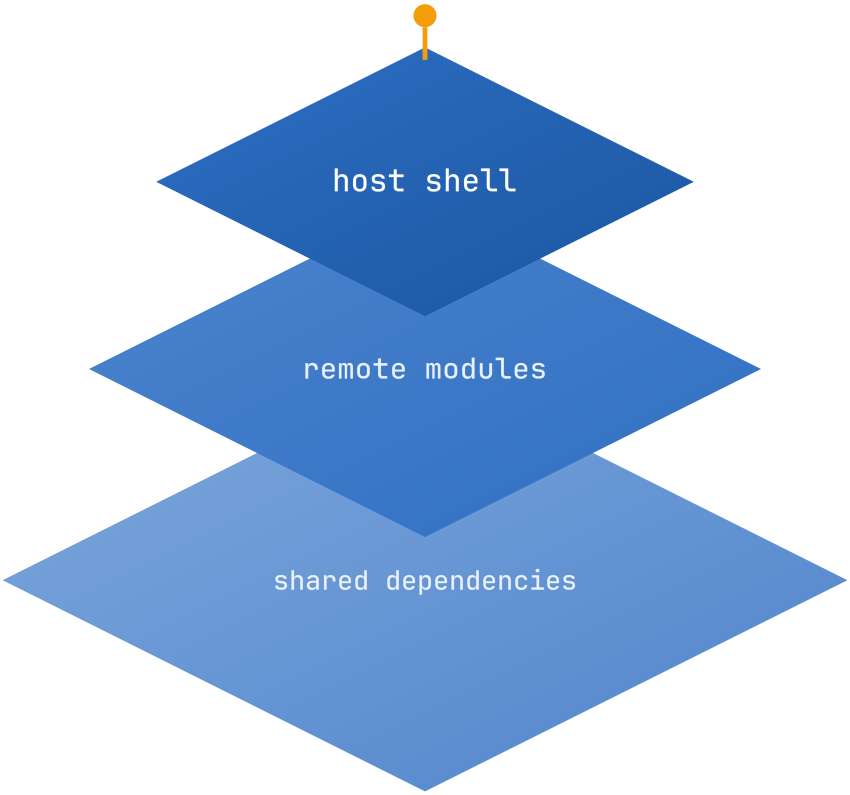

MICRO FRONTENDS WITH

Webpack 5 Module Federation

From Zero to Your First Production MFE

A hands-on guide to building, sharing, and shipping independently deployable React applications.

11 Chapters | Guided Project | Production-Ready

Oluwakemi Oluwadahunsi
Software Engineer & Technical Author
kemi-oluwadahunsi.com

Micro Frontends with Webpack 5 Module Federation

From Zero to Your First Production MFE

Book 1 of 2 · First Edition, 2026

WRITTEN BY

Oluwakemi Oluwadahunsi kemi-oluwadahunsi.com

Copyright Notice

Disclaimer

This book is provided on an 'as is' basis. While every reasonable effort has been made to ensure accuracy, the author makes no warranties regarding its use. Technology evolves rapidly; package versions, APIs, and tooling may have changed since publication, so always consult the official documentation for the most current guidance.

Trademarks

React, Webpack, Next.js, Vue, Angular, and all other product names are trademarks or registered trademarks of their respective owners. Their use here is for identification only and does not imply endorsement or affiliation.

Permissions and Contact

For permissions requests, bulk licensing, speaking engagements, or other enquiries, reach the author at kemi-oluwadahunsi.com.

Table of Contents

Book 1 · From Zero to Your First Production MFE · 4 Parts · 11 Chapters

02 Webpack 5 and Module Federation

Chapters 4 - 6

p. 34 - 95

CHAPTER 4

Webpack 5 Essentials for MFE Development

34 page

Everything you need to know about Webpack 5 before touching Module Federation. Covers the module system, loaders, plugins, chunk splitting, and the parts of the config that matter most for MFEs.

CHAPTER 5

Module Federation: Your First Setup

46 page

Step-by-step: configure a Host application that consumes a Remote, expose components, load them dynamically, and understand exactly what happens at runtime. Your first working federated app.

CHAPTER 6

Shared Dependencies and Version Management

77 page

How to share React, ReactDOM, and other libraries across remotes without loading them multiple times. Version mismatch strategies, singleton enforcement, and avoiding the most common dependency pitfalls.

03 Building with React

Chapters 7 - 9

p. 96 - 151

CHAPTER 7 React Integration in a Federated Architecture

96 page

How to structure React applications for Module Federation. Bootstrapping correctly, lazy loading federated components, handling context across boundaries, and the patterns that prevent most common React MFE bugs.

CHAPTER 8 Routing and Navigation Across MFEs

119 page

Client-side routing in a federated app is trickier than it looks. Covers host-owned routing, memory routing in remotes, deep linking, and keeping navigation consistent across team boundaries.

CHAPTER 9 State Management and MFE Communication

135 page

How to share state and communicate between independent MFEs without creating tight coupling. Covers shared stores, custom events, pub/sub, URL as state, and when to use each pattern.

04 Ship It

Chapters 10 - 11

p. 152 - 197

CHAPTER 10

Error Handling, Fallbacks, and Loading States

152 page

Remote applications fail. Networks are slow. This chapter teaches you how to build resilient federated applications that handle remote failures gracefully with Error Boundaries, Suspense, and intelligent fallback strategies.

CHAPTER 11

Guided Project: Build and Deploy Your First Production MFE

170 page

Everything comes together. Build a complete multi-team e-commerce shell with a Product Catalog remote and a Cart remote. Configure shared deps, routing, state communication, error handling, and deploy to production.

How to Use This Book

This book is designed to be read in order. Each chapter builds directly on the one before it — the Webpack config from Chapter 4 is extended in Chapter 5, the shared dependency strategy from Chapter 6 is applied in every chapter that follows, and the resilience stack from Chapter 10 is wired into the guided project in Chapter 11.

That said, if you are already comfortable with Webpack 5 and Module Federation basics, Chapters 1–4 can be read quickly as a foundations review and you can move faster from Chapter 5 onward.

