



Understanding and implementing micro-frontends on AWS

AWS Prescriptive Guidance



AWS Prescriptive Guidance: Understanding and implementing micro-frontends on AWS

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Introduction	1
Overview	1
Foundational concepts	6
Domain-driven design	6
Distributed systems	8
Cloud computing	8
Alternative architectures	10
Monoliths	10
N-tier applications	10
Microservices	11
Choosing the approach for your requirements	11
Architectural decisions	12
Micro-frontend boundaries	12
How to slice a monolithic application into micro-frontends	13
Micro-frontend composition approaches	15
Client-side composition	15
Edge-side composition	17
Server-side composition	17
Routing and communication	19
Routing	19
Communication between micro-frontends	19
Manage micro-frontend dependencies	20
Share nothing, where possible	20
When you share code	20
Shared state	21
Frameworks and tools	23
General framework considerations	23
Client-side rendering	23
Server-side rendering	24
API integration – BFF	25
Styling and CSS	27
Design systems – A share-something approach	27
Fully encapsulated CSS – A share nothing approach	28
Shared Global CSS – A share-all approach	29

Organization	31
Agile development	31
Team composition and size	32
DevOps culture	32
Orchestrating micro-frontend development across multiple teams	34
Deploying	35
Governance	36
API contracts	36
Cross-interactivity using events	37
Balancing autonomy with alignment	38
Creating micro-frontends	38
End-to-end testing for micro-frontends	38
Releasing micro-frontends	39
Logging and monitoring	39
Alerting	39
Feature flags	41
Service discovery	42
Splitting bundles	42
Canary releases	43
Platform team	44
Next steps	45
Resources	47
Contributors	48
Document history	49

Understanding and implementing micro-frontends on AWS

Luca Mezzalana, Matteo Figus, Alexander Guensche, and Harun Hasdal, Amazon Web Services

As organizations strive for agility and scalability, the conventional monolithic architecture often becomes a bottleneck, hindering rapid development and deployment. Micro-frontends mitigate this by breaking down complex user interfaces into smaller, independent components that can be developed, tested, and deployed autonomously. This approach enhances the efficiency of development teams and facilitates collaboration between backend and frontend, fostering an end-to-end alignment of distributed systems.

This prescriptive guidance is tailored to help IT leaders, product owners, and architects across diverse professional domains to understand micro-frontend architecture and build micro-frontend applications on Amazon Web Services (AWS).

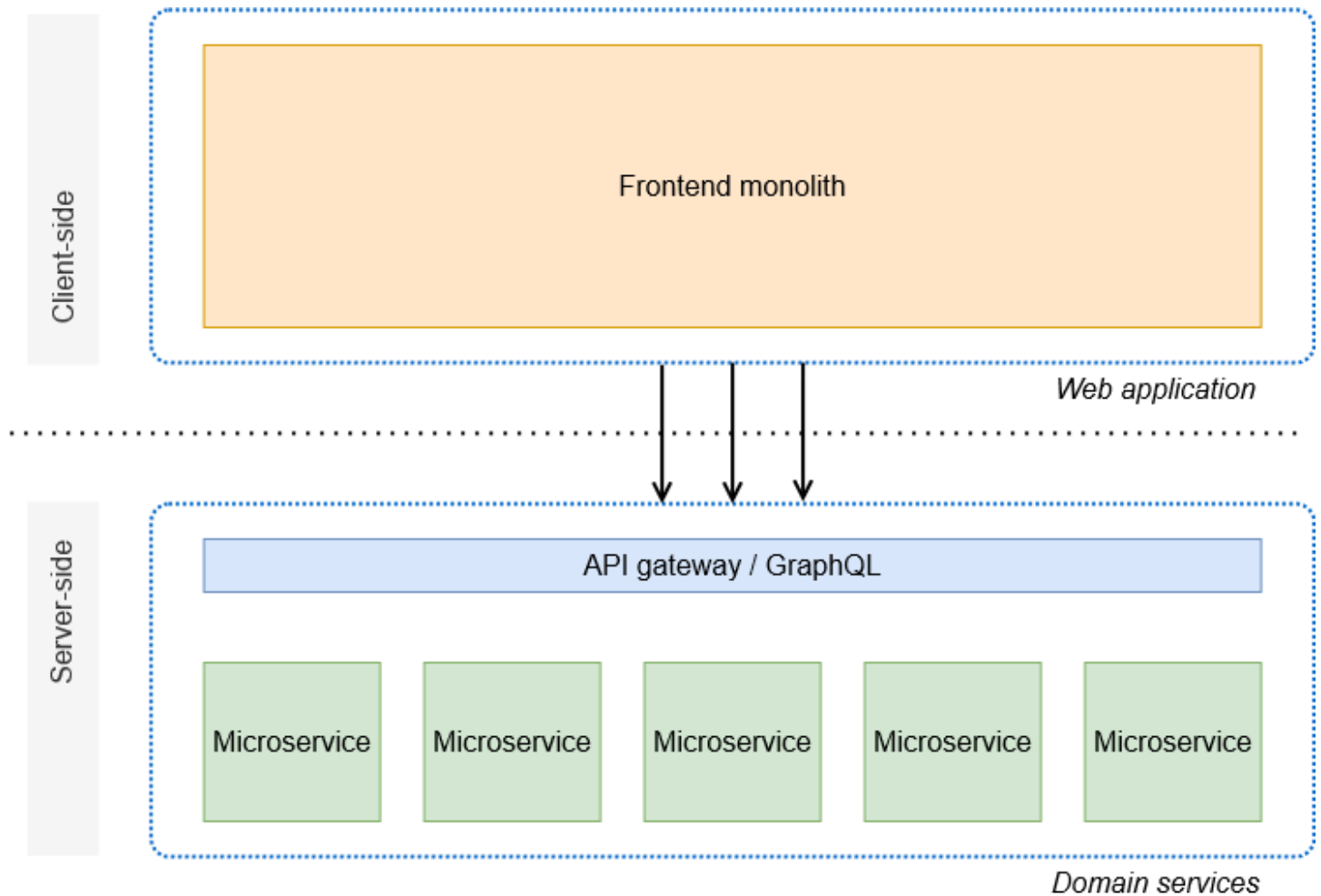
Overview

Micro-frontends are an architecture built on the decomposition of application frontends into independently developed and deployed artifacts. When you split large frontends into autonomous software artifacts, you can encapsulate business logic and reduce dependencies. This supports faster and more frequent delivery of product increments.

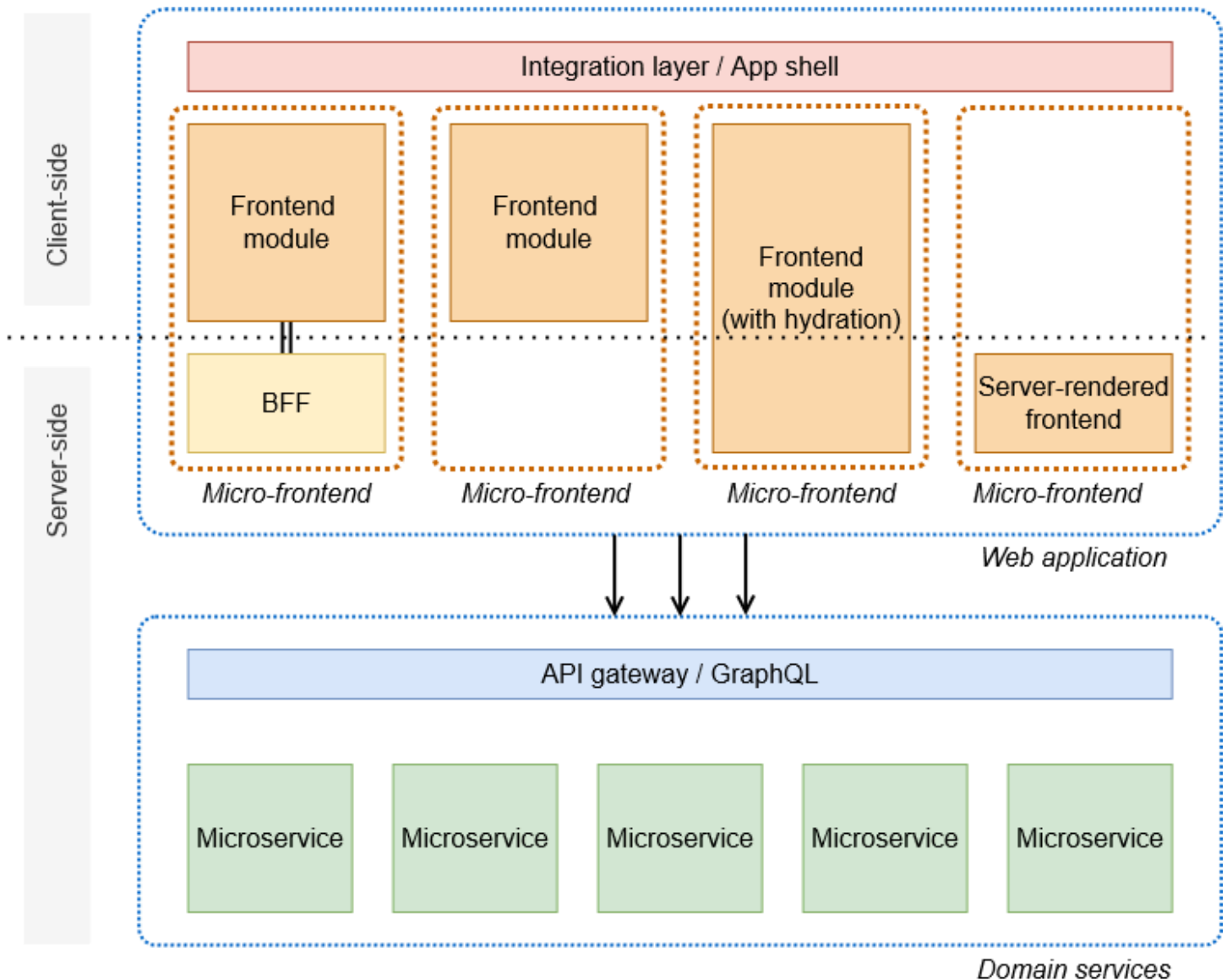
Micro-frontends are similar to *microservices*. In fact, the term micro-frontend is derived from the term microservice, and it aims to convey the notion of a microservice as a frontend. While a microservices architecture typically combines a distributed system in the backend with a monolithic frontend, micro-frontends are self-contained distributed frontend services. These services can be set up in two ways:

- Frontend-only, integrating with a shared API layer behind which runs a microservices architecture
- Full-stack, meaning that each micro-frontend has its own backend implementation.

The following diagram shows a traditional microservices architecture, with a frontend monolith that uses an API gateway to connect to backend microservices.



The following diagram shows a micro-frontend architecture with different implementations of microservices.



As shown in the previous diagram, you can use micro-frontends with client-side rendering or server-side rendering architectures:

- Client-side rendered micro-frontends can directly consume APIs exposed by a centralized API Gateway.
- The team can create a backend-for-frontend (BFF) inside the bounded context to reduce the chattiness of the frontend toward the APIs.
- On the server-side, micro-frontends can be expressed with a server-side approach augmented on the client side by using a technique called hydration. When a page is rendered by the browser, the associated JavaScript is hydrated to allow interactions with UI elements, such as clicking a button.

