aws

Understanding and implementing microfrontends on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Understanding and implementing microfrontends on AWS

Table of Contents

Understanding and implementing micro-frontends on AWS

Amazon Web Services ([contributors](#))

July 2024 ([document history](#))

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.
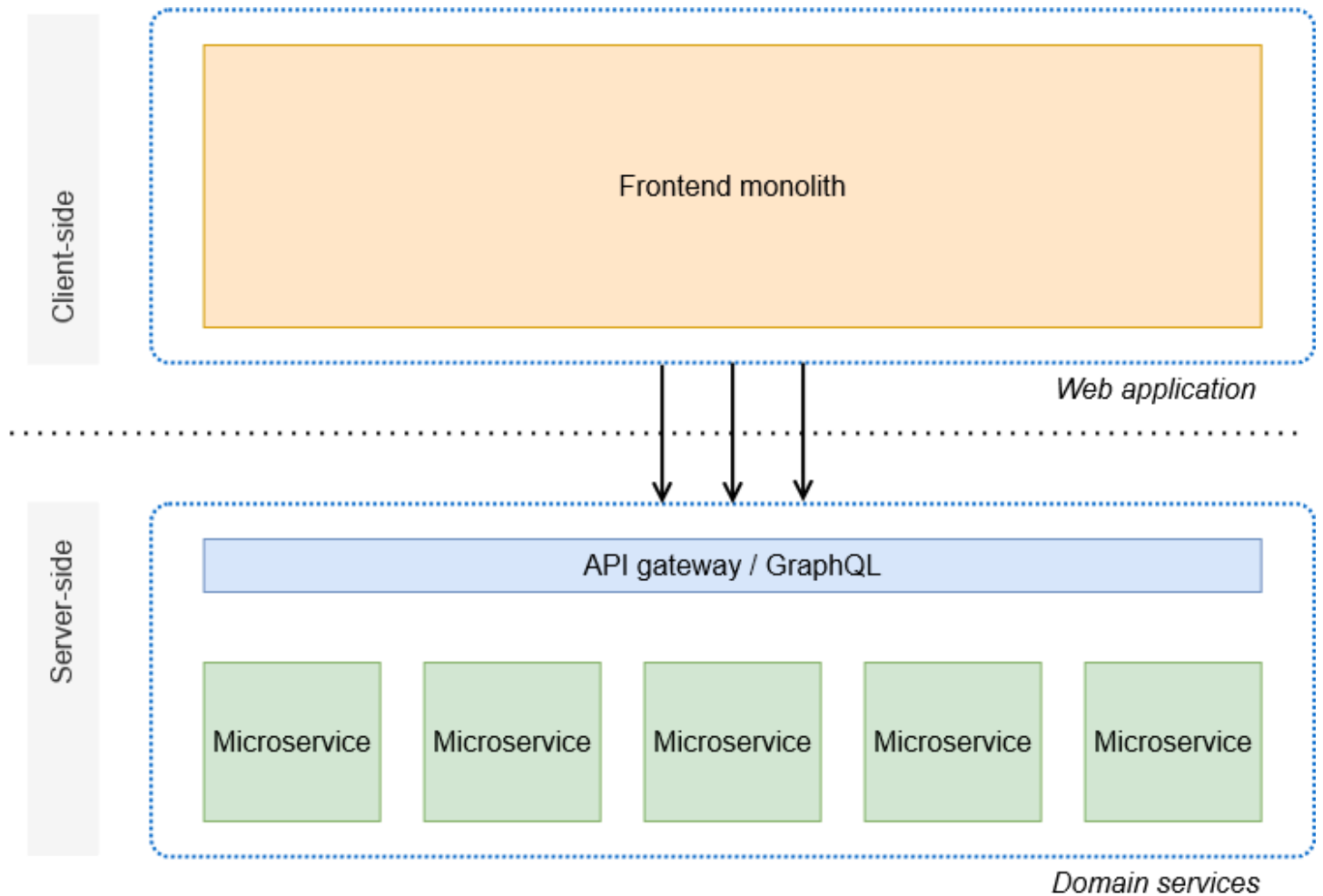
The following diagram shows a micro-frontend architecture with different implementations of microservices.
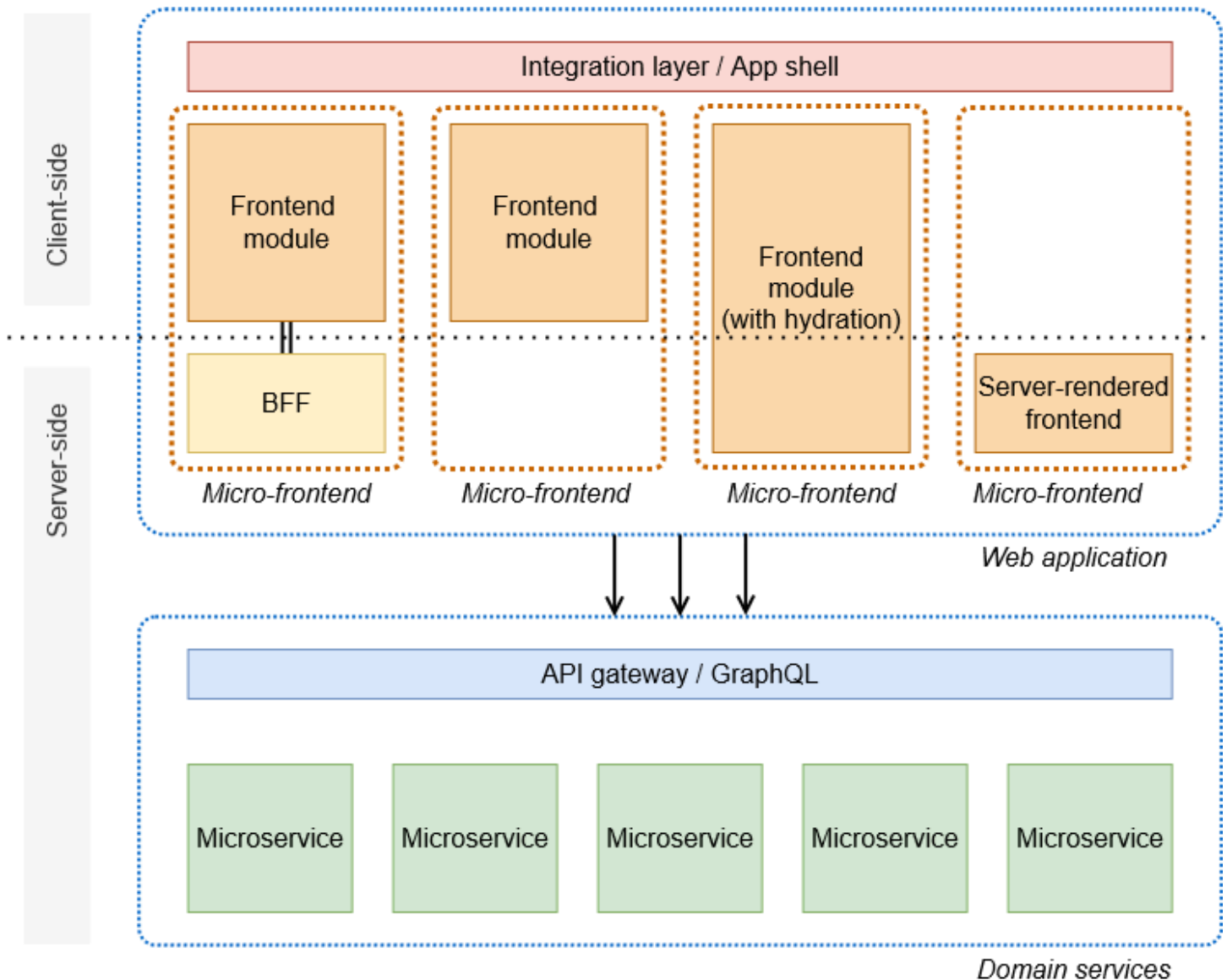
As shown in the previous diagram, you can use micro-frontends with client-side rendering or server-side rendering architectures:

