Improved Micro frontends Anti-patterns Catalog				
Category	**Title**	**Description**	**Occurency in practice example**	**Solution**
Inter-frontends	Cyclic Dependency	Two or more MFEs directly or indirectly depend on each other, resulting in high coupling between screens and fragments, compromising MFEs' independence and modularity. Thus, changes in one MFE require coordination with the others. Circular dependencies lead to challenges in a system's maintenance and evolution, compromising agility and the ability to scale developments efficiently.	Consider an e-commerce application featuring a product details screen implemented on mfe-products. The screen integrates three fragments: one displaying the shopping cart from "mfe-checkout," another showing product recommendations from "mfe-recommender," and a third calculating shipping costs based on the selected delivery address from "mfe-delivery." When a recommended product is added to the cart, "mfe-recommender" notifies "mfe-checkout," which subsequently informs "mfe-delivery" to recalculate the shipping costs. If the shipping address is updated, "mfe-delivery" notifies all other MFEs to verify whether the products they display can be shipped to the new address; if not, those products are disabled. This interaction between the screen and its fragments results in high coupling, where changes or updates in one fragment often necessitate adjustments in others to maintain the overall functionality of the product details screen.	High coupling between MFEs can be effectively mitigated by implementing event-based communication. This eliminates the need for direct communication between MFEs, replacing it with indirect interactions through a centralized event store. The Pub-Sub pattern promotes robust decoupling between MFEs, facilitating the integration of new MFEs without compromising the overall cohesion of the system.
	Knot Micro Frontend	A Knot is composed of three or more MFEs whose communication with each other uses a context-specific interface. This means that navigation and data exchange between screens and fragments heavily depend on the unique context of each MFE involved. Adding new MFEs exacerbates the problem: as the number of MFEs grows, the interface complexity increases due to the introduction of new contexts creating a highly coupled Knot that becomes difficult to maintain and integrate new functionalities.	Suppose an e-commerce system has MFEs for Digital Products (mfe-digital-products) and Payments (mfe-payments). The product details screen of mfe-digital-products navigates to the payment screen of mfe-payments, passing the product data as a parameter. At a later stage, a Physical Products MFE (mfe-physical-products) is implemented, including screens like delivery tracking, address listing, and address registration. For the unpaid physical products, a modification in the mfe-payments payment screen happens for receiving data of either digital or physical products. Later, adding new product types requires constantly adjusting the payment screen of mfe-payments to display the data for these products. Implementing a communication interface with the following fields would allow the addition of new products without requiring adaptations in mfe-payments. This interface definition must allow the transmission of product details in a uniform format for all types of products. Furthermore, it permits the optional transmission of delivery information and specifies whether delivery information can be provided.	A practical solution to address the Knots is to implement domain-driven communication interfaces. These interfaces define a contract based on the domain model, specifying what data and functionalities each MFE needs to interact with others. This approach allows MFEs to understand the integration requirements without being concerned about the specific implementation details of data structures or internal logic within another MFE. As new attributes emerge in new sub-domains, it is important to implement them in the interface such that they can be utilized by any other product, thereby maintaining their generalizability.
	Hub-like Dependency	A screen of a MFE integrates fragments from several other MFEs, becoming a central point of interdependence. Any issue occurring in the main screen or one of its fragments can affect all other fragments present on it.	Consider a digital banking system where the main screen is an MFE that integrates several fragments from other MFEs, such as an investment list, a chart showing bitcoin value variations, account balance, and credit card statement amount. This structure introduces a significant vulnerability: a single faulty fragment can potentially disrupt the entire main screen. Consider a scenario where an issue within the investment list fragment causes it to malfunction. This malfunction could manifest as data display errors, unresponsive controls, or even complete crashes. The consequences of such an incident extend beyond the affected fragment, rendering the entire main screen and other fragments unusable and inaccessible to the user.	To prevent a single fragment failure from crashing the entire main screen, the screen should be kept as simple as possible, and each fragment should implement robust error handling mechanisms. This can be achieved by implementing a strategy where uncaught errors within a fragment gracefully degrade its functionality, displaying a user-friendly fallback message. This approach ensures that users are informed of the issue without hindering their interaction with the remaining functionalities on the main screen.
