

Filling The Gaps in Microservice Frontend Communication: Case for New Frontend Patterns

Amr S. Abdelfattah^a and Tomas Cerny^b

Computer Science, ECS, Baylor University, One Bear Place #97141, Waco, TX 76798-7356, U.S.A.

Keywords: Microservices, Micro-Frontend, Backend for Frontends, Microservices Patterns, API Management.

Abstract: Microservices architecture has exploded in popularity; many organizations use this architectural style to avoid the limitations of large and monolithic backends. Most systems require multiple frontend clients, such that each frontend client expects tailored responses from a backend service. However, there are no best practices for their integration and communication with microservice backends. Backend for Frontends (BFF) is one of the most used patterns for gluing the frontend with the microservices layer. It keeps the frontend layer decoupled from the microservices complications; nevertheless, it is tightly coupled with the frontend layer. Therefore, it introduces barriers in the development process, besides adding risks for business inconsistency. In addition, it negatively impacts the consumed overall data size and time over requests. This risk is boosted by the evolution of the micro-frontend architectural style that encourages the decomposition approach for the frontend components. This paper proposes an alternative pattern that addresses current gaps introduced by the BFF patterns. It supports cloud-native system components to provide the required customization to frontends, along with increasing the frontend awareness to share more responsibilities in the architecture. The new pattern facilitates customizability for client types when interacting with the microservices business layer.


1 INTRODUCTION


Cloud-native systems contain multiple distributed, independent, and self-contained microservices that communicate together as a middleware to achieve scalable systems functionalities (Cerny et al., 2018; Brown and Woolf, 2016). While many best practices are recognized for architecting the cloud-native middleware (Carnell and Sánchez, 2021), there is a significant gap in established practice for connecting the middleware with the user interfaces. These interfaces provide a system frontend and simplify human-computer interaction. Typically, they hide the internal system complexity of decentralization from end users and interface multiple system microservices. However, too few guidelines and best practice examples exist for the connection between microservice middleware and the user interface frontend (Brown and Woolf, 2016).

Practitioners often resort to a common solution called Backend for Frontends (BFF) (Newman, 2015) to address the challenge of accommodating different user interfaces within a system, as mentioned in the

Related Work section. The BFF adds an additional layer between the user interface and the backend, with BFF components communicating with backend services and adapting the data to meet the requirements of a specific user interface. However, because a BFF component is tightly coupled to a specific user interface experience, it is recommended to have individual BFFs for each user interface (i.e., website and mobile application). Despite its demonstrated success, the BFF pattern introduces numerous challenges when integrated with microservice and micro-frontend architectures. These challenges are discussed in the subsequent sections, which cover various perspectives, including Request Performance, Development Process, and Business Inconsistency.

This paper proposes an alternative solution to overcome the above problems. It provides a pattern of communication between user interfaces and the microservices backend layer, called the Frontend Micro-Communication (FMC) Pattern. It optimizes the data and time consumption of the frontend requests. It also avoids the extra development barrier which the BFF introduced into the process. Moreover, it encourages retaining business consistency inside its corresponding microservices.

^a  <https://orcid.org/0000-0001-7702-0059>

^b  <https://orcid.org/0000-0002-5882-5502>

In this paper, we provide a detailed perspective of this solution and assess its properties using a small case study. The results show that the proposed solution achieves the same purpose as the existing solutions of BFF. However, compared with the BFF, the FMC solution shows better performance, especially for the frontend time to receive a response. Moreover, it discusses the methodology for reducing the coupling between the development teams, which encourages the micro-frontend architecture to scale independently.

The paper organization is as follows: Section 2 discusses the related work followed by 3 background and the relations to other frontend communication patterns. Section 4 states the problem definition followed by Section 5 on the proposed FMC pattern. Demonstration and discussion are given by Sections 6 and 7. Finally, Section 8 concludes the paper.

2 RELATED WORK

There are not enough guidelines and good examples to follow for presenting the frontend layer in the microservices architecture Brown et al. (Brown and Woolf, 2016). Nevertheless, multiple literature reviews are driven to investigate the lack of well-identified communication patterns.

Osses et al. (Osses et al., 2018) elaborated on academic architectural tactics and patterns in microservices through a systematic literature review. They concluded that among 44 patterns, the BFF pattern is the most well-identified pattern that is responsible for the communication between the frontend and the microservices' backend components. Different BFFs are implemented for different types of clients, each with an API customized to what that client type needs.

Valdivia et al. (Valdivia et al., 2019) derived a systematic literature review. They emphasized the importance of BFF to facilitate the interactions for the frontend component. The BFF pattern is identified as an entry point pattern, such that it acts as a single API for a client. Moreover, the BFF could have a responsibility to reduce the load of the API gateway layer as well. However, the authors highlighted that it has the disadvantage of being a single point of failure and is inconvenient for high workloads. Therefore, this could be the circumstance when the micro-frontend architecture employs the BFF pattern in its communication architecture.