The three reading modes

- **Cover to cover.** The recommended path if you are new to MFEs or to Webpack 5 Module Federation. Follow every chapter, type the code, and complete the guided project in Chapter 11.
- **Project-first.** Start with Chapter 11 to see the complete picture, then work backwards through the chapters that cover patterns you want to understand more deeply. Every section in Chapter 11 references back to the chapter where the pattern was introduced.
- **Reference mode.** Use the Table of Contents section numbers to jump directly to a specific topic — shared config (6.2), routing (Chapter 8), Error Boundaries (10.2), circuit breakers (10.5). Each section is self-contained enough to read in isolation.

Code examples

Every code example in this book is production-relevant. There are no toy snippets written purely for illustration — each figure shows code you can use directly in a real project, or adapt with minimal changes.

Code figures are labelled with a filename that tells you exactly where the file lives in the project structure. When a figure continues a pattern from a previous chapter, a callout note references the earlier section so you can trace the full evolution.

> **The guided project is the proof**
>
> Chapter 11 is not optional. Every pattern introduced in Chapters 1–10 is applied in the guided project. If a concept felt abstract earlier, Chapter 11 is where it becomes concrete. The complete project is also available in the companion repository linked in Section 11.8.

Callout boxes

- **Blue boxes** contain background context, architectural notes, and important clarifications that expand on the main text.
- **Yellow boxes** flag common mistakes, subtle gotchas, and things that look right but behave unexpectedly in production.
- **Red boxes** mark hard rules — things you must never do in a federated application, with an explanation of exactly why.
- **Green boxes** highlight best practices and production-ready patterns you can apply immediately.

> **Book 2 — At Scale: Advanced Patterns and Production Mastery**
>
> This book ends with a working, deployed MFE application. Book 2 picks up from there and covers the patterns that become essential at team scale: server-side rendering, shared design systems, contract testing, CI/CD pipelines for independent remotes, and the organisational architecture of a mature MFE platform.

Prerequisites and Setup Guide

This book assumes you are a working JavaScript or TypeScript developer with React experience. You do not need prior knowledge of Webpack or Module Federation — both are covered from first principles starting in Chapter 4.

Knowledge prerequisites

Required:

- React function components, hooks (`useState`, `useEffect`, `useContext`), and JSX
- JavaScript ES modules — `import`/`export` syntax, dynamic `import()` calls
- npm or yarn: installing packages, running scripts, reading `package.json`
- Basic command-line usage: navigating directories, running `npm start` and `npm run build`
- Git basics: cloning a repository, creating branches

Helpful but not required:

- Prior exposure to Webpack (any version) — Chapter 4 covers everything you need from scratch
- TypeScript — all examples are in JavaScript; TypeScript extensions are noted where relevant
- react-router-dom v6 — a brief recap is included in Chapter 8

System setup

Figure 0.1 — Required tools and minimum versions

Tool	Minimum version	Notes
Node.js	18.0.0 LTS	Use nvm or fnm to manage versions
npm	9.0.0	Ships with Node 18. Yarn 3+ also works
A code editor	Any	VS Code recommended — Webpack config syntax highlighting
A modern browser	Chrome / Firefox	DevTools console required for debugging shared scope

Verifying your environment

Run the following in your terminal to confirm Node.js and npm are correctly installed before starting Chapter 4:

```
node --version
# → v18.x.x or higher
npm --version
# → 9.x.x or higher
```

Project structure overview

Each chapter that involves code creates files in a self-contained project directory. By Chapter 11 the full structure is:

- `shell-host/` — the Host application, port 3000
- `catalogue-remote/` — the Catalogue Remote, port 3001
- `cart-remote/` — the Cart Remote, port 3002

Each application is fully independent

Every directory has its own `package.json`, its own Webpack config, and its own dev server. They share nothing on disk. Module Federation creates the runtime connections between them.

You can start any application in isolation with `npm start` inside its directory. You do not need all three running to work on any individual chapter.

PART ONE

CHAPTER 1

01

The Problem with Monolithic Frontends

Why large frontend codebases eventually collapse under their own weight

WHAT YOU'LL LEARN IN THIS CHAPTER

- Why monolithic frontends work well at first and why that changes
- The exact pain points that force teams to look for something better
- How deployment bottlenecks and team collisions develop over time
- The coupling trap that makes even small changes feel dangerous
- How backend teams solved the same problem with microservices

1.1 The Beginning Is Always Fine

Every frontend application starts the same way: clean, manageable, and easy to understand. You create a new React or Vue project, write your first components, and everything feels wonderfully simple. One codebase. One team. One deployment. Life is good.

This is the monolithic frontend. In its early days, it is genuinely the right choice. There is nothing wrong with a monolith for a small project or a small team. It is fast to set up, easy to reason about, and simple to deploy. You change a file, run a build, and push it live.

> **What is a Monolithic Frontend?**
>
> A monolithic frontend is a single, unified codebase that contains all the UI, logic, and features of your entire application. Everything lives together, is built together, tested together, and deployed together as one unit.
>
> Think of it like a single large building where every department shares the same floor. It works well when the company is small, but becomes chaotic as more teams move in.

The problems do not arrive on Day 1. They arrive gradually, as the team grows, as the product expands, and as more features are added on top of existing ones. What was once a well-structured codebase starts to feel heavier, slower, and more fragile with every passing sprint.

To understand why this happens, let us walk through the lifecycle of a typical monolithic frontend, from its clean beginning to the moment teams start searching for a better way.

1.2 The Scaling Wall

Imagine you are working at a growing e-commerce company. When you joined, there were five frontend developers. Now there are thirty-five, split across five teams: the Product Catalog team, the Shopping Cart team, the Checkout team, the User Profile team, and the Search team.

All five teams work in the same codebase. Every morning, developers pull the latest code and immediately run into merge conflicts. The Checkout team changed a shared utility function. The Cart team refactored a styling helper. The Search team updated a shared component. Now everyone is blocked, untangling each other's work before they can even start their own.

> **The Real Cost of Merge Conflicts**
>
> Merge conflicts are not just annoying. They are expensive. A study by GitClear found that developers spend an average of 13.5% of their working time resolving merge conflicts and integration issues in large shared codebases. In a team of 35 developers, that is nearly 5 full-time developers doing nothing but untangling other people's code.

This is what engineers call **the scaling wall**. It is the point at which adding more developers to a project starts slowing it down instead of speeding it up. It seems counterintuitive, but it is one of the most predictable problems in software engineering. The more people share the same code, the more coordination is required, and coordination does not scale linearly.