Intra-frontends	Nano Frontend	The frontend decomposes into numerous small MFEs with few screens or fragments. Small MFEs do not justify the cost of their maintenance. Furthermore, the presence of nano frontends can lead to issues of high coupling and the manifestation of other anti-patterns, such as cyclic dependency	In an e-commerce setting, separate MFEs implement the product listing and product details screens. Since both are part of the product context, their implementation should happen within a single MFE encompassing all product screens.	The issue of nano frontends arises when the definition of boundaries is inadequately and excessively granular. Adhering to Domain-driven Design (DDD) principles is necessary to ensure an effective decomposition of MFEs. Therefore, the development team must work closely with the product team to gain a deep understanding of the domains and reflect them accurately in the architecture. To solve this issue, the architecture must be redesigned by grouping MFEs with the same domain is necessary. For minor variations within a domain, consider using templates or component libraries. This approach avoids creating a separate MFE for each slight variation, promoting efficiency and code reuse.
	Mega Frontend	Decomposing the architecture into a few MFEs encompassing numerous screens and fragments manifest this anti-pattern. The MFE inherits the challenges of a monolithic frontend, such as difficulties in testing, slow builds and deployments, high coupling between its components, lack of modularity, and limited scalability.	An e-commerce system is decomposed into just two MFEs, with mfe-users related to users and mfe-shopping related to products and purchases. The latter MFE includes screens that display product listings, product details, purchase confirmations, and purchase history. Decomposing the mfe-shopping into at least two MFEs is necessary: one containing the product listing and product details screens, belonging to the product domain; and another containing the confirmation and purchase history screens, belonging to the purchase domain.	To avoid this problem, the development team must work closely with the product team to gain a deep understanding of the domains and reflect them accurately in the architecture. To fix this issue, the team should reevaluate the architecture and divide the MFEs into more granular units, separating functionalities into smaller and specialized MFEs based on domains. This approach helps reduce complexity, enhance maintainability, and foster a modular and scalable architecture.
	Micro Frontend Greedy	When a developer is uncertain about creating a new MFE, the common practice is to opt for its creation. Whenever a need arises to develop a new set of screens or fragments, a new MFE is instantiated. This can lead to an explosion in the number of MFEs, making the system difficult to understand and increasing the likelihood of both nano and mega frontends emerging.	Within a banking application, an MFE encompasses screens for security validation, utilizing confirmation code submission via email. Subsequently, the need arose to implement a new validation method, now employing facial recognition. The screens in this new flow differ from those in the previous flow, resulting in its implementation through a new MFE. Creating a new MFE might not be advisable, as two MFEs have the same context and functionalities.	To determine where to implement a new feature composed of a set of screens and/or fragments, the domain of the new feature must first be defined. If it falls within the domain of an existing MFE, it should be implemented there. In this case, a summary of all MFEs, their contexts, and domains can help identify the best fit for the new feature. If it belongs to a brand new domain, one or more MFEs should be defined based on the domain definition. Establishing well-defined domains relies on the collaboration between the development and product teams to accurately define boundaries.
Operation	No CI/CD	The company lacks an automated Continuous Integration (CI) and Continuous Delivery (CD) pipeline, so developers must manually execute tests and perform deployments. This manual process becomes burdensome, especially with the potential existence of multiple MFEs. It increases development time, reduces productivity, and raises the risk of errors in the production environment.	Upon releasing a new system version, a developer must conduct manual tests and ensure all unit tests pass. However, developers may skip the tests and manually deploy the changes without realizing some tests are failing, introducing bugs, which is avoidable with an automated CI pipeline. Even if the tests pass, there is still a risk of making mistakes during deployment, which could render the system unavailable. Automating the deployment process with CD ensures correct and consistent execution.	Implement an automated and replicable CI/CD process that extends for new MFEs, ensuring they will have automated test execution and deployment consistently and efficiently. This should be part of the Definition of Done (DoD) of the architecture.
	No Versioning	The MFEs are not versioned. Small and large changes can impact the integration between different MFEs and cause errors. Consequently, the MFEs become less independent, requiring coordinated deployments.	Consider a payment confirmation page with a fragment for calculating shipping costs. Whenever the user inputs shipping information into the fragment, the system generates a delivery charge and adds it to the total purchase amount displayed on the screen. Suppose the delivery charge's return value format changes and the fragment is not versioned. The delivery charge will not be added to the total purchase amount, potentially resulting in a display error or even mistakenly free deliveries. However, if the fragment is versioned, the screen will not be affected by the format change, as it will continue to use the previous version of the fragment and can be updated later when necessary.	Adopting a versioning approach like Semantic Versioning is essential to ensure that changes do not impact functioning versions. For example, consider a fragment that is used in screens across different MFEs in a client-side rendering scenario. Without versioning, any change to the fragment's parameters or return values could break the interaction on all the screens it integrates with. However, with versioning, such updates would not impact the current versions used by other MFEs, as they can continue to request the previous version of the fragment and update at their convenience. This approach helps maintain a stable environment and minimizes disruptions caused by updates.
