdomains, **Low-Complexity Subdomain**

normal-complexity subdomains, **Normal-Complexity Subdomain**
in micro-frontends, **Decentralization**
in microservices, **Decentralization**
organizing for complexity, **A Decentralized Organization**
decision framework (see micro-frontend decision framework)
decoupling
goal of decoupling micro-frontends in monolith to micro-frontends case study,
Technical Goals
in service discovery, **Welcome to the Frontend Discovery Schema**
defects, costs of, rising over time, **Setting the Scene**
DefinePlugin, Module Federation 101
defining bounded contexts, **How to Define a Bounded Context**
defining micro-frontends, **Defining Micro-Frontends-Defining Micro-Frontends**
defining what you want to achieve (playbook for AI), **Step 1: Define What You Want to Achieve-Key takeaways**
delegation to backend API
for serving to vertical-split micro-frontends, **Architecture Evolution**
Dependabot
review of zones using shared components and automatic updates of dependencies,
How to Handle Shared Components
use in micro-frontend automation pipeline case study to manage shared libraries,
Version Control
dependencies
circular, **Unidirectional Sharing**
clashes when using multi-framework approach in horizontal splits, **Multi-framework approach**

configuring dependency updates with Dependabot, using AI in, [Automation](#)
dependency management in monolith to micro-frontends case study, [Dependency Management](#)

dependency management with web components, [Dependency Management](#)
external dependencies impeding delivery of a task or story, [Managing External Dependencies](#)-[Managing External Dependencies](#)

hoisting across packages, [Monorepo](#)

loading asynchronously using Module Federation, [Application Shell](#)

managing in monorepo version control, [Version Control](#)

micro-frontends reducing external dependencies between teams, [Home Micro-Frontend](#)

no risk of clashes in monolith to micro-frontends migration, [Micro-Frontend Decision Framework Applied](#)

shared dependencies between legacy and micro-frontend code, handling in migration, [How Do I Handle Shared Dependencies and Version Management Between Legacy and Micro-Frontend Code?-Personal Experience](#)

best practices and patterns, [Best Practices and Patterns](#)

checklist of questions to ask, [Checklist](#)

personal experience of author, [Personal Experience](#)

quick example, [A Quick Example](#)

trade-offs and pitfalls to consider, [Trade-Offs and Pitfalls](#)

shared dependencies in micro-frontends split by URL, [Splitting by URL](#)

shared, inspecting through browser's developer tools using Module Federation 2.0, [Home Micro-Frontend](#)

shared, pitfalls of with micro-frontends, [Trade-Offs and Pitfalls](#)

sharing across micro-frontends, automation of, [Implementing a Design System](#)

sharing across micro-frontends, help from Module Federation, [Performance](#)

simplified management in monorepos, [Monorepo](#)
tangled web of, with bidirectional sharing, [Unidirectional Sharing](#)
Web Fragments support of dependency sharing at network level, [Web Fragments](#)
deployability
analysis for architectures, [Architecture Analysis](#)
in horizontal-split micro-frontends, [Architecture Characteristics](#)
horizontal-split micro-frontends with server-side composition, [Architecture Characteristics](#)
micro-frontends composed using iframes, [Architecture Characteristics](#)
micro-frontends implemented as web components, [Architecture Characteristics](#)
of SSR architecture, [Architecture Characteristics](#)
in vertical-split micro-frontends, [Architecture Characteristics](#)
deployments
blue-green deployment versus canary releases, [Blue-Green Deployment Versus Canary Releases](#)-[Blue-Green Deployment Versus Canary Releases](#)
deploying micro-frontends, [Discover and Deploy Micro-Frontends](#), [Deployment](#)
disadvantages of monoliths in, [Micro-Frontend Principles](#)
ecommerce platform with micro-frontends, deployment to different environments, [The Problem Space](#)
flexible, in micro-frontends composed by splitting URL, [Splitting by URL](#)
impacts of bidirectional sharing on, [Unidirectional Sharing](#)
independent deployment of micro-frontends, [Independent Deployment](#)
independent deployment of microservices, [Independent Deployment](#)
in monolith to micro-frontends case study, [Platform and Main User Flows](#)
ten-times increase in after moving to micro-frontends, [Personal Experience](#)

design, [The Link Between Organizations and Software Architecture](#)

AI as design reviewer, Example: AI as design reviewer

system, structure as copy of the organization's communication structure, [How Do Committees Invent?](#)

design systems

implementing for vertical-split micro-frontends, [Implementing a Design System](#)-
[Implementing a Design System](#)

integrating design system in monolith to micro-frontends case study, [Integrating a Design System](#)

integrating into micro-frontend implementations, [Architecture Characteristics](#)

investing in robust design system, [Best Practices and Patterns](#)

overengineering, [Trade-Offs and Pitfalls](#)

design tokens, [Implementing a Design System](#)

developer experience, [Developer Experience-Environments Strategies](#)

analysis for architectures, [Architecture Analysis](#)

decision between horizontal and vertical split, impacts of, [Horizontal Versus Vertical Split](#)

defined, [Developer Experience](#)

environments strategies, [Environments Strategies](#)

frictionless micro-frontend blueprints, [Frictionless Micro-Frontend Blueprints](#)

goal of maintaining seamless experience in monolith to micro-frontends case study, [Technical Goals](#)

in horizontal-split architecture with server-side composition, [Architecture Characteristics](#)

in horizontal-split architectures, [Developer Experience](#)

in horizontal-split micro-frontends, [Architecture Characteristics](#)

with iframes, [Architecture Characteristics](#)

improvements for Formula 1 website, with use of first-level URL split micro-frontends, [The Formula 1 Website is Powered by Micro-Frontends](#)

micro-frontends implemented as web components, [Architecture Characteristics](#) in Module Federation applications, [Module Federation](#), Developer Experience seamless, using Webpack with Module Federation, [The Project](#)

in SSR architecture, [Architecture Characteristics](#)

using iframes to compose micro-frontends, [Developer Experience](#)

in vertical-split micro-frontends, [Developer's Experience](#), [Architecture Characteristics](#)

developers

AI augmenting outcomes of, not replacing, [AI and Micro-Frontends: Augmenting, Not Replacing](#)

CI strategies and, [Continuous Integration Strategies](#)

disciplined, necessity when using monorepo, [Monorepo](#)

development environment, [Blue-Green Deployment Versus Canary Releases](#), [Setting the Scene](#)

device groups, BFF creating unique entry point for, [Working with the BFF Pattern](#)

DevOps, about, [Continuous Integration Strategies](#)

diagram generation, using AI in, [Diagram Generation](#)

diffs, AI understanding of, [Keeping the README Alive](#)

discoverability, [Welcome to the Frontend Discovery Schema](#)

(see also service discovery)

challenge in micro-frontends, [Welcome to the Frontend Discovery Schema](#)

Frontend Discovery schema, [Welcome to the Frontend Discovery Schema-Real-World Implementation](#)

benefits of, [The Benefits of This Pattern](#)

in micro-frontend automation pipeline case study, [Deployment](#)
real-world implementation, [Real-World Implementation](#)

integrating discovery pattern with other micro-frontend solutions, [Integrating the Discovery Pattern with Other Solutions](#)

integration of discovery mechanism with Module Federation ecommerce example,
[An Integration Example with Module Federation-An Integration Example with Module Federation](#)

micro-frontend discovery pattern, using with feature flags, [What about Feature Flags?](#)

micro-frontend solutions implementing discovery pattern, market availability,
[Availability in the Market](#)

use of feature flags instead of discovery pattern, [What about Feature Flags?](#)

disintegrators versus integrators, [Architecture Is Not Yet Automatable](#)

distributed monoliths, [Defining Micro-Frontends](#)

Docker

custom cloud automation pipeline with Docker containers, [Setting the Scene](#)

Document Object Model (DOM), [Client-Side Composition](#)

custom events traversing and propagating to window object, [Micro-Frontend Communication](#)

event emitters as DOM-agnostic, [Sharing State Between Micro-Frontends](#)

parsing DOM elements and appending nodes needed in application shell's DOM,
[Composing micro-frontends](#)

shadow DOM in web components, [Web Components Technologies](#)

shadow DOM, web components hiding frameworks for micro-frontend behind,
[Multi-framework approach](#)

web components with shadow DOM, encapsulating SSR micro-frontends as, [Composition Approaches](#)

Web Fragments and, [Web Fragments](#)

domain boundaries

reviewing for micro-frontends, [Authentication](#)

domain-driven design (see DDD)

domains

aggregating APIs by domain, in combination of BFF and micro-frontends, [Working with the BFF Pattern](#)

application shell remaining domain-agnostic as possible, [Application Shell](#)

difficulty of identifying domain boundaries for completely self-sufficient APIs, [Working with the BFF Pattern](#)

evolving autonomously with micro-frontends responsible for local routing, [Catalog Micro-Frontend](#)

DOMParser object, [Composing micro-frontends](#)

DRY principle (Don't Repeat Yourself)

misapplication by developers, [Architecture Evolution](#)

Dunelm, use of micro-frontends, [Dunelm, Data to the Rescue](#)

duplication of code, [Architecture Evolution](#)

favoring over incorrect abstractions, [Architecture Evolution](#)

DX (see developer experience)

dynamic rendering

in client-side horizontal-split architectures, [Search Engine Optimization](#)

E

ecommerce platforms

benefits of SSR for, [When to Use Server-Side Rendering](#)

home page seeking to move from monolith to micro-frontends, [Micro-Frontend or Component?](#)

edge functions

performed by API gateways, [Working with an API Gateway](#)

Edge Side Includes (ESI), [Edge-Side Composition](#), [Horizontal Split](#), Edge Side edge-side composition, [Edge-Side Composition](#), Edge Side

in horizontal-split micro-frontends, [Horizontal Split](#) routing, [Routing Micro-Frontends](#)

elements, custom, in web components, [Web Components Technologies](#)

encapsulation of code, [Architecture Evolution](#)

end-to-end testing, [End-to-End Testing](#)

challenges with horizontal-split architecture, [Horizontal-Split End-to-End Testing Challenges](#)

challenges with vertical-split architecture, [Vertical-Split End-to-End Testing Challenges](#)

in micro-frontend automation pipeline case study, post-build review, [Post-Build Review](#)

technically implementing, recommendations for, [Testing Technical Recommendations](#)

tests in micro-frontend architecture needing to be selective and fast, [Define Your Test Strategy](#)

endpoints

service dictionary list of endpoints available in API layer of micro-frontend, [Working with a Service Dictionary](#)

enterprise applications

use of horizontal-split architectures, [Use Cases](#)

entry points for API layer, [Working with an API Gateway](#)

(see also [API gateway](#))

choosing different entry points, [One API Entry Point per Business Domain](#)

one entry point per business domain, [One API Entry Point per Business Domain](#)

environments

deploying micro-frontends from ecommerce platform to different environments,

[The Problem Space](#)

end-to-end testing in, [End-to-End Testing](#)

environments strategies

in monolith to micro-frontends case study, [Platform and Main User Flows](#)

micro-frontend deployments and, [Blue-Green Deployment Versus Canary Releases](#)

in micro-frontend automation pipeline case study, [Setting the Scene](#)

strategies for building micro-frontends in, [Environments Strategies](#)

error handling

for dynamic routes, [Application Shell](#)

error reporting by micro-frontends, [Observability](#)

in HTML fragments implementation of micro-frontends, [HTML Fragments](#)