The monolithic frontend creates three specific coordination problems that compound as the team grows:

1. **Shared ownership confusion.** When everyone owns the whole codebase, no one truly owns any of it. Decisions slow down because every change needs buy-in from multiple teams. Nobody wants to refactor code they did not write when others will be affected.
2. **Context switching overhead.** A developer working on the Checkout feature now needs to understand the Cart code, the Shared utility library, the global state management setup, and the build configuration, all to ship one small feature.
3. **Communication explosion.** As teams grow, the number of communication channels required grows exponentially. With 5 teams sharing a codebase, the potential for misunderstanding and conflicting decisions grows with every new member added.

Figure 1.1 — Team coordination in a monolithic frontend

Team	What They Own	What They Share	Risk
Product Catalog	Catalog pages	Entire codebase	**HIGH**
Shopping Cart	Cart sidebar	Entire codebase	**HIGH**
Checkout	Checkout flow	Entire codebase	**HIGH**
User Profile	Profile pages	Entire codebase	**HIGH**
Search	Search UI	Entire codebase	**HIGH**

When every team shares the entire codebase, every team is exposed to every other team's risk. A bug introduced by the Search team can block the Checkout team's release.

1.3 Deployment Bottlenecks and the Release Train

Here is a scenario that every developer in a large monolith has experienced at least once, often on a Friday afternoon.

The Checkout team has a critical bug fix ready. It is tested, reviewed, and approved. They are ready to deploy. But they cannot. The Shopping Cart team has an unfinished feature partially committed to the main branch. The Search team has a failing test in a component that is not even related to Checkout. And the build pipeline is already running for an unrelated configuration change.

What is the Release Train Problem?

In a monolithic frontend, all teams must deploy together because it is one application. This creates a 'release train', a scheduled deployment window that all teams must coordinate around. If any team's code is not ready, everyone waits. If any team introduces a bug, everyone's release is at risk.

Teams that move fast get penalised by teams that move slow. Teams that work carefully get blocked by teams that are reckless. Everyone is dependent on everyone else's readiness.

The release train problem produces a few particularly painful consequences that experienced developers know well:

- **Feature flags proliferate.** Teams start wrapping unfinished work in feature flags just to merge without blocking others. Over time, the codebase fills with half-finished features guarded by conditional logic that nobody is confident is safe to remove.
- **Deployments become events.** Instead of a routine operation, deploying becomes a coordination meeting. Someone has to check with all teams, verify readiness, watch the pipeline, and stay on call if anything breaks.
- **Hotfix complexity explodes.** When a production bug needs an immediate fix, you cannot just push the fix. You must carefully cherry-pick only that fix from the main branch, excluding all other changes that have accumulated since the last release.
- **Release frequency drops.** The overhead of each deployment grows so large that teams start bundling more changes into each release. Releases become bigger, riskier, and less frequent, which is the opposite of good software delivery.

The irony is that teams trying to move faster end up moving slower. Each attempt to ship quickly creates coordination debt that slows the next release even further.

1.4 The Coupling Trap

Of all the problems with monolithic frontends, the coupling trap causes the most production incidents and the most developer anxiety. Here is a concrete example of how it works.

Suppose you need to update the button component used in the navigation bar. It seems trivial: change the border radius and the hover colour. Two properties. One component. Should take ten minutes.

But then you discover that this Button component is used in 47 different places across the application. It is used in the checkout flow, the product listing, the user profile, the admin panel, and even the marketing landing pages. It is used with different variants, sizes, and event handlers, some of which depend on the exact current styling to function correctly in their surrounding layout.

What is Tight Coupling?

Tight coupling means that components, modules, or teams are so intertwined that a change in one place has unpredictable effects in many other places. In a monolithic frontend, tight coupling accumulates naturally over time because it is always faster in the short term to reuse what already exists than to create proper abstractions.

The technical debt from tight coupling does not just slow you down. It makes the system fragile. Changes become dangerous, and fear of breaking things leads to the worst outcome of all: teams stop refactoring altogether.

Figure 1.2 — A shared component with cross-team dependencies

src/components/shared/Button.jsx

```
// src/components/shared/Button.jsx — used in 47 places
import { useGlobalTheme } from '../../store/themeStore';
import { trackEvent }     from '../../analytics/tracker';
import { checkPermission} from '../../auth/permissionChecker';

export function Button({ label, onClick, variant }) {
  const theme = useGlobalTheme();
  if (!checkPermission('button:click')) return null;
  return <button onClick={() => { trackEvent(label); onClick(); }}
    className={`btn--${variant} theme--${theme.mode}`}>{label}</button>;
}
```

The figure above shows a Button component that belongs to no single team. It imports directly from three different team's codebases: the theme store, the auth checker, and the analytics tracker. If the Auth team refactors `checkPermission` to be asynchronous, what happens to the Button? And what happens to all 47 places across the app that use the Button?

This is not an edge case. This is Tuesday in a large monolithic codebase.

The Shotgun Surgery Anti-Pattern

When a single change requires you to touch many different files across many parts of the codebase, it is called 'shotgun surgery', a term coined by Martin Fowler. It is one of the most common and costly code smells in large monolithic applications, and it is almost impossible to avoid entirely in a codebase that many teams share over many years.

1.5 What Actually Breaks in Production

So far we have talked about developer experience problems: merge conflicts, slow deployments, and frightening refactors. But these problems eventually manifest as real production incidents that affect real users.

Here are the most common ways tight coupling and monolithic architecture create production failures:

1. **The cascading failure.** One team ships a bug in a shared component used across the entire application. The entire app is affected. A bug in the Search bar UI can bring down the Checkout flow if they share state management.
2. **The broken build.** One team pushes code that breaks the TypeScript compilation or a shared test suite. All other teams are blocked from deploying until the build is fixed, even teams that have nothing to do with the broken code.
3. **The stale cache problem.** Because everything is deployed together, a single cache-busting event invalidates every cached asset simultaneously. Users experience a slow load time after every deployment, even for features they use that have not changed at all.
4. **The impossible rollback.** A critical bug reaches production. The team needs to roll back. But rolling back reverts to the last deployment, which may undo three other teams' work that was shipped at the same time.

A Real-World Example: The Amazon Story

Amazon's engineering teams reached a point in the early 2000s where deploying their monolithic application had become so painful that they could only do it a few times per year. Jeff Bezos issued his famous 'two-pizza team' directive, requiring every team to expose their functionality as an independent service. This became the origin of what we now call microservices, and the same thinking now applies directly to the frontend.