- Micro-frontends can render on the backend and use hyperlinks to route toward a new part of a website.

Micro-frontends are a great fit for organizations that want to do the following:

- Scale with multiple teams working on the same project.
- Embrace decentralization of decision making, empowering developers to innovate inside the identified systems boundaries.

This approach significantly reduces the cognitive load on teams, because they become responsible for specific parts of the system. It boosts business agility because modifications can be made to one part of the system without disrupting the rest.

Micro-frontends are a distinct architectural approach. Although there are different ways to build micro-frontends, they all have common traits:

- A micro-frontend architecture is composed of multiple independent elements. The structure is similar to the modularization that happens with microservices on the backend.
- A micro-frontend is completely responsible for the frontend implementation within its bounded context, which comprises the following:
 - User interface
 - Data
 - State or session
 - Business logic
 - Flow

A bounded context is an internally consistent system with carefully designed boundaries that mediate what can enter and exit. A micro-frontend should share as little business logic and data with other micro-frontends as possible. Wherever sharing needs to happen, it takes place through clearly defined interfaces such as custom events or reactive streams. However, when it comes to some cross-cutting concerns such as a design system or logging libraries, intentional sharing is welcome.

A recommended pattern is to build micro-frontends by using cross-functional teams. This means that each micro-frontend is developed by the same team working from the backend to the

frontend. Team ownership is crucial, from coding to the operationalization of the system in production.

This guidance does not intend to recommend one particular approach. Instead, it discusses different patterns, best practices, trade-offs, and architectural and organizational considerations.

Foundational concepts

Micro-frontend architecture is heavily inspired by three prior architectural concepts:

- *Domain-driven design* is the mental model to structure complex applications into coherent domains.
- *Distributed systems* are an approach for building applications as loosely coupled subsystems that are developed independently and run on their own dedicated infrastructure.
- *Cloud computing* is an approach for running IT infrastructure as services with a pay-as-you-go model.

Domain-driven design

Domain-driven design (DDD) is a paradigm developed by Eric Evans. In his 2003 book [Domain-Driven Design: Tackling Complexity in the Heart of Software](#), Evans postulates that software development should be driven by business concerns rather than technical ones. Evans proposes that IT projects first develop a *ubiquitous language* that helps technical and domain experts to find a shared understanding. Based on that language, they can formulate a mutually understood model of the business reality.

Obvious as that approach might be, many software projects suffer from a disconnect between business and IT. Those disconnects often cause significant misunderstandings, which lead to budgets overruns, decreased quality, or project failure.

Evans introduces multiple other important terms, one of which is the *bounded context*. A bounded context is a self-contained segment of a large IT application containing the solution or implementation to exactly one business concern. A large application will consist of multiple bounded contexts that are loosely coupled through integration patterns. Those bounded contexts can even have their own dialects of the ubiquitous language. For example, a *user* in the payment context of an application might have different aspects from a *user* in the delivery context because the notion of shipping would be irrelevant during payment.

Evans doesn't define how small or large a bounded context should be. The size is determined by the software project, and it might evolve over time. Good indicators of a context's boundaries are the degree of cohesion between the entities (domain objects) and the business logic.

In the context of micro-frontends, domain-driven design can be illustrated by the example of a complex web page such as a flight-booking page.

https://www.example.com/flight-search

DUS, CGN TFA 2 adults, 3 children

23/09/2023 05/10/2023 SEARCH FLIGHTS

RESULTS

	DUS 10:16 TFS 14:33		TFS 13:21 DUS 17:57	498,00 €	BUY
	DUS 10:16 TFS 14:33		TFS 13:21 DUS 17:57	498,00 €	BUY
	DUS 10:16 TFS 14:33		TFS 13:21 DUS 17:57	498,00 €	BUY
	DUS 10:16 TFS 14:33		TFS 13:21 DUS 17:57	498,00 €	BUY

On this page, the main building blocks are a search form, a filters panel, and the results list. To identify the boundaries, you must identify independent functional contexts. In addition, consider nonfunctional aspects, such as reusability, performance, and security. The most important indicator that "things that belong together" is their communication patterns. If some elements in an architecture must communicate frequently and exchange complex information, they likely share the same bounded context.

Individual UI elements such as buttons are not bounded contexts, because they are not functionally independent. Also, the entire page is not a good fit for a bounded context, because it can be broken down into smaller independent contexts. A reasonable approach is to treat the search form as one bounded context and to treat the results list as the second bounded context. Each of these two bounded contexts can now be implemented as a separate micro-frontend.

Distributed systems

To ease maintenance and to support the ability to evolve, the majority of nontrivial IT solutions are modular. In this case, modular means that IT systems consist of identifiable building blocks that are decoupled through interfaces to achieve separation of concerns.

In addition to being modular, distributed systems should be independent systems in their own right. In a merely modular system, each module is ideally encapsulated and exposes its functions through interfaces, but it can't be independently deployed or even be functional on its own. Also, modules generally follow the same lifecycle as other modules that are part of the same system. The building blocks of a distributed system, on the other hand, each have their own lifecycles. Applying the domain-driven design paradigm, each building block addresses one business domain or subdomain and lives in its own bounded context.

When distributed systems interact during build time, a common approach is to develop mechanisms for identifying issues quickly. For example, you might adopt typed languages and invest heavily in unit testing. Multiple teams can collaborate on the development and maintenance of modules, often distributed as libraries for systems to consume with tools such as npm, Apache Maven, NuGet, and pip.

During runtime, interacting distributed systems are typically owned by individual teams. Consuming dependencies causes operational complexity because of error handling, performance balancing, and security. Investments in integration testing and observability are fundamental to reducing risks.

The most popular examples of distributed systems today are microservices. In microservice architectures, backend services are domain-driven (rather than driven by technical concerns such as UI or authentication) and owned by autonomous teams. Micro-frontends share the same principles, extending the solution scope to the frontend.

Cloud computing

Cloud computing is a way of purchasing IT infrastructure as services with a pay-as-you-go model instead of building your own data centers and buying hardware to operate them on premises. Cloud computing offers several advantages:

- Your organization gains significant business agility by being able to experiment with new technologies without having to make large, long-term financial commitments upfront.

- By using a cloud provider such as AWS, your organization can access a broad portfolio of low-maintenance and highly integrable services (such as API gateways, databases, container orchestration, and cloud capabilities). Access to these services frees up your staff to focus on work that differentiates your organization from the competition.
- When your organization is ready to roll out a solution globally, you can deploy the solution to cloud infrastructure around the world.

Cloud computing supports micro-frontends by providing highly managed infrastructure. This makes end-to-end ownership easier for cross-functional teams. While the team should have strong operations knowledge, the manual tasks of infrastructure provisioning, operating system updates, and networking would be a distraction.

Because micro-frontends live in bounded contexts, teams can choose the most suitable service to run them. For example, teams can choose between cloud functions and containers for compute, and they can choose between different flavors of SQL and NoSQL databases or in-memory caches. Teams can even build their micro-frontends on a highly integrated toolkit such as [AWS Amplify](#), which comes with preconfigured building blocks for serverless infrastructure.

Comparing micro-frontends with alternative architectures

As with all architectural strategies, the decision to adopt micro-frontends must be based on evaluation criteria that are guided by your organization's principles. Micro-frontends have advantages and disadvantages. If your organization decides to use micro-frontends, you must have strategies in place to address the challenges of distributed systems

When choosing an application architecture, the most popular alternatives to micro-frontends are monoliths, n-tier applications, and microservices in combination with a single-page application (SPA) frontend. These are all valid approaches, and each of them has advantages and disadvantages.

Monoliths

A small application that doesn't need frequent changes can be delivered very quickly as a monolith. Even in situations where significant growth is expected, a monolith is a natural first step. Later, the monolith can be either retired or refactored into a more flexible structure. By starting with a monolith, your organization can go to market, get customer feedback, and improve the product faster.

However, monolithic applications tend to degrade if not carefully maintained or as the codebase grows in size over time. When multiple teams significantly contribute to the same codebase, they rarely all contribute to its maintenance and operations. This results in an imbalance of responsibilities, which impacts velocity and causes inefficiencies. At the same time, inadvertent coupling between a monolith's modules leads to unintended side effects as the code base evolves. Those side effects can result in malfunctions and outages.

N-tier applications

A more complex application that has a relatively static pace of evolution can be built as a three-tier architecture (presentation, application, data), with a REST or GraphQL layer between the frontend and backend. This is much more flexible, and teams for the different tiers can develop independently to some extent. The disadvantage of an n-tier application is that it's much more difficult to deploy functionality. The frontend and backend are decoupled through an API contract, so breaking changes must be deployed together or the API must be versioned.

Consider the following common scenario: If releasing a new feature requires a data schema change, it might take days for product owners to agree on a set of functionalities with a frontend team. Then the frontend team will ask the backend team to develop and release the functionality on their side. The backend team will work with the data owners to release a database schema update. Next, the backend team will release a new version of the API, so that the frontend team can develop and release their changes. In this scenario, propagating all changes to production might take weeks or even months, because each team has its own backlog, priorities, and mechanisms around developing, testing, and releasing changes.

Microservices

In a microservices architecture, the backend is decomposed into small services, each addressing a particular business concern within a bounded context. Each microservice is also strongly decoupled from other services by exposing a clearly defined interface contract.

It's worth mentioning that bounded contexts and interface contracts should also exist in well-crafted monoliths and n-tier architectures. In a microservices architecture, however, communication happens over the network, usually the HTTP protocol, and services have dedicated runtime infrastructure. This supports independent development, delivery, and operation of each backend service.

Choosing the approach for your requirements

Monoliths and n-tier architectures bundle multiple domain concerns into one technical artifact. This makes aspects such as dependencies and internal data flow easy to manage, but it makes delivery of new functionalities more difficult. To maintain a coherent code base, a team often invests time in refactoring and decoupling because of the large code base they have to handle.

Applications developed by a few teams might not need the additional complexity that comes with moving to micro-frontends. This is especially true if the teams are not paying the penalties of high coupling and long lead times to release changes.

In summary, more complex and distributed architectures are often the right choice for complex and fast-moving applications. For small to midsize applications, a distributed architecture isn't necessarily superior to a monolithic one, especially if the application will not dramatically evolve over a short period of time.

Architectural decisions in micro-frontends

Teams that apply a micro-frontend architecture pattern for their applications must make several decisions about architecture early on:

- [Identification of micro-frontends and definition of boundaries](#)
- [Composing pages and views with micro-frontends](#)
- [Routing, state management, and communication across micro-frontends](#)
- [Managing dependencies for cross-cutting concerns](#)

The following sections cover these topics in more depth.

When making architecture decisions, it's essential to have the correct metrics and to understand the usage patterns application characteristics, and trade-offs. For example, an e-commerce site has different characteristics and usage patterns compared with a video-editing tool or observability dashboards.

Public-facing applications with high traffic and short session depth can be optimized for initial page load metrics such as Time to Interactive (TTI) and First Contentful Paint (FCP). In contrast, an application to which users log in at the start of their day and keep interacting with throughout the day might be optimized for the in-application experience. The application team might optimize for the First Input Delay (FID) metric after each navigation instead of initial page load.