ES modules, [Composing micro-frontends](#)

ESI (Edge Side Includes), [Edge-Side Composition](#), [Horizontal Split](#), Edge Side

event bus, [Micro-Frontend Communication](#)

event storming, [Defining Micro-Frontends](#)

events

designing for event emitter in a micro-frontend architecture, [Account Management Micro-Frontend](#)

each micro-frontend emitting or listening to event without relying on specific position inside the DOM, [Account Management Micro-Frontend](#)

event emitter handling communication between micro-frontends in internal ecommerce site project, [The Project](#)

event emitters or custom events, using for communication between micro-frontends, [Sharing State Between Micro-Frontends](#)

event emitters or events, using for communication in horizontal-split micro-frontends, [Horizontal Split](#)

notifying micro-frontends of, [Micro-Frontend Communication](#)

received and triggered, documenting for micro-frontends, [Team Communication and Best Practices for Maintaining Control of the Final Outcome](#)

use of event emitters and custom events, [Account Management Micro-Frontend](#)

using event emitter for communication between iframes and host page, [Best Practices and Drawbacks](#)

using event emitter or custom events to communicate in horizontal-split micro-frontends, [Micro-frontend communication](#)

“Everything fails, all the time.” (Vogels), [Highly Observable](#)

exploration, AI lowering barriers to, [Prototyping](#)

bouncing ideas and discovering alternatives with agentic AI, [Bouncing Ideas and Discovering Alternatives with Agentic AI](#)

discovering alternatives and challenging assumptions with AI, [Discovering Alternatives and Challenging Assumptions with AI](#)

F

Facebook, automation pipelines scaling with codebase, [Monorepo](#)

failures

isolation of in micro-frontends, [Isolated Failures](#)

isolation of in microservices, [Isolated Failures](#)

resolving quickly more important than preventing, [Highly Observable FAQs](#) (frequently asked questions), [Working Backward](#)

fat servers, [Introducing Micro-Frontends](#)

fate-sharing, elimination using Web Fragments, [Web Fragments](#)
feature flags

hybrid approach combining micro-frontend service discovery with, [What about Feature Flags?](#)

micro-frontend discovery pattern and, [What about Feature Flags?](#)

features

deciding which to migrate first in move to micro-frontends, [How Do I Decide Which Modules or Features to Migrate First?-Personal Experience](#)

features versus components teams, [Features Versus Components Teams-Features Versus Components Teams](#)

new feature pipelines, [Architecture Is Not Yet Automatable](#)
feedback loop

keeping it fast in automation pipelines, [Keep the Feedback Loop Fast](#)

Fetch API, [Application Shell](#)

fetch function, overriding, [Application Shell](#)

first-level URLs, [Vertical Split](#)

fitness functions

implementing using AI, [Automation](#)

using to test architecture's characteristics in automation pipeline, [Code-Quality Review](#)

using to test architecture's characteristics in CI, [Fitness Functions-Fitness Functions](#)

Formula 1 website, use of micro-frontends, [Formula 1](#), [The Formula 1 Website is Powered by Micro-Frontends](#)

Fowler, Martin

explanation of micro-frontends on website, [Foreword for the 1st Edition](#)

frameworks, [Multi-framework approach](#)

(see also multi-framework approach)

OpenComponents framework for server-side composition, [Available Frameworks for server-side composition](#), [Composing Micro-Frontends](#)

server-side rendering frameworks, modern, [Modern Server-Side Rendering Frameworks](#)

use cases, [Use Cases](#)

for vertical-split architecture, [Available Frameworks](#)

frequently asked questions (FAQs), [Working Backward](#)

freshness of data, [Scalability Challenges](#)

frictionless micro-frontend blueprints, [Frictionless Micro-Frontend Blueprints](#)

Frontend Discovery schema, [Welcome to the Frontend Discovery Schema](#)-[Real-World Implementation](#), [Deployment](#)

benefits of, [The Benefits of This Pattern](#)

key features of, [Welcome to the Frontend Discovery Schema](#)

real-world implementation, [Real-World Implementation](#)

frontend layer of new projects, [Monolith to Distributed Systems](#)

frontends

independence from backends, [Best Practices and Patterns](#)

micro-frontends, [Micro-Frontend Principles](#)

migrating to micro-frontends, [Best Practices and Patterns](#)

monolith approach on, disadvantages of, [Introducing Micro-Frontends](#)
revolution in architecture, unidirectional data flow, [Shared code](#)
scaling in microservice applications, [Moving to Microservices](#)

G

General Data Protection Regulation (GDPR), [Architecture Is Not Yet Automatable](#)
generic subdomains, [Defining Micro-Frontends](#)

Git, [Version Control](#)

improvements in operation times for commands, [Monorepo](#)

GitHub, [Version Control](#)

global routing, [Vertical Split](#), [Global and Local Routing](#)

application shell responsible for, [Application Shell](#)

Google

API guidelines, [API Consistency](#)

automation pipelines scaling alongside codebase, [Monorepo](#)

use of monorepos, [Monorepo](#)

Google Cloud, [Micro-Frontend Principles](#)

governance, [Implementing a Design System](#), [Architecture Characteristics](#), [Polyrepo](#)

decentralizing, [Decentralization](#)

of horizontal-split micro-frontends, [Defining Micro-Frontends](#), [Horizontal-Split Architecture](#), [Horizontal Versus Vertical Split](#)

implementing to ease communication flows, [Implementing Governance for Easing the Communication Flows](#)-[Architecture Decision Records](#)

architecture decision records, [Architecture Decision Records](#)

requests for comments, [Requests for Comments](#)

overseeing testing process, [End-to-End Testing](#)

in polyrepo environments, [Polyrepo](#), [Polyrepo](#)
of repositories for shared libraries and dependencies, [Application Shell](#)
of shared code updates, [Architecture Evolution](#)
of shared UI components, [Implementing a Design System](#)

using single storage with a CDN, [Hosting a Client-Side Rendering Micro-Frontend Project](#)

UX consistency and, [Challenges Unique to Micro-Frontends](#)

GPU-accelerated rendering and local compute workflows, [What the Future Holds for AI and Micro-Frontends](#)

GraphQL, [API Strategies](#), [Using GraphQL with Micro-Frontends](#)-[Using GraphQL with Micro-Frontends and a Server-Side Composition](#)

acting as unique entry point for API layer, [Using GraphQL with Micro-Frontends](#)

designing API schema with, [Using GraphQL with Micro-Frontends](#)

schema federation feature, [Using GraphQL with Micro-Frontends](#)

using with micro-frontends and client-side composition, [Using GraphQL with Micro-Frontends and a Client-Side Composition](#)-[Using GraphQL with Micro-Frontends and a Client-Side Composition](#)

using with micro-frontends and server-side composition, [Using GraphQL with Micro-Frontends and a Server-Side Composition](#)

gRPC and HTTP in microservice, [Working with an API Gateway](#)

guardrails for automation strategy, [Empower Your Teams](#)

defining, [Define Your Guardrails](#)

H

hard navigations, [Next.js Multi-Zones](#)

between legacy system and micro-frontends, leveraging absolute URLs, [Best Practices and Patterns](#)

high initial-effort subdomains, [High Initial-Effort Subdomain](#)

high-complexity subdomains, [High-Complexity Subdomain](#)

hoisting dependencies across packages, [Monorepo](#)

home page micro-frontend

loading in application shell in internal ecommerce site project, [Home Micro-Frontend](#)

home zone

acting as proxy for other zones, [Home Zone: The Entry Point of Your Web Application](#)

in Next.js t-shirt shop project, [Home Zone: The Entry Point of Your Web Application](#)

hooks

AI generating React hook to refresh JWTs, [Example scenario](#)

custom React hook, `useMfeInitialization`, [An Integration Example with Module Federation](#), [An Integration Example with Module Federation](#)

provided by version control system, using to generate final micro-frontend artifact, [Performance: The Key Reason for Server-Side Rendering](#)

setting up Git hooks using AI, [Automation](#)

horizontal split, [Micro-Frontend Automation](#)

benefits and downsides of, [Defining Micro-Frontends](#)

choosing over vertical split, [How to Define a Bounded Context](#)

deciding between vertical split and, impacts on developer experience, [Horizontal Versus Vertical Split](#)

horizontal-split architectures, [Horizontal Split](#), [Horizontal-Split Architecture-iframes](#)

analysis of architecture characteristics, [Architecture Characteristics-iframes](#)

analysis of characteristics

summary of, [Architecture Characteristics](#)

challenges in, [Challenges-Authentication](#)

authentication, [Authentication](#)

clashes with CSS class names, [Clashes with CSS class names and how to avoid them](#)

micro-frontend communication, [Micro-frontend communication-Micro-frontend communication](#)

multi-framework approach, [Multi-framework approach](#)

client-side implementation of, [Client Side-Client Side](#)

coarse-grained, in SSR micro-frontends, [Dividing Micro-Frontends](#)

composed on client side, BFF pattern and, [Client-Side Composition with a BFF and a Service Dictionary](#)

deploying micro-frontends from ecommerce platform to different environments, [The Problem Space](#)

developer experience, [Developer Experience](#)

end-to-end testing challenges, [Horizontal-Split End-to-End Testing Challenges](#)

frameworks available for, [Available Frameworks-Shared code](#)

implementing a service dictionary in, [Implementing a Service Dictionary in a Horizontal-Split Architecture](#)

investing time reviewing communication flows, [Managing External Dependencies](#)

micro-frontend refactoring, [Micro-Frontend Refactoring](#)

micro-frontends in My account section of ecommerce site project, [The Project](#)

reduction of number of micro-frontends in a view, [Horizontal-Split Architecture](#)

scenarios for use of, [Horizontal-Split Architecture](#)

search engine optimization in, [Search Engine Optimization](#)

server-side composition, [Server Side-Modern Server-Side Rendering Frameworks](#)

sharing between micro-frontends in same view, [Sharing State Between Micro-Frontends](#)

team communication and best practices to control final outcome, [Team Communication and Best Practices for Maintaining Control of the Final Outcome](#)

technically implementing end-to-end testing, recommendations for, [Testing Technical Recommendations](#)

use cases, [Use Cases](#)

using iframes, [iframes-Web Components](#)

using with vertical-split in internal ecommerce site, [The Project](#)

WebSocket connection to micro-frontend, [WebSocket and Micro-Frontends](#)

host (Module Federation), [Module Federation](#), [Module Federation 101](#)

creating MyAccount host, [Account Management Micro-Frontend](#)

My account host, loading user details and payment methods micro-frontends, [Account Management Micro-Frontend](#)

nesting multiple hosts, [Account Management Micro-Frontend](#)

hosting client-side rendering micro-frontend project, [Hosting a Client-Side Rendering Micro-Frontend Project](#)