Furthermore, Márquez et al. (Márquez and Astudillo, 2018) inspected 17 patterns from microservices-based open-source projects. The BFF is shown as the only frontend communication

pattern that is found in their study. It is introduced as a way to cater to different frontend client types in a system by providing a tailored API for each frontend client type.

These studies depict an image of the BFF as the most common pattern that is utilized for the communication purpose of frontend and backend layers in a microservice architecture.

3 BACKGROUND

This section delves into the intricacies of the BFF pattern and micro-frontend architecture. It details how the frontend layer communicates with the microservices layer.

Regarding the BFF pattern (Brown and Woolf, 2016; Harms et al., 2017), it acts as a composite single API for a frontend client, such that multiple BFFs are implemented corresponding to the different types of clients to provide their customized needs, as shown in Figure 1. It has appeared to eliminate the difficulties of *General Purpose Backend* (Brown and Woolf, 2016) pattern that adds complexity in the frontend, in addition to it increases the data consumption for client applications. The BFF pattern faces two challenges: (1) the frontend requires multiple calls to satisfy a single page requirement, and (2) different client types (e.g., Web, Mobile) may need different response data attributes, resulting in the service sending all attributes to all clients, even if some are not needed.

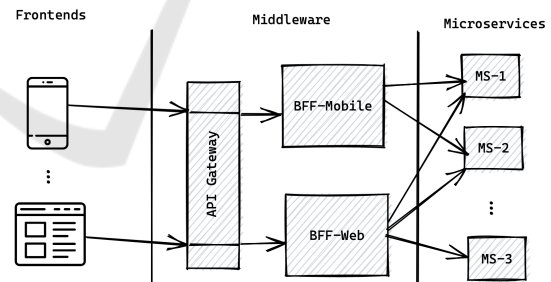


Figure 1: Backend For Frontends Pattern.

The BFF pattern enables the communication between the frontend and microservices to achieve business functionality with minimal requests (Osses et al., 2018; Márquez and Astudillo, 2018). It has multiple responsibilities, including orchestration, translation, and filtering. It orchestrates multiple microservice calls for a single client request, translates results to fit client requirements, and filters unnecessary data for the frontend.

On the other hand, the micro-frontend (Peltonen et al., 2021; Fowler, 2019) architecture appears to apply the distribution approach in the frontend layer.

Nevertheless, the frontend layer can still be implemented as a monolithic application while communicating with a microservices-based backend. The Single Page Application (SPA)¹ (Brown and Woolf, 2016) frontend pattern constructs a web-based client type. It loads only one web document while updating its content via API calls for a more dynamic experience and better performance gains. However, this approach may limit the scalability of frontend development, especially for large projects (Fowler, 2019). To overcome this challenge, the micro-frontend architecture divides the monolithic frontend into smaller, independent, and reusable components, each responsible for a specific aspect of the user interface. This approach enhances deliverability, upgradeability, and maintainability through decoupling within organizations, and features smaller components, more cohesive code, and greater scalability.

The micro-frontend architecture creates a micro-frontend for each page in an application, while a single container application handles rendering common elements such as headers and footers, as well as managing authentication and navigation across the different micro-frontends. However, this approach also has some drawbacks, including the need to manage multiple artifacts (e.g., repositories, tools, domains, build and deploy pipelines) for each micro-frontend component. Additionally, it can result in large payload sizes and duplication of dependencies (Fowler, 2019).

The BFF pattern serves as a communication bridge between frontend and backend layers. It is adapted as the backend for each micro-frontend (Fowler, 2019), creating an additional layer in the system, as illustrated in Figure 2. This layer adds complexity and challenges in managing team dependencies during development as the microservices and micro-frontend layers scale.

4 PROBLEM DEFINITION AND CHALLENGES

The Microservices architecture is based on the concept of decentralization, where self-contained microservices interact with each other to produce the overall system. Each microservice has different functionality and interacts with different Data Stores to serve the business domain of the system. The Frontend, which is the presentation layer containing user interfaces, such as web applications and mobile applications, is integrated with various infrastructure-

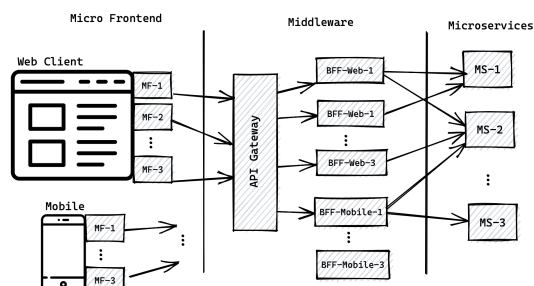


Figure 2: Micro-Frontend Architecture.

supportive services and components (Carnell and Sánchez, 2021).