Public websites must cater to various browser environments. Enterprise applications with known constraints on the client environment can optimize their micro-frontend composition according to their constraints.

There is no single right choice for the architecture decisions. Understand the trade-offs, the context where business operates, usage patterns, and metrics to guide decisions that are suitable for each individual application.

Identifying micro-frontend boundaries

To improve team autonomy, the business capabilities provided by an application can be decomposed into several micro-frontends with minimal dependencies on each other.

Following the DDD methodology discussed previously, teams can break down an application domain into business subdomains and bounded contexts. Autonomous teams can then own the functionality of their bounded contexts and deliver those contexts as micro-frontends.

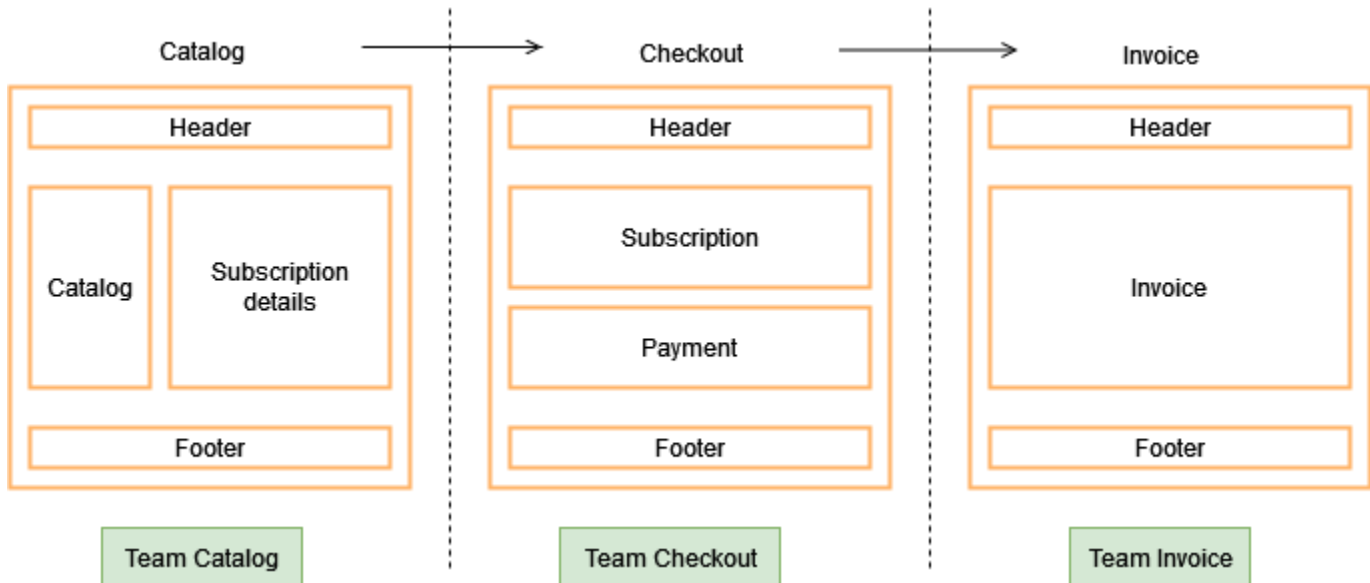
A well-defined bounded context should minimize functional overlap and the need for runtime communication across contexts. The required communication can be implemented with event-driven methods. This is no different from event-driven architecture for microservices development.

A well-architected application should also support the delivery of future extensions by new teams to provide a consistent experience for customers.

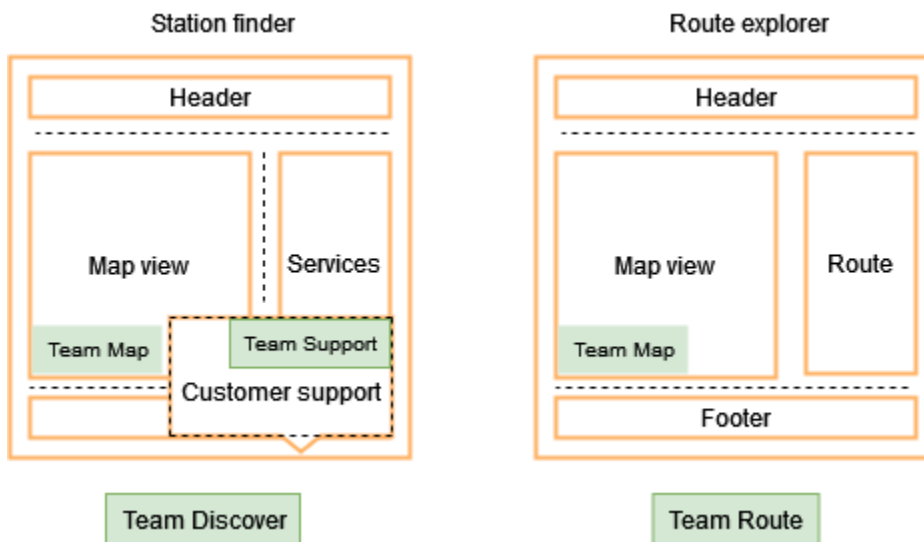
How to slice a monolithic application into micro-frontends

The [Overview](#) section included an example of identifying independent functional contexts on a web page. Several patterns for splitting the functionality on the user interface emerge.

For example, when the business domains form stages of a user journey, a vertical split on the frontend can be applied, where a collection of views in the user journey is delivered as micro-frontends. The following diagram shows a vertical split, where Catalog, Checkout, and Invoice steps are delivered by separate teams as separate micro-frontends.



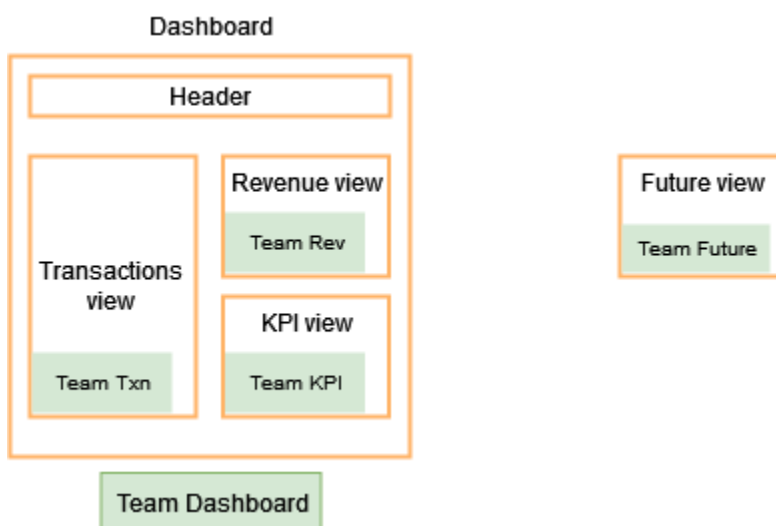
For some applications, vertical split alone might not be enough. For example, some functionality might need to be provided in many views. For these applications, you can apply a mixed split. The following diagram shows a mixed split solution in which micro-frontends for Station finder and Route explorer both use the Map view functionality.



Portal-type or dashboard-type applications typically bring frontend capabilities together in a single view. In these types of applications, each widget can be delivered as a micro-frontend, and the hosting application defines the constraints and interfaces that the micro-frontends should implement.

This approach provides a mechanism for micro-frontends to handle concerns such as viewport sizing, authentication providers, configuration settings, and metadata. These types of applications optimize for extensibility. New features can be developed by new teams to scale the dashboard capabilities.

The following diagram shows a dashboard application developed by three individual teams that are part of Team Dashboard.



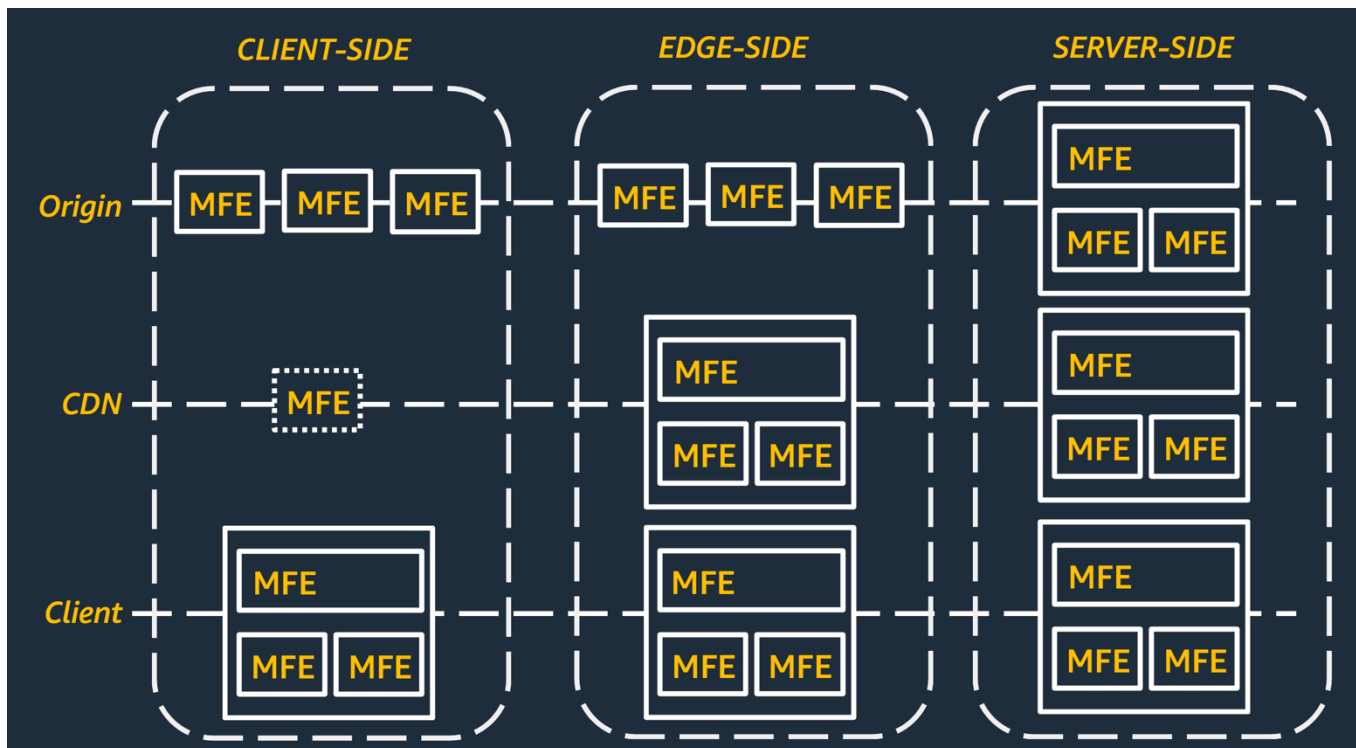
In the diagram, the future view represents new features developed by new teams to scale Team Dashboard and the dashboard capabilities.

Portal and dashboard applications usually compose functionality by using a mixed split in the UI. The micro-frontends are configurable with well-defined settings, including position and size constraints.

Composing pages and views with micro-frontends

You can compose views of an application with client-side composition, edge-side composition, and server-side composition. The composition patterns have different characteristics in terms of necessary team skills, fault tolerance, performance, and cache behavior.