“How Do Committees Invent?” paper (Conway), [How Do Committees Invent?](#)

HTML

entry point file for micro-frontends, [Client-Side Composition](#), [Vertical Split](#)

entry point for micro-frontends using iframes, [Developer Experience](#)

entry point page in vertical-split micro-frontends, [Application Shell](#)

extending to handle dynamic updates using htmx, [Use Cases](#)

page associated with each view in edge-side composition, [Horizontal Split](#)

parsing in vertical-split micro-frontends, [Composing micro-frontends](#)
vertical-split micro-frontends represented as single HTML pages, [Horizontal Versus Vertical Split](#)

HTML fragments

composition approach for SSR micro-frontends, exploring, [HTML Fragments - HTML Fragments](#)

using to compose SSR micro-frontends, [Composition Approaches](#)

HTML templates in web components, [Web Components Technologies](#)

htmx framework, [Modern Server-Side Rendering Frameworks](#)

use cases for, [Use Cases](#)

HTTP

Fetch API as base for calls, [Application Shell](#)

microservice architecture composed with gRPC and, [Working with an API Gateway](#)

HttpOnly cookies, [How to Manage Data](#)

I
ideas, bouncing, and discovering alternatives, using agentic AI, [Bouncing Ideas and Discovering Alternatives with Agentic AI](#)

identifying micro-frontends, [Application Shell](#)

hybrid approach to, [The Project](#)

iframes, [iframes-Web Components](#)

communicating with host page using postMessage method, [iframes](#)

composing micro-frontends in a horizontal split with, best practices and drawbacks, [Best Practices and Drawbacks](#)

legacy app wrapped in, embedding in client-side micro-frontend, [Embedding a Legacy Application](#)

legacy system isolated in, communication with its wrapper via postmessage API,
Anti-Corruption Layer to the Rescue

sandbox attribute, **iframes**

sandbox to prevent clashes in frameworks loaded to, **Multi-framework approach**

use to load micro-frontends, **Client-Side Composition**

using to compose micro-frontends

analysis of architecture characteristics, **Architecture Characteristics**

developer experience, **Developer Experience**

use cases, **Use Cases**

window object in, **Micro-Frontend Communication**

implementation

hidden implementation details in microservices, **Hidden Implementation Details**

hiding details in micro-frontends, **Hidden Implementation Details**

import maps, **Multi-framework approach**, Home Micro-Frontend, **Integrating the Discovery Pattern with Other Solutions**

`generateImportsMap` function, **Integrating the Discovery Pattern with Other Solutions**

in-memory caches

in-memory database cache for SSR micro-frontends, **In-Memory Database Cache**

in SSR architecture, **Scalability Challenges**

includes, **Routing Micro-Frontends**

(see also Edge Side Includes)

including libraries in each micro-frontend, **Application Shell**

Incremental Static Generation (ISG), RSC, **Vercel: A Glance into the Future**

infrastructure

ownership in horizontal-split micro-frontends with server-side composition,
Infrastructure Ownership

setting up correctly for application's traffic flow, **Scalability and Response Time**

initializeMFEs function, **Application Shell**

integration, **Micro-Frontend Automation**

(see also CI/CD pipelines)

complex, with legacy and new micro-frontends coexisting, **Trade-Offs and Pitfalls**

complexity of, in deploying micro-frontends to different environments, **The Problem Space**

Frontend Discovery pattern with existing APIs, **The Benefits of This Pattern**

integration testing, **Code-Quality Review**

integration tests, **Define Your Test Strategy**

integrators versus disintegrators, **Architecture Is Not Yet Automatable**

intention, clarity of, **Context Engineering to the Rescue**

Interface Framework, **Zalando**

internal ecommerce t-shirt website project, **The Project-The Project**

project evolution, **Project Evolution-Caching**

caching of CDN, **Caching**

developing check-out experience, **Developing the Check-Out Experience**

hosting client-side rendering micro-frontend project, **Hosting a Client-Side Rendering Micro-Frontend Project**

project structure, **Project Structure**

technical implementation, **Technical Implementation-Account Management Micro-Frontend**

Account Management micro-frontend, **Account Management Micro-Frontend-Account Management Micro-Frontend**

application shell, [Application Shell](#)-[Application Shell](#)

Catalog micro-frontend, [Catalog Micro-Frontend](#)

Home micro-frontend, [Home Micro-Frontend](#)-[Home Micro-Frontend](#)

introducing micro-frontends in your organization (see organizations)

Inverse Conway Maneuver, [How Do Committees Invent?](#)

ISG (Incremental Static Generation), RSG, [Vercel: A Glance into the Future](#)

isolation of failures

in microservices, [Isolated Failures](#)

in micro-frontends, [Isolated Failures](#)

iterating often (automation pipelines), [Iterate Often](#)

iterative migrations, [Why Go Iterative?](#), [Personal Experience](#)

J

JavaScript

asynchronous loading of bundles using Module Federation, [The Project](#), [Module Federation 101](#)

entry point file for micro-frontends, [Client-Side Composition](#), [Vertical Split](#)

vertical-split micro-frontends and, [Application Shell](#)

loading micro-frontends dynamically inside, [Performance](#)

running when using Web Fragments, [Web Fragments](#)

setting maximum bundle size for performance, [Performance: The Key Reason for Server-Side Rendering](#)

stack trace of exceptions, [Observability](#)

Webpack bundler for, [The Project](#)

Jest, using for unit and integration testing, [Code-Quality Review](#)

Jfrog Artifactory, [Post-Build Review](#)

JSON

Frontend Discovery schema based in, [Welcome to the Frontend Discovery Schema](#)

JSON object with service dictionary, [Working with a Service Dictionary](#)

System function reading config and dynamically loading remote after registering it, [Application Shell](#)

JWTs (JSON Web Tokens), [Application Shell](#), [How to Manage Data](#)

AI generating React hook to refresh JWTs, [Example scenario](#)

K

knowledge sharing, impacts of micro-frontend anarchy, [Micro-Frontend Anarchy](#)

knowledge, institutional, externalizing in form AI assistants can use, [Context Engineering to the Rescue](#)

L

lambda functions

AWS Lambda, event-driven, serverless computing platform, [Deployment](#)

handling SSR on BBC website, [The BBC Architecture: Caching in Action](#)

landing pages

implementing for horizontal-split architecture video-streaming website, [Client Side](#)

largest contentful paint (LCP), [Server-Side Rendering Micro-Frontends](#)

layout and symbolism, cultural sensitivity in, [Architecture Is Not Yet Automatable](#)

lazy-loading components, [Client-Side Composition](#)

LCP (largest contentful paint), [Server-Side Rendering Micro-Frontends](#)

legacy application

embedding in internal ecommerce site project, [Embedding a Legacy Application](#)

integrating with micro-frontends, using ACL, **Anti-Corruption Layer to the Rescue**
Lerna, managing multiple projects in a repository, **Monorepo**, **Version Control**
libraries

ability to isolate, mechanisms for, **Home Micro-Frontend**

checking on implementation of specific libraries across micro-frontends, **Code-Quality Review**

creating shared library to handle authentication tokens in micro-frontends, **Sharing state**

external, sharing across micro-frontends using Webpack with Module Federation, **Module Federation**

handling multiple versions in same application, caution with, **Home Micro-Frontend**

hasty abstractions in shared libraries, **Premature Abstraction**

loading shared libraries into micro-frontends, **Application Shell**

micro-frontends managing different library versions in same app, **Home Micro-Frontend**

shared across micro-frontends, managing with Dependabot, **Version Control**

shared library versions, **Application Shell**

life cycle methods (single-spa), **Available Frameworks**

life cycles, independent, using Web Fragments, **Web Fragments**

Lighthouse, **Performance and Micro-Frontends**

CLI tool, **Post-Build Review**

using to run performance audits on micro-frontends, **Performance: The Key Reason for Server-Side Rendering**

loadRemote method (Module Federation), **Application Shell**

local routing, **Vertical Split**, **Global and Local Routing**

handling by each micro-frontend, [Application Shell](#)

implementation by Catalog micro-frontend in internal ecommerce site project, [Catalog Micro-Frontend](#)

local storage security model, [Integrating Authentication in Micro-Frontends](#)

localization, [Localization](#)

localStorage, [Authentication](#)

sensitive cookies not stored in, [How to Manage Data](#)

logging, [Working with an API Gateway](#)

LogRocket, [Observability](#)

low-complexity subdomains, [Low-Complexity Subdomain](#)

M

maintainability, improved, with micro-frontends, [Best Practices and Patterns](#)

maintenance of best practices, difficulty in polyrepo environment, [Polyrepo](#)

MAJOR.MINOR.PATCH format in semantic versioning, [Implementing Canary Releases](#)

management, how to get on board with migration to micro-frontends, [How Can I Get the Management On Board with This Migration?-Personal Experience](#)

best practices and patterns, [Best Practices and Patterns](#)

checklist of questions to ask yourself, [Checklist](#)

personal experience of author, [Personal Experience](#)

quick example, [A Quick Example](#)

trade-offs and pitfalls, [Trade-Offs and Pitfalls](#)

Mercurial, [Version Control](#)

merge hell in branching strategies, [Monorepo](#)

Mermaid.js, [Diagram Generation](#)

meta shells, [Account Management Micro-Frontend](#), [Account Management Micro-Frontend](#)

Meta, use of monorepos, [Monorepo](#)

meta-shell approach, [Account Management Micro-Frontend](#)

metrics

collection by API gateway, [Working with an API Gateway](#)

measuring performance continuously and objectively, [Performance: The Key Reason for Server-Side Rendering](#)

in micro-frontend performance, [Performance and Micro-Frontends](#)

using in defining bounded contexts, [How to Define a Bounded Context](#)

web performance metrics, [Server-Side Rendering Micro-Frontends](#)

