Micro frontends Anti-patterns Catalog v1				
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 platform with a payment screen implemented in one MFE. This screen contains a fragment from another MFE used to calculate the shipping cost. When the screen displays changes to the items in the cart, the fragment redoes the delivery calculation, which updates the total purchase amount displayed on the screen. This exchange of information between the screen and the fragment results in high coupling. It is essential to assess whether the payment screen and shipping calculation belong to the same domain and if their implementation is part of the same MFE.	High coupling between two or more MFEs indicates they should compose a single MFE. Thus, it is necessary to review the definition of MFE boundaries and ensure they align with the application domains. A careful analysis of the domains and reassessing the functional division of the MFEs can help reduce coupling, improve modularity, and increase the independence between components.
	Knot Micro Frontend	A Knot is a node composed of three or more MFEs whose communication with each other has a low level of abstraction, exposing the specific details of each MFE. The navigation or data exchange between screens and fragments strongly depends on each MFE context. The problem worsens as the addition of new MFEs to the node occurs without implementing a standardized communication interface. This results in a strongly coupled node, making it difficult to maintain and deploy 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: { imageUrl: string; title: string; description: string }[], shippingData: { ... } | null}. This interface definition allows 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 is implementing communication interfaces between MFEs, allowing an MFE to know what is necessary to integrate with another while abstracting the specific details of its data or implementations.
	Hub-like Dependency	A screen of an 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. If any critical issue happens on the main screen, all its functionalities become inaccessible to the end-user	Avoiding screens that serve as a starting point for other functionalities is recommended. When it is not possible to avoid it, we recommend implementing a resilient screen, incorporating error-handling procedures for all MFE fragments, along with a fallback mechanism. The fallback mechanism should allow access to system functionalities in case of any critical issue with the main screen.
Intra-frontends	Nano Frontend	The front end 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. So, redesigning the architecture by grouping MFEs with the same domain is necessary. It promotes a cohesive structure aligned with the business requirements.
	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.	Reevaluate the architecture and divide the MFEs into granular units, separating functionalities into smaller and specialized MFEs. This approach aids in reducing complexity, enhancing maintainability, and fostering 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. It can lead to the creation of nano frontends or mega frontends.	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 assess whether a new screen or fragment can integrate an existing MFE, a comprehensive review of all existing MFEs is essential. It may avoid unnecessary MFE proliferation, promoting a cohesive and sustainable architecture.
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.
	No Versioning	The MFEs are not semantically 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.	It is essential to adopt the Semantic Versioning standard for versioning MFEs, where the developer must assign a Major version when changes introduce incompatibilities, create a Minor version for new functionalities that do not cause incompatibilities, and apply a Patch version for bug fixes that do not introduce incompatibilities. The versioning practice ensures that changes do not impact functioning versions. Consequently, coordinated deployments are unnecessary, as other MFEs can update their versions as needed.
	Lack of skeleton	No skeleton or predefined boilerplate is available as a base for creating new MFEs, which leads to the creation of MFEs from scratch or duplicating an existing MFE. 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.	Create a repository containing the necessary base code to build an MFE called boilerplate, which includes all the required libraries for the MFE's operation, adhere to the code standards established by the team, and have a file with commonly used commands for developers. It is also crucial to include comprehensive documentation detailing the entire process of creating a new MFE, providing instructions on how to add automated CI/CD, integrate the MFE into the existing system, and other relevant aspects.
Development	Common Ownership	A single team is responsible for managing all MFEs. It happens when there is no team division or when they are divided based on technical aspects such as data, front-end, and back-end teams. The team does not leverage the benefits of having independent teams provided by the MFE Architecture.	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.
	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.
	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.