The following diagram shows how the composition happens at the client-side, edge-side, and server-side layers of a micro-frontend architecture.



The client-side, edge-side and server-side layers are discussed in the following sections.

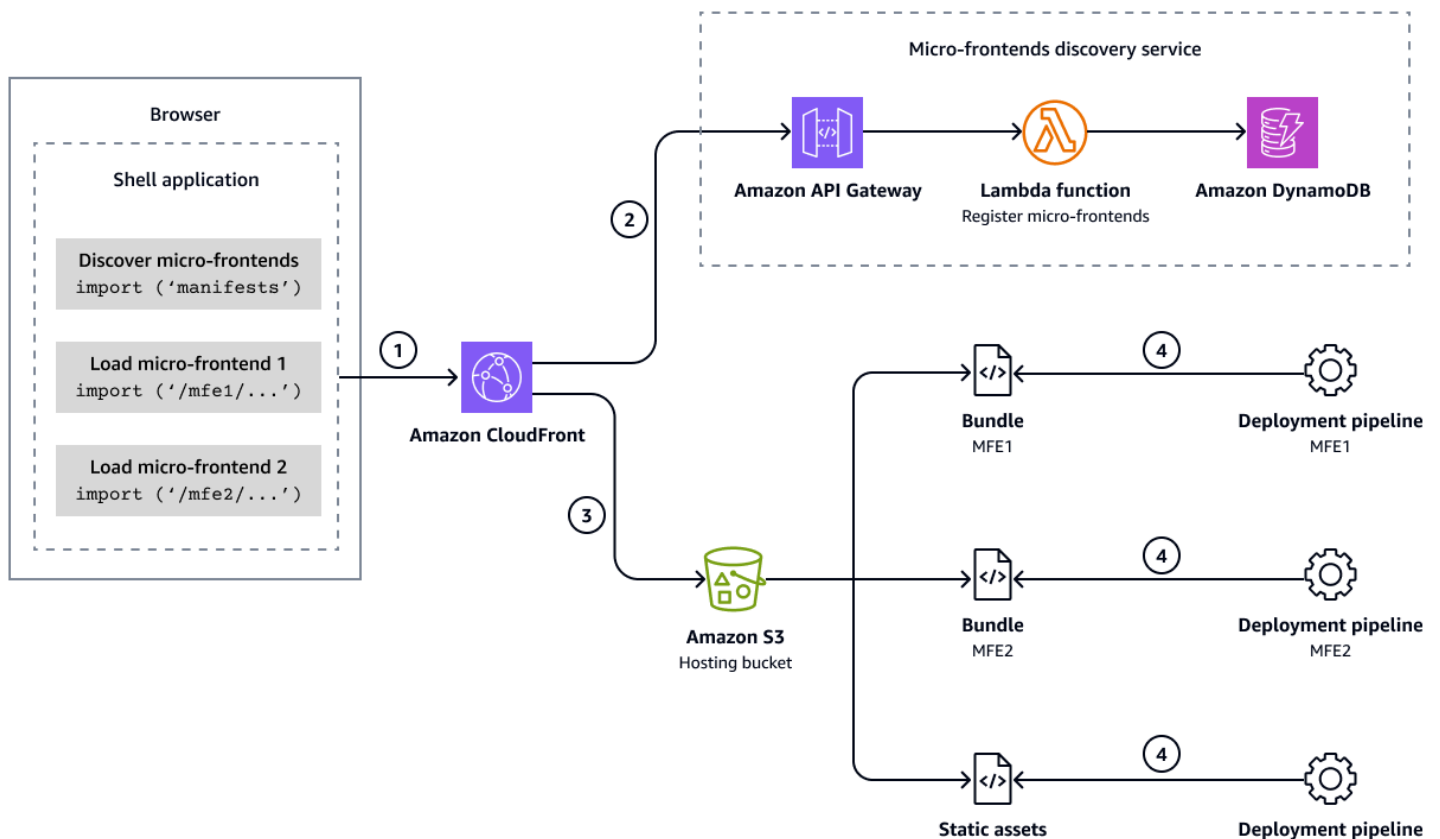
Client-side composition

Dynamically load and append micro-frontends as Document Object Model (DOM) fragments on the client (browser or mobile web view). The micro-frontend artifacts, such as JavaScript or CSS files,

can be loaded from content delivery networks (CDNs) for reduced latency. Client-side composition requires the following:

- A team to own and maintain a shell application or a micro-frontend framework to enable discovery, loading, and rendering micro-frontend components at runtime in the browser
- High skill levels in front-end technologies such as HTML, CSS, and JavaScript, and in-depth understanding of browser environments
- Optimization of the amount of JavaScript loaded in a page, and discipline to avoid global namespace clashes

The following diagram shows an example AWS architecture for serverless client-side composition.



Client-side composition happens in the browser environment through a shell application. The diagram shows the following details:

1. After the shell application is loaded, it makes an initial request to [Amazon CloudFront](#) to discover the micro-frontends to be loaded through a manifest endpoint.
2. Manifests contain information about each micro-frontend (for example, name, URL, version, and fallback behavior). The manifests are served by the micro-frontends discovery service. In

the diagram, this discovery service is represented by Amazon API Gateway, an AWS Lambda function, and Amazon DynamoDB. The shell application uses the manifest information to request individual micro-frontends to compose the page within a given layout.

3. Each micro-frontend bundle is composed of static files (such as JavaScript, CSS, and HTML). The files are hosted in an [Amazon Simple Storage Service \(Amazon S3\)](#) bucket and served through CloudFront.
4. Teams can deploy new versions of their micro-frontends and update the manifest information by using deployment pipelines that they own.

Edge-side composition

Use transclusion techniques such as Edge Side Includes (ESI) or Server Side Includes (SSI) supported by some CDNs and proxies in front of origin servers to compose a page before sending it over the wire to the clients. ESI requires the following:

- A CDN with ESI capability, or a proxy deployment in front of server-side micro-frontends. Proxy implementations such as HAProxy, Varnish, and NGINX support SSI.
- An understanding of the use and limitations of ESI and SSI implementations.

Teams starting new applications typically don't choose edge-side composition for their composition pattern. However, this pattern might provide a pathway for legacy applications that rely on transclusion.

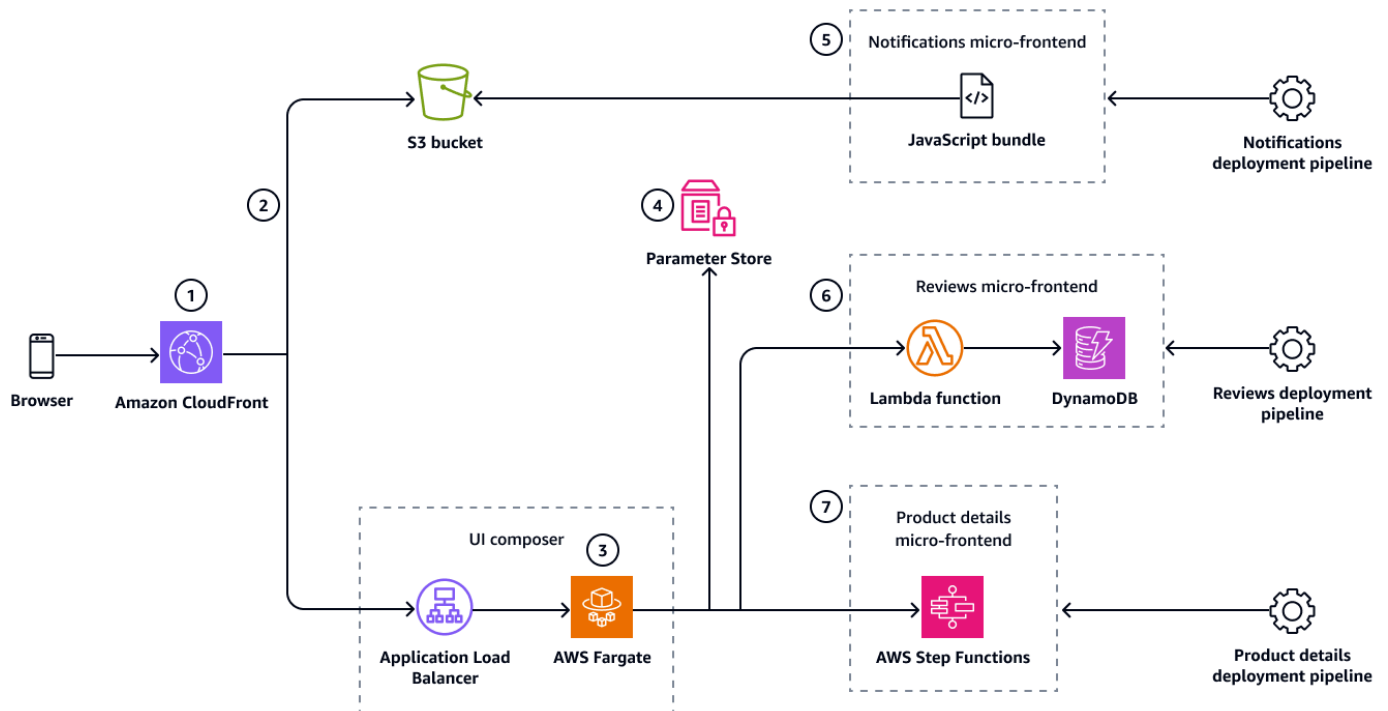
Server-side composition

Use origin servers to compose pages before they are cached on the edge. This can be done with traditional technologies, such as PHP, Jakarta Server Pages (JSP), or templating libraries, to compose the pages by including fragments from micro-frontends. You can also use JavaScript frameworks, such as Next.js, running on the server to compose pages on the server with server-side rendering (SSR).

After the pages are rendered on the server, they can be cached on CDNs to reduce latency. When new versions of micro-frontends are deployed, pages must be re-rendered, and the cache must be updated to deliver the latest versions to customers.

Server-side composition requires an in-depth understanding of the server environment to establish patterns for deployment, discovery of server-side micro-frontends, and cache management.

The following diagram shows server-side composition.



The diagram includes the following components and processes:

1. [Amazon CloudFront](#) provides a unique entry point to the application. The distribution has two origins: the first for static files and the second for the UI composer.
2. Static files are hosted in an [Amazon S3](#) bucket. They are consumed by the browser and the UI composer for HTML templates.
3. The UI composer runs on a containers cluster in [AWS Fargate](#). With a containerized solution, you can use streaming capabilities and multithreaded rendering if needed.
4. [Parameter Store](#), a capability of AWS Systems Manager, is used as a basic micro-frontends discovery system. This capability provides a key-value store used by the UI composer for retrieving the micro-frontend endpoints to consume.
5. The notifications micro-frontend stores the optimized JavaScript bundle in the S3 bucket. This renders on the client because it must react to user interactions.
6. The reviews micro-frontend is composed by a [Lambda](#) function, and the user reviews are stored in [DynamoDB](#). The reviews micro-frontend is rendered fully on the server side, and it outputs an HTML fragment.
7. The product details micro-frontend is a low-code micro-frontend that uses [AWS Step Functions](#). The Express Workflow can be invoked synchronously, and it contains the logic for rendering the HTML fragment and a caching layer.