MFE_discovery service, [Real-World Implementation](#)

micro-frontend anarchy antipattern, [Micro-Frontend Anarchy-Micro-Frontend Anarchy](#)

micro-frontend decision framework, [Micro-Frontend Decision Framework](#)

application to architectures, [Discovering Micro-Frontend Architectures](#)

applied in monolith to micro-frontends migration strategy, [Micro-Frontend Decision Framework Applied](#)

defining micro-frontends, [Defining Micro-Frontends-Defining Micro-Frontends](#)

following for internal ecommerce site example, [The Project](#)

summary of, [Micro-Frontend Communication](#)

micro-frontends

acting as adapters, [Embedding a Legacy Application](#)

applying microservice principles to, [Applying Principles to Micro-Frontends-Highly Observable](#)

culture of automation, [Culture of Automation](#)

decentralization, [Decentralization](#)

hiding implementation details, **Hidden Implementation Details**

high observability, **Highly Observable**

independent deployment, **Independent Deployment**

isolated failures, **Isolated Failures**

modeling around business domains, **Modeled Around Business Domains**

architectures, selecting right type, **Micro-Frontend Architectures and Challenges**

challenges unique to, **Challenges Unique to Micro-Frontends**

companies that have adopted micro-frontends, **Micro-Frontends in Practice-DAZN**

composition of, **Testing Your Micro-Frontend Boundaries-Server-Side Composition**

client-side composition, **Client-Side Composition**

edge-side composition, **Edge-Side Composition**

server-side composition, **Server-Side Composition**

defined, **Micro-Frontend Principles**

flexibility to load different versions of, **Application Shell**

multiple micro-frontends consuming same API, consolidating, **Multiple Micro-Frontends Consuming the Same API**

uses and limitations of, **Micro-Frontends Are Not a Silver Bullet**

using with microservices, **Introducing Micro-Frontends**

micro-frontends layer in server-side composition, **Composing Micro-Frontends**

“Micro-frontends in the Trenches” show, **Data to the Rescue**

microservices

advantages and disadvantages of, **Micro-Frontend Principles**

BFF hiding complexity by creating single entry point for client to consume APIs, **Working with the BFF Pattern**

micro-frontends with, [Introducing Micro-Frontends](#)

moving to from monolith, [Moving to Microservices](#)

principles of, [Microservices Principles-Highly Observable](#)

culture of automation, [Culture of Automation](#)

decentralization, [Decentralization](#)

hiding implementation details, [Hidden Implementation Details](#)

high observability, [Highly Observable](#)

independent deployment, [Independent Deployment](#)

isolated failures, [Isolated Failures](#)

modeling around business domains, [Modeled Around Business Domains](#)

server-side composition with API gateway, [Server-Side Composition with an API Gateway](#)

service discovery, [Welcome to the Frontend Discovery Schema](#)

Microsoft, API guidelines, [API Consistency](#)

middleware

in internal ecommerce site project, [Application Shell](#)

in Next.js, [Next.js Multi-Zones](#)

verifying authentication and authorization requests in multi-zone architecture, [How to Manage Data](#)

middleware functions

using for sharing authentication tokens, [Sharing state, Authentication](#)

migrations

Formula 1 website, from monolith to micro-frontends, [The Formula 1 Website is Powered by Micro-Frontends](#)

migrating legacy application, use of multi-framework strategy, [Multi-framework approach](#)

migrating to micro-frontends, [Migrating to Micro-Frontends](#)

How can I get management on board with this migration?, [How Can I Get the Management On Board with This Migration?-Personal Experience](#)

How should I actually plan the migration?, [How Should I Actually Plan the Migration?-Personal Experience](#)

how to decide which modules or features to migrate first, [How Do I Decide Which Modules or Features to Migrate First?-Personal Experience](#)

how to handle shared dependencies and version management between legacy and micro-frontend code, [How Do I Handle Shared Dependencies and Version Management Between Legacy and Micro-Frontend Code?-Personal Experience](#)

how to manage cross-cutting concerns during migration, [How Do I Manage Cross-Cutting Concerns Like Auth, Routing, or State During Migration?-Personal Experience](#)

iterative migration, [Why Go Iterative?-Why Go Iterative?](#)

questions covered, [Why Go Iterative?](#)

user experience and consistency, how to maintain, [How Do I Maintain User Experience and Consistency During Migration?-Personal Experience](#)

which problems am I trying to solve with micro-frontends, [Which Problem\(s\) Am I Trying to Solve with Micro-Frontends?-Personal Experience](#)

monolith to microservices, use of BFF, [Working with the BFF Pattern](#)

using AI in code migrations, [Code Migration](#)

[MinChunkSizePlugin](#), [Module Federation 101](#)

minimum viable product (MVP), [Monolith to Distributed Systems](#)

mob programming, [Community of Practice and Town Halls](#)

mobile applications

use of BFF pattern with, [Working with the BFF Pattern](#)

use of micro-frontends, [The Link Between Organizations and Software Architecture](#)

MobX, [Dependency Management](#)

MobX state tree, [Technology Choice](#)

model-view-controller (MVC), [Shared code](#)

model-view-intent (MVI), [Shared code](#)

model-view-presenter (MVP), [Shared code](#)

model-view-viewmodel (MVVM), [Shared code](#)

modular monoliths, [Working with a Service Dictionary](#)

implementation of service dictionary with, [Working with a Service Dictionary](#)

modularity, [The Link Between Organizations and Software Architecture](#)

analysis for architectures, [Architecture Analysis](#)

in horizontal-split micro-frontends, [Architecture Characteristics](#)

horizontal-split micro-frontends with server-side composition, [Architecture Characteristics](#)

of horizontal-split architectures, [Client Side](#)

micro-frontends composed using iframes, [Architecture Characteristics](#)

micro-frontends implemented as web components, [Architecture Characteristics](#)

of SSR architecture, [Architecture Characteristics](#)

in vertical-split micro-frontends, [Architecture Characteristics](#)

Module Federation framework, [Composing micro-frontends](#), [Multi-framework approach](#)

basic concepts, understanding, [Module Federation 101](#)-[Module Federation 101](#)

benefits for client-side rendering applications, [Account Management Micro-Frontend](#)

different scope (or container) created by leveraging shareScope property, [Home Micro-Frontend](#)

ecommerce example implementing service discovery, [An Integration Example with Module Federation-An Integration Example with Module Federation](#)

inspecting shared resources through browser's developer tools in version 2.0, [Home Micro-Frontend](#)

key aspects, initialization and registering remotes at runtime, [Application Shell](#) plug-ins for multiple bundlers, [Module Federation 101](#)

shared dependency management in micro-frontends, [Application Shell](#)

use in implementing vertical-split architecture, [Available Frameworks](#)

use in internal ecommerce site project, reasons for, [The Project](#)

use in micro-frontend architectures, [Module Federation-Use Cases](#)

composition, [Composition](#)

developer experience, [Developer Experience](#)

exposing micro-frontends or shared libraries for asynchronous integration, [Module Federation](#)

host and remote parts, [Module Federation](#)

performance and, [Performance](#)

sharing code, [Shared code](#)

simplicity of sharing code across projects, problems with, [Module Federation](#)

use cases, [Use Cases](#)

using for client-side rendering, [Client-Side Rendering Micro-Frontends](#)

using in React application, sharing service dictionary via React context APIs, [Implementing a Service Dictionary in a Horizontal-Split Architecture](#)

using to build SSR micro-frontends, [Composition Approaches](#)

Vercel's approach to environment-agnostic Module Federation, [Vercel: A Glance into the Future](#)

modules or features to migrate first, deciding in move to micro-frontends, [How Do I Decide Which Modules or Features to Migrate First?-Personal Experience](#)

best practices and patterns, [Best Practices and Patterns](#)

checklist of questions to ask yourself, [Checklist](#)

personal experience of author, [Personal Experience](#)

quick example, [A Quick Example](#)

trade-offs and pitfalls to consider, [Trade-Offs and Pitfalls](#)

monitoring, microservices and, [Micro-Frontend Principles](#)

monolith to micro-frontends, case study, [From Monolith to Micro-Frontends: A Case Study-Summary](#)

context, [The Context-Techical Goals](#)

ACME Inc., video streaming service, [The Context](#)

platform and main user flows, [Platform and Main User Flows-Platform and Main User Flows](#)

technical goals, [Technical Goals](#)

technology stack, [Technology Stack-Technology Stack](#)

implementation details in migration, [Implementation Details-Localization](#)

application initialization, [Application Initialization](#)

application shell responsibilities, [Application Shell Responsibilities](#)

backend integration, [Backend Integration](#)

canary releases for micro-frontends, [Implementing Canary Releases](#)

communication bridge, [Communication Bridge](#)

dependency management, [Dependency Management](#)

global and local routing, [Global and Local Routing](#)

integrating authentication in micro-frontends, [Integrating Authentication in Micro-Frontends](#)

localization, [Localization](#)

migration strategy, [Migration Strategy](#)

mounting and unmounting micro-frontends, [Mounting and Unmounting Micro-Frontends](#)

sharing components, [Sharing Components](#)

sharing state, [Sharing State](#)

migration strategy, [Migration Strategy-Splitting the SPA into Multiple Subdomains](#)

micro-frontend decision framework applied to, [Micro-Frontend Decision Framework Applied](#)

splitting SPA into multiple subdomains, [Splitting the SPA into Multiple Subdomains-Splitting the SPA into Multiple Subdomains](#)

technology choice, [Technology Choice-Technology Choice](#)

monoliths

disadvantages of, [Micro-Frontend Principles](#)

distributed, [Defining Micro-Frontends](#)

implementation of service dictionary with, [Working with a Service Dictionary](#)

micro-frontends working in combination with, [Backend Patterns For Micro-Frontends](#)

migration to microservices, use of BFF, [Working with the BFF Pattern](#)

modular, [Working with a Service Dictionary](#)

in new projects, [Monolith to Distributed Systems](#)

splitting into microservices, [Moving to Microservices](#)

monorepo, [Project Structure, Monorepo-Monorepo](#)

advantages of, [Monorepo](#)

challenges of [Monorepo](#)

combined with polyrepo in future version control system, [A Possible Future for a Version Control System](#)

context engineering and, [Context Engineering to the Rescue](#)

example, all projects in same repository, [Monorepo](#)

in micro-frontend automation pipeline case study, [Setting the Scene, Version Control](#)

paper on monorepos, [Monorepo](#)

mounting and unmounting micro-frontends, [Mounting and Unmounting Micro-Frontends](#)

multi-environment strategy, [Blue-Green Deployment Versus Canary Releases](#)

multi-framework approach, [Multi-framework approach](#)

contributing to micro-frontend anarchy, [Micro-Frontend Anarchy](#)

in horizontal-split architectures, [Multi-framework approach](#)

offering significant benefits in certain contexts, [Micro-Frontend Anarchy](#)

viewing as temporary solution rather than a long-term architectural strategy,
[Micro-Frontend Anarchy](#)

multi-tenant applications

use of horizontal-split architectures, [Use Cases](#)

multi-zone architectures, [Composing Micro-Frontends, Modern Server-Side Rendering Frameworks, Composition Approaches](#)

multi-zone micro-frontend architectures, [Next.js Multi-Zones-Vercel: A Glance into the Future](#)

handling shared components, [How to Handle Shared Components](#)

home zone, entry point of web application, [Home Zone: The Entry Point of Your Web Application](#)

managing data, [How to Manage Data](#)

multirepo (see polyrepo)

MVC (model-view-controller) pattern, [Shared code](#)

MVI (model–view–intent), [Shared code](#)

MVP (minimum viable product), [Monolith to Distributed Systems](#)

MVP (model–view–presenter), [Shared code](#)

MVVM (model–view–viewmodel), [Shared code](#)

N

namespacing events in micro-frontends, [Account Management Micro-Frontend](#)

naming conventions, regulating in polyrepos, [Polyrepo](#)

Native Federation library, [Composing micro-frontends](#)

import maps, API leveraging, [Integrating the Discovery Pattern with Other Solutions](#)

integrating discovery pattern with, [Integrating the Discovery Pattern with Other Solutions](#)

use in implementing vertical-split architecture, [Available Frameworks](#)

navigation in multi-zone setup, [Next.js Multi-Zones](#)