Enterprise systems commonly use multiple user interfaces and implementing a frontend layer, whether monolithic or micro-frontend layer, requires adaptation to the microservices architecture to meet the frontend client needs. To manage interactions and communications between the frontend clients and microservices, different patterns are used, which vary from system to system.

The purpose of this paper is to present a new communication pattern for the frontend in the microservice architecture, specifically addressing the challenges associated with the widely used BFF pattern. The goal of the proposed pattern is formulated as follows:

Designing a frontend communication pattern, for the purpose of filling the gap of providing a client-specific service interface in a microservice architecture, concerning the request performance, the development process, and business inconsistency.

Prior to introducing the proposed pattern, we analyze the deficiency in frontend communication within the microservice architecture from two main perspectives: the challenges faced when implementing the existing BFF pattern within the microservice architecture, and the feasibility of extending the BFF pattern to suit the micro-frontend architecture type.

First off, integrating pattern with the microservices distribution style poses several challenges. The BFF is an additional layer in the architecture (see Figure 1) to proxy communication between the frontend and backend, making it a potential single point of failure for high workloads (Valdivia et al., 2019). Additionally, the BFF can call multiple microservices to fulfill a frontend request, which impacts request consumption time, such that waiting for all inter-service requests to conclude can increase the frontend’s waiting time.

¹SPA: <https://developer.mozilla.org/en-US/docs/Glossary/SPA>, accessed on 02/13/2023.

Furthermore, the BFF consumes large amounts of data from business microservices, filters out the required attributes for clients, and creates duplicate logic in the system when a separate BFF component is implemented for each frontend client type (Newman, 2015). This duplication increases the risk of embedding business logic in the BFF layer and causes inconsistency when changing the logic in one BFF component. For example, if the logic changes in one BFF, inconsistencies may occur in the other BFFs, leading to a greater risk of business inconsistencies during system modifications.

The coupling of the frontend and BFF teams adds a barrier to the development process, where changes required by the frontend team also require modifications by the BFF team or could cause a ripple effect. This barrier becomes problematic, especially with large systems that have multiple teams.

Secondly, the BFF pattern can be extended to a backend for each micro-frontend in the integration of the BFF with the micro-frontend architecture (Fowler, 2019), as illustrated in Figure 2. However, this can be an overhead to the system, especially if the micro-frontend only requires one microservice endpoint to invoke. Moreover, each BFF has its own business logic, which could result in business inconsistency among different client types.

Additionally, Modifications and new features in a micro-frontend may also necessitate changes to the backend response. Therefore, the BFF may impede expansion and the attainment of micro-frontend scalability, and it may cause a bottleneck in development processes, particularly if it is not owned by its corresponding frontend team.

5 FMC PATTERN

The FMC pattern provides an alternative approach for managing the communication between the frontend and microservices backend. Unlike the BFF pattern, the FMC pattern retains system scalability and reduces the coupling between the development teams.

This section follows the pattern writing language and style explained in (Wellhausen and Fießer, 2011). It details the description and the implementation methodology of the proposed FMC pattern. It includes multiple perspectives of the pattern as follows: *Context, Problem and Forces, Solution, and Consequences* of benefits and liabilities.

5.1 Context

The context of this pattern is building a microservices-based system and different fron-

end client types. Such that the frontend clients consume the microservices APIs for receiving tailored responses.

5.2 Problem and Forces

Reduce the development barriers between frontend and microservices teams that result from the BFF layer. Minimize data communication overhead and API response time to the microservice frontend. Mitigate the risks of business inconsistency between different frontend client types.

Some forces restrict the BFF pattern to solving these problems, as follows:

1. Development Process: The BFF pattern develops an additional project for handling the communication between the frontend and backend. This project involves an additional repository to control its source code and manage its deployment pipeline. This project could be implemented in different languages that require a different team to develop and manage changes. However, there is coupling between the frontend and the BFF layers, therefore, modifications in the frontend require adaptation from BFF components as well. Thus, the barriers happen as a result of every single change.

2. Business Inconsistency: The BFF has an orchestration responsibility to invoke multiple microservices to construct the required collective response. This functionality adds a risk that the business logic becomes embedded in the BFF layer instead of within the underlying business microservices. Creating a BFF component per client type increases the risk that logic becomes inaccessible to other client types unless it is replicated to all BFF components.

3. Data Consumption: The BFF layer receives a response from the business microservices that contains all data attributes. It filters them out according to the needs of the invoking client type. Therefore, this communication increases the data consumption between the business and the BFF layers, even if this data is useless for the frontend client.

4. Response Time: A frontend request to the BFF is translated into multiple microservices requests from the BFF component to the microservices business layer. The frontend waits until all requests are concluded to receive a complete response from the BFF. Thus, it waits for the longest response time among the invoked inter-services.