1.6 The Backend Solved This First

If all of this sounds familiar, it should. The backend world faced exactly the same problem and solved it over a decade ago with **microservices architecture**.

Instead of one giant backend application handling everything from user auth to order processing to payment handling, microservices split the backend into small, independently deployable services. Each service has its own codebase, its own deployment pipeline, its own database, and its own team.

The result was transformational. Teams could deploy their services independently, scale them independently, and build in different programming languages where that was the right choice. A bug in the Notification Service could no longer take down the entire checkout process.

Frontend teams looked at this and asked the obvious question: can we do the same thing for the UI?

Figure 1.3 — Backend microservices vs Micro Frontend parallels

Backend Microservices	Micro Frontends
User Service	User Profile MFE
Product Service	Product Catalog MFE
Order Service	Shopping Cart MFE
Payment Service	Checkout MFE

Search Service	Search MFE
API Gateway	Shell Application (Host)

Every major concept from backend microservices has a direct parallel in Micro Frontend architecture.

The analogy is not perfect. The frontend and backend have different constraints, different user-facing concerns, and different tooling. But the core principle transfers almost directly:

> **The Core Principle**
>
> **Split your application along team boundaries. Give each team independent ownership of their slice. Let them build, test, and deploy on their own schedule, without needing permission or coordination from the rest of the organisation.**
>
> This is the foundational idea behind Micro Frontends. It is the answer to every problem we have discussed in this chapter.

1.7 Signs Your Team Is Hitting the Monolith Wall

Micro Frontends are not the right solution for every team or every project. They introduce real complexity that smaller teams should not take on. But if several of the following signs describe your current situation, it is worth taking the idea seriously.

- Deployments require coordination meetings with multiple teams before they can happen.
- Developers regularly block each other with merge conflicts in shared files or shared components.
- A change to one part of the application frequently causes unexpected breakages in unrelated parts.
- The build time has grown to the point where running tests locally takes more than a few minutes.
- Teams feel ownership of features but not of their code, because the code does not truly belong to any one team.

- Onboarding a new developer takes weeks because the codebase is too large to understand holistically.
- The frontend team has grown to more than 8 to 10 developers working in the same repository simultaneously.

> **The Honest Counter-Argument**
>
> If your team is small (under 5 developers), your product is early-stage, or your application does not have clearly separable feature areas, a monolith is probably still the right choice. Micro Frontends solve organisational scale problems. If you do not have that problem, you are trading simplicity for complexity with no benefit.
>
> The best engineers choose tools that match the problem. Keep this in mind throughout this book.

Chapter Summary

- Monolithic frontends work well for small teams and early-stage products, but they accumulate problems as teams and codebases grow.
- The scaling wall hits when adding developers slows the team down due to coordination overhead, merge conflicts, and shared ownership confusion.
- The release train problem means all teams are coupled to each other's readiness. One team's bug blocks everyone's deployment.
- Tight coupling means that changing one component can have ripple effects across the entire application, making every change feel risky.
- Production failures in monolithic frontends tend to be wide. One bug can affect features far removed from where the change was made.
- The backend world solved the same problems with microservices. Micro Frontends apply the same thinking to the UI layer.
- Micro Frontends are not right for every team. They solve organisational scale problems and introduce complexity that smaller teams should avoid.

Up Next: Chapter 02 — Introducing Micro Frontends

Now that we understand exactly what goes wrong with monolithic frontends, Chapter 2 introduces Micro Frontends as the solution. We cover what they are, how they work conceptually, and what makes a well-designed MFE system different from simply splitting a monolith into pieces.

PART ONE

CHAPTER 2

02

Introducing Micro Frontends

What they are, how they work, and what you give up to get there

WHAT YOU'LL LEARN IN THIS CHAPTER

- The precise definition of a Micro Frontend and what makes it different
- The three core principles every MFE must uphold: ownership, isolation, independence
- How teams are structured vertically around features, not layers
- The real trade-offs — bundle size, complexity, and operational overhead
- How MFEs compare to monorepos and why they are not the same thing
- A simple framework for deciding whether MFEs are right for your team

In Chapter 1 we tore down the monolith. We watched it scale past its breaking point, saw deployment pipelines grind to a halt, and traced the coupling trap from a seemingly trivial button change all the way to a production incident. If you came away from that chapter with the feeling that there has to be a better way, you were right.

This chapter introduces that better way. But before we can build anything, we need to be precise about what a Micro Frontend actually is. The term gets used loosely, often to describe anything from a monorepo to a distributed bundle, and that vagueness causes real problems on real teams. By the end of this chapter you will have a clear, working definition, a set of principles to test any architecture against, and an honest picture of the costs involved.

> **Before You Continue**
>
> This chapter is intentionally conceptual. We are not writing any Webpack config or Module Federation code yet. That comes in Chapter 4. The goal here is to build the mental model that makes all of that configuration meaningful rather than just mechanical.
>
> If you are tempted to skip ahead to the code, resist it. The teams that struggle most with MFEs are usually the ones who started with the tooling before they understood the architecture.

2.1 Defining Micro Frontends

A Micro Frontend is a self-contained, independently deployable slice of a frontend application that is owned end-to-end by a single team.

Every word in that definition matters. Let us unpack each one.

Self-contained means the MFE encapsulates its own UI, its own logic, and its own data fetching. It does not reach into another team's code to get what it needs. It has a defined contract with the outside world, and everything inside that boundary is its own business.

Independently deployable means the team owning this MFE can ship a new version without coordinating with any other team, without running a shared release pipeline, and without touching a shared codebase. This is the property that solves the deployment

bottleneck from Chapter 1.

Owned end-to-end by a single team means one team is responsible for the full vertical slice: the UI, the API calls, and the business logic. There is no separate UI team, no separate API team, no separate testing team. The same people who build it are the ones who run it in production.

> **What is a Micro Frontend?**
>
> A Micro Frontend is a self-contained, independently deployable slice of a frontend application that is owned end-to-end by a single team. It communicates with other MFEs through well-defined contracts rather than shared code, and it can be built, tested, and deployed without any other team's involvement.
>
> The key word is *independently*. If deploying your MFE still requires coordinating with another team, you do not yet have a Micro Frontend. You have a module in a distributed monolith.