Netflix

subdomains, examples of, [Defining Micro-Frontends](#)

use of micro-frontends, [Netflix](#)

New Relic, [Use Cases](#), [Observability](#)

Next.js framework, [Horizontal Split](#), [Composing Micro-Frontends](#)

multi-zone feature, [Modern Server-Side Rendering Frameworks](#)

multi-zones, use in building micro-frontends, [Next.js Multi-Zones](#)-[Next.js Multi-Zones](#)

home zone, entry point of web app, [Home Zone: The Entry Point of Your Web Application](#)

React Server Components in, [Modern Server-Side Rendering Frameworks](#)

use cases for, [Use Cases](#)

NGINX reverse proxy, [Modern Server-Side Rendering Frameworks](#)

Node.js framework, [Composition Approaches](#)

multi-zone approach to composing SSR micro-frontends, [Composition Approaches](#)

normal-complexity subdomains, [Normal-Complexity Subdomain](#)

notification event, [Account Management Micro-Frontend](#)

O

object storage, [Technology Choice](#)

Object.freeze method, [Account Management Micro-Frontend](#)

observability, [How to Define a Bounded Context](#)

high observability in micro-frontends, [Highly Observable](#)

high observability in microservices, [Highly Observable](#)

integration into monolithic application, [Monolith to Distributed Systems](#)

microservices and, [Micro-Frontend Principles](#)

platform in monolith to micro-frontends case study, [Platform and Main User Flows](#)

real-world observability in production, [Performance: The Key Reason for Server-Side Rendering](#)

in successful micro-frontend architectures, [Observability](#)

observability tools, [Observability](#)

observer pattern (or pub/sub pattern), [Horizontal Split](#)

omnidirectional sharing across micro-frontends, [Unidirectional Sharing](#)

harm done to deployment and rollback strategies due to shared dependencies,
Unidirectional Sharing

on-demand environments, **Environments Strategies, Testing Technical Recommendations**

onboarding for new hires

advantage of monorepo for, **Monorepo**

faster onboarding with micro-frontends, **Best Practices and Patterns, Micro-Frontend Decision Framework Applied**

goal of tracking onboarding time in monolith to micro-frontends case study, **Technical Goals**

OpenComponents framework, **Available Frameworks**

OpenTable's OpenComponents, use of micro-frontends, **OpenTable**

organizations

introducing micro-frontends in, **Introducing Micro-Frontends in Your Organization-Summary**

benefits of using micro-frontends, **Why Should We Use Micro-Frontends?- Why Should We Use Micro-Frontends?**

creating trade-off analysis, **Creating a Trade-Off Analysis-Organization Capabilities**

data to convince management, **Data to the Rescue**

decentralization, implications for micro-frontends, **Decentralization Implications with Micro-Frontends-Low-Complexity Subdomain**

decentralized organization, **A Decentralized Organization-Decentralization Implications with Micro-Frontends**

features versus components teams, **Features Versus Components Teams- Features Versus Components Teams**

how committees invent, **How Do Committees Invent?**

implementing governance to ease communication flows, [Implementing Governance for Easing the Communication Flows-Architecture Decision Records](#)

link between organizations and software architecture, [The Link Between Organizations and Software Architecture](#)

techniques to enhance communication flow, [Techniques for Enhancing the Communication Flow-Managing External Dependencies](#)

migration to micro-frontends best suited for, [Personal Experience](#)

organizational goals in monolith to micro-frontends case study, [Technical Goals](#)

organizational readiness for micro-frontends, [Trade-Offs and Pitfalls](#)

organizational structure, relationship with culture and software architecture, [Common Antipatterns in Micro-Frontend Implementations](#)

overhead for small teams/apps, [Trade-Offs and Pitfalls](#)

P

package management and versioning in multi-zone architecture shared components, [How to Handle Shared Components](#)

page loads, fast initial loads to improve user retention, [When to Use Server-Side Rendering](#)

pages

in migration to micro-frontends, favoring entire pages or groups of pages, [Best Practices and Patterns](#)

page routing in micro-frontend architecture, [Application Shell](#)

splitting micro-frontends per page or per group of pages, [Composing Micro-Frontends](#)

parent component to its children, unidirectional data flow, [Unidirectional Sharing](#)

PayPal, use of micro-frontends, [PayPal](#)

performance, [Architecture Characteristics](#)

analysis for architectures, [Architecture Analysis](#)

considerations driving architectural decisions, [Architecture Is Not Yet Automatable](#)

in horizontal-split micro-frontends, [Architecture Characteristics](#)

of horizontal-split architecture with server-side composition, [Architecture Characteristics](#)

iframes and, [Architecture Characteristics](#)

iframes causing issues with, [iframes](#)

improvements from SSR architectures, [When to Use Server-Side Rendering](#)

improvements in Formula 1 website from use of micro-frontends, [The Formula 1 Website is Powered by Micro-Frontends](#)

issues caused by multi-framework approach, [Multi-framework approach](#)

key reason for SSR, [Performance: The Key Reason for Server-Side Rendering](#)-
[Performance: The Key Reason for Server-Side Rendering](#)

metrics for, [Fitness Functions](#)

metrics, time-to-first-byte (TTFB) and time-to-interactive (TTI), [Application Shell](#)

and micro-frontends, [Performance and Micro-Frontends](#)-[Performance and Micro-Frontends](#)

vertical-split micro-frontends, [Performance and Micro-Frontends](#)

micro-frontends implemented as web components, [Architecture Characteristics](#)

monitoring, challenges of in micro-frontends, [Challenges Unique to Micro-Frontends](#)

of SSR architecture, [Architecture Characteristics](#)

testing for post-build review of micro-frontend automation pipeline, [Post-Build Review](#)

web performance metrics, [Server-Side Rendering Micro-Frontends](#)

persistent layer, **Monolith to Distributed Systems**

Piral Cloud, micro-frontend service discovery implementation, **Availability in the Market**

placeholder tags, **Client-Side Composition**

planning migration to micro-frontends, how to, **How Should I Actually Plan the Migration?-Personal Experience**

checklist of questions to ask yourself, **Checklist**

personal experience of author, **Personal Experience**

quick example, **A Quick Example**

trade-offs and pitfalls to consider, **Trade-Offs and Pitfalls**

platform and main user flows in monolith to micro-frontends case study, **Platform and Main User Flows-Platform and Main User Flows**

playbook for working with AI

automating through AI, **Step 6: Automate Through AI-Key takeaways**

effective prompting, **Effective prompting**

how to approach it, **How to approach it**

importance of, **Why it matters**

defining what you want to achieve, **Step 1: Define What You Want to Achieve-Key takeaways**

effective prompting, **Effective prompting**

how to approach it, **How to approach it**

importance of, **Why it matters**

pitfalls to avoid in prompting, **Pitfalls to avoid**

engineering the context instead of just prompting, **Step 2: Engineer the Context; Don't Just Prompt-Key takeaways**

effective prompting, **Effective prompting**

embedding context in codebase, **Strategy: Embed context in the codebase**

how to approach it, **How to approach it**

importance of, **Why it matters**

prompt refinement example, **Prompt refinement example**

iterating in small, focused steps, **Step 4: Iterate in Small, Focused Steps-Key takeaways**

antipattern, one prompt to rule them all, **Antipattern: The “one prompt to rule them all”**

effective prompting, **Effective prompting**

follow-up prompt, **Now, consider this follow-up prompt:**

how to approach it, **How to approach it**

importance of, **Why it matters**

why this works, **Why this works**

reasoning normally as a developer, **Step 3: Reason Like You Normally Would as a Developer-Key takeaways**

example, AI as design reviewer, **Example: AI as design reviewer**

how to approach it, **How to approach it**

importance of, **Why it matters**

pitfalls to avoid, **Pitfalls to avoid**

scenarios for using this pattern, **When to use this pattern**

reviewing and refactoring output, **Step 5: Review and Refactor the Output-Key takeaways**

example scenarios, **Example scenario-Example scenario**

how to approach it, **How to approach it**

importance of, **Why it matters**

prompt pattern for review, **Prompt pattern for review**

polyrepo, Project Structure, Polyrepo-Polyrepo

benefits of, Polyrepo

caveats, Polyrepo

combined with monorepo in future version control system, A Possible Future for a Version Control System

PR/FAQs, Working Backward

pre-provisioning compute resources, Scalability Challenges

precomputation, Scalability Challenges

premature abstraction, Premature Abstraction, Trade-Offs and Pitfalls

preproduction environment (see staging environment)

presentation layer (frontend), Monolith to Distributed Systems

press release (PR), Working Backward

problems I'm trying to solve with micro-frontends, deciding, Which Problem(s) Am I Trying to Solve with Micro-Frontends?-Personal Experience

checklist of questions to ask yourself, Checklist

personal experience of author, Personal Experience

quick example, A Quick Example

trade-offs and pitfalls to consider, Trade-Offs and Pitfalls

problems slowing down the business, identifying, Data to the Rescue

production environment, Environments Strategies, Blue-Green Deployment Versus Canary Releases, Setting the Scene

responsibility for running system in production, Architecture Is Not Yet Automatable

testing in production, challenges of, End-to-End Testing

projects

coupling, risk in monorepos, Monorepo

difficulties with discoverability using polyrepos, [Polyrepo](#)
scaffolding, using AI in, [Project Scaffolding](#)

prompting
[effective](#), [Effective prompting](#)
[reasoning](#) as a developer, [Effective prompting](#)
[engineering](#) the context instead of just prompting, [Step 2: Engineer the Context; Don't Just Prompt](#)-Key takeaways
[iterating](#) in small, focused steps, [Step 4: Iterate in Small, Focused Steps](#)-Key takeaways
[pitfalls](#) to avoid, [Pitfalls to avoid](#)
[prompt pattern](#) for reviewing AI output, [Prompt pattern for review](#)
when aiming for automation, [Effective prompting](#)

[proof of concept](#), [Best Practices and Patterns](#)

prototyping, using AI in, [Prototyping](#)

proxy servers
[in micro-frontend automation pipeline case study](#), [Post-Build Review](#)
[using in end-to-end tests](#) of micro-frontends, [Testing Technical Recommendations](#)
[Webpack DevServer proxy configuration](#), [Testing Technical Recommendations](#)

[publish/subscribe \(pub/sub\) pattern](#), [Horizontal Split, Sharing State Between Micro-Frontends](#)

[events with pub/sub approach](#), unidirectional data flow and, [Unidirectional Sharing](#)
[using between iframes and host page](#), [Best Practices and Drawbacks](#)

Q

[quality of code](#), reviewing, [Code-Quality Review](#)-[Code-Quality Review](#)

query strings or URLs, micro-frontend communication via, [Micro-Frontend Communication, Horizontal Split](#)

Qwik framework, [Modern Server-Side Rendering Frameworks](#)

use cases for, [Use Cases](#)

R

rate limiting, [Working with an API Gateway](#)