5.3 Solution

The FMC pattern eliminates the need for the frontend to wait for the longest inter-service response to be

ready by utilizing a single-request-multiple-response communication approach, which leads to better performance in terms of response time and data consumption. Unlike the BFF pattern, it avoids adding extra layers to the architecture that can introduce barriers in development and couple the extra layer and instead leverages microservices architecture components.

This pattern is designed over three layers of the architecture as follows: Frontend, API gateway (Middleware), and microservices business layer. The three modifications are highlighted in red color in Figure 3, such that the *Socket* as an example of full-duplex communication (e.g., *WebSocket*²) methods that allow both the client and the server to send and receive data from each other. This communication protocol is integrated for communication between the API gateway and the frontends. The *Contract* is the Consumer-driven Contract (Fowler, 2006) for describing the frontend requirements and their requests compositions. Finally, the *Mobile* and *Web* red boxes symbolize the integrated adapters for data communication with the microservices. They sort out the attributes in the response based on the client types.

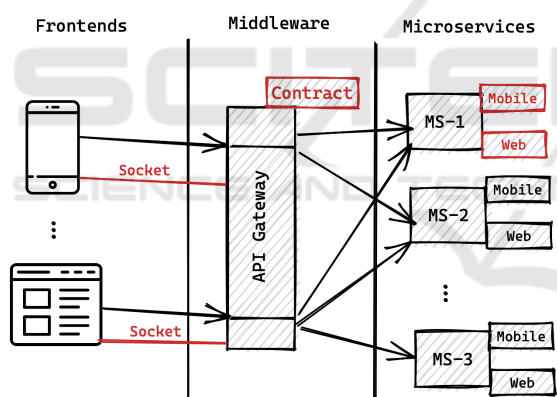


Figure 3: The Proposed FMC Pattern.

The FMC pattern flows as the middleware receives the frontend request, then it translates the received request into multiple microservices requests based on a predefined *Consumer-driven Contract* located in the middleware. For example, the sampled contract in Listing 1 indicates the middleware to construct three microservices requests per the "service1" request. After that, the middleware detects and injects the client type into the constructed requests header. Then, the business microservices perform their business logic, and they check the client type to apply tailored adapters for customizing the response attributes

²WebSocket: <https://datatracker.ietf.org/doc/html/rfc6455>, accessed on 02/13/2023.

accordingly. These adapters follow the pluggable adapter design pattern; they use Data Transfer Object (DTO) (Cerny et al., 2018) and Wish List (Stocker et al., 2018) patterns to avoid impacting the microservices business cohesion. The middleware receives the responses for each microservices request individually. Then it forwards them through the *Socket* protocol to the frontend client without waiting for all requests to respond. Finally, the frontend actively receives multiple *Socket* responses regarding its single-initiated request. The pseudocode in Listing 2 demonstrates the frontend calls "/service1" to receive two microservices responses separately. Therefore, the frontend can render partial responses to the user and decrease the application response time.

Listing 1: Consumer-driven Contract Sample.

```

1 /service1:
2   microservices:
3     - /MS1:
4       ForClient: True
5     - /MS2:
6       ForClient: True
7     - /MS3:
8       ForClient: False
9   - Dependencies:
10      MS1.response.dataId -> MS2.request.data.id
    
```

Listing 2: Frontend Snippet for a Service call (Single-call multiple responses).

```

1 ServiceConsumer.call("/service1") {
2   # MS1
3   callback1(response1) { ... }
4   # MS2
5   callback2(response2) { ... }
6   onFinish() { ... }
7 }
    
```

The sequence flow of the pattern is depicted in Figure 4 as follows:

1. The *Frontend* client sends a specific service request through the *API Gateway*. And also, the frontend generates and embeds a unique request ID to the request body for identifying this request.
2. The *API Gateway* sends an immediate response to the *Frontend* to acknowledge that request is in progress. This immediate response could contain an alert message or cached response to display until the completion of the request.
3. The *Frontend* connects a socket channel with the *API Gateway* parameterized with the related request ID.
4. The *API Gateway* uses the *Consumer Contract* to map the received request into the corresponding