- 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.

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 midsized 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. For more information about separation of concerns, see the Serverless Land diagram.

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.
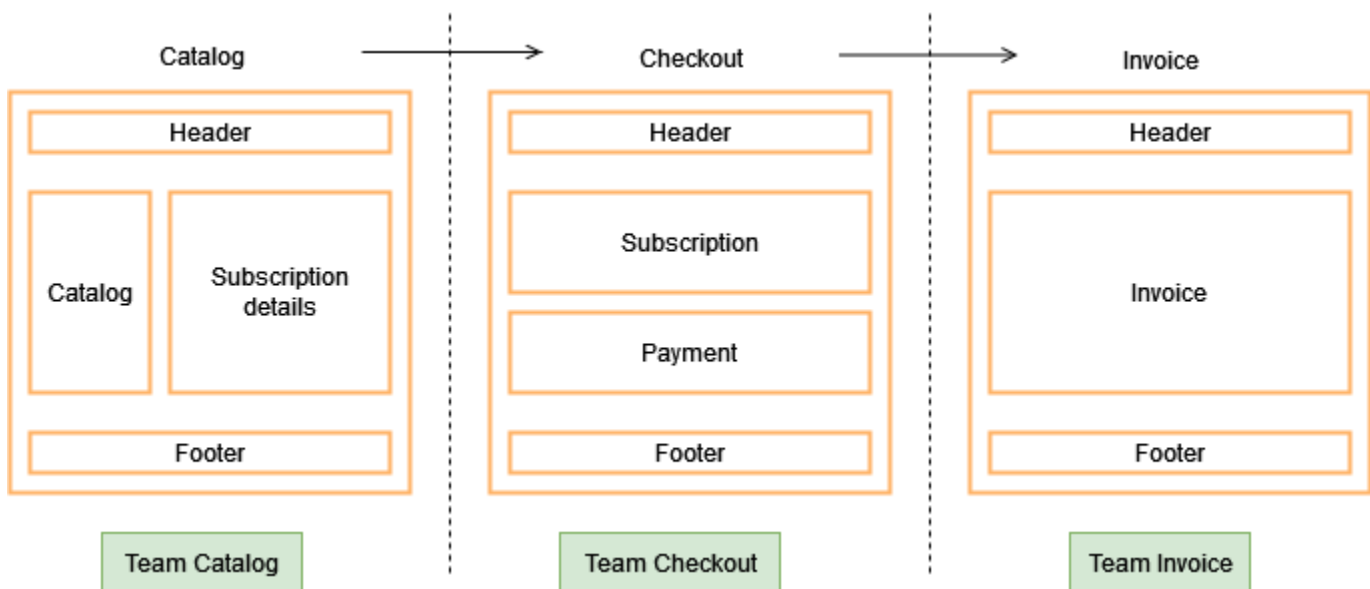
For some applications, vertical split alone might not be enough. For example, some functionality might need to be provided in many views. For these applications, you can apply a mixed split. The following diagram shows a mixed split solution in which micro-frontends for Station finder and Route explorer both use the Map view functionality.
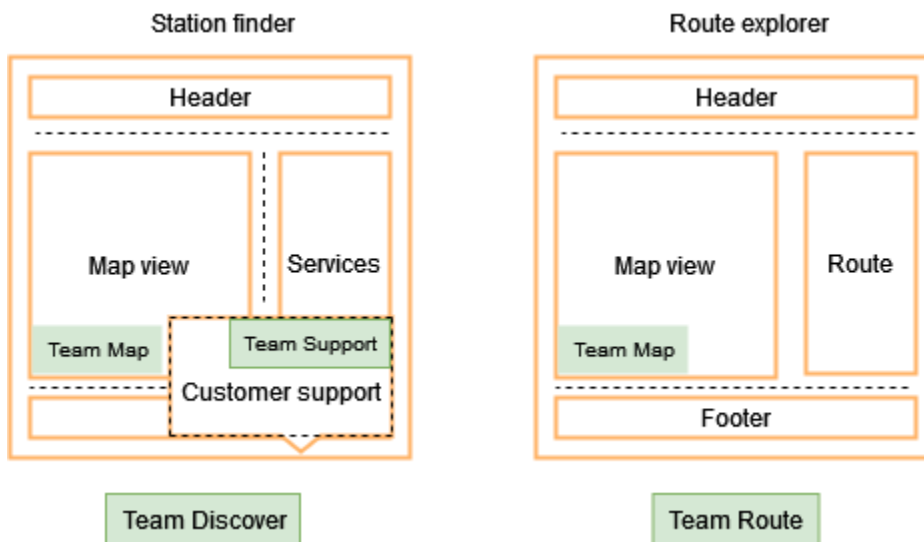
Portal-type or dashboard-type applications typically bring frontend capabilities together in a single view. In these types of applications, each widget can be delivered as a micro-frontend, and the hosting application defines the constraints and interfaces that the micro-frontends should implement.

