aws SUMMIT

INDIA | MAY 25, 2023

BMARC008

Building large scale micro frontends

Ajay Nagar (he/his)
Sr. Cloud Application Architect
AWS India

Jayesh Shinde (he/his)
Cloud Application Architect
AWS India

Agenda

- Background
- Principles of Micro-Frontends
- Architecture patterns
- Implementation techniques
- Example illustrations
- Anti patterns
- Conclusions

Background

Driving factors

Large Enterprises

- Monolithic in-house apps with operational complexity & large deployment teams
- Plethora of applications and lack of visual and functional integrations between them
- With business transformation shift from application to business centric development
- Frequent independent deployments
- Multiple targeted platform like web and mobile
- Inconsistent user experience
- Different UI frameworks / tech stacks

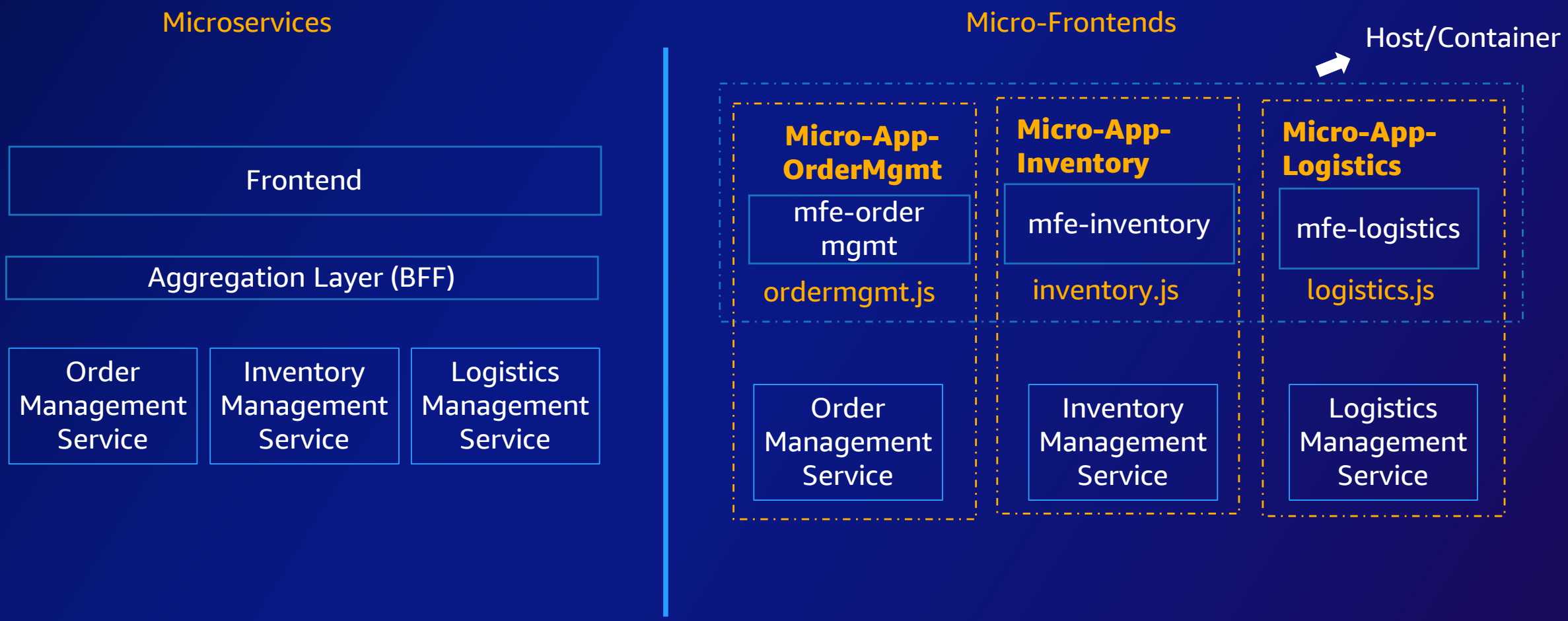
Principles of Micro-Frontends

What are Micro Frontends?