React, [Shared code](#), [Dependency Management](#)

context APIs, [Implementing a Service Dictionary in a Horizontal-Split Architecture](#)

custom hook retrieving micro-frontends available from frontend discovery, [An Integration Example with Module Federation](#)

micro-frontend retrieving React and ReactDOM libraries avoiding version clashes, [Home Micro-Frontend](#)

technology choice in monolith to micro-frontends case study, [Technology Choice using Module Framework](#) along with, [The Project](#)

React Router, [Application Shell](#), [Catalog Micro-Frontend](#)

React Server Components (RSCs), [Modern Server-Side Rendering Frameworks](#), [Vercel: A Glance into the Future](#)

reactive programming paradigm, [Technology Choice](#)

reactive streams, [Horizontal Split](#)

use of, [Account Management Micro-Frontend](#)

using for micro-frontend communication, [Micro-frontend communication](#)

README, using AI to maintain, [Keeping the README Alive](#)

Redis, [In-Memory Database Cache](#)

caches used by BBC site, [The BBC Architecture: Caching in Action](#)

Redux, [Technology Choice](#)

refactoring

micro-frontend refactoring in horizontal-split architectures, [Micro-Frontend Refactoring](#)

refresh tokens, [How to Manage Data](#)

registration of micro-frontends (single-spa), [Available Frameworks](#)

regulatory factors impacting architecture, [Architecture Is Not Yet Automatable](#)
releases

faster, safer with micro-frontends, [Best Practices and Patterns](#)

implementing canary releases in monolith to micro-frontends case study,
[Implementing Canary Releases](#)

strategies in migration to micro-frontends, [Best Practices and Patterns](#)

remote (Module Federation), [Module Federation](#), [Module Federation 101](#)

creating MyAccount host having two remotes, [Account Management Micro-Frontend](#)

defining all remotes in Webpack configuration, [Application Shell](#)

enforcing hierarchical relation between host and remote, [Account Management Micro-Frontend](#)

registering remotes at runtime, [Application Shell](#)

rendering mode for micro-frontends

choosing thoughtfully, [HTML Fragments](#)

repositories, [Version Control-A Possible Future for a Version Control System](#)

cloning in micro-frontend automation pipeline initialization, [Pipeline Initialization](#)

hybrid approach, combining monorepo and polyrepo strengths, [A Possible Future for a Version Control System](#)

monorepo strategy, [Monorepo-Monorepo](#)

polyrepo strategy, [Polyrepo-Polyrepo](#)

requests for comments (RFCs)

benefits of using, [Team Communication and Best Practices for Maintaining Control of the Final Outcome](#)

contents and structure of, [Requests for Comments](#)

sharing with teams when breaking change must be introduced in API, [APIs Come First, Then Implementation](#)

response time

in horizontal-split architecture with server-side composition, [Scalability and Response Time](#)

REST APIs

GraphQL becoming as popular in micro-frontends, [API Strategies](#)

reusability

concerns raised by use of BFF pattern, [Working with the BFF Pattern](#)

enhancement using monorepos, [Monorepo](#)

micro-frontends and, [Application Shell](#)

representing form of coupling, [Managing External Dependencies](#)

reusable micro-frontend development guide across projects, [Context Engineering to the Rescue](#)

reusing micro-frontends across a project, horizontal split architecture, [Managing External Dependencies](#)

rewrites in Next.js multi-zones, [Next.js Multi-Zones](#)

home zone, [Home Zone: The Entry Point of Your Web Application](#)

right approach for the right solution, [The Right Approach for the Right Subdomain](#)

rollbacks

faster, in migration to micro-frontends, [Best Practices and Patterns](#)

Rolldown, [Module Federation 101](#)

rollouts

progressive, in migration to micro-frontends, **Best Practices and Patterns**

Rollup, **Developer Experience**

routing

Catalog micro-frontend in internal ecommerce site, implementing local routing,
Catalog Micro-Frontend

challenges of in micro-frontends, **Challenges Unique to Micro-Frontends**

client-side, **The Project**

use with vertical-split micro-frontends, **Vertical Split**

global and local, in monolith to micro-frontends case study, **Global and Local Routing**

hard navigation between legacy system and micro-frontends, leveraging absolute URLs, **Best Practices and Patterns**

implementing dynamic routes, including error handling for unexpected cases,
Application Shell

loading routes dynamically using initializeMFEs, **Application Shell**

managing in migration to micro-frontends, **How Do I Manage Cross-Cutting Concerns Like Auth, Routing, or State During Migration?**

of micro-frontends, **Routing Micro-Frontends-Routing Micro-Frontends**

of micro-frontends in monolith to micro-frontends migration, **Micro-Frontend Decision Framework Applied**

Route objects consisting of path and corresponding micro-frontend, **Application Shell**

single source of truth for, in migration to micro-frontends, **Best Practices and Patterns**

summary of in micro-frontend decision framework, **Micro-Frontend Communication**

top-level routes, assigning by first-level URL, [Splitting by URL](#)
using React Router, [Application Shell](#)
using rewrites or middleware in Next.js multi-zones, [Next.js Multi-Zones](#)

RSCs (React Server Components), [Modern Server-Side Rendering Frameworks](#),
[Vercel: A Glance into the Future](#)

Rspack, [Module Federation 101](#)

rulebook for reusable micro-frontend development, [Context Engineering to the Rescue](#)
runtime sharing of components, [How to Handle Shared Components](#)

S

SameSite cookies, [How to Manage Data](#)

scaffolding

creating command-line tool for scaffolding micro-frontends, [Frictionless Micro-Frontend Blueprints](#)

using AI in project scaffolding, [Project Scaffolding](#)

scalability

analysis for architectures, [Architecture Analysis](#)

challenges in scaling SSR applications, [Scalability Challenges](#)

considerations with pages personalized per user, [Server-Side Composition](#)

horizontal-split architecture with server-side composition, [Scalability and Response Time](#)

in horizontal-split micro-frontends, [Architecture Characteristics](#)

of horizontal-split architecture with server-side composition, [Architecture Characteristics](#)

micro-frontends composed using iframes, [Architecture Characteristics](#)

micro-frontends implemented as web components, [Architecture Characteristics](#)

micro-frontends scaling with organization growth, [Best Practices and Patterns](#)
scaling server-side composition micro-frontends, [Routing Micro-Frontends](#)
scaling SSR with smart caching strategies, [Don't Fear the Cache: Scaling SSR with Smart Caching Strategies](#)
of SSR architecture, [Architecture Characteristics](#)
in vertical-split micro-frontends, [Architecture Characteristics](#)
scaling in microservice applications, [Moving to Microservices](#)
schema federation, [Using GraphQL with Micro-Frontends](#)
schema stitching, [The Schema Federation](#)
scopes
shareScope property of Module Federation, [Home Micro-Frontend](#)
using with import maps or SystemJS, [Home Micro-Frontend](#)
Scrum of Scrums, [Managing External Dependencies](#)
search engine optimization (SEO)
crawability, [Server-Side Rendering Micro-Frontends](#)
in horizontal-split architectures, [Search Engine Optimization](#)
SSR architecture for content-heavy websites with crucial SEO, [When to Use Server-Side Rendering](#)
strong requirements for, use of server-side composition, [Server Side](#)
second-level URLs and beyond, [Vertical Split](#)
Secure cookies, [How to Manage Data](#)
security
testing for architectures, [Fitness Functions](#)
semantic versioning, [Implementing Canary Releases](#)
Sentry, [Observability](#)

server islands (Astro.js), [Composition Approaches](#)

server-side composition, [Server-Side Composition](#), [Server Side-Modern Server-Side Rendering Frameworks](#)

with API gateway, [Client-Side Composition with an API Gateway and a Service Dictionary](#)-[Server-Side Composition with an API Gateway](#)

with BFF and service dictionary, [Server-Side Composition with a BFF and a Service Dictionary](#)

BFFs and API gateways better suited to micro-frontends than service dictionary, [Working with a Service Dictionary](#)

composing micro-frontends, [Composing Micro-Frontends](#)-[Composing Micro-Frontends](#)

in horizontal-split micro-frontends, [Horizontal Split](#)

in horizontal-split micro-frontends, [Horizontal Split](#)

horizontal-split micro-frontends with

analysis of architecture characteristics, [Architecture Characteristics](#)

use cases, [Use Cases](#)

infrastructure ownership in, [Infrastructure Ownership](#)

micro-frontend communication in, [Micro-Frontend Communication](#)

OpenComponents framework for, [Available Frameworks](#)

rejection in monolith to micro-frontends migration, [Micro-Frontend Decision Framework Applied](#)

routing, [Routing Micro-Frontends](#)

scalability and response time, [Scalability and Response Time](#)

using GraphQL with micro-frontends and, [Using GraphQL with Micro-Frontends and a Server-Side Composition](#)

server-side rendering (SSR) frameworks, [Horizontal Split](#)

analysis of architecture characteristics, [Architecture Characteristics](#)

modern frameworks, [Modern Server-Side Rendering Frameworks](#)

use cases for, [Use Cases](#)

server-side rendering (SSR) micro-frontends

API strategies, [API Strategies](#)

BBC website architecture, using caching, [The BBC Architecture: Caching in Action](#)

caches, types to consider, [Types of Caches Every Developer Should Know-Warm Cache](#)

challenges in scaling SSR applications, [Scalability Challenges](#)

dividing micro-frontends, [Dividing Micro-Frontends](#)

Formula 1 website, powered by micro-frontends, [The Formula 1 Website is Powered by Micro-Frontends](#)

multi-zone architectures

handling shared components, [How to Handle Shared Components](#)

home zone, entry point of web app, [Home Zone: The Entry Point of Your Web Application](#)

managing data, [How to Manage Data](#)

performance, key reason for SSR, [Performance: The Key Reason for Server-Side Rendering-Performance: The Key Reason for Server-Side Rendering](#)

scaling SSR with smart caching strategies, [Don't Fear the Cache: Scaling SSR with Smart Caching Strategies](#)

splitting by URL, [Splitting by URL-Splitting by URL](#)

SSR at compile time instead of runtime, [Micro-Frontend-Specific Operations](#)

use cases for SSR, [When to Use Server-Side Rendering](#)

using Next.js multi-zones, [Next.js Multi-Zones](#)

web performance metrics and, [Server-Side Rendering Micro-Frontends](#)

serverless, [Scalability Challenges](#)