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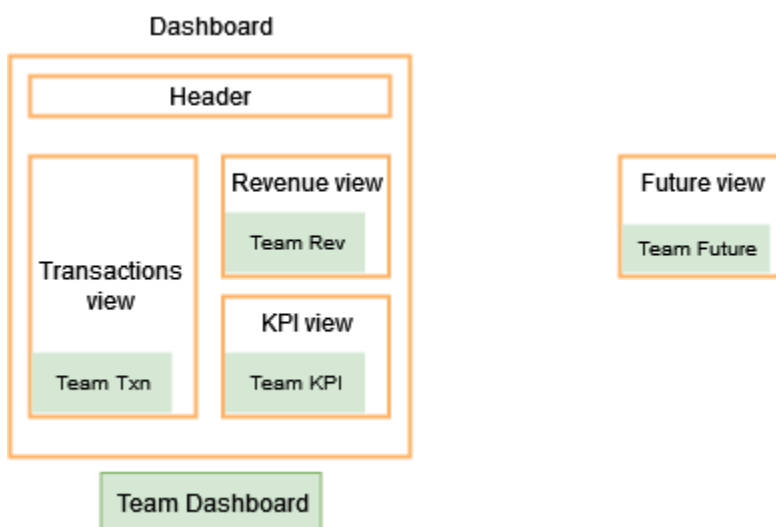
In the diagram, the future view represents new features developed by new teams to scale Team Dashboard and the dashboard capabilities.

