International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I6P103

Volume 72 Issue 6, 15-24, June 2024

Original Article

Applications of Micro-Frontend Application Development in a Customer Support CRM

Tanmaya Gaur

Principal Architect, Customer Support, T-Mobile US, Washington, USA.

Corresponding Author : tanmay.gaur@gmail.com

Received: 16 April 2024 Revised: 22 May 2024 Accepted: 03 June 2024 Published: 15 June 2024

Abstract *- Micro-frontends extend the concept of micro-services to the world of UI. The idea behind Micro Frontends is to develop applications as a composition of features which are owned and developed completely isolated and by independent teams. These experiences are strung together either at runtime or build-time to deliver a single cohesive application experience to the end user. Customer Relationship Management (CRM) is a system that helps businesses manage their interactions with current and potential customers. CRM systems can provide various functions, such as customer service, sales automation, contact management and more. CRM systems are essential for businesses that want to improve their customer satisfaction, retention, and loyalty, as well as increase their sales and revenue. A traditional CRM system is traditionally monolithic, where the system is built as a single unit that shares the same codebase, database, and user interface. Developing and maintaining a CRM system as a monolith can be challenging to scale, especially for large and complex businesses that have multiple teams, departments, and products. These teams often fall back to breaking down the CRM into multiple isolated applications to deal with the maintainability and operability challenges. This whitepaper explores Micro-frontend Architecture in delivering CRM Applications. This allows operational flexibility but helps standardize the experience for users and the tech stack for the enterprise. The paper will attempt to provide an overview of considerations, outline key features, address the challenges in development and illustrate how composable designs can help application teams tackle these obstacles.*

Keywords *- Customer relationship management, CRM, Telecom, Web Development, Micro-frontend.*

1. Introduction

This paper will propose developing enterprise Apps, specifically CRMs, as a Micro-frontend instead of traditional web development. For large Enterprises, CRM Can quickly become huge and unwieldy. Taking the example of a telco's CRM, it supports everything from sales scenarios like device upgrades and adding new lines to service scenarios like rate plan change, device, network troubleshooting, etc.

Moreover, as the telco ends up supporting multiple product offerings and lines of business, the required functions of the application start to bloat. Many enterprises end up either building multiple different applications to support their needs. This comes at the cost of having to maintain multiple applications and often tech stacks while also requiring customer agents using these apps must deal with constant app swivels and experience inconsistencies.

Building an enterprise's CRM as a traditional monolith leads to significant challenges, such as

- Difficulty in scaling, testing, and deploying the system, as any change in one part of the system may affect the whole system.
- Lack of flexibility and customization, as different teams and products may have different needs and preferences for their CRM functions.
- Increased complexity and technical debt as the system grows larger and more interdependent over time.
- Reduced performance and user experience, as the system may become slow, buggy, and outdated.

To overcome these challenges, the topic of this paper proposes developing the CRM application as a micro-frontend. Micro-frontend is a modular approach to building web applications, where each feature or function of the application is developed and deployed independently as a separate unit. Each micro-frontend has its own codebase and user interface and communicates with other micro-frontends through well-defined interfaces. This way, each micro-frontend can be developed, tested, and deployed independently without affecting the entire application.

At the highest level, some immediate benefits of using a micro-frontend style development for a CRM system spring to mind.

- Improved scalability, testability, and operability, as each micro-frontend can be scaled, tested, and deployed separately without affecting the rest of the application.
- Increased flexibility and customization, as different teams and products can make the pattern and design choices for their micro-frontends and tailor them to their specific needs and preferences.
- Reduced complexity and technical debt, as each micro-frontend has a clear boundary and responsibility and can be easily replaced or updated without affecting the rest of the system.
- Enhanced performance and user experience, as each micro-frontend can be optimized for its own functionality and deliver a fast, reliable, and consistent user interface.

This sounds great, right! But with great flexibility comes hidden costs and challenges, which teams often only encounter as they begin to scale. This paper aims to explore options for building a CRM as a collection of micro frontends integrated into a unified customer support dashboard. While the paper focuses on a Telco CRM, the discussion can be easily applied by any large application team exploring the micro-frontend architecture.

This paper will discuss the implementation options, organizational impacts, and challenges of using a micro-frontend style development for a customer support CRM system. At the same time, there are some online documents on patterns to implement micro-frontends and hello world examples. There is minimal to no documentation which details the architecture, strategies, considerations, and best practices around this implementation paradigm. That is the research gap this paper will try to address.

2. Implementation Options

There are multiple options available to implement micro-frontend style Apps. This section lists out some commonly used patterns, each with its own advantages and disadvantages. It is important to understand the use case and determine the best option for specific requirements and needs. While this section only focuses on the web-components strategy, there are good documents scattered on the internet which detail the other options and their implications.

In most cases, it may make sense to optimize by developing global Concerns with the container app such that they are always available to all other experiences. Each of the approaches below utilizes a container or router application to launch or navigate other micro-frontend experiences.

2.1 Server-Rendered

Server-rendered web architecture (see Figure 1), which has been around almost as long as web development and is a suitable example of a micro-frontend, based on development and isolation strategy. Every reload asks the backend server to return a specific resource specified in the URL or request. These resources and dependencies can be independently developed, and the base html for the experience is returned on the first request. Once the html is returned and loaded, it requests relevant dependencies.