For more information about server-side composition, see the blog post [Server-side rendering micro-frontends – the architecture](#).

Routing and communication across micro-frontends

Routing options depend on the composition approach. Communication can be optimized by reducing coupling between front-end components.

Routing

Applications that use client-side composition with vertical split can use server-side routing (multipage application) or client-side routing (single-page application). If they use a mixed split for the UI composition, client-side routing is necessary to support deeper routing hierarchies of micro-frontends on a page.

Applications that use edge-side composition and server-side composition align better with server-side routing, or routing with edge compute such as Lambda@Edge with Amazon CloudFront.

Communication between micro-frontends

With micro-frontend architectures, we recommend reducing the coupling between frontend components. One approach to reduce coupling is to move away from synchronous function calls to asynchronous messaging.

Browser runtimes and user interactions are asynchronous by nature. Events can be exchanged between producers and consumers through messages. The events provide a well-defined interface for communication across micro-frontends.

If you follow DDD practices to identify your bounded contexts for micro-frontends, the next step is to identify events that must be communicated across the boundaries.

The messaging mechanism for events can be native DOM events (CustomEvents), JavaScript event emitters, or reactive stream libraries provided by the platform teams. Micro-frontends publish events and subscribe to events relevant for their bounded context. With this method, the publishers and subscribers don't need to be aware of each other. The contract is the event definition. For a visual representation of this, see the *Communicate with events* section of the [Bounded context with event architectures diagram](#).

Managing dependencies for cross-cutting concerns

Conscious dependency management is crucial for the success of a distributed architecture such as micro-frontends. Dependency management is one of the most challenging parts of micro-frontend development.

In a micro-frontend architecture, two important aspects of dependency management are the performance penalty from transferring large code artifacts to the client, and the overhead in compute resources. Ideally, your organization needs to mandate how dependencies in a distributed frontend architecture are maintained.

Three viable strategies for mandating dependency maintenance are *share nothing*, using web standards such as import maps, and *module federation*. Other approaches are anti-patterns because they violate basic principles of distributed architectures.

Share nothing, where possible

The share-nothing approach postulates that no dependencies between independent software artifacts should be shared at all, or at least not at integration or runtime. This means that if two micro-frontends depend on the same library, each must bake in the library at build time and ship it separately. Also, each micro-frontend must validate that the library does not pollute global namespaces and shared resources.

This leads to redundancies, but it is a conscious trade-off with maximum agility. With no runtime dependencies shared, teams have maximum flexibility to evolve the software in any way they see useful as long as they do so in the scope of their solution and don't break any interface contracts.

On a platform where micro-frontends follow the share-nothing principle, it's important to keep micro-frontends as lightweight as possible. It requires developers who are skilled and diligent in optimizing their micro-frontends for performance and who don't sacrifice user experience for developer experience.

When you share code

When you make the decision to share some code, you can share it as libraries or runtime modules. For example, the frontend core team delivers libraries for micro-frontend consumption through CDNs. The business value teams can load the libraries at runtime, or they can use package repositories to publish their libraries. Micro-frontend teams can develop against a specific version of the packaged library at build time, similar to mobile applications using hybrid frameworks.

A third option is to use a private package registry to support build-time integration of common libraries. This reduces the risk that a breaking change in the library contract initiates errors at runtime. However, this more conservative approach requires more governance in place to synchronize all the micro-frontends with newer library versions.

To improve the page load times, micro-frontends can externalize the library dependencies to be loaded from cached chunks from a CDN such as Amazon CloudFront.

To manage runtime dependencies, micro-frontends can use import-maps (or libraries such as `System.js`) to specify where each module is loaded from at runtime. webpack Module Federation is another approach to point to a hosted version of a remote module and resolve common dependencies across independent micro-frontends.

Another approach is to facilitate dynamic loading of import-maps with an initial request to a [discovery endpoint](#).

Shared state

To reduce the coupling of micro-frontends, it's important to avoid a global state management accessible from all the micro-frontends in the same view, similar to monolithic architectures. For example, having a global Redux store accessible from all micro-frontends increases coupling.

A pattern to eliminate shared state is to encapsulate it within micro-frontends, and communicate with asynchronous messages as discussed previously.

When absolutely necessary, introduce well-defined interfaces for global state, and opt in for read-only sharing to avoid unexpected behavior:

- When a vertical split is present, you can use URL components and browser storage to access information from the host environment.
- When you have a mixed split, you can also use the DOM standard custom events or JavaScript libraries, such as event emitters or bi-directional streams, to pass information to micro-frontends.

If you need to share several pieces of information across micro-frontends, we recommend revisiting the micro-frontend boundaries. The need to share might be caused by business evolution or a subpar initial design.

It's also possible to use server-side sessions, where each micro-frontend fetches the required data by using a session identifier. To reduce coupling, it's important to eliminate shared state and to keep micro-frontend specific session data separate.

Frameworks and tools

There is no shortage of frontend frameworks, such as Angular and Next.js, but most of them are not created with micro-frontends in mind. Therefore, they are sometimes missing mechanisms to address the challenges of micro-frontend architecture.

General framework considerations

This guide doesn't aim to recommend, or to compare, individual frameworks. As multiple micro-frontends often run on the same web application page, loading and runtime performance are major concerns. It's important to choose a framework that introduces as little overhead as possible.

Frameworks are divided based on the rendering layer:

- Client-side rendering (CSR)
- Server-side rendering (SSR)

Frontend architectures include other capabilities, such as static site generation (SSG). However, SSG is performed one time only. Micro-frontends are mainly composed at runtime, so CSR and SSR are the main options.

Client-side rendering

For CSR, there are two popular options:

- Single SPA framework
- Module Federation

Single SPA is a lightweight choice for composing micro-frontends. It solves the most common challenges in micro-frontend architectures, such as composing multiple micro-frontends in the same page and avoiding dependency clashes.

Module Federation started as a plugin, offered by webpack 5, and it solves a vast majority of the challenges in micro-frontend architectures, including dependencies management across different artifacts. Module Federation 2.0 works natively with Rspack, webpack, esbuild, and now with JavaScript.

Consider not using a framework at all. Modern browsers, with a market share of 98 percent overall according to caniuse.com, offer features such as custom elements natively, and they are adequate for a micro-frontends application. Where necessary, combine custom elements with lightweight libraries for event propagation, internationalization, or other specific concerns.

Server-side rendering

On the SSR side, the two main options are more complicated:

- Embrace an existing framework such as Next.js, and apply a micro-frontends principle that uses Module Federation.
- Use HTML-over-the-wire to exchange HTML fragments that represent micro-frontends, and compose these fragments inside a template at runtime. An example of this approach is Podium.

API integration – Backend for frontend

The [Backends for Frontends \(BFF\) pattern](#) is typically used in microservices environments. In the context of micro-frontends, a BFF is a server-side service that belongs to a micro-frontend. Not all micro-frontends need to have a BFF. However, if you're using a BFF, it must run inside the same bounded context and not be shared across other bounded contexts.

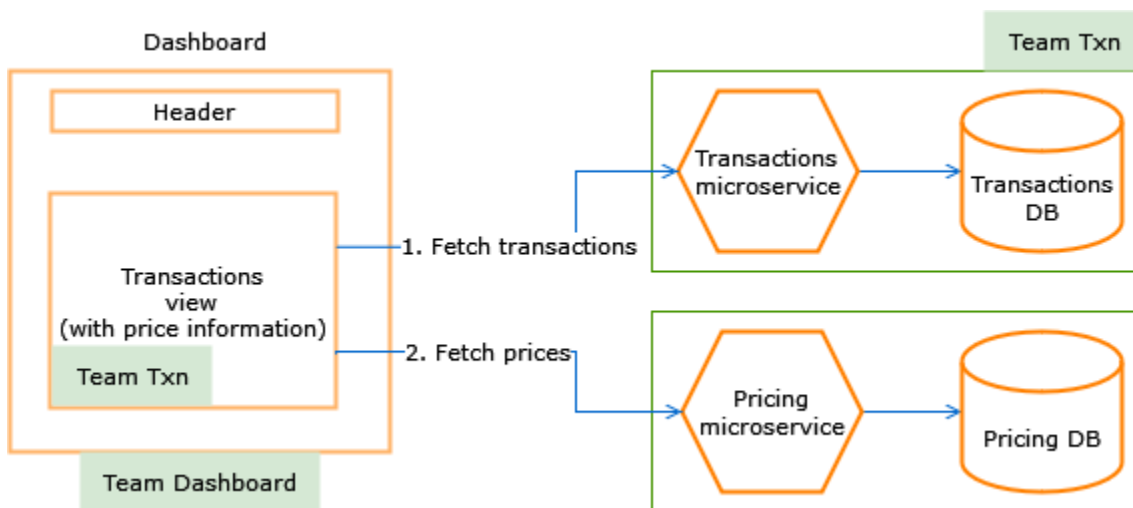
Unlike a traditional service, a BFF doesn't follow a domain model. Instead, it's an API layer for the micro-frontend to preprocess data before it reaches the client. Areas where this is useful include the following:

- Authorization toward private APIs
- Aggregation of data from different sources
- Transformation of data to reduce network load and to ease the consumption of data by the client

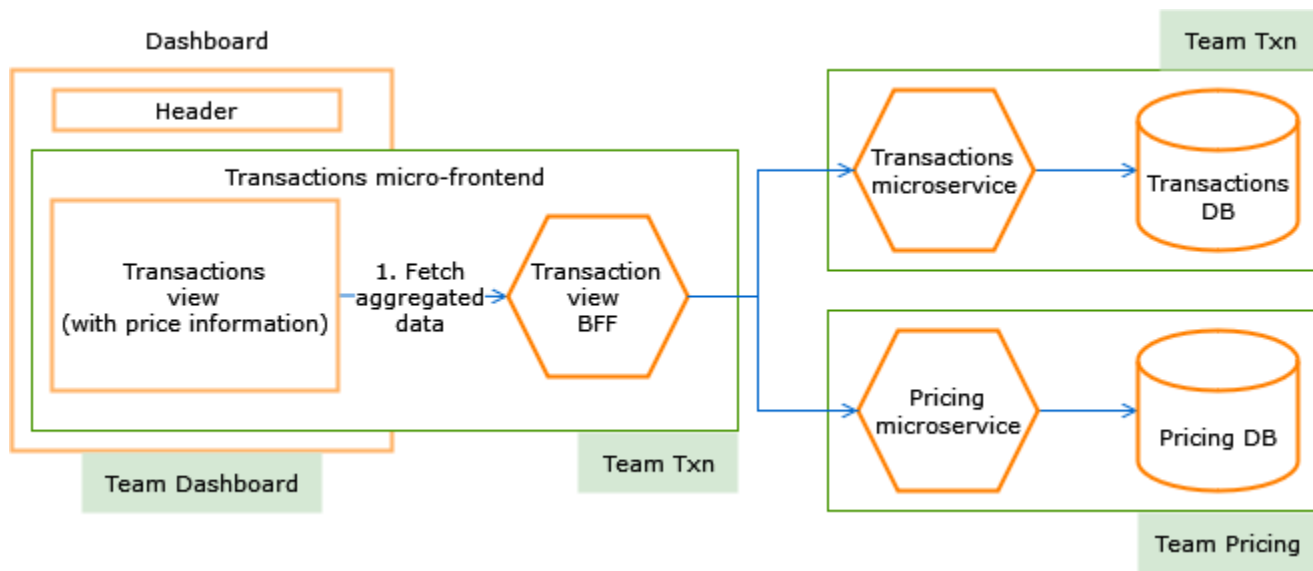
As such, a BFF is owned by the micro-frontend, not by the domain service tier. BFFs can be deployed by using the following:

- AWS AppSync GraphQL APIs
- A set of AWS Lambda functions
- As a container running on Amazon ECS, Amazon EKS, or AWS AppRunner

The following diagram shows that without the BFF pattern, micro-frontends must connect to individual microservice API endpoints to fetch and aggregate data.