It is also worth being clear about what a Micro Frontend is not. An MFE is not simply a component. Components live inside a codebase and are deployed as part of the larger application. An MFE is deployed on its own, lives at its own URL or entry point, and can be updated without the host application knowing anything about the change beyond the contract.

An MFE is also not a microservice. Microservices live on the server. Micro Frontends live in the browser. They share the same philosophy but operate on completely different layers of the stack. You can have one without the other, though in practice many teams adopt both together.

2.2 Core Principles: Ownership, Isolation, Independence

Every successful Micro Frontend architecture is built on three non-negotiable principles. Break any one of them and you will eventually end up recreating the problems you were trying to escape.

Principle 1: Ownership

Every MFE has a single team that owns it completely. That team makes all the decisions about its internal implementation: which libraries to use, how to structure state, how to write tests. No one else has the authority to change those choices without that team's agreement.

Ownership is not just about code. It is about operational accountability. The team that builds the MFE also monitors it in production, responds to its incidents, and is responsible for its performance. This tight loop between building and running is what drives quality. It is very hard to cut corners on observability when you are the one getting paged at 2am.

Principle 2: Isolation

An MFE must not leak its internal state, styles, or event handlers into the surrounding application. CSS from one MFE should not bleed into another. A global event listener in one MFE should not intercept events intended for another. A JavaScript error in one MFE should not crash the entire shell.

Isolation is what allows teams to work in parallel without stepping on each other. Without it, you are back to the coordination overhead of the monolith, just with more moving parts.

> **Isolation Is Not Free**
>
> Achieving isolation requires deliberate effort. You need CSS scoping strategies (CSS Modules, Shadow DOM, or a strict naming convention), JavaScript sandboxing (Error Boundaries for React), and explicit contracts for cross-MFE communication. None of this is difficult, but it does not happen automatically. Chapters 7 through 9 cover each of these in detail.
>
> The teams who skip isolation work early almost always pay for it later when a style from the Cart team starts affecting the layout of the Header team.

Principle 3: Independence

Independence means the MFE can be built, tested, and deployed without any other MFE being involved. The team has a pipeline that deploys only their code. They can do it on a Tuesday afternoon without sending a message in Slack first.

Independence is the hardest principle to maintain as an organisation grows. Shared dependencies, shared design systems, and shared infrastructure all create subtle couplings that erode independence over time. We will cover how to manage all of these in Parts 2 and 3 of this book.

Figure 2.1 — The three core MFE principles and what violates them

Principle	What it looks like in practice	Common violation
Ownership	One team, full vertical slice, on-call responsibility	Multiple teams committing to the same remote
Isolation	Scoped styles, Error Boundaries, no global state leakage	Global CSS classes shared across MFEs
Independence	Single-team pipeline, no cross-team release gates	Shared deploy script that runs all remotes together

2.3 Team Topologies and Vertical Slicing

The organisational structure of a team that builds MFEs looks fundamentally different from a traditional layered team. Understanding this difference is important because the architecture you build will reflect the team that built it. This is Conway's Law in practice.

In a traditional frontend organisation, teams are structured horizontally by technical layer. There is a Design team, a Frontend team, a Backend team, and a DevOps team. A feature that needs a new button, a new API endpoint, and a new deployment configuration has to pass through all four teams sequentially. Every handoff is a potential delay and a potential miscommunication.

In an MFE organisation, teams are structured vertically by business domain or user journey. The Checkout team owns the checkout UI, the checkout API, the checkout database schema, and the checkout deployment pipeline. They can ship a new feature from idea to production without asking anyone else for permission or resources.

> **Conway's Law and Your Architecture**
>
> Melvin Conway observed in 1967 that organisations inevitably produce designs that mirror their communication structure. If your engineering organisation is divided into a UI team and an API team, your system will have a hard boundary between UI and API, regardless of what your architecture diagrams say.
>
> This is why team structure is not a soft concern in MFE architecture. It is a technical decision. Before you split your codebase, consider whether your teams are actually structured to own independent verticals. If they are not, the architecture will fight you.

Vertical slicing means cutting through all technical layers to deliver a complete user-facing feature. A vertically sliced team looks something like this:

- A product designer embedded in or closely aligned with the team
- Two to four frontend engineers who own the remote application
- One to two backend engineers who own the APIs that remote consumes
- Shared or embedded DevOps capability to own the deployment pipeline
- A product owner who sets direction and prioritises the backlog

This structure is sometimes called a stream-aligned team in the Team Topologies framework coined by Matthew Skelton and Manuel Pais. The key characteristic is that the team has everything it needs to deliver value to users without depending on another team's schedule.

2.4 Trade-offs You Must Understand Up Front

Micro Frontends solve the problems described in Chapter 1 well. But they introduce new problems in exchange. A good architect does not pretend these do not exist. They understand them clearly and plan for them deliberately.

Payload and Bundle Duplication

In a monolithic frontend, React is loaded once and shared across the entire application. In a naive MFE setup, every remote could load its own copy of React. For a user on a slow

mobile connection, this is the difference between a fast application and one that never fully loads.

Module Federation solves this with the `shared` configuration key, which tells Webpack to load a dependency only once and share it across all remotes. We cover this in detail in Chapter 6. For now, know that bundle duplication is a solvable problem, but it requires deliberate configuration. It does not solve itself.

Operational Complexity

A monolithic frontend is one deployment. Three MFEs are three deployments, three CI pipelines, three monitoring dashboards, and three sets of environment variables to keep in sync. At ten MFEs, the operational surface area is significant.

This is not a reason to avoid MFEs, but it is a reason to invest in platform tooling before you scale. Teams that jump to ten MFEs before they have a solid deployment and observability platform almost always end up with a system that is harder to operate than the monolith it replaced.

Initial Development Overhead

The first time you set up Module Federation, configure shared dependencies, establish routing conventions, and agree on a cross-MFE communication contract, it will take longer than it would to just build a new feature in the monolith. This is the setup tax, and it is real.

The setup tax pays for itself. The teams that absorb it early are the ones who can ship independently six months later while the monolith teams are still waiting for a release window. But you should budget for it honestly rather than pretending it does not exist.

The Trade-off Summary

You gain: team autonomy, independent deployments, parallel development, isolated blast radius for failures, and the ability to scale different parts of the frontend independently.