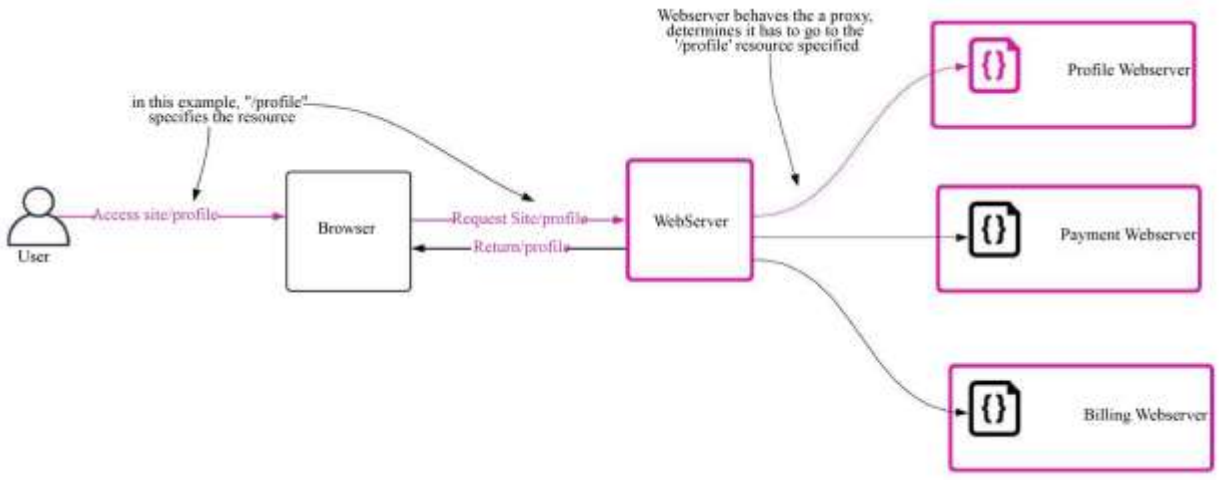

Fig. 1 Pattern for server rendered web architecture

2.2. Iframes

Going with the theme of patterns that have existed for a long time, let us discuss using Iframes (see Figure 2) to build a web application; iframe is an html element that loads another html page within the document. The page launching the iframe is the parent and the embedded html page launched is referred to as the child. The two-run is completely standalone. All communication relies on post messages for parent-to-child communication.

Is it perfect? Of course not. But this does fit the bill of allowing multiple micro frontends to co-exist at runtime.

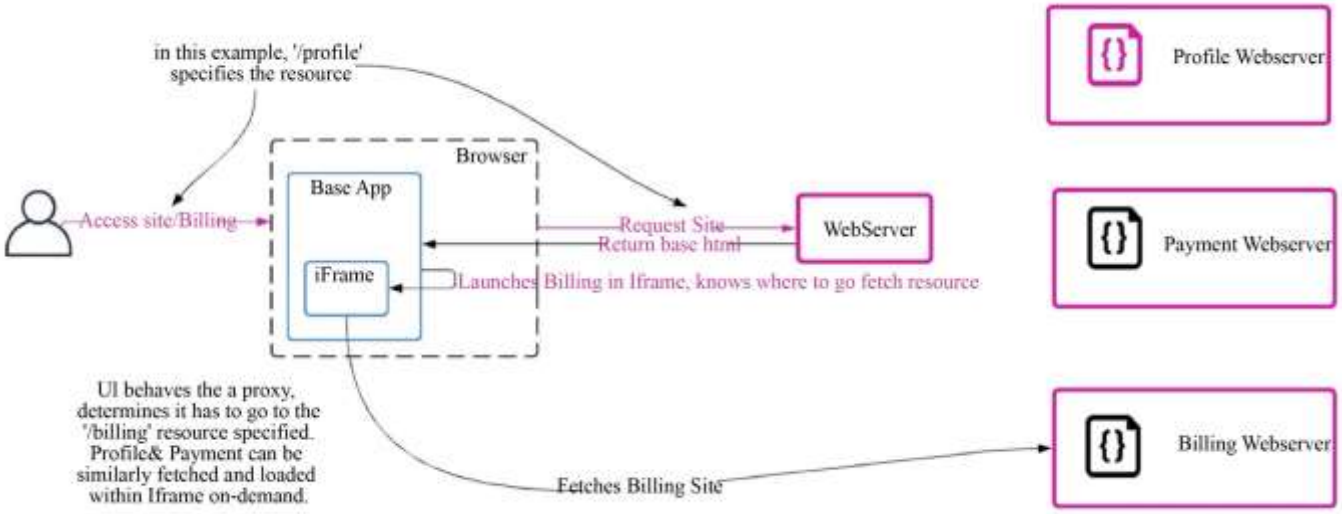

Fig. 2 Pattern for iframe rendered web architecture

2.3. Build Time Scaffolding

The approach (see Figure 3) utilizes a build time instead of run time compilation and thus loses some of the flexibility that was discussed earlier on. The approach is to code each micro-frontend as an independent library which gets compiled together at build time. The flexibility of being able to deploy without impacting the rest of the app is lost, as the whole app needs to be recompiled prior to each code drop.