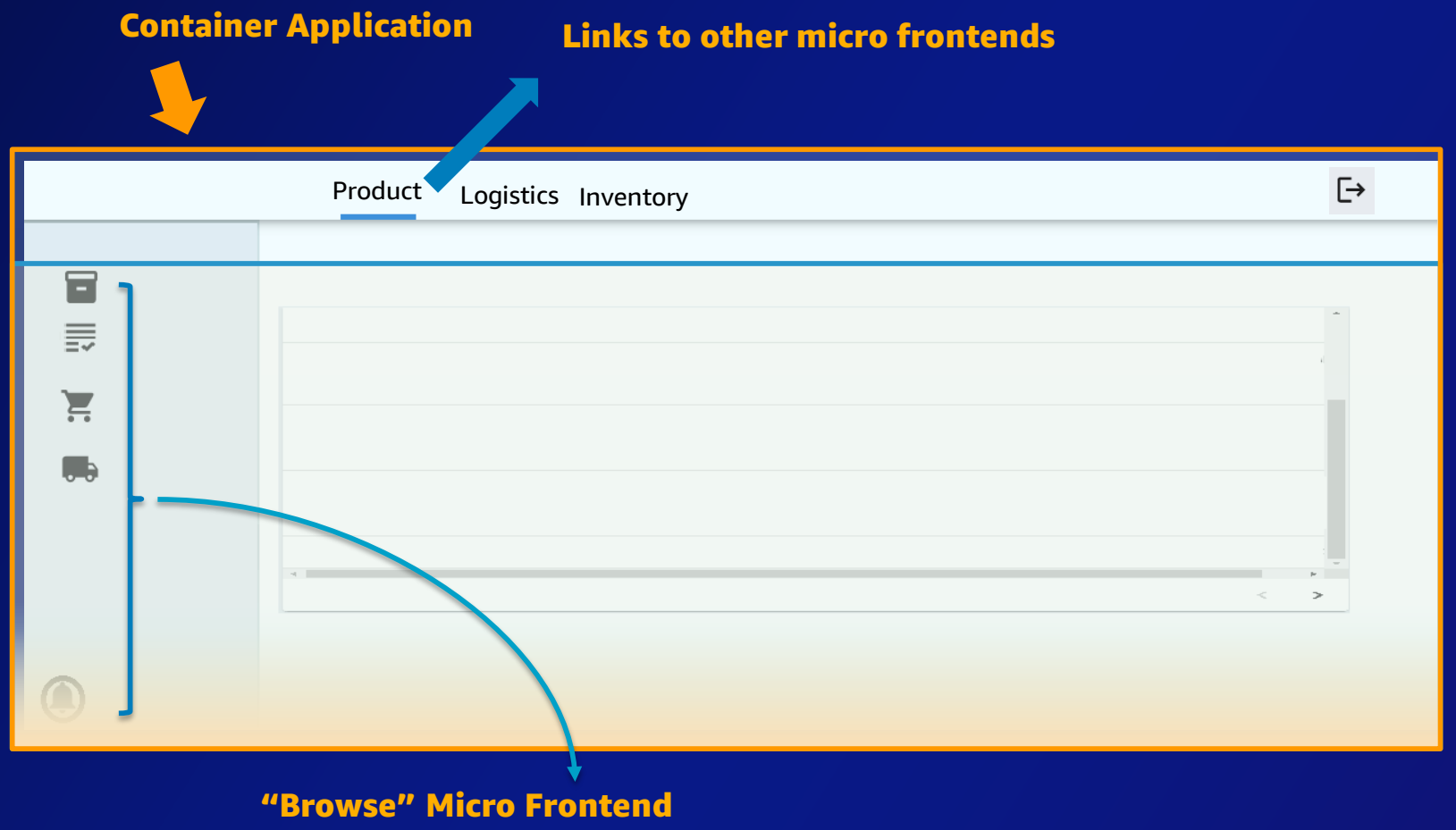
Micro frontends is an architecture patterns for decomposing frontend monoliths into smaller, simpler chunks that can be developed, tested and deployed independently, while still appearing to customers as a single cohesive product.

“An architectural style where independently deliverable frontend applications are composed into a greater whole”

How does it work?



Micro-Frontends – an example



1. Host/Driver application wires together micro apps
2. Micro-App is a vertical slice of functionality
3. Technical representation of a business subdomain
4. Independent development and deployment
5. Polygot in terms of UI framework
6. Solution interface appears to end-users as a single cohesive whole