	Lack of skeleton	No skeleton or predefined boilerplate is available as a base for creating new MFEs. This leads to the creation of MFEs from scratch or based on an existing MFE and inheriting its issues. The consequences include wasted time, increased risk of errors, duplicated code across MFEs, and a need for more standardization in development.	At the beginning of a specific system development, the developers create an MFE from scratch without adhering to a specific pattern. The second MFE is developed by copying files and code blocks from the first, changing specific parts. The exact process happens when creating new MFEs. Then, a diverse set of MFEs emerges, which hampers the establishment of automated pipelines, fosters code duplication, and complicates developer interchange between teams.	Whenever a new technology is used to implement a MFE, the development team must create a repository containing the necessary base code, known as a boilerplate. The boilerplate should enable the creation of new MFEs with the same technology by simply cloning it. Keeping the boilerplate updated with new design patterns and library versions is crucial. Additionally, the development team should create comprehensive documentation detailing the entire process of creating a new MFE, regardless of the technology. This documentation should provide instructions on adding automated CI/CD, integrating the MFE into the existing system, and addressing other relevant aspects.
	Common Ownership	A single team is tasked with managing all MFEs, which can occur either due to a lack of team division or when teams are segmented based on technical aspects such as data, frontend, and backend. However, one of the key benefits of MFE architecture is independence, so adopting MFE Architecture without distinct teams to operate independently negates this advantage.	A small company chooses to adopt the MFE architecture. Due to the insufficient number of developers, it is not feasible to create teams by domain. So, developers compose a single team responsible for all MFEs. In this scenario, the cost of maintaining different micro frontends is not justified and is only an additional challenge for the development team.	Context should be the defining factor when structuring development teams. Therefore, defining the boundaries of teams and MFEs is essential according to Domain-driven Design (DDD), so a team will be responsible only for MFEs within its domain. Creating shared libraries can facilitate boundary definition and promote greater team independence.

Improved Micro frontends Anti-patterns Catalog				
Category	**Title**	**Description**	**Occurency in practice example**	**Solution**
Development	Golden Hammer	All MFEs utilize the same technology, even if it does not meet the specific needs of each MFE. It happens due to developers' familiarity with only one specific technology. This approach limits the architecture, failing to take advantage of the benefits of the possibility of a heterogeneous architecture, which is one of the main attractions of adopting MFEs.	A web application contains MFEs implemented using ReactJS framework with Client-side Rendering, even those encompassing essential pages such as the landing page. This technological uniformity overlooks the necessity for Search Engine Optimization (SEO) strategies to ensure better rankings on search engines like Google. It would be advisable to utilize ReactJS with Server-side Rendering or employ a static rendering framework such as NextJS, enabling better optimization for search engines.	To choose the most suitable technology that addresses the specific challenges of each MFE, which includes adopting the correct programming languages, frameworks, and libraries during its development. When uncertain about a particular technology, conducting a proof-of-concept (POC) can validate its suitability. Testing new technologies through POCs helps validate their suitability without compromising the establishment of standardized patterns within the company. However, it's important to note that increasing the variety of technologies can increase the complexity of the architecture.
	Micro Frontend as the goal	Adopting the MFE architecture in inappropriate contexts can lead to more issues than benefits, especially in systems with few screens and low complexity or in companies lacking a sufficient number of developers to create dedicated teams for different application domains. In such situations, the maintenance costs of the architecture may outweigh the expected benefits, making its implementation unfeasible.	A personal notes application is divided into the notes and user domains, each comprising its own MFE. The notes domain contains functionalities for note management, containing operations such as listing, creating, editing, and deleting notes. The user's domain encompasses login, registration, and profile management functionalities. In this context, using MFEs results in unnecessary maintenance and development challenges due to the low volume of screens and the low probability of increasing complexity in the application. Adopting a monolithic frontend is a suitable option.	Software teams must consider carefully different aspects of adopting MFE architecture. Considering the system's complexity, the feasibility of maintaining automated CI/CD pipelines and the team's restructuring according to different domains is necessary.