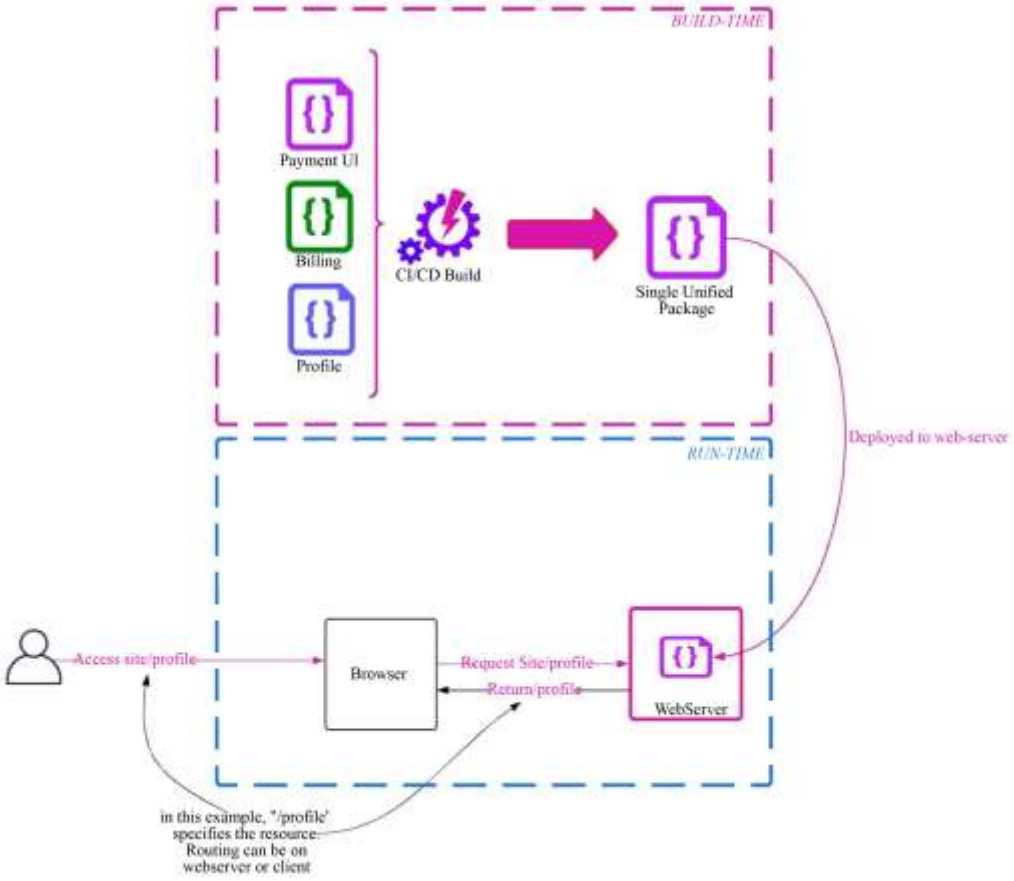

Fig. 3 Build-time scaffolding to develop modularized web applications

2.4. Web-Components

There has long existed a JavaScript approach to building a UI experience on the fly by appending HTML to the experience. Essentially, a JavaScript method executed based on certain triggers can, at runtime, create an HTML and then append or remove the into the Document object Model.

An extension of this approach was used to build a micro-frontend concept using the web component standard (see Figure 4). Web Components is a suite of different technologies allowing you to create reusable custom elements, among other things like shadow DOM, templates, and html imports. The custom elements specifically provide a way for developers to build their own fully-featured experiences, which can be exposed as standalone DOM elements. The createCustomElement() function defined in the spec provides ways to register these standalone components with the browser.

Custom elements bootstrap themselves when launched - they start automatically when they are added to the DOM and are automatically destroyed when removed from the DOM. Once a custom element is added to the DOM for any page, it looks and behaves like any other HTML element and does not require any special knowledge of Angular terms or usage conventions.

Web development frameworks like Angular are also on board with this emerging trend. Angular, for example, supports Angular elements, which are Angular components packaged as custom elements. Custom elements or extensions like angular elements give you the base framework to code micro-frontend style apps, which get strung together at runtime.