Benefits

Business Agility
Faster Time to Market

Be Technology Agnostic

Build a Resilient Site

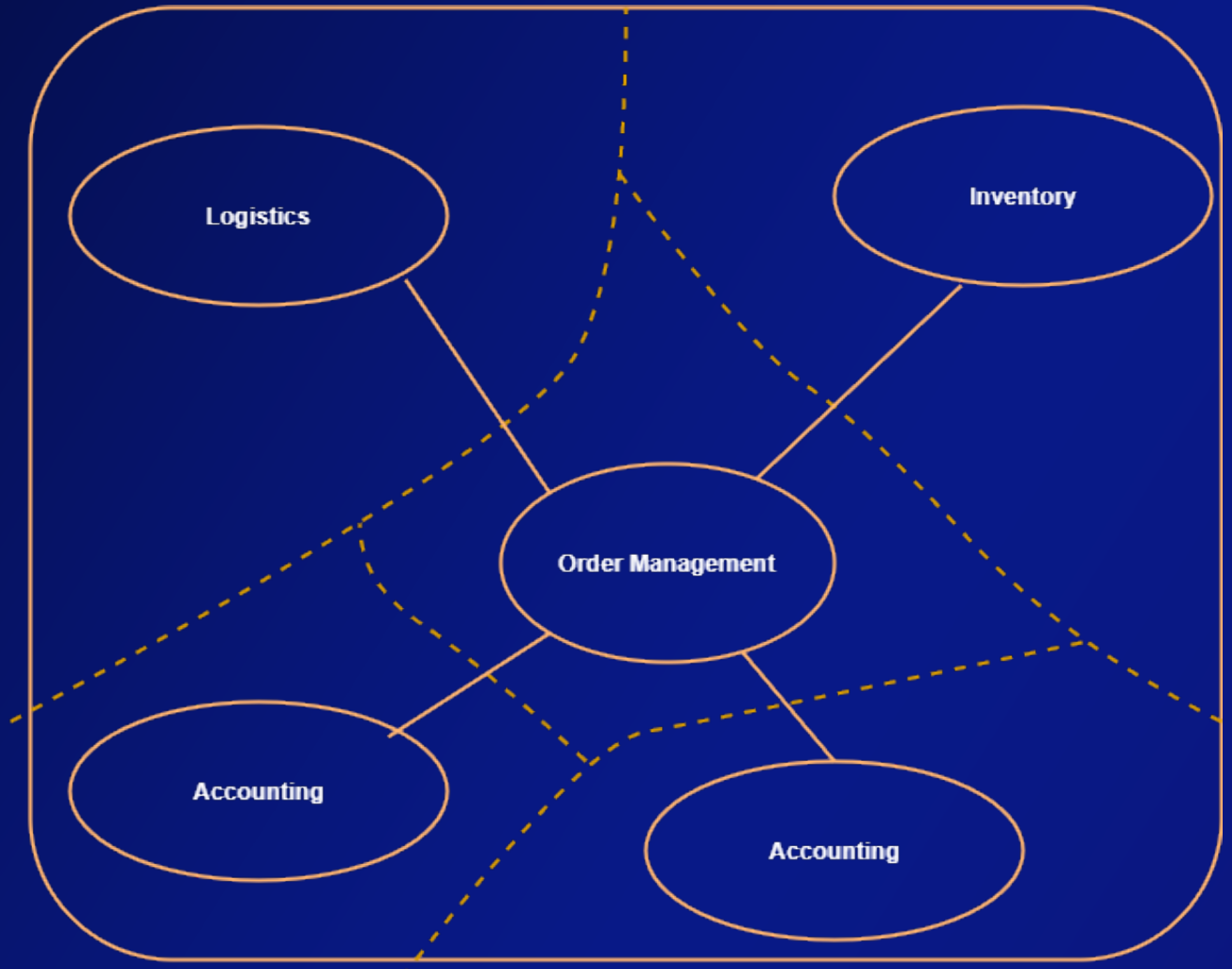
Isolate Team Code

Favor Native Browser Features over Custom APIs

Micro-Frontend tenets

- Follows domain driven design
- Decentralized architecture with enhanced reusability
- Smaller, more cohesive and maintainable codebases
- More scalable , agile organizations with decoupled, autonomous teams
- Simple, decoupled codebases & failure isolation
- Faster page load due to a smaller bundle
- Less risky implementation of major changes
- Faster feature roll-out due to independent deployments
- Support multiple frameworks
- Higher app stability due to loose coupling