You give up: simplicity of a single codebase, cheap cross-cutting refactors, trivial shared state, and low operational overhead.

MFEs make large teams fast. They make small teams slower. The inflection point is usually around three to four teams working on the same frontend surface area.

2.5 MFEs vs Monorepos vs Monoliths

One of the most common sources of confusion in conversations about Micro Frontends is the conflation of code organisation with deployment strategy. These are separate concerns, and mixing them up leads to architectural decisions based on misunderstandings.

A **monolith** is a single deployable unit. All the code is deployed together, all at once. In a frontend context, a monolith means one build process, one output bundle, and one deployment.

A **monorepo** is a code organisation strategy. It means keeping multiple projects, packages, or applications in a single version control repository. A monorepo can contain a monolith, multiple independently deployed MFEs, a design system package, and a shared utility library, all in the same repo. The monorepo says nothing about how the code is deployed.

A **Micro Frontend** is a deployment strategy. It means the frontend is split into independently deployable units. You can implement MFEs from a monorepo. Many successful MFE architectures do exactly that, using the monorepo for shared tooling and design system packages while each MFE has its own independent deployment pipeline.

Figure 2.2 — Code organisation vs deployment strategy

	Single repo	Monorepo	Polyrepo
Monolith	Common	Possible	Rare
MFEs	Uncommon	Very common	Also common

The important takeaway is that choosing a monorepo does not mean you have Micro Frontends, and choosing Micro Frontends does not require a polyrepo. The deployment strategy and the code organisation strategy are independent decisions that happen to interact. Choose each one on its own merits.

2.6 Is This the Right Choice for Your Team?

Micro Frontends are not the right choice for every team in every situation. Applying them where they do not fit is a common and costly mistake. Here is a simple framework for making the decision honestly.

The core question is not technical. It is organisational: **are you suffering from team coordination overhead on your frontend?** If two or more teams are blocked from shipping because they share a codebase and a release pipeline, Micro Frontends are likely the right tool. If a single team owns the entire frontend and ships without coordination pain, they are probably not.

Signs that MFEs are the right fit:

- Multiple teams have commits in the same frontend repository and regularly block each other
- Your release pipeline requires sign-off or testing across teams before any deployment can happen
- Different parts of your frontend have genuinely different release cadences (e.g. the marketing pages ship daily, the checkout ships weekly)
- Teams have meaningfully different technology preferences and forcing a single stack creates friction

- Post-mortems regularly show that one team's change broke another team's feature

Signs that MFEs are probably not the right fit yet:

- A single team owns the entire frontend, and coordination is not a problem
- The product is early-stage and the architecture is still changing rapidly
- The team does not yet have solid CI/CD, monitoring, or observability in place
- The frontend codebase is small enough that one engineer can understand all of it in a day

> **The Goldilocks Zone**
>
> MFEs are most valuable when a product is past early-stage chaos but has not yet hit the scale of a full platform engineering investment. If you have two or three teams shipping features on the same frontend and your release process is becoming a bottleneck, that is the ideal moment to introduce MFEs.
>
> Too early, and you pay the setup tax before you need the benefits. Too late, and you are trying to extract MFEs from a deeply coupled monolith while simultaneously shipping product. Neither is insurmountable, but both are harder than getting the timing right.

If you have read through the signs above and the case for MFEs is clear for your team, the rest of this book is your step-by-step guide to building them. If the case is not yet clear, finishing Part 1 will give you the architectural context to revisit the decision with more information.

Chapter Summary

- A Micro Frontend is a self-contained, independently deployable slice of a frontend application owned end-to-end by a single team.
- The three non-negotiable principles are ownership (one team, full accountability), isolation (no CSS leakage, scoped state, Error Boundaries), and independence (deploy without coordinating with other teams).
- Teams in an MFE organisation are structured vertically by business domain, not horizontally by technical layer. Conway's Law ensures this structure will be reflected in the architecture.
- MFEs trade simplicity and cheap cross-cutting refactors for team autonomy, independent deployments, and isolated failure blast radius. Both sides of that trade are real and must be planned for.
- A monorepo is a code organisation strategy. A Micro Frontend is a deployment strategy. They are independent decisions that can and often do coexist.
- MFEs are the right choice when coordination overhead between teams on a shared frontend is already causing pain. They are not the right choice for a single team on an early-stage product.

Up Next: Chapter 3 — Architecture Patterns and When to Use Each

Now that you know what a Micro Frontend is and how teams are organised around them, Chapter 3 maps out the major integration patterns available to you: iframes, Web Components, JavaScript bundles, and Module Federation. You will come away with a clear decision framework for choosing the right pattern for your specific context before a single line of Webpack config is written.

PART ONE

CHAPTER 3

03

Architecture Patterns and When to Use Each

iFrames, Web Components, JavaScript bundles, and Module Federation

WHAT YOU'LL LEARN IN THIS CHAPTER

- The difference between build-time and run-time integration — and why it matters
- How iFrames work as an MFE integration layer and when they are the right call
- What Web Components offer and the limitations you need to plan around
- How JavaScript bundle integration works without a framework
- Why Module Federation is the dominant pattern for React MFEs today
- A practical decision framework for choosing the right pattern for your context

You now know what a Micro Frontend is, what principles it must uphold, and how teams organise around them. The next question is a practical one: how do you actually connect a collection of independently deployed applications into a coherent user experience?

There is no single answer. There are four major integration patterns, each with different strengths, different limitations, and different contexts where they shine. Understanding all of them — even the ones you will not use — gives you the vocabulary to make the right choice for your situation and to explain that choice to your team.

This chapter maps each pattern clearly. By the end you will have a decision framework you can apply to your own architecture before a single line of Webpack configuration is written.

3.1 Integration at Build Time

Before we look at individual patterns, it is important to understand the fundamental split that separates them: the moment at which the integration between MFEs actually happens.

In **build-time integration**, MFEs are composed into the final application during the build process. One central build step pulls in code from multiple packages or repositories, compiles everything together, and produces a single deployable output. The individual MFEs do not exist as independent deployables in production; they are baked into one bundle.

Build-time integration is straightforward to set up and works naturally with standard tooling. But it reintroduces one of the core problems from Chapter 1: to deploy a change in any one MFE, you have to rebuild and redeploy the entire application. Independent deployment is impossible by definition.