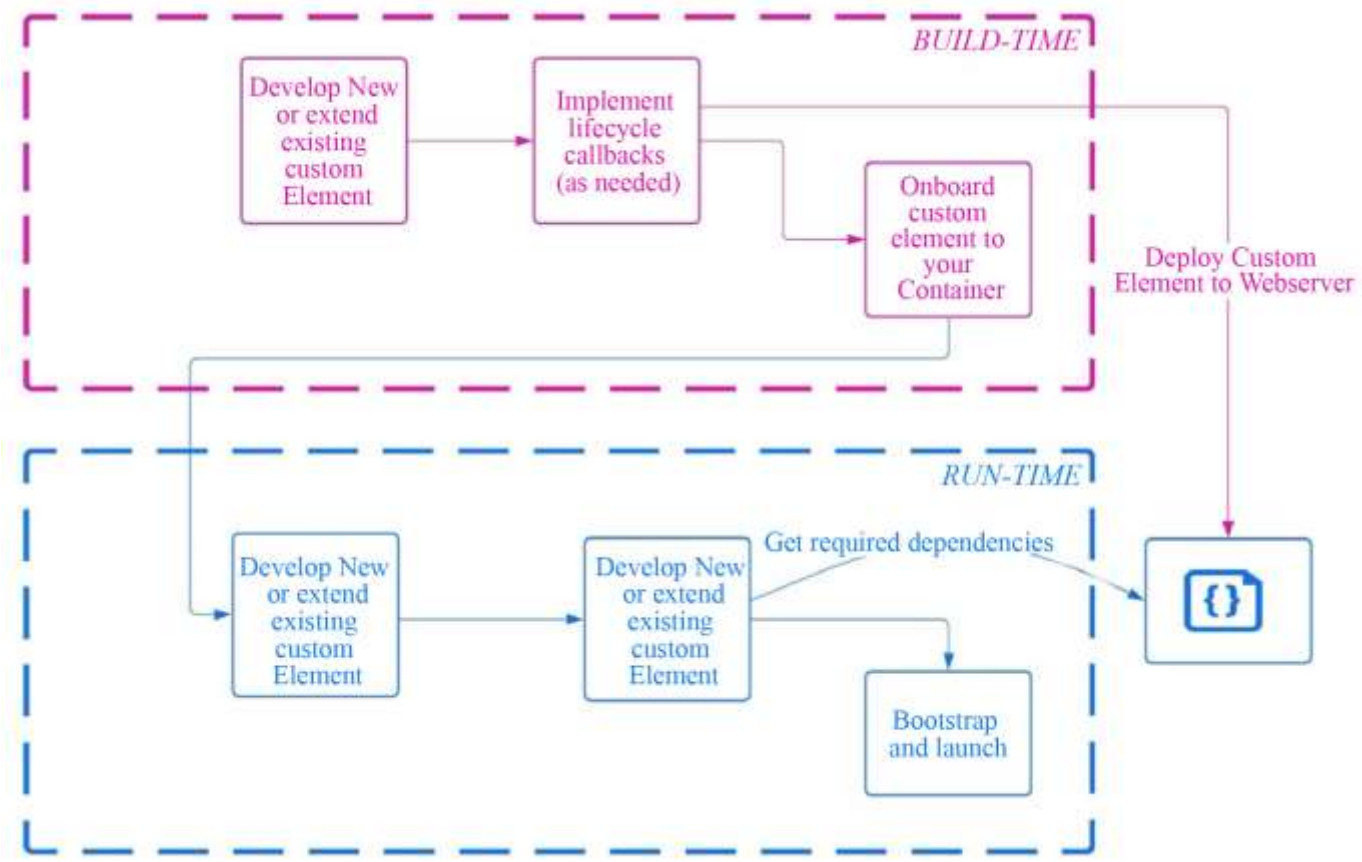

Fig. 4 Web component composing a micro-frontend at run-time

3. Development Team Structure and Evolution

Traditional development has evolved quite significantly since web development began (see Figure 5)

- The initial development used in the 1990s was monolithic architecture, where a single (One-Tier)system consolidates all components, spanning the user interface to data storage, within a single executable or process.
- Development evolved in the 2000s, and Two-Tier architecture started becoming more popular. The two-tier pattern bifurcates an application into a client and a server. The client manages the user interface, while the server handles backend functions.

- By 2010, development had shifted towards three-tier and later n-tier by 2010. The three-tier architecture is that where each tier runs on its own infrastructure, each tier can be developed simultaneously by a separate development team. It allows for one tier to be updated or scaled as needed without impacting all the other tiers.
- The N-Tier architecture builds upon the principles of the Three-Tier model, allowing for an arbitrary number of specialized tiers. Each tier focuses on specific functionalities, fostering a more modular and distributed approach to developing applications.
- The latest trend is where the n-tier applications use modern technologies like containers and micro-services and are often built in cloud-native technologies. The microservices architecture allows a tier to be broken down into small, independent services, each focusing on a specific business capability.