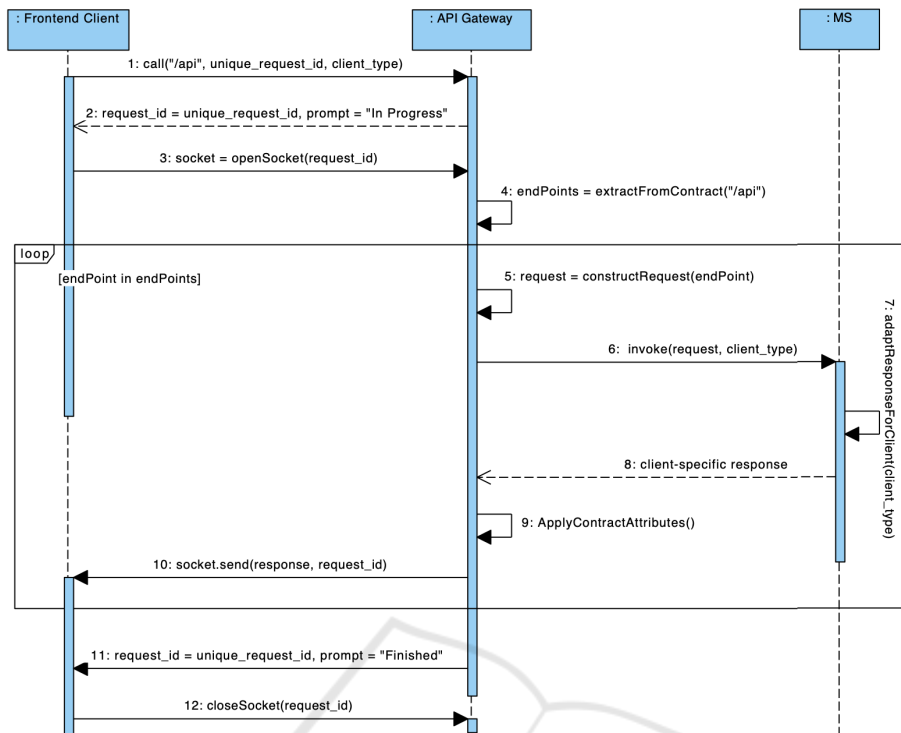


Figure 4: The FMC Pattern sequence flow.

4. The *Frontend Client* sends a request to the *API Gateway*.
5. The *API Gateway* constructs the corresponding microservices requests.
6. The *API Gateway* invokes these requests after it injects the client type and the request ID into their headers.
7. The *Microservice* executes the request. Then, it checks the client type attribute to prepare a corresponding client-specific tailored response data.
8. The *Microservice* responds to the *API Gateway* with the customized response.
9. The *API Gateway* applies some filters to the received response based on *Consumer Contract* specified attributes. These attributes are detailed in the below list.
10. The *API Gateway* embeds the request ID into the received responses, then it sends them to the *Frontend* through the connected socket channel.
11. The *API Gateway* sends a conclusion signal through the socket once the composite requests of the original request have been completed.
12. The *Frontend* closes the socket channel, such that the socket channel duration matches the request execution time.

The *Consumer-driven Contract* specifies two attributes (Dependencies and ForClient) to control the request and response behaviors. As listed in Listing 1, The *Dependencies* attribute describe the dependencies

between microservices requests to fulfill single frontend requests. The *ForClient* specifies which response parts should be forwarded to the frontend. These attributes are detailed as follows:

- *Dependencies*: It coordinates the dependencies between the listed microservices and each other. It could postpone a service call until the dependent service responds, in addition to transferring data from the received response to the upcoming requests. For example, as shown in Listing 1, a dependency of $MS1.response.dataId \rightarrow MS2.request.data.id$. This means *MS2* waits for the response of *MS1*, and then the API gateway should extract the attribute with the key *dataId* from the *MS1* response. After that, it inserts the value of this attribute into the *MS2* request body according to the *request.data.id* keypath. Notice that, the API gateway neither performs data conversion nor knows the logical meaning of these attributes, it follows the generalization and abstraction approaches for applying the dependency relation between the parties.

- *ForClient*: This attribute controls if a specific microservice response should be forwarded to the frontend client. It differentiates a microservice needed for business logic and the one the frontend needs its response to the display. Thus, this optimizes the data consumption between the middleware and the frontend layers so that the frontend client receives the needed data only.

5.4 Consequences

The FMC pattern proposes a solution to the mentioned problem and its forces. This solution has consequences of benefits and liabilities to illustrate the pattern tradeoffs.

The benefits are addressed as follows:

1. Development Process: This pattern approaches the avoidance of the development process barrier that happens in the BFF pattern. It avoids adding additional projects to the development teams and additional layers to the microservices architecture. It overcomes the coupling between the BFF layer and the frontend; however, it accomplishes the purpose behind that by providing the frontend clients with customized responses.

2. Business Inconsistency: The combination between the API gateway and the consumer-driven contract manages the orchestration required for accomplishing the frontend request. Therefore, the business-related microservices are the only components responsible for the business logic and its response customization in the system; Especially the API gateway follows the generalization approach, such that it does not convert or modify the received responses before forwarding them to the frontend.

3. Data Consumption: The business layer responds with the only attributes required for the invoking client type. This optimizes the data transfer and consumption per each request.