Instead, with the BFF pattern in the following diagram, micro-frontends can communicate with their own backend and fetch aggregated data.



Teams can develop BFFs for different channels such as mobile, web, or specific views, with requirements to optimize backend interactions by reducing chattiness.

Styling and CSS

Cascading Style Sheets (CSS) is a language for centrally determining the presentation of a document instead of hard-coding formatting for text and objects. The cascading feature of the language is designed to control priorities between styles by using inheritance. When you work on micro-frontends and create a strategy to manage dependencies, the cascading feature of the language can be a challenge.

For example, two micro-frontends co-exist on the same page, each defining its own styling for the body HTML element. If each fetches its own CSS file and attaches it to the DOM by using a `style` tag, the CSS file overrides the first if they both have definition for common HTML elements, class names or element IDs. There are different strategies to deal with these problems, depending on the dependency strategy that you pick for managing styles.

Currently, the most popular approach to balance performance, consistency, and developer experience consists of developing and maintaining a design system.

Design systems – A share-something approach

This approach uses a system to share styling when appropriate while supporting occasional divergence to balance consistency, performance, and developer experience. A design system is a collection of reusable components, guided by clear standards. Design system development is typically driven by one team with input and contributions from many teams. In practical terms, a design system is a way to share low-level elements that can be exported as a JavaScript library. Micro-frontend developers can use the library as a dependency to build simple interfaces by composing premade available resources and as a starting point to make new interfaces.

Consider the example of a micro-frontend that needs a form. The typical developer experience consists of using premade components available in the design system to compose text boxes, buttons, dropdown lists, and other UI elements. The developer doesn't need to write any styling for the actual components, only for how they look together. The system to build and release can use webpack Module Federation or a similar approach to declare the design system as an external dependency, so that the form's logic is packaged without including the design system.

Multiple micro-frontends can then do the same to take care of shared concerns. When teams develop new components that can be shared between multiple micro-frontends, those components are added to the design system after they reach maturity.

A main advantage of the design system approach is the high level of consistency. While micro-frontends can write styles and occasionally override those from the design system, there is very little need for that. The main low-level elements don't change often, and they offer basic functionality that is extendable by default. Another advantage is performance. With a good strategy to build and release, you can produce minimal shared bundles that are instrumented by the application shell. You can improve even further when multiple micro-frontend specific bundles are asynchronously loaded on demand, with minimal footprint in terms of network bandwidth. Last but not least, the developer experience is ideal because people can focus on building rich interfaces without reinventing the wheel (such as writing JavaScript and CSS every time a button needs to be added to a page).

The downside is that a design system of any sort is a dependency, so it must be maintained and sometimes updated. If multiple micro-frontends require a new version of a shared dependency, you can use either of the following:

- An orchestration mechanism that can occasionally fetching multiple versions of that shared dependency without conflicts
- A shared strategy to move all the dependents to use a new version

For example, if all micro-frontends depend on the version 3.0 of a design system and there is a new version called 3.1 to be used in a shared manner, you can implement feature flags for all micro-frontends to migrate with minimal risk. For more information, see the [Feature flags](#) section. Another potential downside is that design systems usually address more than styling. They also include JavaScript practices and tools. These aspects require reaching consensus through debate and collaboration.

Implementing a design system is a good long-term investment. It's a popular approach, and it should be considered by anyone working on complex frontend architecture. It typically requires frontend engineers and product and design teams to collaborate and define mechanisms to interact with each other. It's important to schedule time to reach the desired state. It's also important to have sponsorship from the leadership so that people can build something reliable, well-maintained, and performant in the long term.

Fully encapsulated CSS – A share nothing approach

Each micro-frontend uses conventions and tools to overcome the cascading feature of CSS. An example is ensuring each element's style is always associated with a class name instead of the

element's ID, and class names are always unique. In this way, everything is scoped to individual micro-frontends, and the risk of unwanted conflicts is minimized. The application shell is typically in charge of loading micro-frontends' styles after they are loaded into the DOM, although some tools bundle the styles together by using JavaScript.

The main advantage of sharing nothing is the reduced risk of introducing conflicts between micro-frontends. Another advantage is the developer's experience. Each micro-frontend shares nothing with other micro-frontends. Releasing and testing in isolation is simpler and quicker.

A main disadvantage of the share-nothing approach is the potential lack of consistency. No system is in place to assess consistency. Even if duplicating what's shared is the goal, it becomes challenging when balancing speed of release and collaboration. A common mitigation is to create tools to measure consistency. For example, you can create a system to take automated screenshots of multiple micro-frontends rendered in a page with a headless browser. You can then manually review the screenshots before a release. However, that requires discipline and governance. For more information, see the [Balancing autonomy with alignment](#) section.

Depending on the use case, another potential disadvantage is performance. If a large amount of styling is used by all the micro-frontends, the customer must download a lot of duplicated code. That will negatively affect the user experience.

This share-nothing approach should be considered only for micro-frontend architectures that involve only a few teams, or micro-frontends that can tolerate low consistency. It can also be a natural initial step while an organization is working on a design system.

Shared Global CSS – A share-all approach

With this approach, all the code related to styling is stored in a central repository where contributors write CSS for all micro-frontends by working on CSS files or by using preprocessors such as Sass. When changes are made, a build system creates a single CSS bundle that can be hosted in a CDN and included in each micro-frontend by the application shell. Micro-frontend developers can design and build their applications by running their code through a locally hosted application shell.

Apart from the obvious advantage of reducing risk of conflicts between micro-frontends, the advantages of this approach are consistency and performance. However, decoupling styles from markup and logic makes it harder for developers to understand how styles are used, how they can evolve, and how they can be deprecated. For example, it might be quicker to introduce a new class

name than to learn about the existing class and the consequences of editing its properties. The disadvantages of creating a new class name are bundle-size growth, which affects performance, and the potential introduction of inconsistencies in the user experience.

While a shared global CSS can be the starting point of a monolith-to-micro-frontends migration, it's rarely beneficial for micro-frontend architectures that involve more than one or two teams collaborating together. We recommend investing in a design system as soon as possible and implementing a share-nothing approach while the design system is in development.

Organization and ways of working

As with all architectural strategies, micro-frontends have implications far beyond the technology that an organization chooses to implement. The decision to build micro-frontend applications must align with business, product, organization, operations, and even culture (for example, empowering teams and decentralizing decision making). In return, this type of micro-frontend architecture supports truly agile, product-driven development, because it significantly reduces communication overhead between otherwise independent teams.

Agile development

The idea of agile software development has become so universal in recent years that virtually every organization claims to work agile. While a conclusive definition of *agile* is beyond the scope of this strategy, it's worth reviewing key elements that are relevant to micro-frontend development.

The foundation of the agile paradigm is the [Agile Manifesto](#) (2001), which postulated four main tenets (for example, "Individuals and interactions over processes and tools") and twelve principles. Process frameworks such as Scrum and the Scaled Agile Framework (SAFe) have emerged around the Agile Manifesto and have found their way into everyday practices. However, the philosophy behind them is largely misunderstood or ignored.

In the context of micro-frontends architecture, the following agile principles are important to embrace:

- "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

This principle emphasizes how important it is to work in increments and deliver software to production as regularly and as often as possible. From a technological perspective, this refers to continuous integration and continuous delivery (CI/CD). In CI/CD, the tools and processes for building, testing, and deployment are integral parts of each software project. The principle also implies that the runtime infrastructure and the operational responsibilities must be owned by the team. That ownership is especially important in distributed systems where independent subsystems might have significantly differing requirements for infrastructure and operations.

- "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

"The best architectures, requirements, and designs emerge from self-organizing teams."

Both of these principles emphasize the benefits of ownership, independence, and end-to-end responsibility. A micro-frontend architecture will be successful when (and only when) teams truly own their micro-frontends. End-to-end responsibility from the conception through design and implementation on to delivery and operation ensures that the team can actually exercise ownership. This independence is required, both technically and organizationally, for the team to have autonomy over the strategic direction. We don't recommend using a micro-frontend platform in a centralized organization that uses a waterfall development model.

Team composition and size

For a software team to exercise ownership, it must govern itself, including how and what the team delivers, within the boundaries imposed by the organization.

To be effective, teams must be able to deliver software independently and have the authority to decide the best way to deliver it. A team that receives feature requirements from an external product manager or UI designs from an external designer, without being involved in the planning of these items, cannot be considered autonomous. The features might violate existing contracts or functionality. Such violations will require further discussions and negotiations, with the risk of delaying the delivery and introducing unnecessary conflicts between teams.

At the same time, teams should not become too large. While a larger team has more resources and can accommodate individual absences, communication complexity grows exponentially with each new member. It isn't possible to state a universally valid maximum team size. The number of people needed for a project depends on factors such as team maturity, technical complexity, pace of innovation, and infrastructure. For example, Amazon follows the two-pizza rule: A team that is too large to be fed on two pizzas should be split into smaller teams. That can be a challenge. The split should happen along natural boundaries and should give each team autonomy and ownership over their work.

DevOps culture

DevOps refers to a software engineering practice where the steps of the development lifecycle are tightly integrated from organizational and technical perspectives. Contrary to popular belief, DevOps is very much about culture and mindset, and very little about roles and tooling.

Traditionally, a software organization would have teams of specialists, such as for design, implementation, testing, deployment, and operations. Whenever a team completed their job, they

would hand the project over to the next team. However, the delivery of software through silos of specialized teams causes friction during handovers. At the same time, when specialists are forced to work with a narrow focus, they lack knowledge in neighboring domains, and they don't have a systemic view of the product. Those deficits can lead to low coherence of the software product.

For example, when a software architect designs a solution that is to be implemented by someone on a different team, they might overlook inherent aspects of the implementation (such as a dependency mismatch). Developers then take shortcuts (such as a monkey patch), or a formalized back-and-forth is initiated between the architect and the development team. Because of the overhead of managing these processes, development is no longer agile (in the sense of flexible, adaptive, incremental, and informal).

Although the term DevOps mainly pertains to culture, it implies the technologies and processes that make DevOps possible in practice. DevOps is closely related to CI/CD. When a developer has finished implementing an increment of software, they commit it to a version-control system such as Git. Traditionally, a build system would then build and integrate the software, and have it tested and released in a more or less unified and centralized process. With CI/CD, the building, integration, testing, and release of the software is inherent and automated. Ideally, the process is part of the software project itself through configuration files that are tailored to the given project specifically.

As many steps as possible are automated. For example, manual testing practices should be reduced, because nearly all types of tests can be automated. When the project is set up in that way, updates to a software product can be delivered several times each day with high confidence. Another technology that supports DevOps is infrastructure as code (IaC).