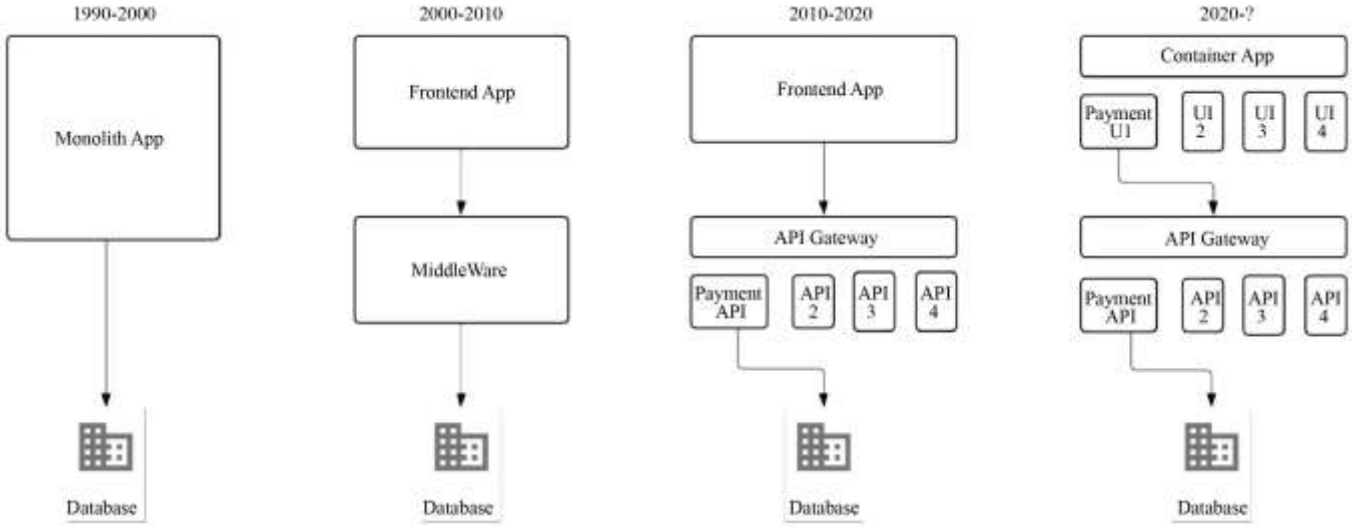

Fig. 5 Evolution of web development architecture

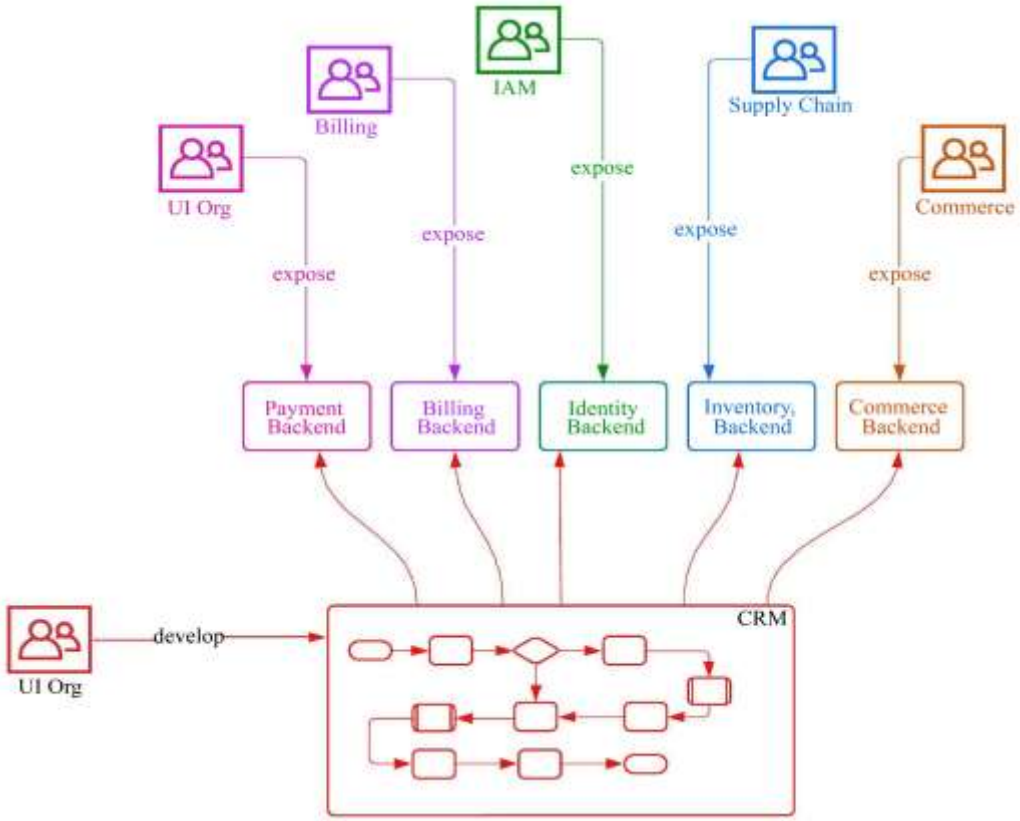

Fig. 6 Early days of web development

While the micro-services architecture allowed more modern modular solutions to be applied to the backend Tiers, the UI Applications continued to stay monolith. While not as popular as micro-services, micro-frontend technologies are now evolving to help bring the advantages of micro-services to UI Tier as well.

Let's dive deeper into what an organization would have looked like prior to the advent of microservices becoming standardized. Let us take the example of an enterprise with large organizations managing domains like Payment, Billing, Identity Management, Supply Chain, and commerce. In the initial days of web development (see Figure 6), Each of these organizations would have their independent teams exposing their data through legacy technologies, which a large monolith UI organization Team would consume directly or via legacy middleware.

As enterprises matured, so did the API development practices. Each backend organization started owning Micro-services exposed through standardized API tiers and gateways (see Figure 7). This was cleaner and maintainable and gave organizations better control over the data they expose.