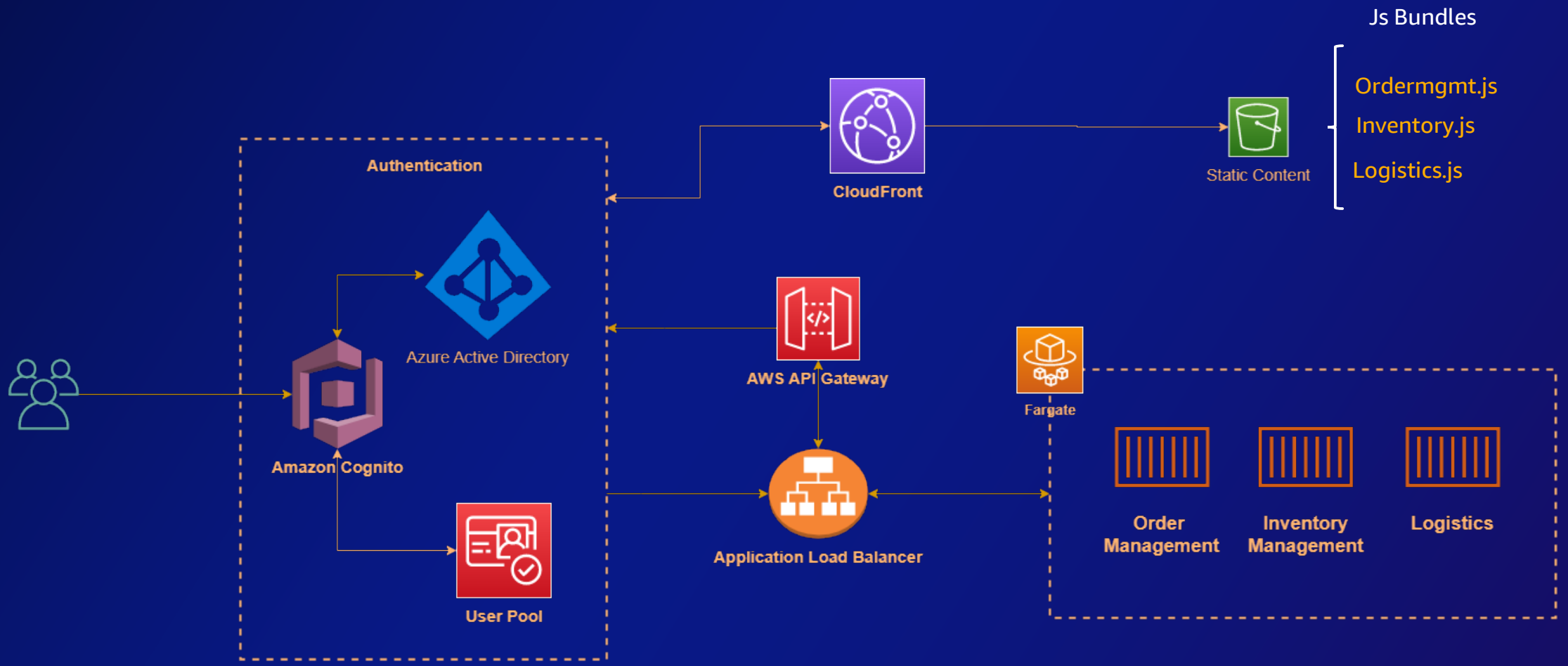
Domain driven design



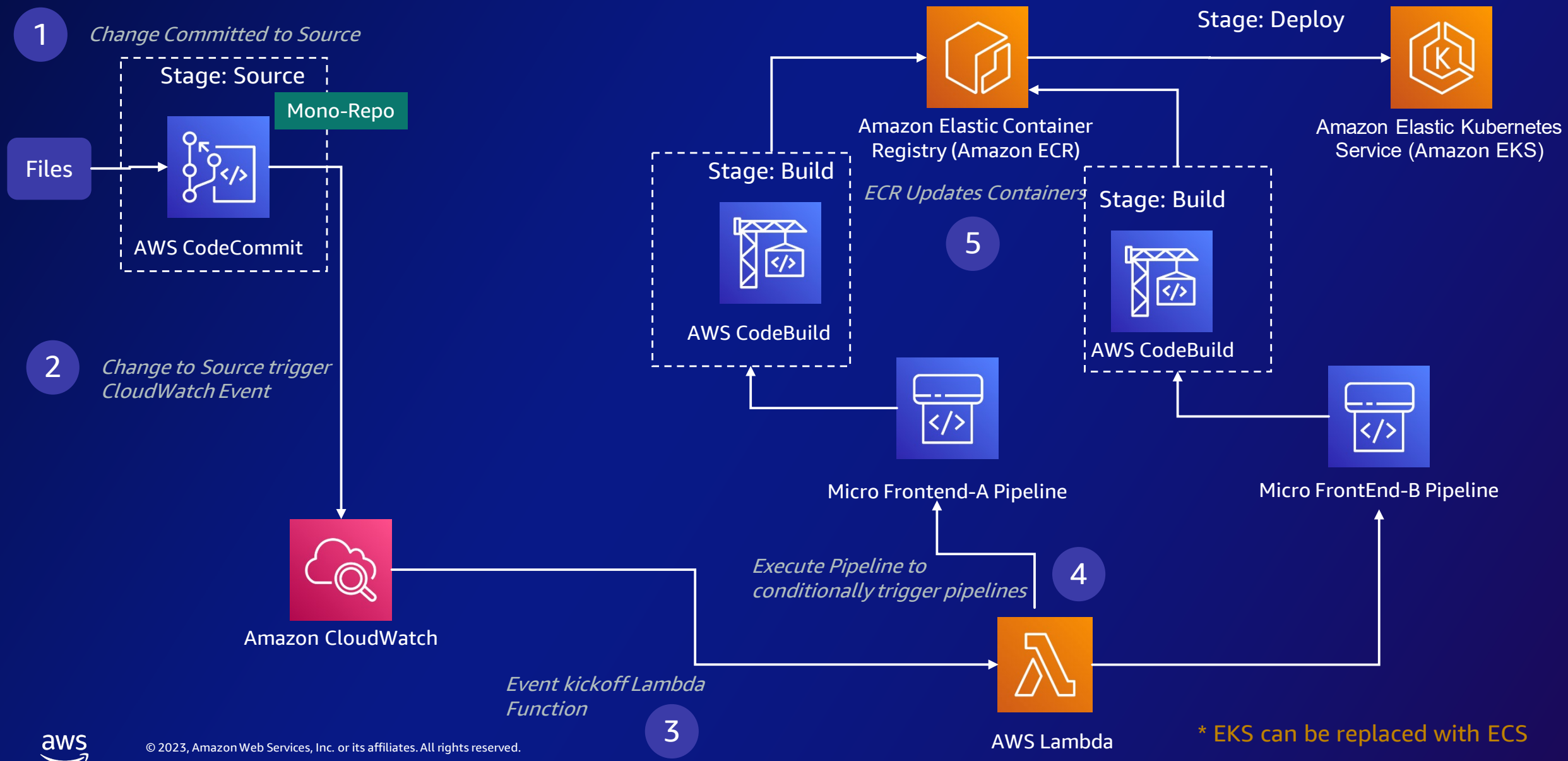
Sample - Bounded Context of an ecommerce application

Apply ubiquitous language boundaries used in the business context to identify different domains and subdomains of the business. Domains are top down view of the business.