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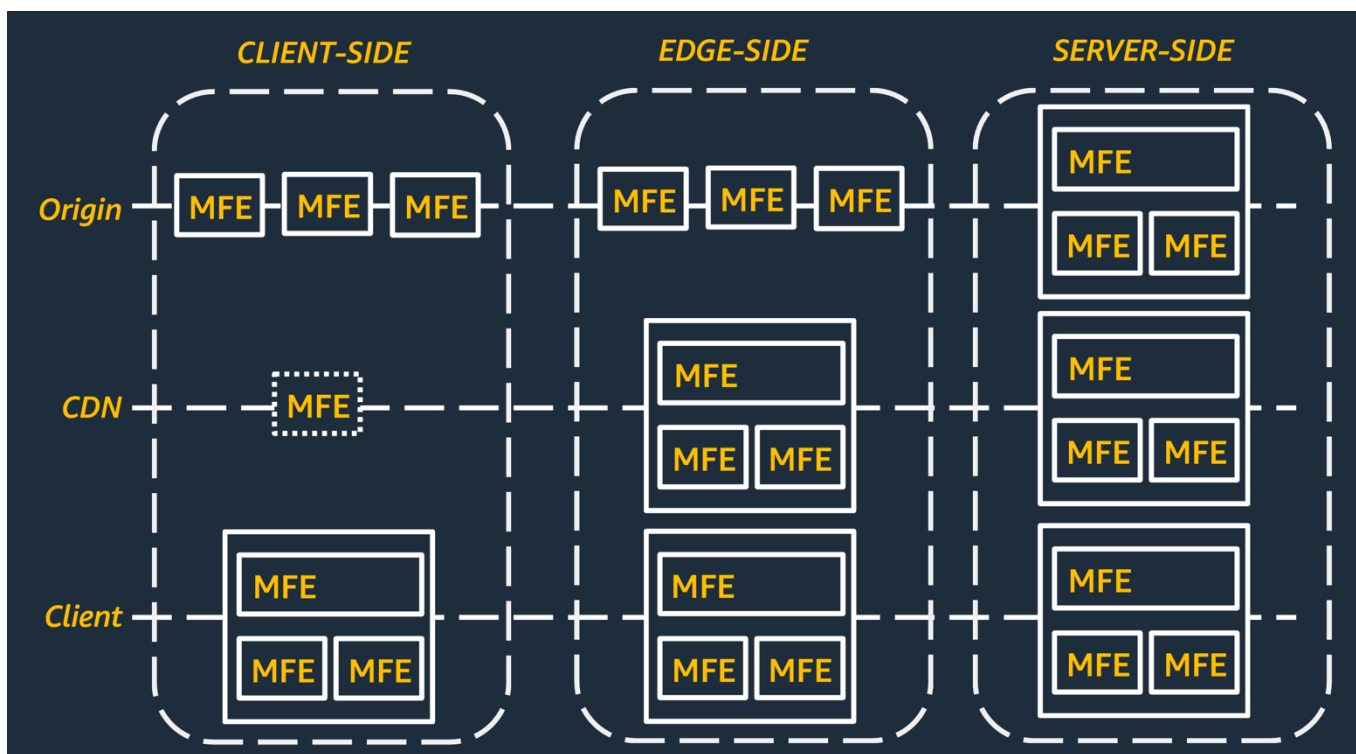
The client-side, edge-side and server-side layers are discussed in the following sections.

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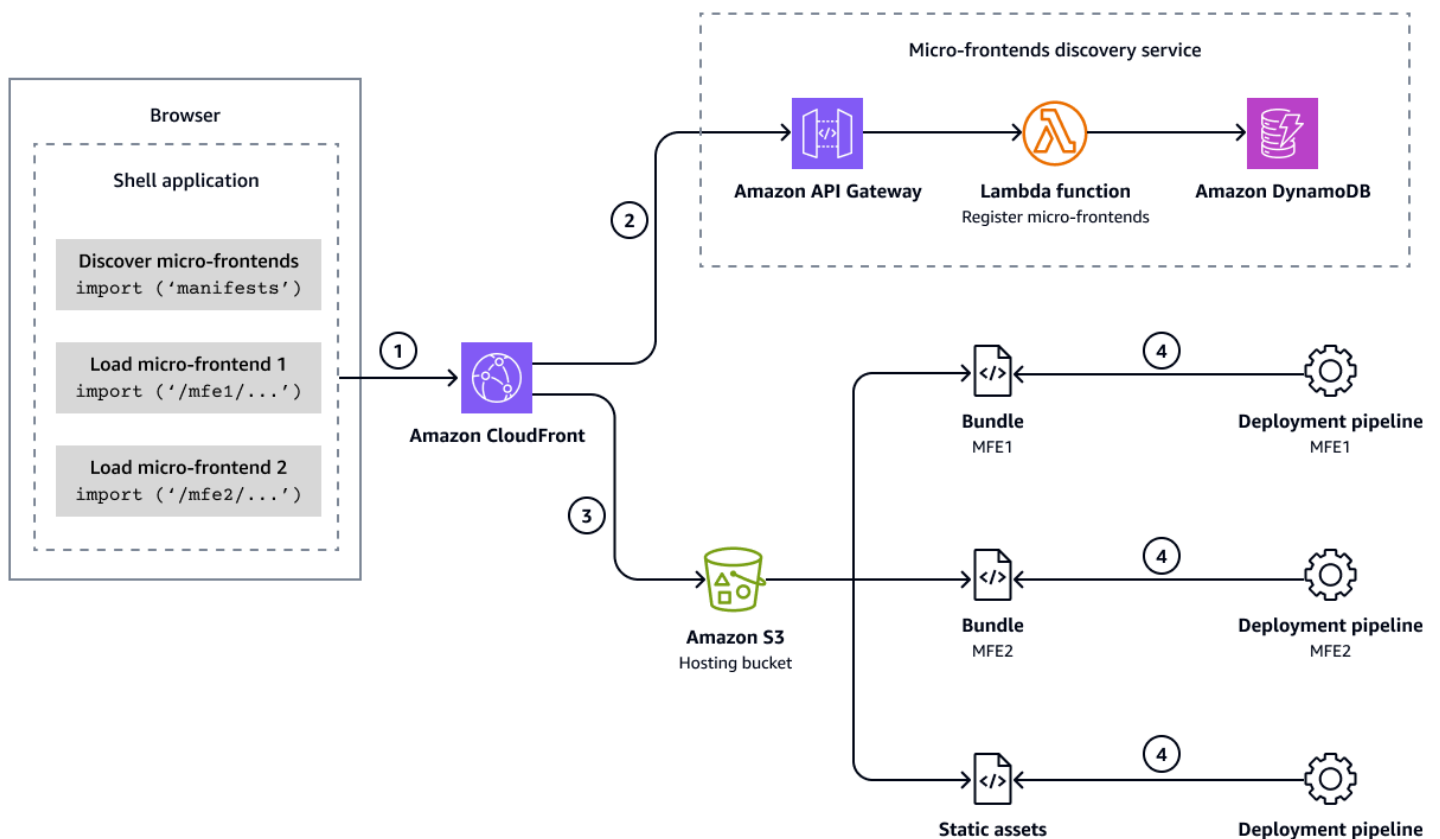
Client-side composition happens in the browser environment through a shell application. The diagram shows the following details:

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.
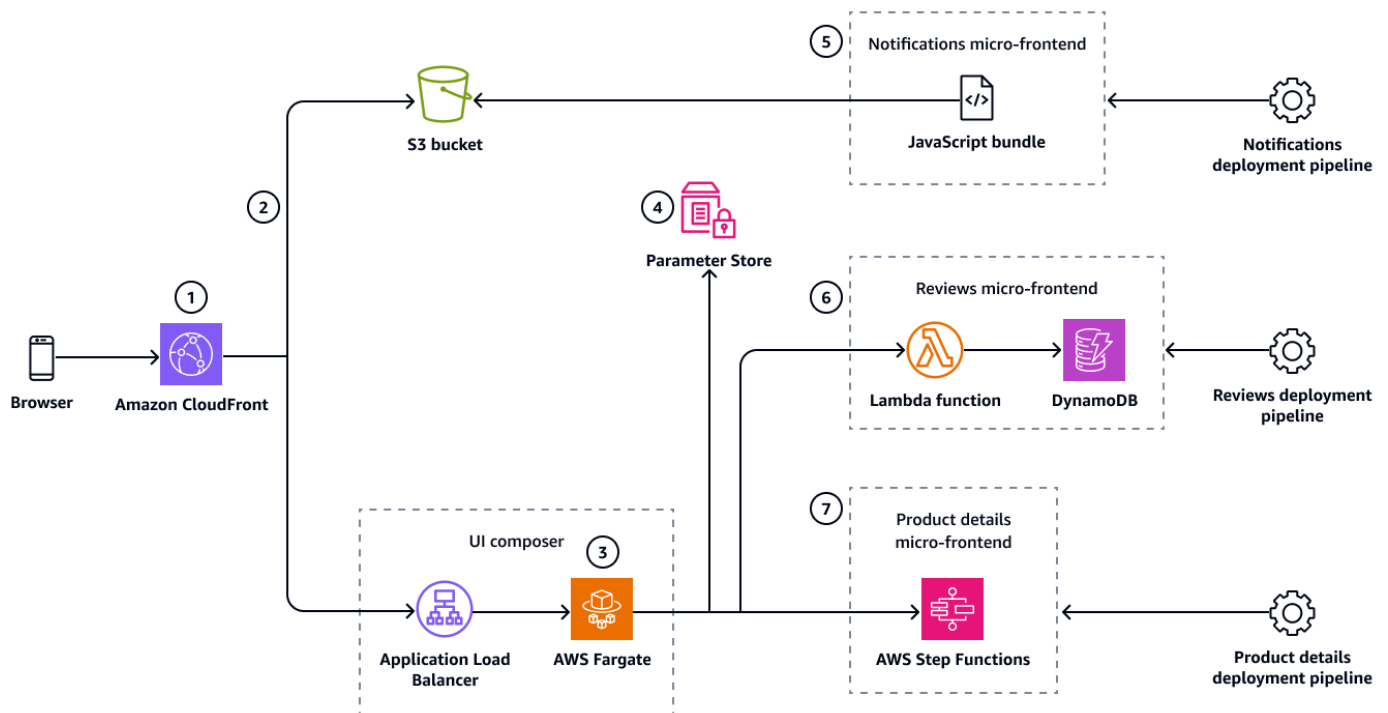
The diagram includes the following components and processes:

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 frontend 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 bidirectional 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 not 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.
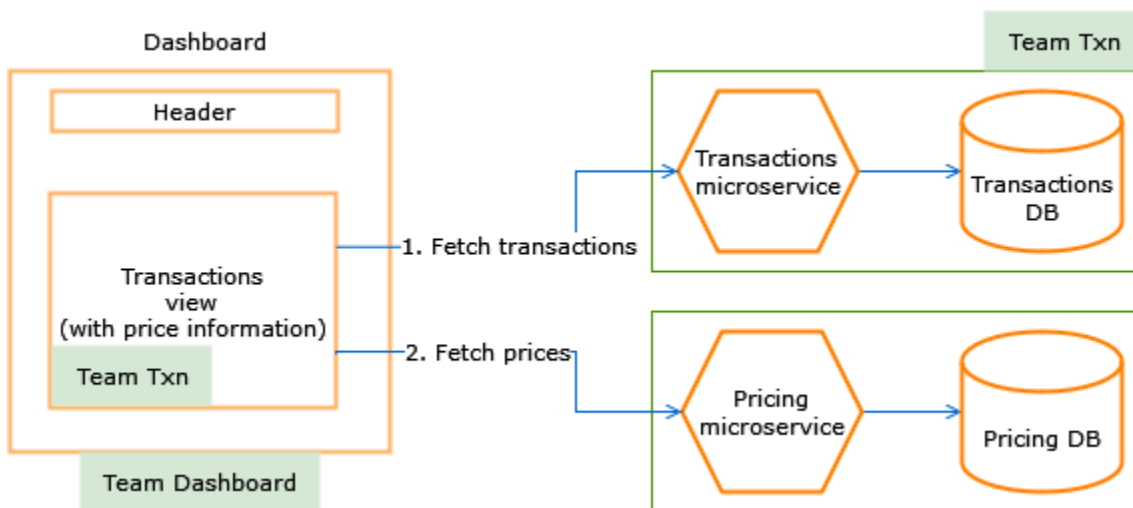
Instead, with the BFF pattern in the following diagram, micro-frontends can communicate with their own backend and fetch aggregated data.
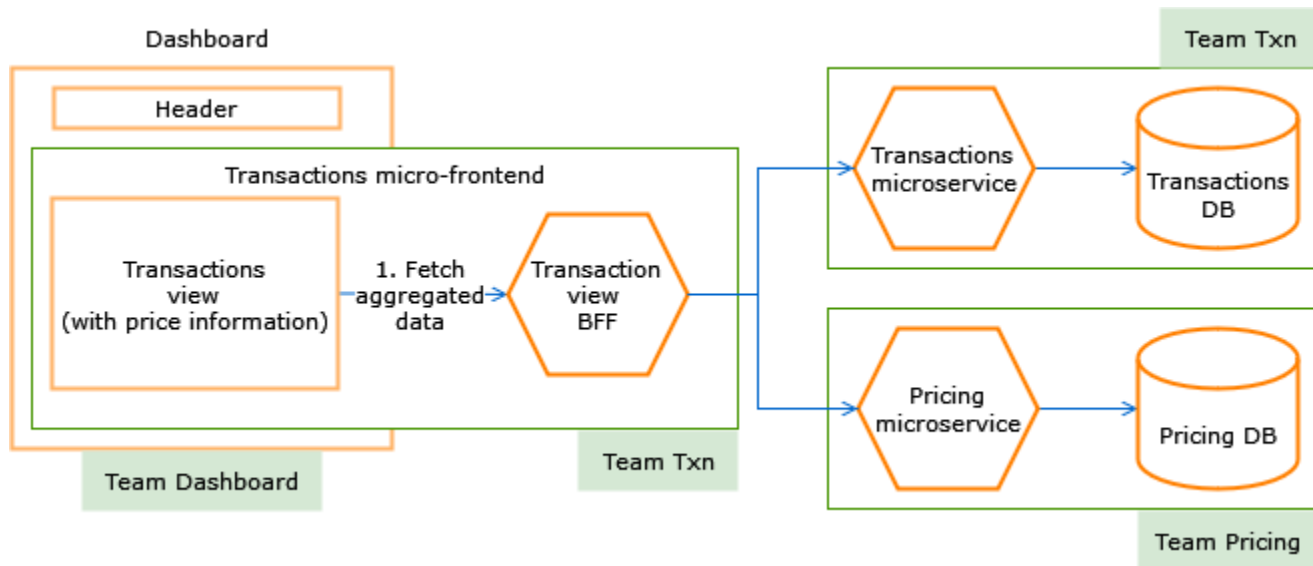
Teams can develop BFFs for different channels such as mobile, web, or specific views, with requirements to optimize backend interactions by reducing chattiness.

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 override to 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-frontend 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 priniciple 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.
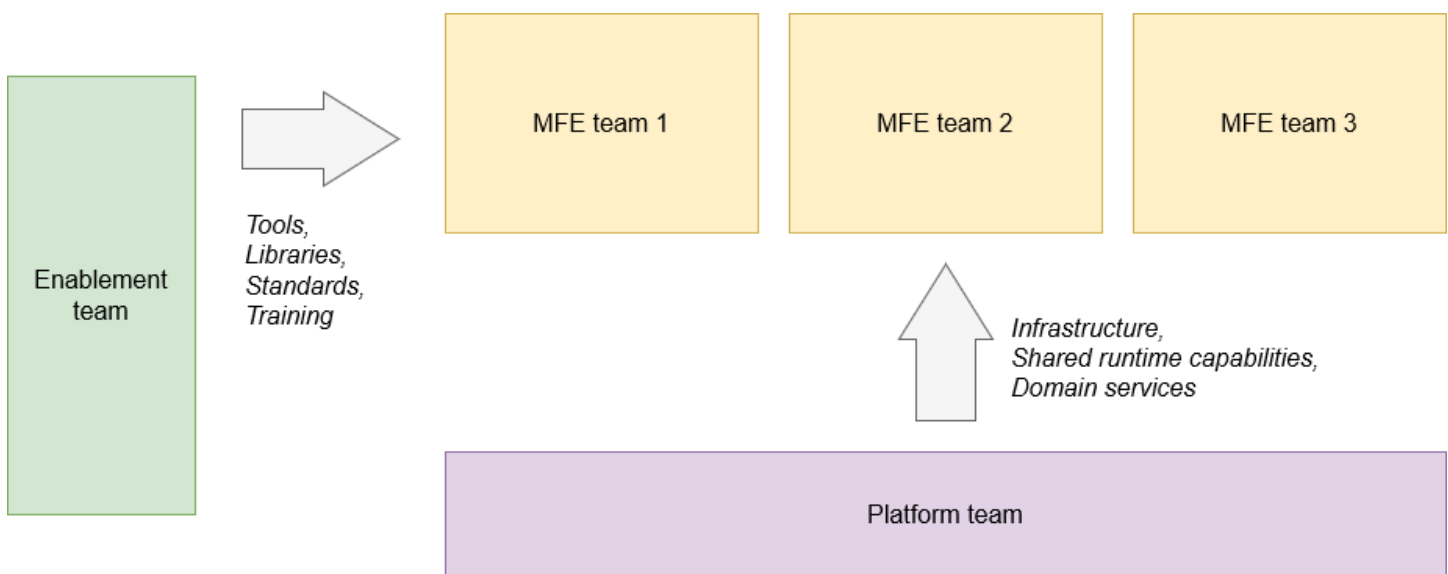
The platform team supports the micro-frontend teams by freeing them from undifferentiated heavy lifting. This support can include infrastructure services such as container runtimes, CI/CD pipelines, collaboration tooling, and monitoring. However, setting up a platform team should not lead to an organization 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-aboutus-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