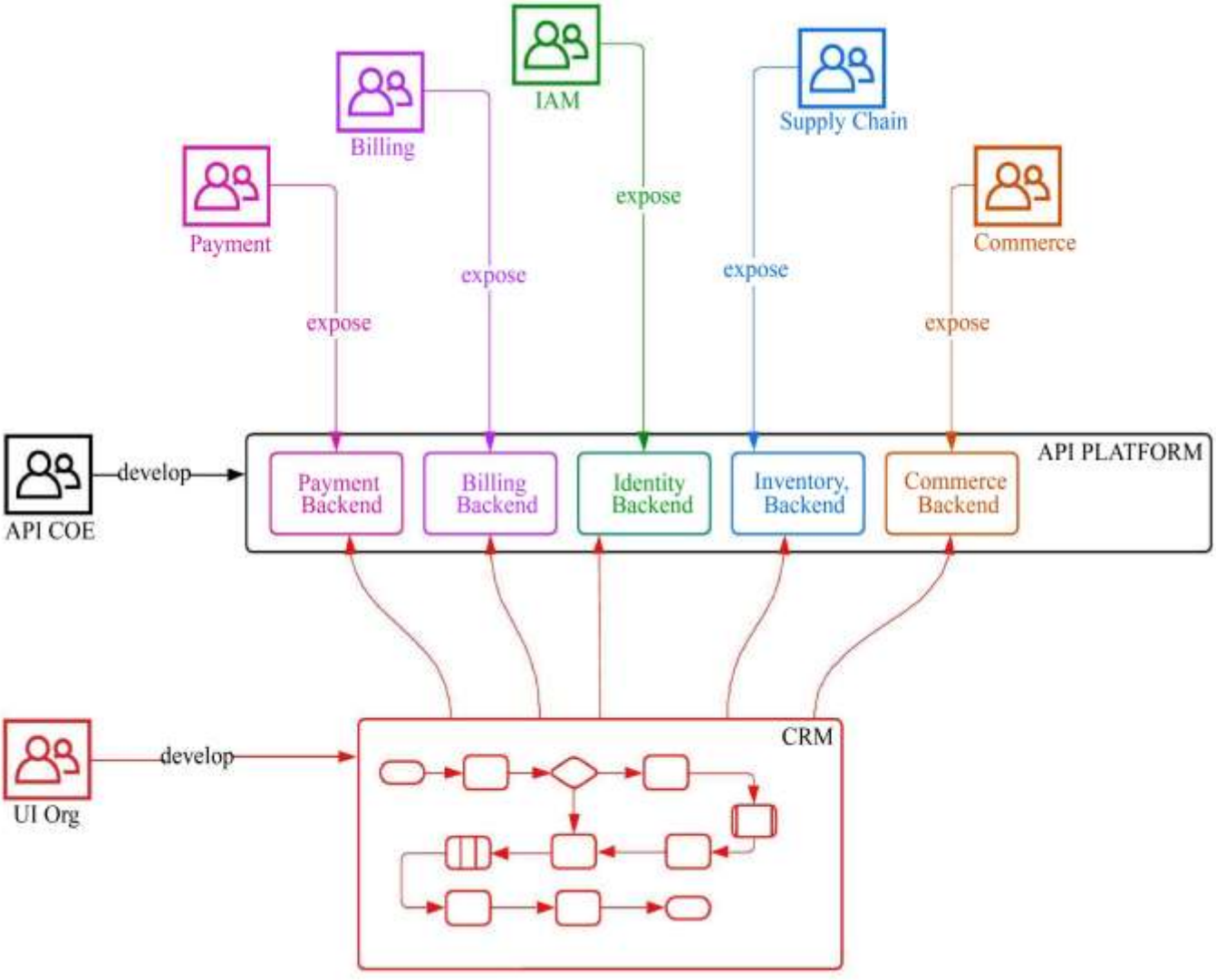

Fig. 7 Micro-frontends and API platforms positively impact web development

So, what does micro-frontend bring to the table? It allows the UI to be developed as decoupled codebases with independent deployability. This reduces the scope of any given deployment, which in turn reduces the scope of testing and the risk of outages across the broader application. Given this, it allows organizations the ability to have independent, autonomous teams (see Figures 8 and 9).

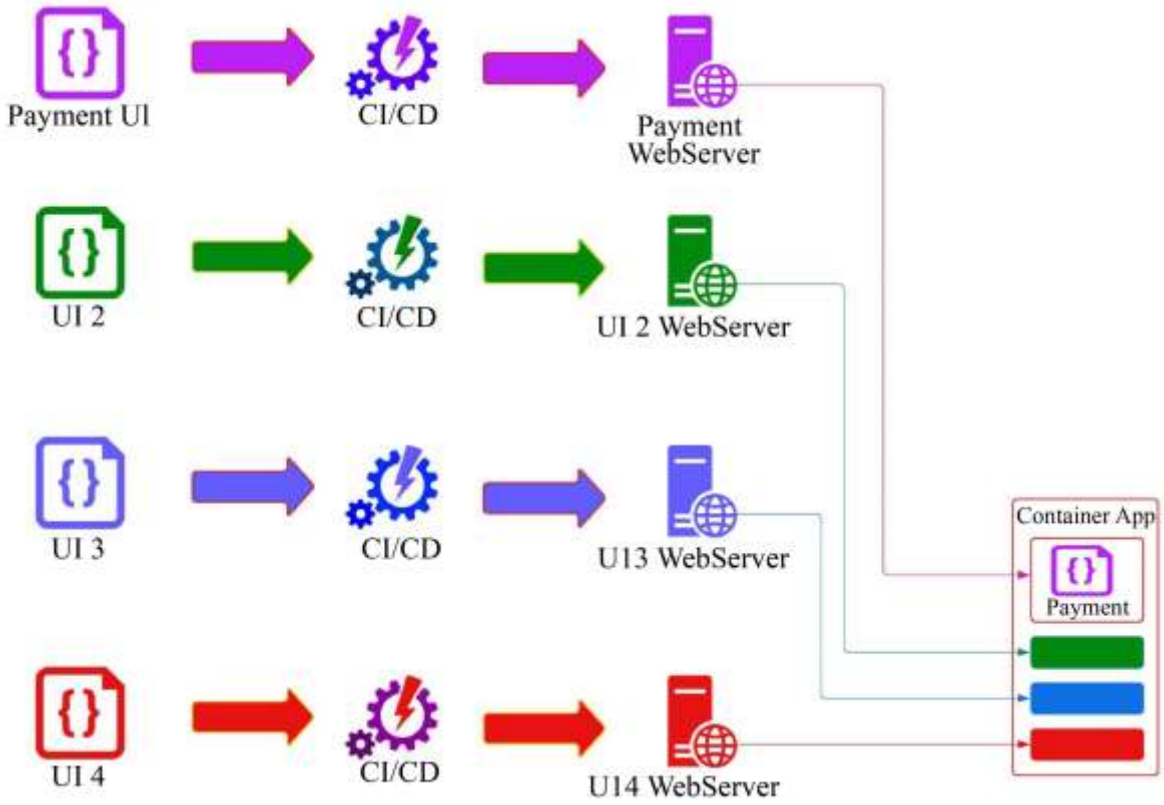

Fig. 8 Micro-frontends follow independent lifecycle from development to production