Illustrative architecture



Continuous integration and Deployment



Architecture patterns

Micro-Frontend patterns

1 Backend for Front-End

Building scalable backend services specific to interfaces

2 Web Components

Reusable components/modules with context based data sharing

3 Real Time via iFrame

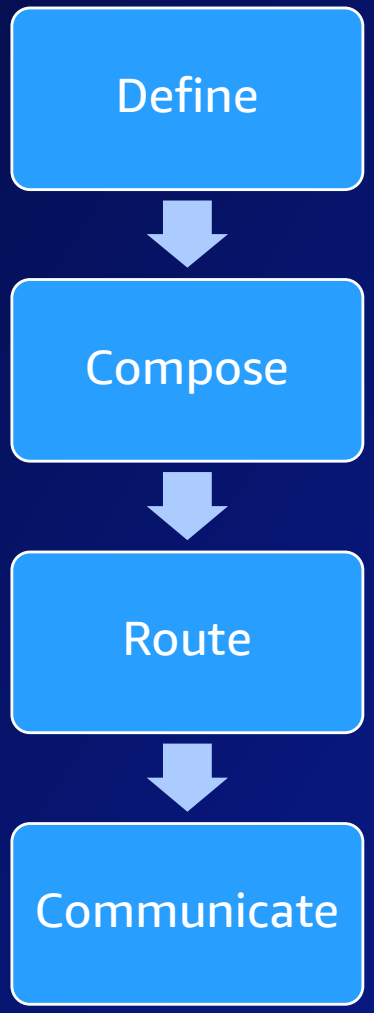
Event bus for coordinating events across iFrames