View composition practices	Autonomy	Consistency	Complexity
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 Mezzalira, Principal Go to Market Specialist Solutions Architect Serverless UK, AWS

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	July 12, 2024

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

> Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:
>
> - Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
>
> - Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
>
> - Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
>
> - Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
>
> - Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
>
> - Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See [attribute-based access control](#).

abstracted services

See [managed services](#).

ACID

See [atomicity, consistency, isolation, durability](#).

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than [active-passive migration](#).

active-passive migration

A database migration method in which in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include SUM and MAX.

AI

See [artificial intelligence](#).

AIOps

See [artificial intelligence operations](#).

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see [ABAC for AWS](#) in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the [AWS CAF website](#) and the [AWS CAF whitepaper](#).

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

> A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

> See business continuity planning.

behavior graph

> A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

> A system that stores the most significant byte first. See also endianness.

binary classification

> A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as "Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

> A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

> A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

> A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

>   See [AWS Cloud Adoption Framework](#).

canary deployment

>   The slow and incremental release of a version to end users. When you are confident, you deploy
>   the new version and replace the current version in its entirety.

CCoE

>   See [Cloud Center of Excellence](#).

CDC

>   See [change data capture](#).

change data capture (CDC)

>   The process of tracking changes to a data source, such as a database table, and recording
>   metadata about the change. You can use CDC for various purposes, such as auditing or
>   replicating changes in a target system to maintain synchronization.

chaos engineering

>   Intentionally introducing failures or disruptive events to test a system's resilience. You can use
>   [AWS Fault Injection Service (AWS FIS)](#) to perform experiments that stress your AWS workloads
>   and evaluate their response.

CI/CD

>   See [continuous integration and continuous delivery](#).

classification

>   A categorization process that helps generate predictions. ML models for classification problems
>   predict a discrete value. Discrete values are always distinct from one another. For example, a
>   model might need to evaluate whether or not there is a car in an image.

client-side encryption

>   Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or Bitbucket Cloud. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, AWS Panorama offers devices that add CV to on-premises camera networks, and Amazon SageMaker AI provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See [computer vision](#).

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see [Data classification](#).

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see [Building a data perimeter on AWS](#).

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a [star schema](#), a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a [disaster](#). For more information, see [Disaster Recovery of Workloads on AWS: Recovery in the Cloud](#) in the AWS Well-Architected Framework.

DML

See [database manipulation language](#).

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see [Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway](#).

DR

See [disaster recovery](#).

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to [detect drift in system resources](#), or you can use AWS Control Tower to [detect changes in your landing zone](#) that might affect compliance with governance requirements.

DVSM

See [development value stream mapping](#).

E

EDA

   See exploratory data analysis.

EDI

   See electronic data interchange.

edge computing

   The technology that increases the computing power for smart devices at the edges of an IoT
   network. When compared with cloud computing, edge computing can reduce communication
   latency and improve response time.

electronic data interchange (EDI)

   The automated exchange of business documents between organizations. For more information,
   see What is Electronic Data Interchange.

encryption

   A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

   A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys
   can vary in length, and each key is designed to be unpredictable and unique.

endianness

   The order in which bytes are stored in computer memory. Big-endian systems store the most
   significant byte first. Little-endian systems store the least significant byte first.

endpoint

   See service endpoint.

endpoint service

   A service that you can host in a virtual private cloud (VPC) to share with other users. You can
   create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts
   or to AWS Identity and Access Management (IAM) principals. These accounts or principals
   can connect to your endpoint service privately by creating interface VPC endpoints. For more

information, see [Create an endpoint service](#) in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, [MES](#), and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see [Envelope encryption](#) in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.

- lower environments – All development environments for an application, such as those used for initial builds and tests.

- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.

- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the [program implementation guide](#).

ERP

See [enterprise resource planning](#).

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a [star schema](#). It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see [AWS Fault Isolation Boundaries](#).

feature branch

See [branch](#).

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see [Machine learning model interpretability with AWS](#).

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into "2021", "May", "Thu", and "15", you can help the learning algorithm learn nuanced patterns associated with different data components.

few-shot prompting

Providing an LLM with a small number of examples that demonstrate the task and desired output before asking it to perform a similar task. This technique is an application of in-context learning, where models learn from examples (*shots*) that are embedded in prompts. Few-shot prompting can be effective for tasks that require specific formatting, reasoning, or domain knowledge. See also zero-shot prompting.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

FM

See foundation model.

foundation model (FM)

A large deep-learning neural network that has been training on massive datasets of generalized and unlabeled data. FMs are capable of performing a wide variety of general tasks, such as understanding language, generating text and images, and conversing in natural language. For more information, see What are Foundation Models.