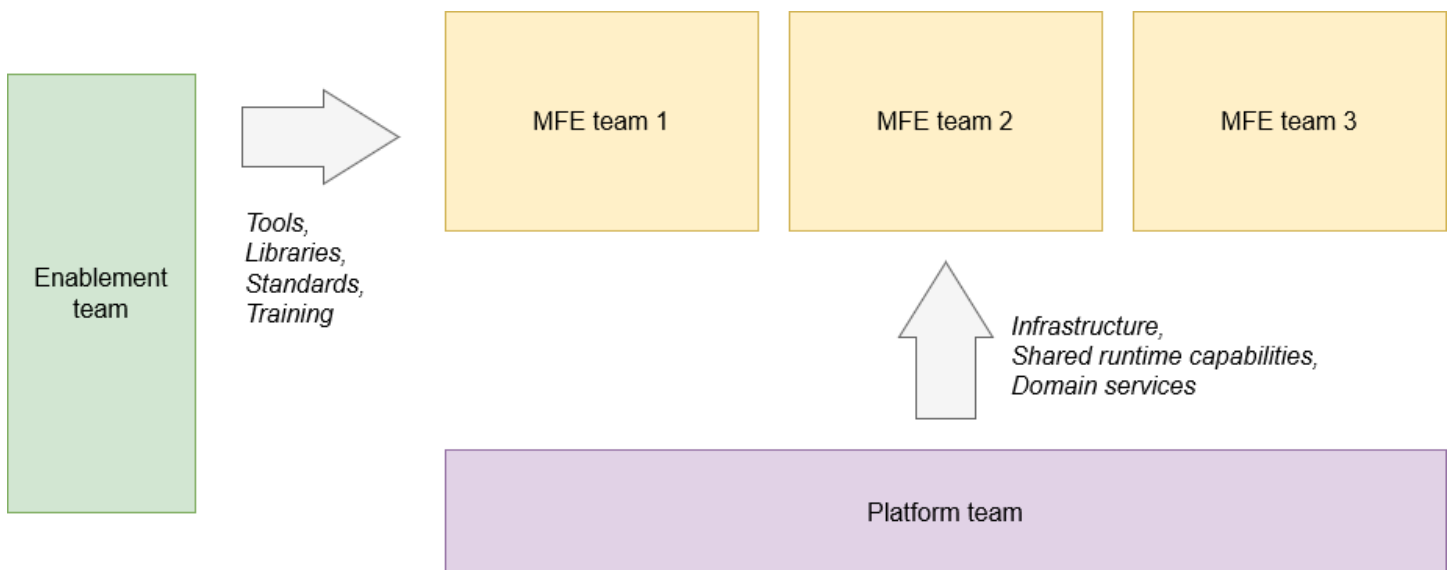
Traditionally, setting up and maintaining IT infrastructure would require the manual work of installing and maintaining hardware (setting up cables and servers in a data center) and operational software. This was necessary, but it had numerous drawbacks. Setup was time-consuming and error-prone. Hardware was often over-provisioned or under-provisioned, leading to either excess spending or degraded performance. By using IaC, you can describe the infrastructure requirements for an IT system through a configuration file from which cloud services can be deployed and updated automatically.

What does all of this have to do with micro-frontends? DevOps, CI/CD, and IaC are ideal complements to a micro-frontend architecture. The benefits of micro-frontends rely on fast and frictionless delivery processes. A DevOps culture can thrive only in environments where teams own software projects with end-to-end responsibility.

Orchestrating micro-frontend development across multiple teams

When scaling micro-frontend development across multiple cross-functional teams, two problems emerge: First, teams start to develop their own interpretation of the paradigm, make framework and library choices, and create their own tooling and helper libraries. Second, fully autonomous teams must take responsibility for generic capabilities such as low-level infrastructure management. Therefore, it makes sense to introduce two additional teams in a multi-team micro-frontend organization: an enablement team and a platform team. These concepts are widely adopted in modern IT organizations with distributed systems and are well-documented in [Team Topologies](#).

The following diagram shows the enablement team providing tools, libraries, standards and testing to three micro-frontend teams. The platform team provides infrastructure, shared runtime capabilities, and domain services to those same three micro-frontend teams.



The platform team supports the micro-frontend teams by freeing them from undifferentiated heavy lifting. This support can include infrastructure services such as container runtimes, CI/CD pipelines, collaboration tooling, and monitoring. However, setting up a platform team should not lead to an organization in which development is detached from operations. The opposite is true: The platform team offers an engineering product, and micro-frontend teams have ownership and runtime responsibility of their services on the platform.

The enablement team provides support by focusing on governance and ensuring consistency across the micro-frontend teams. (The platform team should not be involved with this.) The

enablement team maintains shared resources such as a UI library, and it creates standards such as choice of framework, performance budgets, and interoperability conventions. At the same time, it provides new teams or team members with training in applying standards and tooling as defined by governance.

Deploying

The north star for autonomy in micro-frontend teams is to have an automated pipeline with a path to production that is independent from other micro-frontend teams. Teams that follow the share-nothing principle can implement independent pipelines. Teams that share libraries or depend on a platform team must decide how to manage dependencies in deployment pipelines.

Typically, each pipeline does the following:

- Builds frontend assets
- Deploys the assets to hosting for consumption
- Ensures that registries and caches are updated so that the new versions can be delivered to customers

The actual pipeline steps vary depending on the technology stack and the page composition approach.

For client-side composition, this means uploading application bundles to hosting buckets, and releasing to consumption through caching at a CDN. Applications that use browser caching with service workers should also implement ways to update the service-worker caches.

For server-side composition, this usually means deploying the new version of the server component and updating the micro-frontend registry to make the new version discoverable. You can use blue/green or canary deployment patterns to gradually roll out new versions.

Governance

Multiple personas typically work on micro-frontends, and each one works under different constraints toward common business goals. While communication and collaboration between people is key to success, overcommunicating and implementing overly complicated processes slows down the development cycle. This results in decreased morale, and it lowers the quality bar.

The most successful companies that implement micro-frontends by using multiple teams create mechanisms to balance autonomy with alignment. They empower decision makers to take action locally and to escalate hierarchically only when needed. Mechanisms include the following:

- [API contracts](#)
- [Cross-interactivity using events](#)
- [Balancing autonomy with alignment](#)
- [Feature flags](#)
- [Service discovery](#)

API contracts

Each micro-frontend is a system capable of encapsulating opinions, logic, and complexity. Cross-cutting concerns usually include the following:

- **Design systems** – Tooling to develop UIs distributed as libraries
- **Composition** – The way a micro-frontend needs to interact with an application shell to render and to inherit its context
- **Logic handling** – Interaction with APIs to handle persistent state
- **Interactivity with other micro-frontends** – Scenarios such as publishing and consuming events or navigating from one micro-frontend to another

To accelerate consumption and troubleshooting, it's common to invest in standardizing the way these interfaces are declared and documented, including micro-frontend dependencies. Wikis curated by humans are a good start. A more scalable approach is to store this information as structured metadata in code. You can then centralize it for consumption by using automation to track historic changes and provide full-text search.

When micro-frontends involve a large number of teams, you need a strategy to coordinate between teams. Sharing API contracts in a unified way becomes a must because it reduces communication overhead and improves developer experience.

[OpenAPI](#) is a specification language for HTTP APIs that supports defining API interfaces and contracts in a unified way. You can implement REST APIs by [using OpenAPI in Amazon API Gateway](#). You can also use a wide variety of open source frameworks that you can host in containers or virtual machines. A significant advantage is that OpenAPI can automatically generate documentation in a consistent format, so multiple teams can share knowledge with a minimal initial investment.

When multiple teams work on micro-frontends, they often form groups. In these groups, people can meet and learn from each other while thinking about and contributing to the bigger picture. These initiatives typically define and document ownership boundaries, discuss cross-cutting concerns, and identify early on any duplication of efforts to solve common problems.

Cross-interactivity using events

In some scenarios, multiple micro-frontends might need to interact with each other to react to state changes or user actions. For example, multiple micro-frontends on the page can include collapsible menus. A menu appears when the user chooses a button. The menu is hidden when the user clicks anywhere else, including another menu that is rendered within a different micro-frontend.

Technically, a shared state library such as Redux can be used by multiple micro-frontends and coordinated by a shell. However, that creates significant coupling between applications, resulting in code that is harder to test, and it might slow performance during rendering.

One common, effective approach is to develop an event bus that's distributed as a library, orchestrated by the application shell, and used by multiple micro-frontends. In this way, each micro-frontend publishes and listens to particular events asynchronously, basing its behavior on its own internal state only. Then, multiple teams can maintain a shared wiki page that describes events and documents behaviors that have been agreed on by user experience designers.

In an implementation of the event-bus example, a dropdown component uses the shared bus to publish an event called `drop-down-open-menu` with a payload of `{"id": "homepage-about-us-button"}`. The component adds a listener to the `drop-down-open-menu` event to ensure that if an event is fired for a new ID, the dropdown component is rendered to hide its

collapsible section. In this way, the micro-frontend can react to changes asynchronously with increased performance and better encapsulation, making easier for multiple teams to design and test behaviors.

We recommend using standard APIs implemented natively by modern browsers to improve simplicity and maintainability. The [MDN Event reference](#) provides information about using events with client-side rendered applications.

Balancing autonomy with alignment

Micro-frontend architectures are strongly biased toward team autonomy. However, it's important to distinguish between areas that can support flexibility and diverse approaches to solve problems, and areas where standardization is necessary to achieve alignment. Senior leaders and architects must identify these areas early on and prioritize investments to balance security, performance, operational excellence, and reliability of micro-frontends. Finding this balance involves the following: micro-frontend creation, testing, release, and logging, monitoring, and alerting.

Creating micro-frontends

Ideally, all teams are strongly aligned to maximize benefits in terms of end-user performance. In practice, this can be hard, and it might require more effort. We recommend starting with some written guidelines that multiple teams can contribute to through open and transparent debate. Teams can then gradually adopt the Cookiecutter software pattern, which supports creating tools that provide a unified way to scaffold a project.

Using this approach, you can bake in opinions and constraints. The downside is that these tools require significant investment for creation and maintenance, and to ensure that blockers are addressed quickly without affecting developer productivity.

End-to-end testing for micro-frontends

Unit testing can be left to owners. We recommend implementing a strategy early on to cross-test micro-frontends running on a unique shell. The strategy includes the capability to test applications before and after a production release. We recommend developing processes and documentation for technical and nontechnical people to test critical functionalities manually.

It's important to ensure that changes don't degrade either the functional or the nonfunctional customer experience. An ideal strategy is to gradually invest in automated testing, both for key features and for architecture characteristics such as security and performance.

Releasing micro-frontends

Each team might have its own way to deploy their code, bake in opinions, and own infrastructure. The cost of complexity for maintaining such systems is usually a deterrent. Instead, we recommend investing early on to implement a shared strategy that can be enforced by shared tools.

Develop templates with the CI/CD platform of choice. Teams can then use the preapproved templates and shared infrastructure to release changes to production. You can begin investing in this development work early because these systems rarely need significant updates after an initial period of testing and consolidation.