4. Response Time: The frontend clients receive multiple partial responses per request. Therefore, they can start gradually rendering the received data. This technique boosts the usability of the application, such that it enables the user to interact with the interface while it displays partial responses until the whole response data is received and rendered.

The liabilities are analyzed as follows:

1. Development Process: Extending the functionality of the API gateway to handle the customer-driven contract logic. However, this logic could be implemented in a generic technique. It should be constructed into a library to be reused in multiple projects. Moreover, including the full-duplex communication (i.e., Socket connection) requires more effort for handling and recording accurate logs, and for investigating the system issues. Furthermore, the FMC adds responsibility on microservices to adjust the response attributes according to the client type.

2. Business Inconsistency: The frontend code should be designed to accept gradual responses, such that it receives multiple responses per single request. Thus, the frontend participates in the orchestration functionality as well.

3. Data Consumption: Extra responsibility is added to the microservices business layer. It adapts the response data according to the client type. However, multiple patterns help in implementing this in a standard approach, such as DTO and Wish List patterns.

4. Response Time: Achieving the single-request-multiple-response approach requires multiple handlers from the frontend side to receive these responses through the socket protocol. Therefore, the frontend prepares a callback for each expected partial response as demonstrated in Listing 2.

6 USE CASE

This section highlights a use case that showcases and confirms the effectiveness of the FMC pattern. It also compares its performance with the BFF pattern, specifically in terms of the size of the response and the time taken to consume requests from multiple frontend types

6.1 Testbench Description

We have created a cloud-native software application³ as part of our study that serves the purpose of showcasing organizational management for licenses. This application, which employs the Java Spring Boot framework, consists of three layers: a frontend, an API gateway, and a business layer. We implemented the API gateway utilizing Spring Cloud Gateway, integrated with Spring Data JPA for data persistence, Spring WebFlux for WebSocket implementation of full-duplex and reactive communication, Spring Cloud Configuration Server, and Eureka Service Discovery. We deployed the testbench through Docker containers, utilizing the Compose tool for the deployment dependencies.

The business layer contains two microservices called *License-Service* and *Organization-Service*. These contain endpoints as detailed in Table 1. The *License-Service's* `/list` returns a list of licenses, and `/details/license-id` returns the details of a specific license. The *Organization-Service* provides a single endpoint called `/details/org-id` to return specific organization information. The endpoints retrieve stored data. However, we have added a constant delay to simulate more load. This delay helps to compensate for the variable overhead we remove, as detailed in the procedure section.

³Implemented Testbench: <https://github.com/amr-abdelfattah/Micro-communication-Pattern>, accessed on 02/13/2023.

Table 1: Teshbench Endpoints.

Frontend Request	License-Service	Organization-Service
<code>/license/list</code>	<code>/list</code>	—
<code>/license/details/{license-id}</code>	<code>/details/{license-id}</code>	<code>/details/{org-id}</code>

Two databases are developed using JPA in-memory data. These databases are initiated by a randomly generated dataset of 20 license items and 30 organization items. This testbench supports two different frontend client types (mobile and web); These clients expect two requests from the backend side to fulfill their needs as listed in Table 1; however, they require customized responses.

As shown in Table 1, the first frontend request is `/license/list`, which communicates with the *License-Service* with endpoint `/list` to return a list of *License Model*; however, each client type requires different fields in the returned list as reported in Table 2.

The second request is `/license/details/license-id` which communicates with both *License-Service* with endpoint `/details/license-id`, and *Organization-Service* with endpoint `/details/org-id` to return the details of a *License Model* and its attached *Organization Model* information as well. Moreover, Table 2 shows the detailed attributes required of the models per each client; Such that `/license/list` returns different attributes from one client to the other; however, the `/license/details/license-id` returns the identical information for both clients.

Table 2: Testbench Data Models.

Models	Client type	Endpoint	Required Fields
License Model	Database	—	id, title, shortDescription, longDescription, date, url, licenseType, orgID
	Mobile	<code>/list</code>	id, title, shortDescription , date, licenseType, orgID
		<code>/details</code>	id, title, longDescription, date, url, licenseType, orgID
	Web	<code>/list</code>	id, title, longDescription , date, licenseType, orgID
		<code>/details</code>	id, title, longDescription, date, url, licenseType, orgID
	Organization Model	Database	—
Mobile		<code>/details</code>	name, information
Web		<code>/details</code>	name, information

6.2 Procedure

This procedure initiates the launch of two system instances, one utilizing the FMC pattern and the other containing the BFF pattern. To ensure accurate measurements and eliminate external influences, certain restrictions are implemented. The same data items are utilized for both patterns to prevent variation due to different response sizes. An in-memory database is integrated to eliminate the processing overhead from

varying database engines. Additionally, a local connection is employed to avoid communication overhead from the internet and external servers. A local connection refers to a direct connection between two devices within the same network, as opposed to a remote connection over the internet.