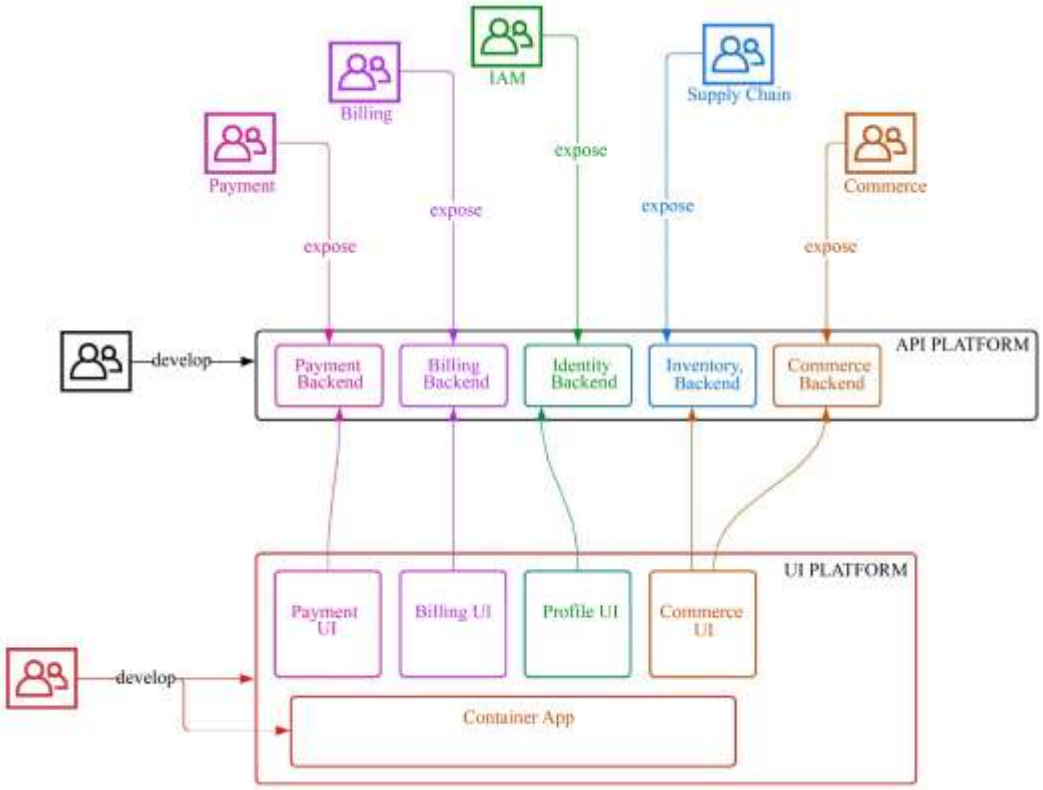

Fig. 9 Micro-frontend development architecture

So, does micro-frontend really change how an organization structures its UI teams? Not necessarily. The UI can still continue to be owned by a single UI organization, and that still works. However, the real power of micro-frontends lies in the fact that you can now have the actual business organizations own their UI functions as well. As an example, Payment teams can own the payment UI. IAM teams can own AuthN, AuthZ and profile-related UI functions.

Just like any enterprise looking to standardize its software tooling practices across a Tier, good governance and automation are key to managing the micro-frontend. The next section will go over challenges with micro-frontend during initial development and at scale.

4. Micro-frontend Implementation Challenges

This section will focus specifically on Challenges with web component-style micro-frontends. These challenges apply to other patterns as well, with some variations. This paper will not dive deeper into these challenges but that may be a topic for future research and publications.

4.1. Duplication

Duplication comes from two aspects.

- Each independently executable micro-frontend references its own dependencies to allow it to launch standalone. This can often mean that the application has multiple micro-frontends referencing the same libraries independently. The fact that there may be different versions of the same library being referenced makes things worse and often causes conflicts.
  An example would be if multiple micro-frontends were using angular elements. Each of them references angular independently. Given the large footprint of the angular base framework, this duplication can result in significant additional downloads, resource utilization and performance impacts.
  There are certain modern enhancements like module federation architecture which help us optimize for this issue.
- Without proper governance, teams may duplicate certain experience or non-functional functions. At scale, awareness and discoverability can start becoming a big challenge. Good documentation and architectural and product governance over the platform, with an observable process for new development, is crucial to avoid this pitfall.

4.2. Performance

Directly linked to the duplication challenge is the impact on performance. Most user agents have fixed CPU, Memory, or RAM limits. Monoliths often have a base module which launches reusable functions utilized by every other module. Micro-frontends often have multiple instances of the base framework running in parallel.

Optimizing dependencies and developing common functions in base containers or designing them to reside in microapps are techniques to ease the dependency on resources, thereby improving performance.

If a web development implementation utilizes a bulky base framework like angular, dynamic module federation techniques are crucial to scale. Module federation allows an app to share third-party packages across remotes. This means that when you load a federated module from a remote, Webpack does not need to re-download a copy of these packages. Is it worth the effort? It is likely one of the most exciting new features in Webpack and is a game-changer in JavaScript architecture.

4.3. CSS Leaks

This issue stems from one micro-frontend's CSS impacting others due to incorrectly scoped CSS. In regular monolith UIs, this kind of issue would be easy to identify during testing and easily fixable. What makes matters worse is that for a micro-frontend at scale, this may cause unintentional impacts only on a certain permutation of the test scenario. A certain microapp launched after another microapp. This, in addition to the fact that micro-frontend promotes incremental code drops without wide regression, such issues often only get identified in production.

Do not mistake a CSS leak as a non-functional low severity experience issue. There can be situations where the CSS leak causes a particular functional CTA to completely disappear from the viewable area of the DOM, creating an outage for parts of the website.

To remediate this issue, good CSS governance with scopes per micro-frontend or utilizing enhanced web standards like Shadow DOM allows us to mitigate this issue successfully. Ideally, these are built and advised as foundational aspects of the micro-frontend strategy.

4.4. Clean Separation of Concerns (between Micro-frontends)

While seeming independent, Modules across complex CRM apps need to talk. There are aspects of a session state that need to be shared in real time to provide for these needs. An example in a telco use-case would be if you log into a customer account and cancel a line; any relevant cache any other micro-frontend has saved needs to be wiped clean.

Depending on how granular a development team builds a micro-frontend, they may also need to share some common state, e.g. if payments are a micro-frontend used both by upsell and add remove services flows, the flows may need to supply some context to the payment micro-frontend on launch so it can launch appropriate experience.

One way to achieve this would be to keep no state cache on the client and make everything an API call to a backend. This, of course, has larger cost, complexity, and performance implications.

The best solution is to provide a common layer for the messaging and state sharing as part of the base framework. Its best to abstract the microapps from each other and have them share state through well-defined methods that serve as the abstraction.