G

generative AI

A subset of AI models that have been trained on large amounts of data and that can use a simple text prompt to create new content and artifacts, such as images, videos, text, and audio. For more information, see What is Generative AI.

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

golden image

A snapshot of a system or software that is used as a template to deploy new instances of that system or software. For example, in manufacturing, a golden image can be used to provision software on multiple devices and helps improve speed, scalability, and productivity in device manufacturing operations.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries.

Detective guardrails detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See [high availability](#).

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. [AWS provides AWS SCT](#) that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

holdout data

A portion of historical, labeled data that is withheld from a dataset that is used to train a [machine learning](#) model. You can use holdout data to evaluate the model performance by comparing the model predictions against the holdout data.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see [What is IoT?](#)

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see [Machine learning model interpretability with AWS](#).

IoT

See [Internet of Things](#).

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the [operations integration guide](#).

ITIL

See [IT information library](#).

ITSM

See [IT service management](#).

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large language model (LLM)

A deep learning AI model that is pretrained on a vast amount of data. An LLM can perform multiple tasks, such as answering questions, summarizing documents, translating text into other languages, and completing sentences. For more information, see What are LLMs.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

LLM

See large language model.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

> See [manufacturing execution system](#).

Message Queuing Telemetry Transport (MQTT)

> A lightweight, machine-to-machine (M2M) communication protocol, based on the [publish/subscribe](#) pattern, for resource-constrained [IoT](#) devices.

microservice

> A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see [Integrating microservices by using AWS serverless services](#).

microservices architecture

> An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see [Implementing microservices on AWS](#).

Migration Acceleration Program (MAP)

> An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

> The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the [AWS migration strategy](#).

migration factory

> Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners,

migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

    See [origin access control](#).

OAI

    See [origin access identity](#).

OCM

    See [organizational change management](#).

offline migration

    A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

    See [operations integration](#).

OLA

    See [operational-level agreement](#).

online migration

    A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

    See [Open Process Communications - Unified Architecture](#).

Open Process Communications - Unified Architecture (OPC-UA)

    A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

    An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic PUT and DELETE requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See [operational readiness review](#).

OT

See [operational technology](#).

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The [AWS Security Reference Architecture](#) recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see [Permissions boundaries](#) in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See [personally identifiable information](#).

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See [programmable logic controller](#).

PLM

See [product lifecycle management](#).

policy

An object that can define permissions (see [identity-based policy](#)), specify access conditions (see [resource-based policy](#)), or define the maximum permissions for all accounts in an organization in AWS Organizations (see [service control policy](#)).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see [Enabling data persistence in microservices](#).

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see [Evaluating migration readiness](#).

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see [Preventative controls](#) in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in [Roles terms and concepts](#) in the IAM documentation.

privacy by design

A system engineering approach that takes privacy into account through the whole development process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see [Working with private hosted zones](#) in the Route 53 documentation.

proactive control

A [security control](#) designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the [Controls reference guide](#) in the AWS Control Tower documentation and see [Proactive controls](#) in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See [environment](#).

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

prompt chaining

Using the output of one [LLM](#) prompt as the input for the next prompt to generate better responses. This technique is used to break down a complex task into subtasks, or to iteratively refine or expand a preliminary response. It helps improve the accuracy and relevance of a model's responses and allows for more granular, personalized results.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

RAG

See Retrieval Augmented Generation.

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

Retrieval Augmented Generation (RAG)

A generative AI technology in which an LLM references an authoritative data source that is outside of its training data sources before generating a response. For example, a RAG model might perform a semantic search of an organization's knowledge base or custom data. For more information, see What is RAG.

rotation

> The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

> The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

> See recovery point objective.

RTO

> See recovery time objective.

runbook

> A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

> An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

> See supervisory control and data acquisition.

SCP

> See service control policy.

secret

> In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata.

The secret value can be binary, a single string, or multiple strings. For more information, see [What's in a Secrets Manager secret?](#) in the Secrets Manager documentation.

security by design

A system engineering approach that takes security into account through the whole development process.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: [preventative](#), [detective](#), [responsive](#), and [proactive](#).

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as [detective](#) or [responsive](#) security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see [Service control policies](#) in the AWS Organizations documentation.

service endpoint

> The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see [AWS service endpoints](#) in *AWS General Reference*.

service-level agreement (SLA)

> An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

> A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

> A target metric that represents the health of a service, as measured by a [service-level indicator](#).

shared responsibility model

> A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see [Shared responsibility model](#).

SIEM

> See [security information and event management system](#).

single point of failure (SPOF)

> A failure in a single, critical component of an application that can disrupt the system.

SLA

> See [service-level agreement](#).

SLI

> See [service-level indicator](#).

SLO

> See [service-level objective](#).

split-and-seed model

> A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your

organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

system prompt

A technique for providing context, instructions, or guidelines to an LLM to direct its behavior. System prompts help set context and establish rules for interactions with users.

T

tags

> Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

> The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

> A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

> See environment.

training

> To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

> A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

> An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see [Using AWS Organizations with other AWS services](#) in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the [Quantifying uncertainty in deep learning systems](#) guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See [environment](#).

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zero-shot prompting

Providing an LLM with instructions for performing a task but no examples (*shots*) that can help guide it. The LLM must use its pre-trained knowledge to handle the task. The effectiveness of zero-shot prompting depends on the complexity of the task and the quality of the prompt. See also few-shot prompting.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.