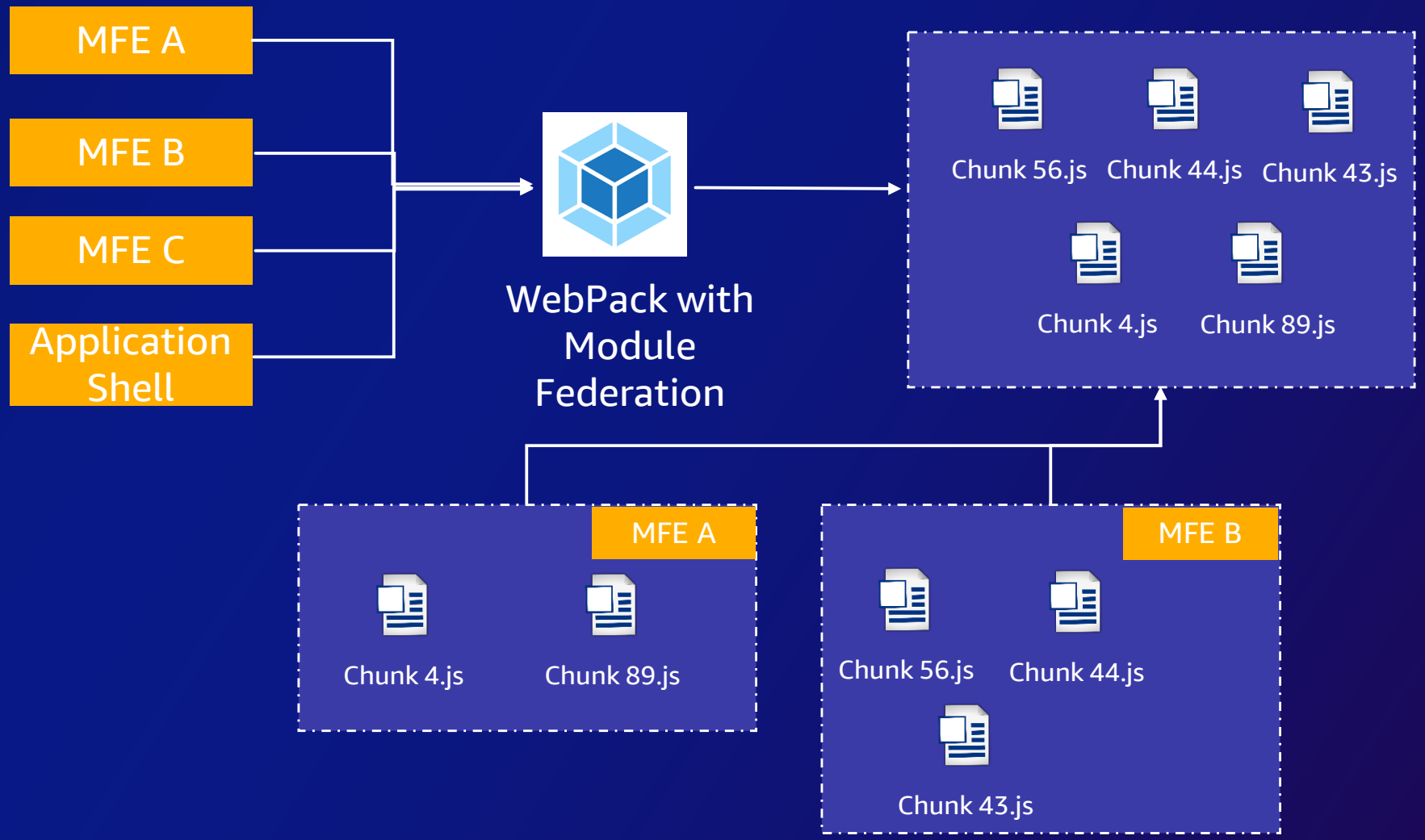
4 Server Side Rendering

Dynamically generate templates to render in browser using hydration process

Implementation techniques

Key design considerations





Bootstrap

Server Side / Edge Orchestrator

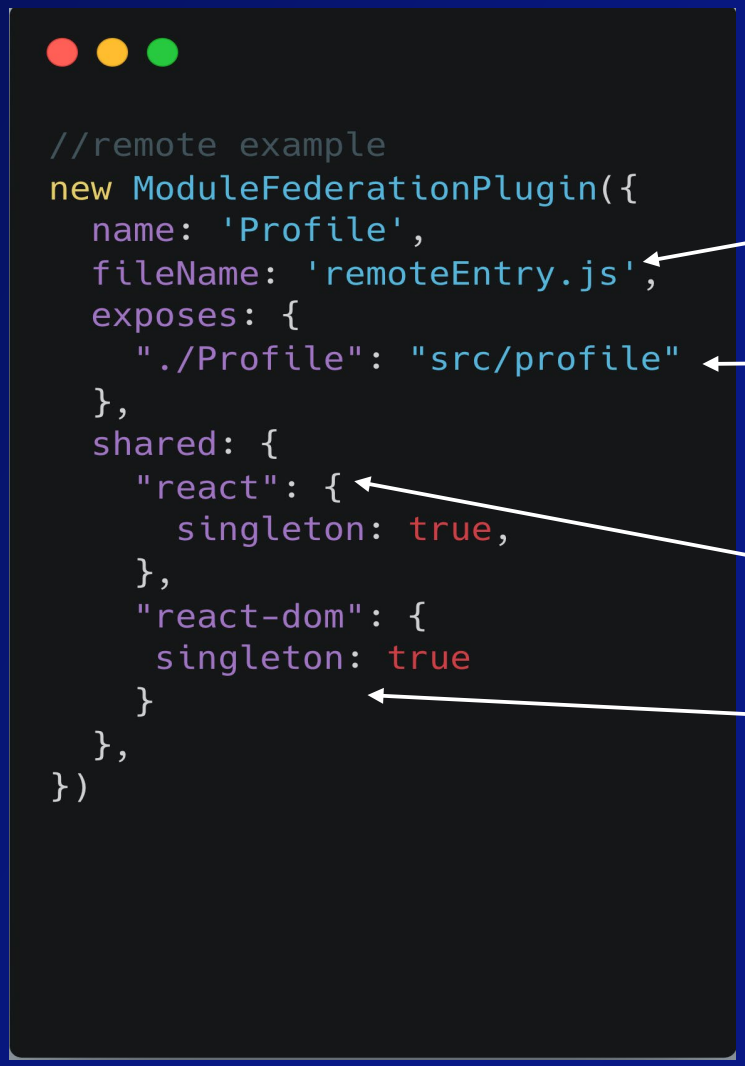
- Edge Solution over CDN
- Server side loading/unloading
- Bursty Traffic Scaling
- Multiple subdomains
- App Communication Challenges
- Application Initialization

Bootstrap doesn't have the entire URLs map of our applications, instead, it loads in memory a map of which micro-frontend should be loaded based on the user status and the URL requested via user's interactions or deep link.

Client Side Orchestrator

- Application Initialization
- Configuration Sharing Across Apps
- More Control and Stable
- Load/Unload Apps on User State
- Explicit Routing
- Platform Abstract

Module Federation configuration



```
//remote example
new ModuleFederationPlugin({
  name: 'Profile',
  fileName: 'remoteEntry.js',
  exposes: {
    "./Profile": "src/profile"
  },
  shared: {
    "react": {
      singleton: true,
    },
    "react-dom": {
     singleton: true
    }
  },
})
```