4.5. Overall Governance

Isn't governance a crucial aspect of any enterprise initiative developing a UI application? That is very true, and this paper is not discounting that. What is being called out is that this becomes especially harder when you develop a UI as a micro-frontend. Multiple teams who are working on independent codebases find it tough to identify already available features, functions, and utilities that are ripe for reuse. Lack of discoverability of such reuse opportunities can lead to duplication and performance issues as well as cause the development estimates to be higher. Also, the operational costs of these duplicate implementations are no less significant and will be the next topic.

It is key to foster governance early in the architecture and design phase. It is ideal for this governance to be a dedicated team. It is also ideal that a lot of this governance be automated, as that makes life easy for this dedicated team as the initiative scales. Something beneficial is to have a boilerplate implementation of the micro-frontend that all development teams can enforce to use. The same with CI/CD. Having control over the Build and deploy pipeline also allowed us to apply automated controls across the lifecycle of a micro-frontend's SDLC.

4.6. Runtime Conflicts

An earlier topic touched on the duplication of dependencies and the fact that these dependencies may not always be the same version. This can manifest in multiple ways. As an example, the two versions of a library the application depends on May not be backwards compatible. These issues are tricky to identify and debug, just like CSS issues, as they again need a specific test scenario to execute to reproduce. Additionally, at times, the only possible fix may require one of the microapps to change the dependency to align. This can cause team conflicts and needs some negotiation. Often, this version change may not be a straightforward fix.

4.7. Integration Testing and Environments

Like micro-services, micro-frontends provide us with the benefits of developing and deploying the application as smaller units of code, which, in theory, negates the need for app-wide regression. The difference, however, is the fact that unlike micro-services, which run on discrete independent servers and are isolated at runtime, micro-frontends are all returned to a single browser and run on a single executable environment. As such, there are integration issues and conflicts like the CSS Java-script related challenges which were discussed earlier. Given the higher payload size and code duplication, there are also performance testing considerations for larger apps. This integration testing needs and having to maintain integration test environments is similar to traditional web development. This, however, takes away the independence of the team's micro-frontend promises unless there is a high level of automation in testing and ci/cd and democratization between teams or to a dedicated test authority.

4.8. Long-term Operational Costs

As touched on in the last topic, the duplication associated with micro-frontends often comes with a higher cost of development. Now let us consider an issue with an angular vulnerability, if a microapp did not implement module federation, such a security issue which needs a quick fix will need to be fixed across multiple projects quickly. There can be scenarios in which a single line of code in a traditional Application's base module ends up requiring teams to touch every single micro-frontend they deploy.

While not always remediable, this was one scenario which is mitigated if teams have a good base boilerplate codebase and a high amount of automation in integration, deployment, and testing.

5. Conclusion

To conclude, A micro-frontend does bring about a new era to web development architecture and capabilities. This also means that development organizations encounter very different complexities than are encountered in traditional application development. These complexities are not only development and tech stack but also impact an organization's structure. To make the best use of technology like micro-frontend, especially in enterprise use-cases like CRM solutions, it makes sense to understand these implications and challenges upfront and to solve them early.

The goal of this paper was not to dissuade the use of micro-frontend but to instead point out the challenges encountered in running it at scale. Although the development paradigm lets developers divide a large frontend application into smaller independent UI micro-frontends and code them independently, this does not force each team should do so completely in a silo. It is almost necessary to over-share to be successful. It is beneficial for teams to partner across solutions, create reusable components, and make discoverability a priority.

References

[1] Andrey Pavlenko et al., "Micro-Frontends: Application of Microservices to Web Front-Ends," *Journal of Internet Services and Information Security*, vol. 10, no. 2, pp. 49-66, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[2] Antonello Zanini, 5 Pitfalls of Using Micro Frontends and How to Avoid Them, The Sitepoint Website, 2022. [Online]. Available: https://www.sitepoint.com/micro-frontend-architecture-pitfalls

[3] The IBM Website, What is Three-Tier Architecture?. [Online]. Available: https://www.ibm.com/topics/three-tier-architecture

[4] The Mozilla Website, CustomElementRegistry: define() method, mdn web doc. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry/define

[5] The Angular Website. Angular Elements Overview. [Online]. Available: https://angular.io/guide/elements

[6] The MartinFowler Website, Micro Frontends, 2019. [Online]. Available: https://martinfowler.com/articles/micro-frontends.html

[7] Microfrontend Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Microfrontend

[8] Web Components Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Web_Components

[9] Single SPA, Concept: Microfrontends. [Online]. Available: https://single-spa.js.org/docs/microfrontends-concept/

[10] Github.com, Micro Frontend Resources. [Online]. Available: https://github.com/billyjov/microfrontend-resources

[11] Slideshare.net, Micro-Frontends. [Online]. Available: https://www.slideshare.net/SrikanthJallapuram/micro-frontends-78813796

[12] Emilija Stefanovska, and Vladimir Trajkovik, "Evaluating Micro Frontend Approaches for Code Reusability," *Communications in Computer and Information Science*, vol. 1740, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[13] Manfred Steyer, 6 Things You Should Know About MicroFrontends @ngCopenhagen, 2020. [Online]. Available: https://speakerdeck.com/manfredsteyer/6-things-you-should-know-about-microfrontends-at-ngcopenhagen-juni-2020

[14] Nilesh Savani, "The Future of Web Development: An In-depth Analysis of Micro-Frontend Approaches," *International Journal of Computer Trends and Technology*, vol. 71, no. 11, pp. 65-69, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[15] Stack Overflow, Micro Frontend Architecture Advice. [Online]. Available: https://stackoverflow.com/questions/47922293/micro-frontend-architecture-advice