The experiment sends the two frontend requests (see Table 1) using both client types configurations. We repeat the request calls five times to eliminate the random chances. Each time we automatically measure the response time and the response size using the Postman⁴ client application. These executions are very close in their time and size measurements. Thus, we calculated and illustrated the average among these executions.

6.3 Results

This section analyzes and concludes the consumed data size and time over the two patterns. Regarding these performance aspects, the analysis shows the impact of the BFF pattern on the system. This is, however, a small system, and it uses monolithic frontend clients. The results conclude the FMC pattern performance impact compared with the BFF one.

The results show that the FMC pattern consumes more data than the BFF pattern for both clients; however, as shown in Figure 5, this difference is relatively small. This data increases because the FMC pattern injects a few keys in the partial responses. These keys help in binding the responses with their related request in the frontend. Moreover, observing the size differences through multiple trials highlights the constant size change per service. On the other hand, the BFF pattern introduces an additional perspective on data consumption. It receives all service attributes, then it extracts the required ones for the client. Figure 6 depicts the percentage of sent data to the client after filtering out the unnecessary attributes. Therefore, that extra overhead is obvious in `/list (mobile)` that consumes much amount of data than needed for the mobile client. For example, the BFF endpoint receives 200KB size, however, the filtered data size is only 5KB to be sent to the mobile client. This is the impact of only one attribute (`longDescription`), which is unnecessary for the mobile client. Overall, the BFF consumes much data from two perspectives. However, the FMC pattern encourages the microservices business layer to send the needed attributes only corresponding to the invoking client type.

⁴Postman Application: <https://www.postman.com>, accessed on 02/13/2023.

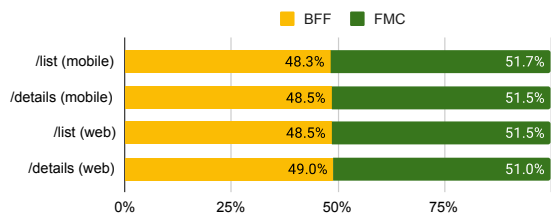


Figure 5: Response Size Percentage Comparison (Client-side perspective).

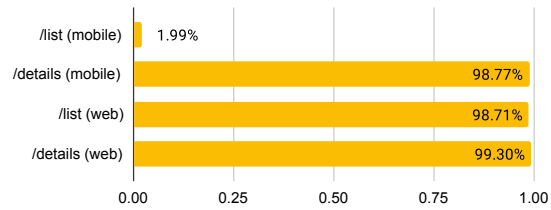


Figure 6: BFF Filtered Data Percentage.

From the response time perspective, we highlight the time consumed per each individual request data until reaches the frontend client. The behavior is relatively similar for web and mobile clients. The differences between the two patterns are illustrated in Figure 7, which shows that the FMC pattern stands out from the BFF. The BFF consumes extra time for filtering the response data, besides the extra communications for the requests between the BFF and the microservices business layer. This impact is shown in the time of `/list` and `/org/details` endpoints. Moreover, `/license/details` highlights the drawback of the BFF pattern, such that the frontends wait for all inter-services requests to respond so that the BFF layer sends the filtered data as a single response.

In contrast, the FMC pattern sends the response gradually to the frontend in separate parts. Diving deeper into this case, the first scenario is both `/license/details/license-id` and `/org/details/org-id` are independent endpoints, such that the client has both parameters `license-id` and `org-id`. Thus, both requests are parallel executed. In the BFF, the frontend receives both the license and organization data models together in 10 seconds, which is the longest time required for `/org/details/org-id` request. However, in the FMC, it receives the license data model in 5 seconds, while the organization data model is received after 10 seconds in total. The second scenario is `/org/details/org-id` depends on `/license/details/license-id` for finding the required `org-id` parameter. Therefore, both BFF and FMC can not parallel these requests. In the BFF, the frontend receives both the license and organization data models together in 15 seconds, which is the to-

tal response times for both requests. However, in the FMC, it receives the license data model in 5 seconds, while the organization data model is received after 15 seconds in total. To sum up, the BFF has a slower response time than the FMC, regardless of request dependencies. This difference may be further amplified when the system is deployed on remote servers, due to extra communication overhead and remote server calls, leading to increased response time.

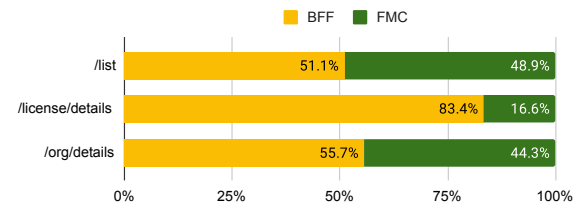


Figure 7: Response Time Percentage Comparison.