File exported by Webpack MF

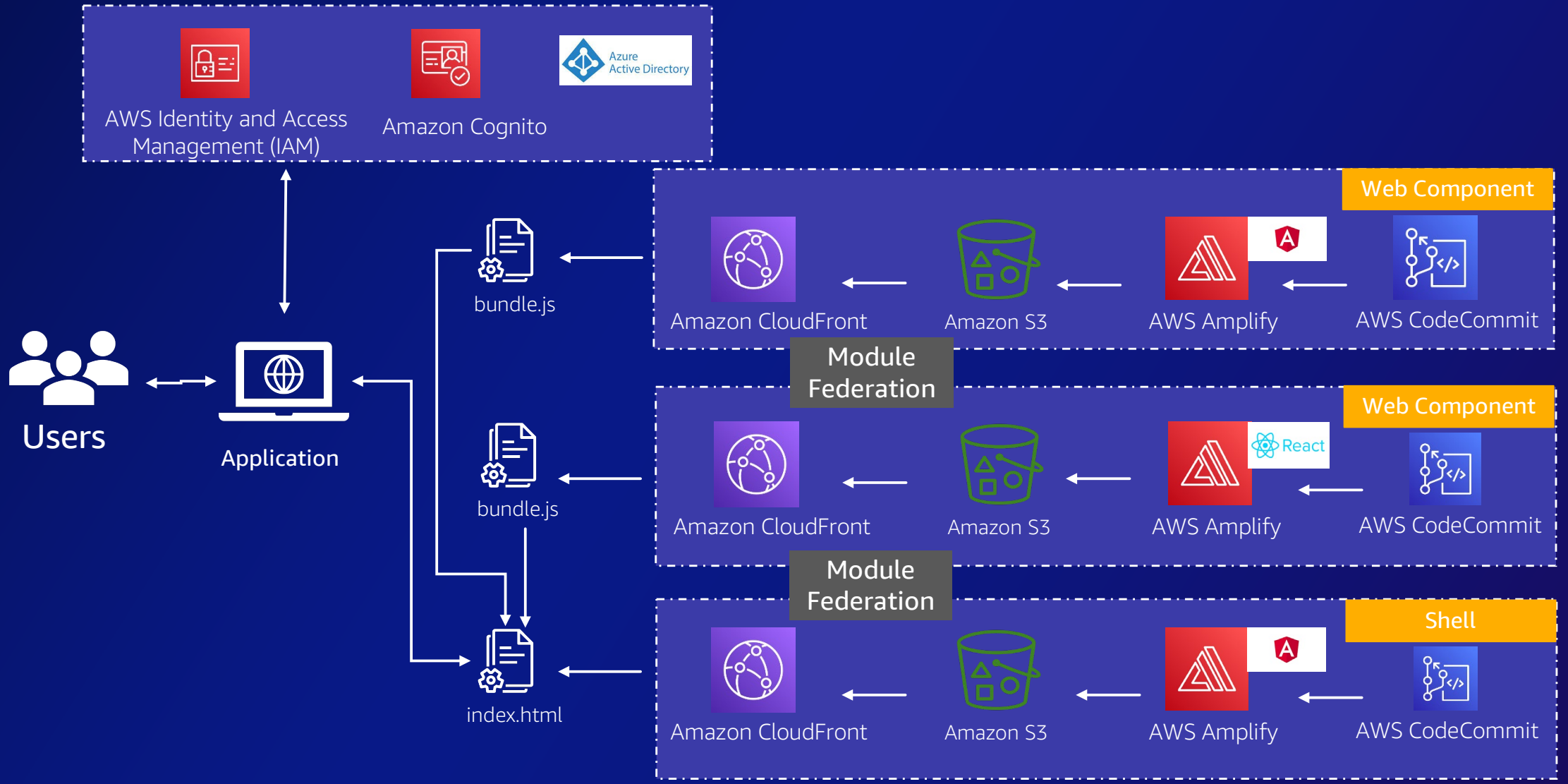
Module(s) shared with host

Shared Dependencies

Only one dependency loaded

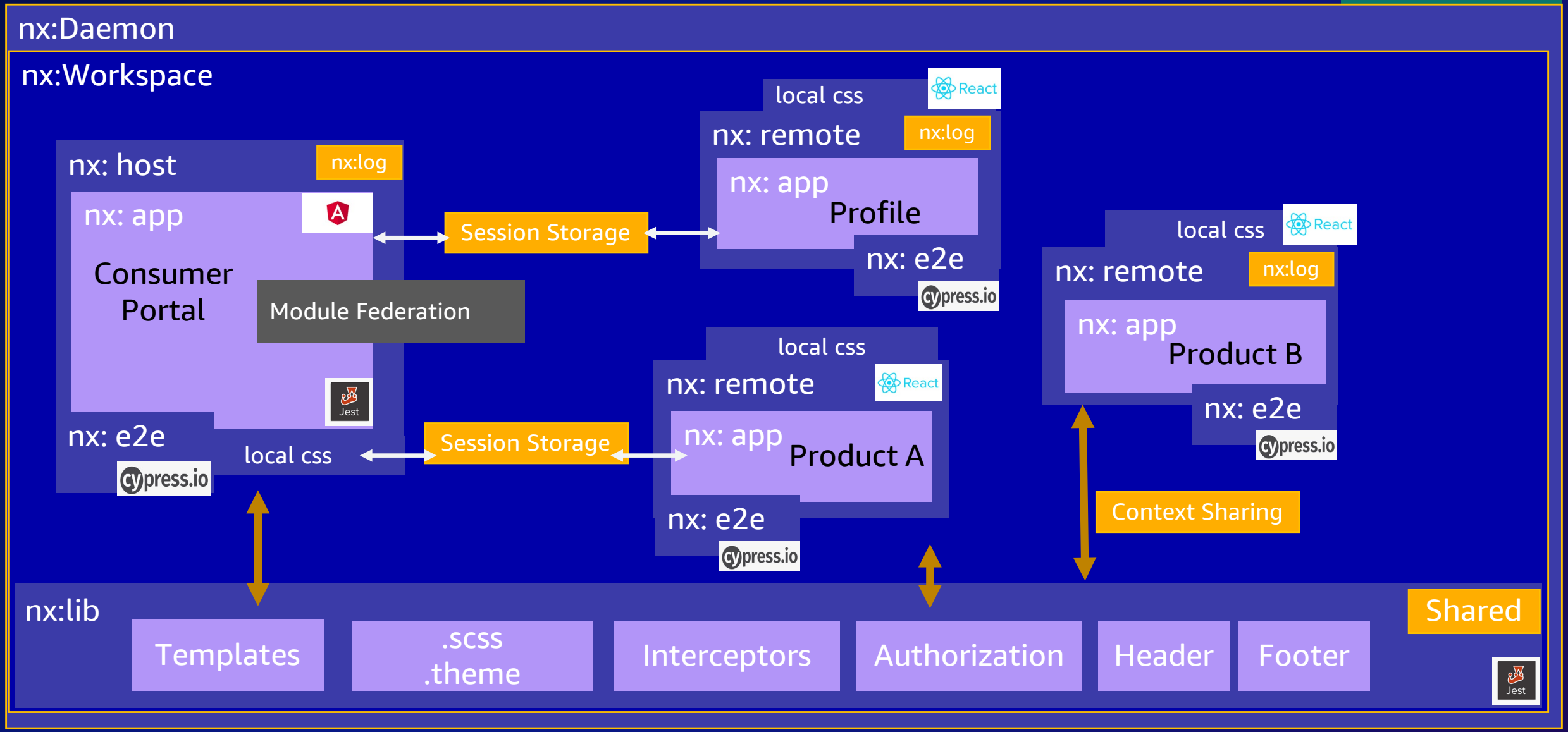
Module Federation is not a Micro Frontend

Amplify based implementation



NRWL based implementation

Mono-Repo



Code snippets

```tsx
// import { StrictMode } from 'react';
import * as ReactDOM from 'react-dom/client';
import { BrowserRouter } from 'react-router-
dom';
import App from './app/app';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(

    <BrowserRouter>
      <App />
    </BrowserRouter>

);
```

Bootstrap.tsx

```tsx
// @ts-check

const { withModuleFederation } = require('@nrwl/react/module-federation');
const baseConfig = require('./module-federation.config');

/**
 * @type {import('@nrwl/react/module-federation').ModuleFederationConfig}
 **/
const prodConfig = {
  ... baseConfig,
  /*
   * Remote overrides for production.
   * Each entry is a pair of an unique name and the URL where it is deployed.
   *
   * e.g.
   * remotes: [
   *   ['app1', '//app1.example.com'],
   *   ['app2', '//app2.example.com'],
   * ]
   *
   * You can also use a full path to the remoteEntry.js file if desired.
   *
   * remotes: [
   *   ['app1', '//example.com/path/to/app1/remoteEntry.js'],
   *   ['app2', '//example.com/path/to/app2/remoteEntry.js'],
   * ]
   */
  remotes: [
    ['app1', process.env['NX_BASE_PATH'] + '/app1/remoteEntry.js'],
    ['app2', process.env['NX_BASE_PATH'] + '/app2/remoteEntry.js'],
    ['profile', process.env['NX_BASE_PATH'] + '/profile/remoteEntry.js'],