Logging and monitoring

Each team can have different business and system metrics that they want to track for operational or analytics purposes. The Cookiecutter software pattern can be applied here as well. The delivery of events can be abstracted and made available as a library that multiple micro-frontends can consume. To balance flexibility and provide autonomy, develop tools for logging custom metrics and creating custom dashboards or reports. The reporting promotes close collaboration with product owners and reduces the end-customer feedback loop.

By standardizing the delivery, multiple teams can collaborate to track metrics. For example, an e-commerce website can track the user journey from the "Product details" micro-frontend to the "Cart" micro-frontend, to the "Purchase" micro-frontend to measure engagement, churn, and issues. If each micro-frontend logs events by using a single library, you can consume this data as whole, explore it holistically, and identify insightful trends.

Alerting

Similar to logging and monitoring, alerting benefits from standardization with room for a degree of flexibility. Different teams might react differently to functional and non-functional alerts. However, if all teams have a consolidated way to initiate alerts based on metrics that are collected and analyzed on a shared platform, the business can identify cross-team issues. This capability is useful during incident-management events. For example, alerts can be initiated by the following:

- Elevated number of JavaScript client-side exceptions on a particular browser version
- Time to render significantly degraded over a given threshold
- Elevated number of 5xx status codes when consuming a particular API

Depending on the maturity of your system, you can balance your efforts on different parts of your infrastructure, as shown in the following table.

Adoption	Research and development	Ascent	Maturity
Create micro-frontends.	Experiment, document, and share learnings.	Invest in tooling to scaffold new micro-frontends. Evangelize adoption.	Consolidate tooling for scaffolding. Push for adoption.
Test micro-frontends end to end.	Implement mechanisms for manually testing all related micro-frontends.	Invest in tooling for automated security and performance testing. Investigate feature flags and service discovery.	Consolidate tooling for service discovery, testing in production, and automated end-to-end testing.
Release micro-frontends.	Invest in a shared CI/CD infrastructure and automated multiple-environment releases. Evangelize adoption.	Consolidate tooling for CI/CD infrastructure. Implement manual rollback mechanisms. Push for adoption.	Create mechanisms to initiate automated rollbacks on top of system and business metrics and alerts.
Observe micro-frontend performance.	Invest in a shared monitoring infrastructure and library for consistent logging of system and business events.	Consolidate tooling for monitoring and alerting. Implement cross-team dashboards to monitor general health and improve incident management.	Standardize logging schemas. Optimize for cost. Implement alerting based on complex business metrics.

Feature flags

Feature flags can be implemented in micro-frontends to facilitate the coordination of testing and releasing features in multiple environments. The feature flag technique consists of centralizing decisions in a Boolean-based store, and driving behavior based on that. It's often used to silently propagate changes that can be kept hidden until a specific moment in time, while unlocking new releases for new features that would otherwise be blocked, reducing team velocity.

Consider the example of teams working on a micro-frontend feature that will be launched on a specific date. The feature is ready, but it needs to be released together with a change on another micro-frontend that is independently released. Blocking the release of both micro-frontends would be considered an anti-pattern and would increase risk when deployed.

Instead, the teams can create a Boolean feature flag in a database that they both consume during render time (perhaps through an HTTP call to a shared Feature Flags API). The teams can even release the change in a test environment where the Boolean value is set to `True` to verify cross-project functional and nonfunctional requirements before launching to production.

Another example of feature flag use is implementing a mechanism to override the value of a flag by setting a specific value through the `QueryString` parameter or storing a particular test string in a cookie. Product owners can iterate on features without blocking the release of other features or bug fixes until launch date. On the given date, changing the flag value on the database instantly makes the change visible in production, without the need for cross-team coordinated releases. After a feature is released, the development teams clean up the code to remove the old behavior.

Other use cases include releasing a context-based feature flag system. For example, if a single website serves customers in multiple languages, a feature might be available only for visitors of a particular country. The feature flag system can be dependent on the consumer sending the country context (for example by making use of the `Accept-Language` HTTP header), and there can be a different behavior depending on that context.

While feature flags are a powerful tool for facilitating collaboration between developers and product owners, they rely on people's diligence to avoid a significant degradation of the code base. Keeping flags active on multiple features can increase complexity when troubleshooting issues, increase JavaScript bundle size, and ultimately accumulate technical debt. Common mitigation activities include the following:

- Unit testing each feature behind a flag to reduce the probability of bugs, which can introduce longer feedback loops in the automated CI/CD pipelines that run the tests

- Creating tools to measure bundle size increases during code changes, which can be mitigated during code reviews

AWS offers a range of solutions for optimizing A/B testing on the edge by using Amazon CloudFront functions or Lambda@Edge. These approaches help to reduce the complexity of integrating a solution or the existing SaaS product that you are using to assert your assumptions. For more information, see [A/B testing](#).

Service discovery

The frontend discovery pattern improves the development experience when developing, testing, and delivering micro-frontends. The pattern uses a shareable configuration that describes the entry point of micro-frontends. The shareable configuration also includes additional metadata that is used for safe deployments in each environment by using canary releases.

Modern frontend development entails using a wide variety of tools and libraries to support modularity during development. Traditionally, this process consisted of bundling code into individual files that could be hosted in a CDN with the goal of keeping network calls at a minimum during runtime, including initial load (when an app opens in a browser) and usage (when a customer performs actions such as choosing buttons or inserting information).

Splitting bundles

Micro-frontend architectures solve the performance issues caused by very large bundles generated by individually bundling a large set of functionalities. For example, a very large e-commerce website can be bundled into a 6 MB JavaScript file. Despite compression, the size of that file might negatively impact the user's experience when loading the app and downloading the file from an edge-optimized CDN.

If you split the app into home page, product details, and cart micro-frontends, you can use a bundling mechanism to produce three individual 2 MB bundles. This change might improve the performance for first load by 300 percent when users consume the home page. The product or cart micro-frontends bundles are loaded asynchronously only if and when the user visits the product page for an item and decides to purchase it.

Many frameworks and libraries are available based on this approach, and there are advantages for both customers and developers. To identify business boundaries that can result in decoupling

dependencies in code, you can map different business functions to multiple teams. The distributed ownership introduces independence and agility.

When you split build packages, you can use a configuration to map micro-frontends and drive the orchestration for the initial load and for post-load navigation. Then, the configuration can be consumed during runtime rather than during build time. For example, the client-side frontend code or server-side backend code can make an initial network call to an API to dynamically fetch the list of micro-frontends. It also fetches the metadata that's required for composition and integration. You can configure failover strategies and caching for reliability and performance. Mapping the micro-frontends helps make individual deployments of micro-frontends to be discoverable by previously deployed micro-frontends that are orchestrated by a shell app.

Canary releases

A canary release is a well-established and popular pattern for deploying micro-services. Canary releases bucket the target users of a release into multiple groups, and release changes gradually as opposed to an immediate replacement (also known as a blue/green deployment). An example of a canary release strategy is to roll out a new change to 10 percent of the target users, and add 10 percent every minute, with a total duration of 10 minutes to reach 100 percent.

The goal of a canary release is to get early feedback about the changes, monitoring the system to reduce the impact of any issues. When automation is in place, business or system metrics can be monitored by an internal system that can stop the deployment or start a rollback.

For example, a change might introduce a bug that, in the first couple minutes of a release, results in a loss of revenue or a performance degradation. Automated monitoring can initiate an alarm. With the service discovery pattern, that alarm can stop the deployment and immediately roll back, affecting only 20 percent of users instead of 100 percent. The business benefits from the reduced scope of the issue.

For an example architecture that uses DynamoDB as storage to implement a REST Admin API, see the [Frontend Service Discovery on AWS solution](#) on GitHub. Use the AWS CloudFormation template to integrate the architecture in your own CI/CD pipelines. The solution includes a REST Consumer API for integrating the solution with your frontend applications.

Do you need a platform team?

Some companies have a team that's responsible for owning and maintaining code, infrastructure, and processes that are adopted by other teams to work on micro-frontends. Common responsibilities include:

- Create and maintain a CI/CD pipeline that can be used with repositories containing micro-frontends. Build and test code changes, and release them in multiple environments.
- Create and maintain observability-related tools such as shared dashboards, alerting mechanisms, and systems to react to issues.
- Create and maintain shared libraries for event handling, shared-service consumption, and third-party dependencies.
- Create and maintain tools that continuously monitor nonfunctional qualities such as performance, security, and reliability of the system.
- Create and maintain design systems.
- Create, maintain, and support the application shell for the micro-frontend system.

Depending on the scale of the project, you can manage these responsibilities by using one of the following approaches:

- Create a dedicated platform team whose only responsibility is to work on shared tools.
- Create a group composed of members from multiple teams. Group members split their time between working on micro-frontends and working on shared tooling. This is also known as a tiger team.

While the tiger-team approach is an effective way to stay customer focused, a tiger team often evolves into a platform team if the project gains traction and responsibilities. For both platform teams and tiger teams, the most successful companies working on micro-frontends form these teams so that multiple people with multiple backgrounds and skills can contribute. Team members might include backend engineers, frontend engineers, user experience (UX) designers, and technical product managers. This diversity pushes people to continuously engage in healthy debates and design with simplicity in mind.

Next steps

This guide covered architectural and organizational patterns, trade-offs for key decisions, and governance concerns related to micro-frontends. The tables summarize the trade-offs of practices discussed in this document in terms of the following dimensions:

- **Autonomy** – Each micro-frontend team's ability to independently evolve their implementation and release to end users.
- **Consistency** – The overall experience of the application where each micro-frontend behaves as expected. High consistency means micro-frontends are consistent with the rest of the application and are not detrimental to the user experience of the overall application.
- **Complexity** – The amount of infrastructure, code, and effort required to implement and test micro-frontends, the overall application, and governance controls.

Practice	Autonomy	Consistency	Complexity
Building with micro-frontends instead of monolithic applications	High	Medium	High
Code-sharing practices	Autonomy	Consistency	Complexity
Share nothing	High	Low	Low
Share cross-cutting concerns	Medium	High	Medium
Share business logic	Low	High	Medium
Share through libraries at build time	Medium	High	Low
Share at runtime	High	High	High

Micro-frontend discovery practices	Autonomy	Consistency	Complexity
Configure during application build	Low	High	Low
Server-side discovery	High	High	Medium
Client-side (runtime) discovery	High	High	Medium

View composition practices	Autonomy	Consistency	Complexity
Server-side composition	High	Medium	High
Edge-side composition	Medium	Medium	High
Client-side composition	High	Medium	Medium

To learn more about the concepts introduced in this guidance, see the [Resources](#) section.

Resources

- [Micro-frontends in context](#)
- [Domain Driven Design](#)
- [EDA Visuals](#)
- [Frontend Discovery](#)
- [Frontend Service Discovery on AWS](#)
- [Agile Manifesto](#)
- [MDN Event reference](#)
- [OpenAPI](#)

Contributors

The following individuals contributed to this guide.

- Matteo Figus, Principal Solutions Architect, AWS
- Alexander Guensche, Senior Solutions Architect, AWS
- Harun Hasdal, Senior Solutions Architect, AWS
- Luca Mezzalana, Principal Go to Market Specialist Solutions Architect Serverless UK, AWS

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	July 12, 2024