Build-Time Integration: Use With Caution

Build-time integration is sometimes presented as a stepping stone toward 'real' MFEs. In practice it gives you the organisational complexity of separate packages with none of the deployment independence that makes MFEs valuable.

There are legitimate uses — shared component libraries, design tokens, and utility packages are all reasonable build-time integrations. But if you are building features that teams need to ship independently, build-time integration is not the right approach.

In **run-time integration**, MFEs are composed in the browser. The shell application loads remote applications dynamically at runtime, and the user sees the assembled experience without ever knowing it came from multiple separate deployments. This is where the patterns worth investing in live.

All three of the patterns we cover in sections 3.3 through 3.5 are run-time integration strategies. The remainder of this chapter and indeed the remainder of this book focuses entirely on run-time integration.

3.2 Integration at Run Time

Run-time integration means that when a user visits your application, the shell requests and loads remote applications dynamically. The key architectural concept is the **shell** (also called the host or container): a lightweight application whose primary job is to load, compose, and route between the independently deployed MFEs.

The shell does not contain feature code. It is infrastructure. It handles top-level routing, shared navigation, authentication state, and the orchestration of which MFEs to load for a given URL. Each MFE is a self-contained application that the shell loads on demand.

The Shell Application

The shell is the one piece of the architecture that teams do share. It is the entry point of the application and the integration layer. Because of this, the shell tends to be owned by a platform or infrastructure team rather than any individual product team.

Keep the shell thin. Every feature that lives in the shell is a feature that every team is coupled to. The shell should load MFEs and stay out of their way.

There are three primary mechanisms for how the shell loads and embeds a remote MFE at runtime: iFrames, Web Components, and JavaScript-based integration (of which Module Federation is the most powerful implementation). Each mechanism is a different answer to the same question: how does the shell know about, load, and render a remote application it did not know about at build time?

3.3 iFrames: The Simple Option

The iframe is the oldest and most isolated integration mechanism available on the web platform. Each MFE runs inside its own iframe, which is essentially a completely separate browser context embedded inside the shell page. The shell does not know anything about the MFE's internals, and the MFE cannot accidentally affect the shell's DOM, styles, or JavaScript context.

What iFrames do well:

- Complete isolation by default — CSS, JavaScript, and DOM are fully sandboxed
- Technology agnostic — the MFE inside can be built in any framework or language
- Security boundary — the browser enforces same-origin policies automatically
- Simple mental model — each MFE is just a URL that the shell loads

Where iFrames create problems:

- User experience breaks down at boundaries — scroll position, focus management, and keyboard navigation do not cross iframe boundaries naturally

- Routing is difficult to synchronise — deep linking into an iframed MFE requires explicit cross-frame communication via postMessage
- Performance overhead — each iframe loads a separate browser context, including its own copy of any shared libraries
- Accessibility limitations — screen readers and focus traps behave unpredictably across iframe boundaries
- Responsive layout is awkward — iframes have a fixed size by default and require JavaScript to resize dynamically

When iFrames Are the Right Choice

iFrames are well suited for genuinely isolated widgets that have minimal interaction with the surrounding page: embedded dashboards, third-party tools, payment forms, and isolated report viewers all work well as iframes.

If the embedded content needs to feel like a seamless part of a single application, shares navigation, or requires tight UI coordination with surrounding content, iFrames will fight you every step of the way.

Figure 3.1 — Minimal iframe-based shell integration

shell/src/App.jsx

```
// shell/src/App.jsx
// The shell loads each MFE as an iframe pointed at its own origin
export function App() {
  return (
    <main>
      <nav>...</nav>
      <iframe src="https://checkout.example.com" title="Checkout" />
      <iframe src="https://catalog.example.com"  title="Catalog"  />
    </main>
  );
}
```

3.4 Web Components

Web Components are a suite of native browser APIs that allow you to define custom HTML elements with encapsulated behaviour and styling. An MFE built as a Web Component exposes itself as a custom element — something like `<checkout-app></checkout-app>` — which the shell renders just like any other HTML element.

The key technology that enables style isolation in Web Components is the **Shadow DOM**: a separate, encapsulated DOM tree attached to a custom element. Styles defined inside a Shadow DOM do not leak out, and styles from the parent page do not bleed in. This is the strongest native CSS isolation available without an iframe.

What Web Components do well:

- Framework-agnostic contract — any framework can consume a custom element because it is standard HTML
- Native CSS isolation via Shadow DOM without build tooling
- Runs in the same browser context as the shell, enabling shared routing and seamless navigation
- Standards-based, no additional loader or plugin required

Where Web Components create friction:

- Shadow DOM and React have a historically difficult relationship — React's synthetic event system does not propagate through Shadow DOM boundaries correctly in all versions
- Server-side rendering support is limited — Shadow DOM is a browser API and does not have a universal SSR story
- Sharing state between the shell and a Web Component requires explicit attribute and event communication, which can become verbose
- Bundle size is not automatically shared — two Web Components using React will each load their own copy unless you manage sharing manually

Figure 3.2 — Registering and consuming a Web Component MFE

checkout-mfe/src/index.js

```
// checkout-mfe/src/index.js  — the MFE registers itself as a custom element
import { CheckoutApp } from './CheckoutApp';
class CheckoutElement extends HTMLElement {
  connectedCallback() {
    ReactDOM.render(<CheckoutApp />, this);
  }
}
customElements.define('checkout-app', CheckoutElement);

// shell/index.html  — the shell just uses the custom element tag
<script src="https://checkout.example.com/bundle.js"></script>
<checkout-app></checkout-app>
```

Web Components in a React-Dominant Stack

If your entire stack is React, Web Components add interface complexity without giving you much that Module Federation does not already provide. The framework-agnostic contract is most valuable when you genuinely have different frameworks on different remotes.

For a homogeneous React stack, Module Federation gives you better developer experience, automatic shared dependency management, and tighter TypeScript integration.

3.5 JavaScript / Module Federation

JavaScript-based integration is the most flexible and most capable approach to run-time MFE composition. At its simplest, it means loading a remote JavaScript bundle at runtime and executing it in the shell's browser context. Module Federation is Webpack 5's built-in, production-grade implementation of this pattern, and it is what the rest of this book is built on.

The core idea is straightforward: a Remote application exposes specific modules (components, utilities, stores) via Webpack configuration. A Host application declares

which remotes it wants to consume and from which URLs. At runtime, Webpack handles the loading, version negotiation of shared dependencies, and module resolution automatically.