7 DISCUSSION

The data analysis emphasizes the positive impact of the FMC pattern, this is, however, a primitive experiment. This impact is obvious, especially for request consumption time. It also impacts the consumed data size compared to the BFF pattern consumption. However, both patterns send almost the same amount of data to clients. In summary, the BFF added an extra layer of communication that increases the time and data size overhead throughout requests. Diving deeper into investigating the other two impact factors of this pattern. Those are the *development lifecycle* and the *business inconsistency*.

The BFF provides as many separate implementations as the number of clients and the number of micro-frontends as well. In the experiment above, although the identical response for `license/details` as shown in Table 2, the BFF provides two separate implementations for the clients. This adds obvious overhead to the development process and its teams. Furthermore, the FMC pattern reduces the coupling between the different development teams.

For example, the following scenario illustrates the coupling between the different layers and, thus, between the different teams. The frontend clients require additional attributes to be added to a response payload for one of the requests. The BFF pattern requires a change to all BFF components. Therefore, the BFF-related source code repository is changed, and then it goes through all integration and deployment pipelines until it's released to the frontend usage. This is beside the required changes for the responsible business microservice in case it has not returned those attributes already.

On the other hand, the FMC pattern requires only the change for the business microservice, which provides the required attributes for the corresponding client types. Although that adds extra responsibility on the microservices and their developers, it does not impact the microservices' cohesion property. The FMC pattern embeds a set of adapters for customizing responses. Therefore, it encourages the embedded adapters to follow the wish list pattern. Thus, it is only the wish list parameters that would be changed for customizing responses to clients.

Regarding the *business inconsistency* perspective, assume a business change has been requested, such that it should impact the business microservices and the data format for all frontend clients. Therefore, inconsistency happens when the developer forgets to change any of the BFF-related components for a specific client. The business would be inconsistent for that missed client. On the other hand, the FMC pattern emphasizes that the business is concentrated in the business microservices layer. There are no modifications to the original response until it reaches the frontend that is responsible for combining and rendering it. Thus, this pattern prevents the risks of these business inconsistencies.

In conclusion, the proposed FMC pattern shows multi-aspect enhancements compared with the common BFF pattern. Moreover, these results encourage a potential fit for the FMC pattern with the evolution of micro-frontend architecture and its integration with microservices distributed architecture.

8 CONCLUSION

This paper is motivated by current gaps for frontend integration in microservice systems. These gaps lead to challenges in the development process and business inconsistency over system modifications. In comparison with the BFF, the proposed FMC pattern offers better-fitted integration with the microservice development process and reduced team coupling. It also leads to reduce the overall data and time consumption. Although FMC presents liabilities for applying the pattern, they are scattered through the cloud-native components. In contrast to the BFF, the FMC avoids adding extra layers to the architecture.

Our future work includes applying FMC to a large benchmark, converting a BFF-based system to FMC for performance comparison, and conducting a user study with microservice experts to evaluate FMC's development process enhancement.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research, <https://research.redhat.com>.

REFERENCES

- Brown, K. and Woolf, B. (2016). Implementation patterns for microservices architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*, pages 1–35.
- Carnell, J. and Sánchez, I. (2021). *Spring Microservices in Action, Second Edition*. Manning.
- Cerny, T., Donahoo, M. J., and Trnka, M. (2018). Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45.
- Fowler, M. (2006). Consumer-driven contracts: A service evolution pattern.
- Fowler, M. (2019). Micro frontends.
- Harms, H., Rogowski, C., and Lo Iacono, L. (2017). Guidelines for adopting frontend architectures and patterns in microservices-based systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 902–907.
- Márquez, G. and Astudillo, H. (2018). Actual use of architectural patterns in microservices-based open source projects. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 31–40. Ieee.
- Newman, S. (2015). Backends for frontends.
- Osses, F., Márquez, G., and Astudillo, H. (2018). An exploratory study of academic architectural tactics and patterns in microservices: A systematic literature review. *Avances en Ingeniería de Software a Nivel Iberoamericano, CibSE*, 2018:71–84.
- Peltonen, S., Mezzalana, L., and Taibi, D. (2021). Motivations, benefits, and issues for adopting micro-frontends: a multivocal literature review. *Information and Software Technology*, 136:106571.
- Stocker, M., Zimmermann, O., Zdun, U., Lübke, D., and Pautasso, C. (2018). Interface quality patterns: Communicating and improving the quality of microservices apis. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–16.
- Valdivia, J. A., Limón, X., and Cortes-Verdin, K. (2019). Quality attributes in patterns related to microservice architecture: a systematic literature review. In *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 181–190. IEEE.
- Wellhausen, T. and Fießer, A. (2011). How to write a pattern? a rough guide for first-time pattern authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*, pages 1–9.