AWS Lambda, event-driven, serverless computing platform, [Deployment](#)

leveraging AWS Lambda@Edge in monolith to micro-frontends case study, [Technology Choice](#)

use of serverless functions in BBC website, [The BBC Architecture: Caching in Action](#)

service dictionaries, [Working with a Service Dictionary](#)-[Implementing a Service Dictionary in a Horizontal-Split Architecture](#), [Summary](#)

client-side composition with BFF and service dictionary, [Client-Side Composition with a BFF and a Service Dictionary](#)-[Client-Side Composition with a BFF and a Service Dictionary](#)

client-side micro-frontend composition with API gateway and, [Client-Side Composition with an API Gateway and a Service Dictionary](#)

high-level architecture on using a service dictionary for testing in production, [Working with a Service Dictionary](#)

implementing in horizontal-split architectures, [Implementing a Service Dictionary in a Horizontal-Split Architecture](#)

implementing in vertical-split architectures, [Implementing a Service Dictionary in a Vertical-Split Architecture](#)

server-side composition with BFF and service dictionary, [Server-Side Composition with a BFF and a Service Dictionary](#)

service discovery, [Welcome to the Frontend Discovery Schema](#), [API Integration and Micro-Frontends](#)

benefits of, [Welcome to the Frontend Discovery Schema](#)

flow of, working with microservices, [Welcome to the Frontend Discovery Schema](#)

integration example with Module Federation, [An Integration Example with Module Federation](#)-[An Integration Example with Module Federation](#)

integration in micro-frontend client-side application, [Real-World Implementation](#)

micro-frontend service discovery pattern, implemented in automation pipeline case study, [Deployment](#)

use by UI composer to fill micro-frontend placeholders, [HTML Fragments](#)

sessionStorage, [Authentication](#)

sensitive cookies not stored in, [How to Manage Data](#)

setupFetch module, [Application Shell](#)

shared code

pitfalls with micro-frontends, [Trade-Offs and Pitfalls](#)

shared data, micro-frontend communication via, [Micro-Frontend Communication](#)

sharing data in different views, [Micro-Frontend Communication](#)

shared libraries, versions of, [Application Shell](#)

shared resource

inspecting through browser's developer tools using Module Federation 2.0, [Home Micro-Frontend](#)

sharing between micro-frontends (antipattern), [Sharing State Between Micro-Frontends](#), [Sharing State Between Micro-Frontends](#)

sharing code

using Module Federation in micro-frontend architectures, [Shared code](#)

Shopify, [Using GraphQL with Micro-Frontends](#)

shopping cart contents, [Challenges Unique to Micro-Frontends](#)

simplicity, [Architecture Characteristics](#)

analysis for architectures, [Architecture Analysis](#)

in horizontal-split architecture with server-side composition, [Architecture Characteristics](#)

in horizontal-split micro-frontends, [Architecture Characteristics](#)

micro-frontends composed using iframes, [Architecture Characteristics](#)

micro-frontends implemented as web components, [Architecture Characteristics](#)

in SSR architecture, [Architecture Characteristics](#)

single source of truth for routing, [Best Practices and Patterns](#)

single-page applications (SPAs), [Monolith to Distributed Systems](#), [DDD with Micro-Frontends](#)

entry point for vertical-split micro-frontends, [Application Shell](#)

microservices architecture with SPA, [Moving to Microservices](#)

performance and, [Performance and Micro-Frontends](#)

porting frontend application from SPA to micro-frontends, [Multi-framework approach](#)

SPA for frontend in monolith to micro-frontends case study, [Technology Stack](#)

splitting SPA into multiple subdomains in monolith to micro-frontends case study, [Splitting the SPA into Multiple Subdomains](#)-[Splitting the SPA into Multiple Subdomains](#)

use with Web Fragments, [Web Fragments](#)

vertical-split micro-frontends represented as, [Horizontal Versus Vertical Split](#)

single-spa framework, [Available Frameworks](#)

import-map-injector utility library, [Integrating the Discovery Pattern with Other Solutions](#)

integrating discovery pattern with, [Integrating the Discovery Pattern with Other Solutions](#)

Snowpack, [Developer Experience](#)

sociotechnical aspects (in architecture), [Architecture and Trade-Offs](#)

soft navigations, [Next.js Multi-Zones](#)

software architecture

link between organizations and, [The Link Between Organizations and Software Architecture](#)

relationship with culture and organizational structure, [Common Antipatterns in Micro-Frontend Implementations](#)

sociotechnical aspect of, emphasis in DDD, [A Decentralized Organization](#)

SonarQube static analysis tool, [Code-Quality Review](#)

Sonatype Nexus Repository, [Post-Build Review](#)

sparse-checkout command (Git), [Monorepo](#)

SPAs (see single-page applications)

“split-brain” scenarios, [Trade-Offs and Pitfalls](#)

splitting micro-frontends

splitting authentication micro-frontend to reduce cognitive load, [Architecture Evolution](#)

using code encapsulation, [Architecture Evolution](#)

spot instances, [Environments Strategies](#)

Spotify’s Backstage, [Frictionless Micro-Frontend Blueprints](#)

sprints, [Managing External Dependencies](#)

SSRs (server-side rendering) frameworks, [Horizontal Split](#)

framework-specific solutions to composing SSR micro-frontends, [Composition Approaches](#)

use with horizontal-split micro-frontends, [Horizontal Split](#)

stack traces, [Observability](#)

staging environment, [Environments Strategies](#), [Blue-Green Deployment Versus Canary Releases](#), [Setting the Scene](#)

Staltz, André, website, [Shared code](#)

state

avoiding sharing state across horizontal-split micro-frontends, [Horizontal Split](#)

handling of state management by each micro-frontend, **Account Management Micro-Frontend, Sharing State Between Micro-Frontends**

managing during migration to micro-frontends, **Best Practices and Patterns**

shared state in micro-frontends, **Challenges Unique to Micro-Frontends, Micro-Frontend Communication**

sharing in monolith to micro-frontends case study, **Sharing State**

sharing in vertical-split architectures, **Sharing state**

static analysis, **Fitness Functions**

storage

choice between single storage with a CDN and multiple storages with a unified CDN, **Hosting a Client-Side Rendering Micro-Frontend Project**

using single or multiple storages with CDN, **Hosting a Client-Side Rendering Micro-Frontend Project**

strangler fig pattern, **Embedding a Legacy Application, Working with the BFF Pattern**

applying in moving to micro-frontends in case study, **Technology Choice**

using in migration to micro-frontends, **Best Practices and Patterns**

strangler pattern on the frontend, applying, **Personal Experience**

subdomains, **Defining Micro-Frontends**

examples of Netflix subdomains, **Defining Micro-Frontends**

high-complexity, **High-Complexity Subdomain**

in internal ecommerce t-shirt website project, **The Project**

intersecting and sharing similarities

assigning to co-located team, **How Do Committees Invent?**

grouping by user journeys, **How Do Committees Invent?**

overlapping with bounded contexts, **DDD with Micro-Frontends**

presented across views in horizontal-split micro-frontends, **Horizontal Split**

requiring high initial effort, [High Initial-Effort Subdomain](#)

selecting horizontal-split or vertical-split architecture for in a project, [The Project](#) separating different DDD subdomains, [Working with the BFF Pattern](#)

splitting SPA into multiple subdomains in monolith to micro-frontends case study, [Splitting the SPA into Multiple Subdomains-Splitting the SPA into Multiple Subdomains](#)

types of, [Defining Micro-Frontends](#)

supporting subdomains, [Defining Micro-Frontends](#)

SystemJS, [Composing micro-frontends](#), [Home Micro-Frontend](#)

T

t-shirt ecommerce project (see internal ecommerce t-shirt website project)

Tailor.js project, [Zalando](#)

team structures, [Horizontal-Split Architecture](#)

design team, [Implementing a Design System](#)

distributed, [DDD with Micro-Frontends](#)

system designs and, [DDD with Micro-Frontends](#)

Team Topologies (Skelton and Pais), [A Decentralized Organization](#)

teams

communication flows, link with software architecture, [How Do Committees Invent?](#)

empowering in automation strategy, [Empower Your Teams](#)

features versus components teams, [Features Versus Components Teams-Features Versus Components Teams](#)

friction between, seen as organizational problem, [Managing External Dependencies](#)

in micro-frontend anarchic approach, [Micro-Frontend Anarchy](#)

no risk of blocking using polyrepos, [Polyrepo](#)

overhead for small teams with micro-frontends, [Trade-Offs and Pitfalls](#)

responsible for subdomains in internal ecommerce site project, [The Project](#)

team autonomy and parallel development with micro-frontends, [Best Practices and Patterns](#)

team communication and best practices to control final outcome, [Team Communication and Best Practices for Maintaining Control of the Final Outcome](#)

two-pizza team rule, [Working with a Service Dictionary](#)

use of service dictionaries, [Working with a Service Dictionary](#)

technical goals in monolith to micro-frontends case study, [Technical Goals](#)

technology stack, [Technology Stack](#)-[Technology Stack](#), [Context Engineering to the Rescue](#)

templates, [Context Engineering to the Rescue](#)

HTML, associated with specific routes, [Horizontal Split](#)

HTML, in web components, [Web Components Technologies](#)

testability

analysis for architectures, [Architecture Analysis](#)

in horizontal-split micro-frontends, [Architecture Characteristics](#)

of horizontal-split architecture with server-side composition, [Architecture Characteristics](#)

micro-frontends composed using iframes, [Architecture Characteristics](#)

micro-frontends implemented as web components, [Architecture Characteristics](#)

of SSR architecture, [Architecture Characteristics](#)

in vertical-split micro-frontends, [Architecture Characteristics](#)

testing

AI use cases in, Testing

challenges when deploying different versions of micro-frontends to different environments, [The Problem Space](#)

defining test strategy, [Define Your Test Strategy](#)

disadvantages of monoliths in, [Micro-Frontend Principles](#)

enabling testing of end-to-end flows in isolation, [Environments Strategies](#)

of horizontal-split micro-frontends, [Horizontal Versus Vertical Split](#)

impacts of bidirectional sharing on, [Unidirectional Sharing](#)

locally, using Webpack DevServer proxy, [Developer Experience](#)

in micro-frontend automation pipeline case study, [Setting the Scene](#)

of micro-frontends, [Testing Micro-Frontends-Testing Technical Recommendations](#)

end-to-end testing, [End-to-End Testing](#)

horizontal-split end-to-end testing challenges, [Horizontal-Split End-to-End Testing Challenges](#)

technically implementing end-to-end tests, [Testing Technical Recommendations](#)

vertical-split end-to-end testing challenges, [Vertical-Split End-to-End Testing Challenges](#)

with new endpoint versions in production, [Working with a Service Dictionary](#)

in Module Federation applications, [Module Federation](#)

of vertical-split micro-frontends, [Horizontal Versus Vertical Split](#)

testing environment, [Environments Strategies](#), [Blue-Green Deployment Versus Canary Releases](#)

testing strategies, [Testing Micro-Frontends](#)

thin clients, [Introducing Micro-Frontends](#)

time-to-first-byte (TTFB), [Application Shell](#)

time-to-interactive (TTI), [Application Shell](#)

reduction through streaming of HTML page, [HTML Fragments](#)

time-to-live (TTL) values, [Scalability Challenges](#)

configuring different cache TTLs based on content volatility, [The BBC Architecture: Caching in Action](#)

different values for CDN cache, [Caching](#)

short, on BBC website for massive concurrent requests, [The BBC Architecture: Caching in Action](#)

tokens

authentication, access tokens and refresh tokens, [How to Manage Data](#)

storage of, [How to Manage Data](#)

storing and retrieving authentication tokens, [Authentication](#)

tools

automation, constant investment in using monorepo, [Monorepo](#)

polyrepos needing less investment in, [Polyrepo](#)

town halls, [Community of Practice and Town Halls](#)

trade-offs

with architectures, [Architecture and Trade-Offs](#)

constant across entire organization, reviewing team structure and architecture, [Features Versus Components Teams](#)

creating analysis of in convincing organization to adopt micro-frontends, [Creating a Trade-Off Analysis-Organization Capabilities](#)

architecture characteristics, [Architecture Characteristics](#)

business requirements, [Business Requirements](#)

organization capabilities, [Organization Capabilities](#)

transclusion, [Client-Side Composition](#)

use with HTML fragments, [HTML Fragments](#)

trunk-based development, [Monorepo](#), [Setting the Scene](#)

ts-arch (architectural unit test framework), [Code-Quality Review](#)