What Module Federation does well:

- Shared dependencies are managed automatically — React loads once and is shared across all remotes, eliminating the bundle duplication problem
- Native TypeScript support — remote module types can be shared and consumed with full type safety
- Hot module replacement works across remotes in development
- No custom element wrapping required — remote React components are imported and used exactly like local components
- Lazy loading is built in — remotes are only fetched when actually needed

What Module Federation requires:

- Both the host and every remote must use Webpack 5 (or a compatible bundler with a Module Federation plugin)
- Shared dependency version ranges must be configured carefully — mismatches can cause silent runtime failures
- The remote URL must be known at runtime, which requires environment configuration to be handled correctly across environments

Figure 3.3 — Module Federation config for a Host and a Remote

webpack.config.js

```
// Remote: webpack.config.js — exposes a CheckoutApp component
new ModuleFederationPlugin({
  name: 'checkout',
  filename: 'remoteEntry.js',
  exposes: { './CheckoutApp': './src/CheckoutApp' },
  shared: { react: { singleton: true }, 'react-dom': { singleton: true } },
}),

// Host: webpack.config.js — declares the checkout remote
new ModuleFederationPlugin({
  name: 'shell',
  remotes: { checkout: 'checkout@https://checkout.example.com/remoteEntry.js' },
  shared: { react: { singleton: true }, 'react-dom': { singleton: true } },
}),
```

Do not worry if the configuration above looks unfamiliar right now. Every part of it is explained in detail in Chapters 4 and 5. The purpose here is to show the shape of the integration: the Remote declares what it exposes, the Host declares which remotes it consumes, and Webpack handles everything in between at runtime.

3.6 Pattern Decision Framework

With four patterns on the table, the practical question is: how do you choose? The answer depends on three factors: your team's technology stack, the level of UI integration required, and your operational maturity.

Work through these questions in order. The first one that produces a clear answer is your decision point.

1. **Does the MFE need to feel like a seamless part of a single application?** If yes, eliminate iFrames. They cannot deliver seamless UX. Go to question 2.
2. **Is your stack genuinely multi-framework (React, Vue, Angular together)?** If yes, Web Components are a strong choice as the framework-agnostic contract. If no, go

to question 3.

3. **Are all your remotes using Webpack 5 or a compatible bundler?** If yes, Module Federation is almost certainly your best option. If no, evaluate whether migrating to Webpack 5 is feasible before committing to a different pattern.
4. **Is the MFE a truly isolated widget with minimal shell interaction?** Payment forms, embedded dashboards, third-party tools. If yes, an iFrame is actually appropriate and its isolation is an asset, not a limitation.

Figure 3.4 — MFE integration patterns at a glance

Pattern	Isolation	UX quality	Bundle sharing	Framework agnostic	Best for
iFrame	Complete	Poor	None	Yes	Isolated widgets, payment forms
Web Components	Shadow DOM	Good	Manual	Yes	Multi-framework teams
JS Bundle	None by default	Good	Manual	Partial	Simple custom loaders
Module Federation	Error Boundaries	Excellent	Automatic	Partial	React-dominant stacks

The last row is highlighted for a reason. For teams building React applications with Webpack 5, Module Federation is the pattern that gives you the best developer experience, the most capable runtime, and automatic shared dependency management. It is the pattern this book is built on.

You Do Not Have to Pick Just One

Many production architectures use more than one pattern. A common setup is Module Federation for the core product MFEs and an iframe for an embedded third-party analytics dashboard or payment gateway. The patterns are not mutually exclusive.

What matters is that each pattern choice is deliberate. Use Module Federation where you need seamless composition and shared dependencies. Use iFrames where isolation and simplicity are more valuable than UX integration.

Part 1 of this book ends here. You now have the full conceptual foundation: you understand why monolithic frontends break, what Micro Frontends actually are, how to structure teams around them, and which integration pattern to reach for. Part 2 is where we get our hands dirty. Chapter 4 begins with the Webpack 5 fundamentals that make Module Federation possible.

Chapter Summary

- Build-time integration composes MFEs at build time into a single deployable. It sacrifices independent deployment and is rarely the right choice for feature MFEs.
- Run-time integration composes MFEs in the browser via a shell application. iFrames, Web Components, and Module Federation are all run-time strategies.
- iFrames provide the strongest isolation and the simplest mental model, but deliver poor UX for seamlessly integrated applications. Best for isolated widgets and embedded third-party tools.
- Web Components expose MFEs as custom HTML elements with Shadow DOM isolation. Best when teams use different frameworks and need a framework-agnostic contract.
- Module Federation is Webpack 5's run-time integration system. It handles shared dependencies automatically, supports lazy loading, and enables remote components to be consumed like local ones. It is the dominant pattern for React-centric MFE architectures.
- Choose based on three factors: required UX integration level, framework homogeneity, and operational maturity. Most production systems use more than one pattern deliberately.

Up Next: Chapter 4 — Webpack 5 Essentials for MFE Development

Part 2 begins. Chapter 4 covers the Webpack 5 concepts you need before you touch Module Federation: how Webpack thinks about modules, what chunk splitting does, and the minimal configuration that makes everything in Part 2 work. If you are already comfortable with Webpack 5, you can skim this chapter and use it as a reference. If you are not, read it carefully. The configuration in Chapter 5 will make much more sense if you do.

End of Sample

You have just finished Part 1: Foundations of Micro Frontends. This free sample covers Chapters 1 to 3, why monolithic frontends break, what Micro Frontends actually are, and how to choose the right integration pattern.

The full book continues with the part you came for: Webpack 5 and Module Federation hands-on, React integration, routing, state and communication across MFEs, resilient error handling, and a complete guided project where you build and deploy a multi-team application from scratch.

What you get in the full book

- Chapters 4 to 6: Webpack 5 essentials, your first federated setup, shared dependencies
- Chapters 7 to 9: React integration, routing across MFEs, state and communication patterns
- Chapter 10: Error boundaries, fallbacks, retries, and circuit breakers for resilient remotes
- Chapter 11: A full guided project, a host shell with product catalog and cart remotes, deployed

Get the complete book

Available now on Leanpub. The full source code for the guided project is on GitHub:

https://github.com/Kemi-Oluwadahunsi/mfe-project-book-1