    ['master-data', process.env['NX_BASE_PATH'] + '/master-data/remoteEntry.js'],
    ['configurator', process.env['NX_BASE_PATH'] +
'/configurator/remoteEntry.js'],
};

module.exports = withModuleFederation(prodConfig);
```

webpack.config.js

```tsx
const UserManagement = React.lazy(
  async () => await import('user-management/Module')
);
const Profile = React.lazy(async () => await import('profile/Module'));
const Reports = React.lazy(async () => await import('reports/Module'));
const Login = React.lazy(async () => await import('../app/login/login'));
const Middleware = React.lazy(
  async () => await import('../app/middleware/middleware')
);
const Dashboard = React.lazy(async () => await
import('./dashboard/dashboard'));
export const App: React.FC = () => {
  const url1 = window.location.href;
  const url2 = url1.replace('#', '?');
  const url3 = new URL(url2);
  const params = new URLSearchParams(url3.search);
  const navigate = useNavigate();
  React.useEffect(() => {
    if (params.get('id_token')) {
      TokenService.setAccessToken(params.get('id_token'));
      navigate('/middleware');
    }
  }, []);
```

App.tsx

Anti patterns

Anti patterns

1. Too many components?
2. Multiple frameworks
3. Bi-directional data flow
4. Tight coupling

Architecture is always a trade-off, just find a balanced approach for your context

skillbuilder.aws

Your time is now

Build in-demand cloud skills your way

Thank you!

Please complete the session survey

Ajay Nagar
Sr. Cloud Application Architect
AWS India

Jayesh Shinde
Cloud Application Architect
AWS India