TTFB (time-to-first-byte), [Application Shell](#)

TTI (time-to-interactive), [Application Shell](#), [HTML Fragments](#)

TTL (time-to-live) values, [Scalability Challenges](#)

configuring different cache TTLs based on content volatility, [The BBC Architecture: Caching in Action](#)

different values for CDN cache, [Caching](#)

short, on BBC website for massive concurrent requests, [The BBC Architecture: Caching in Action](#)

two-pizza teams, [Working with a Service Dictionary](#)

U

UAT (user acceptance testing), [Setting the Scene](#)

UI

swapping micro-frontends directly from in any environment, [Vercel: A Glance into the Future](#)

UI components library in design system, [Implementing a Design System](#)

micro-frontend that hosts, [Implementing a Design System](#)

UI composers, [HTML Fragments](#)

UI events, micro-frontend communication via, [Micro-Frontend Communication](#)

event emitter and custom events, diagram of, [Micro-Frontend Communication](#)

UI frameworks

using multiple, [Multi-framework approach](#)

unidirectional data flow, **Module Federation 101**

sharing code across micro-frontends, benefits of, **Shared code**

unidirectional sharing across micro-frontends, **Unidirectional Sharing-Unidirectional Sharing**

unit testing, **Code-Quality Review**

unit tests, **Define Your Test Strategy**

using AI to generate, **Testing**

updates, frequent, micro-frontends with, **Caching**

URLs

application shell handling only first-level routes, **Application Shell**

leveraging absolute URLs for hard navigation between legacy system and micro-frontends, **Best Practices and Patterns**

splitting by first-level URLs for SSR micro-frontends, **Splitting by URL**

useMfeInitialization custom React hook, **An Integration Example with Module Federation**

user acceptance testing (UAT), **Setting the Scene, Platform and Main User Flows**

user details micro-frontend (example), **Account Management Micro-Frontend**

user experience (UX)

consistency of, challenges in micro-frontends, **Challenges Unique to Micro-Frontends**

performance optimizations and, **Performance and Micro-Frontends**

user experience and consistency during migration to micro-frontends, how to maintain, **How Do I Maintain User Experience and Consistency During Migration?-Personal Experience, Micro-Frontend Decision Framework Applied**

best practices and patterns, **Best Practices and Patterns**

checklist of questions to ask yourself, **Checklist**

integrating design system in monolith to micro-frontends case study, [Integrating a Design System](#)

personal experience of author, [Personal Experience](#)

quick example, [A Quick Example](#)

trade-offs and pitfalls to consider, [Trade-Offs and Pitfalls](#)

user flows in monolith to micro-frontends case study, [Platform and Main User Flows- Platform and Main User Flows](#)

users

cultures in different regions, impact on architecture, [Architecture Is Not Yet Automatable](#)

grouping subdomains by user journeys, [How Do Committees Invent?](#)

user preferences, [How to Manage Data](#)

V

value, generating quickly as possible, [The Link Between Organizations and Software Architecture](#)

in monolith to micro-frontends case study, [Technical Goals](#)

Vercel, innovations from, [Vercel: A Glance into the Future](#)

version control, [Version Control-A Possible Future for a Version Control System](#)

in micro-frontend automation pipeline case study, [Version Control pipeline initialization](#), [Pipeline Initialization](#)

monorepo strategy, [Version Control-Monorepo](#)

polyrepo strategy, [Polyrepo-Polyrepo](#)

version control systems, [Version Control](#)

possible future for, [A Possible Future for a Version Control System](#)

versioning

of APIs, [Working with a Service Dictionary](#)

of GraphQL endpoint, [Using GraphQL with Micro-Frontends](#)

handling version management between legacy and micro-frontend code in migration, [How Do I Handle Shared Dependencies and Version Management Between Legacy and Micro-Frontend Code?-Personal Experience](#)

best practices and patterns, [Best Practices and Patterns](#)

checklist of questions to ask, [Checklist](#)

personal experience of author, [Personal Experience](#)

quick example, [A Quick Example](#)

trade-offs and pitfalls to consider, [Trade-Offs and Pitfalls](#)

semantic, [Implementing Canary Releases](#)

of shared resources, avoiding version clashes using Module Federation, [Home Micro-Frontend](#)

vertical split, [Micro-Frontend Automation](#)

decision between horizontal split and, impacts on developer experience, [Horizontal Versus Vertical Split](#)

vertical split, micro-frontends, [Defining Micro-Frontends](#)

vertical-split architectures, [Vertical Split, Vertical-Split Architectures-Horizontal-Split Architecture](#)

analysis of architecture characteristics, [Architecture Characteristics-Horizontal-Split Architecture](#)

architecture evolution, [Architecture Evolution-Architecture Evolution](#)

in automation pipeline case study, [Setting the Scene](#)

benefits of BFF pattern used with vertical split composed on client side, [Client-Side Composition with a BFF and a Service Dictionary](#)

Catalog micro-frontend in internal ecommerce site, [Catalog Micro-Frontend](#)

challenges in, [Challenges-Multi-framework approach](#)

composing micro-frontends, [Composing micro-frontends](#)

multi-framework approach, [Multi-framework approach](#)

sharing state, [Sharing state](#)

chosen in monolith to micro-frontends migration, [Micro-Frontend Decision Framework Applied](#)

client-side micro-frontend composition with API gateway and service dictionary, [Client-Side Composition with an API Gateway and a Service Dictionary](#)

developer experience, [Developer's Experience](#)

end-to-end testing challenges, [Vertical-Split End-to-End Testing Challenges](#)

frameworks available for, [Available Frameworks](#)

handling of authentication and authorization, [Application Shell](#)

implementing a design system, [Implementing a Design System-Implementing a Design System](#)

implementing a service dictionary in, [Implementing a Service Dictionary in a Vertical-Split Architecture](#)

performance and, [Performance and Micro-Frontends-Performance and Micro-Frontends](#)

problem of too many external dependencies, [Managing External Dependencies](#)
server-side render at compile time instead of runtime, [Micro-Frontend-Specific Operations](#)

technically implementing end-to-end testing, recommendations for, [Testing Technical Recommendations](#)

use cases for, [Application Shell](#)

using vertical split with independent infrastructure to compose SSR micro-frontends, [Composition Approaches](#)

using WebSockets with micro-frontends, [WebSocket and Micro-Frontends](#)

using with hybrid-split in internal ecommerce site, [The Project](#)

Vite, [Module Federation 101](#)

volatility of code, [Architecture Is Not Yet Automatable](#)

Vue.js

micro-frontends using Vue.js 3.0.0 with Module Federation, [Module Federation](#)

W

warm caches, [Warm Cache](#)

web applications

ACME Inc's three-tier web application platform (example), [Technology Stack](#)

web components, [Web Components-Web Fragments](#)

dependency management, [Dependency Management](#)

micro-frontends implemented as

analysis of architecture characteristics, [Architecture Characteristics-Web Fragments](#)

use cases, [Use Cases](#)

migrating components from Angular to, [Integrating a Design System](#)

with shadow roots, use in composing SSR micro-frontends, [Composition Approaches](#)

technologies in, [Web Components Technologies](#)

using in micro-frontends design system, [Implementing a Design System](#)

Web Fragments, [Web Fragments](#)

WebGPU API, [What the Future Holds for AI and Micro-Frontends](#)

Webpack

configuration, [Application Shell](#)

existing knowledge in inside the organization, [The Project](#)

home page configuration in internal ecommerce site project, [Home Micro-Frontend](#)

JavaScript bundler for internal ecommerce site project, [Module Federation 101](#)

Module Federation no longer coupled to, [Module Federation 101](#)

performing code optimizations for micro-frontend builds, [Build](#)

Webpack 5, Module Federation library, [Composing micro-frontends](#)

Webpack DevServer proxy, [Developer Experience](#)

configuration, [Testing Technical Recommendations](#)

Webpack with Module Federation

covering almost all micro-frontend use cases, [Use Cases](#)

WebSockets, [API Integration and Micro-Frontends](#)

using with micro-frontends, [WebSocket and Micro-Frontends](#)

window object

shared dependencies added to, [Dependency Management](#)

window.__FEDERATION__.__SHARE__ object, [Home Micro-Frontend](#)

working backward, [Working Backward](#)

wrappers, [Embedding a Legacy Application](#)

X

X (Twitter)

automation pipelines scaling alongside codebase, [Monorepo](#)

XML

ESI markup language based on, [Edge-Side Composition](#)

treating HTML entry point document as XML, [Composing micro-frontends](#)

XSS (cross-site scripting), [Integrating Authentication in Micro-Frontends](#)

Z

Zalando, use of micro-frontends, [Zalando](#)

Zephyr Cloud, micro-frontend service discovery implementation, [Availability in the Market](#)

zones (Next.js multi-zones), [Next.js Multi-Zones](#)

About the Author

Luca Mezzalira has been associated with the industry since 2004 and has lent his expertise predominantly in the solution architecture field. He earned accolades for revolutionizing the scalability of frontend architectures with micro-frontends, from increasing the efficiency of workflows to delivering quality in products. Luca is known as an excellent communicator who believes in using an interactive approach for understanding and solving problems of varied scopes.

As Principal Serverless Specialist Solutions Architect at AWS, he helps customers to design and implement serverless workloads efficiently. He also shares with the community best practices to develop cloud-native architectures solving technical and organizational challenges in his social accounts.

Colophon

The animal on the cover of *Building Micro-Frontends* is the Jamaican tody (*Todus todus*). It is one of five species making up the *Todus* genus of birds, which are endemic to the Greater Antilles in the Caribbean. The Jamaican tody can only be found on the island of Jamaica, where it is commonly known as the rasta bird, the robin, and the robin red breast. It lives primarily in forested areas across the island.

The Jamaican tody is small and vibrantly colored, with a green head, red throat and bill, and green-white or yellow-white breast. It is about 4.25 inches (9 cm) in size, with an average weight of 6.4 grams and a wingspan of around 1.8 inches (4.6 cm).

Jamaican todies typically travel in pairs and are most conspicuous in the spring and summer breeding months, when they may be spotted performing wing-rattling or courtship feeding behaviors. Their characteristic vocalizations include a loud beep sound and throat-rattle. Jamaican todies nest and lay their eggs in burrows dug in the soil. They feed almost entirely on insects and their larvae but occasionally consume fruit as well.

The current conservation status of the Jamaican tody is of Least Concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from Lydekker's *Royal Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.