UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Berker Demirer

Scaling Up a Frontend Monolith: Pipedrive Case Study

Master's Thesis (30 ECTS)

Supervisor(s):
Kadir Aktaş (MSc.)
Prof. Gholamreza Anbarjafari
Prof. Satish Srirama

Tartu 2020

Scaling Up a Frontend Monolith: Pipedrive Case Study

Abstract:

There have been considerable efforts on scaling up the backend monoliths and creating microservices. Companies that run on monolithic services have started to experience the negative impact of slower deployment cycles and service shortages as their customer base, and team expands. To mitigate the problem, some of the companies have switched to a microservice architecture, which provides better fault tolerance, faster deployment cycles and better software development processes. However, most of these developments were focused on splitting the backend part of the applications while leaving the frontend to stay as a monolith. Complex business logic included in the frontend monolith introduced the bottleneck for agile development and raised the necessity of having micro-frontends. This thesis focuses on the Pipedrive's transition from frontend monolith to frontend microservices by evaluating this process on architectural and implementational level.

Keywords:
Agile, scaling, monolith, microservices, micro-frontends

CERCS:
P175 Informatics, systems theory

Frontend monoliidi skaleerimine: Pipedrive näitel

Lühikokkuvõte:

Monoliit-aplikatsioonide skaleerimiseks ja selle lõhkumiseks mikroteenusteks on tehtud palju jõupingutusi. Firmad kes pakuvad teenuseid, mis jooksevad monoliitsel arhitektuuril, kogevad tihtipeale aeglasemaid arendustsükleid ja teenuse ebastabiilsust kui nende kasutajaskond ja arendusmeeskond suureneb. Et probleemi lahendada, on paljud firmanud pööranud pilgu just mikroteenuste poole, mis pakuvad üldiselt suuremat stabiilsust, kiiremaid arendustsükleid ja võimaldavad paremaid tarkvaraarenduse protsesse. Kuid palju neist arengutest on pühendatud just backend-poole skaleerimisele ja *frontend* jäetakse tihti monoliitseks. Pipedrive keeruline äriloogika frontend-monoliidis osutus pudelikaelaks, mis hakkas hälvama agiilseid arendusmetoodikaid ja lahenduseks on frontendi mikroteenused. See lõputöö keskendubki Pipedrive üleminekule *frontend* monoliidist mikroteenustele, hinnates protsessi arhitektuurlsel ja rakenduslikul tasandil.

Võtmesõnad:
Agiilne, skaleerimine, monoliit, mikroteenused, mikro-frontendid

Table of Contents

Table of Figures

List of acronyms

CRM	Customer Relationship Management
API	Application Programming Interface
UI	User Interface
SPA	Single Page Application
CDN	Cloud Delivery Network
CI/CD	Continuous Integration and Continuous
DDD	Domain Driven Development
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheet
BEM	Block Element Modifier
SCSS	Sassy CSS
DOM	Document Object Model
RPC	Remote Procedure Call
XSS	Cross Site Scripting
NCP	Nginx consul proxy

1 Introduction

Microservice architecture or microservices is a particular method of designing software applications that aims on building single-function modules with well-defined interfaces and components. They are modeled around a business domain as suites of independently deployable services. The common characteristics of microservices include: independent and automated deployability, easy to develop, and decentralized control of programming languages.

In recent years, microservices have enabled companies and organizations to develop scalable and expandable applications. Many organizations use microservices to avoid running into the limitations of monolithic frontend or backend. This architecture style enables them to use structures that are necessary for developing any complex and modern web applications.

When implementing a microservice architecture, it is important to apply the architecture on both backend and frontend. A common solution is to split the frontend application into micro-frontends to avoid having one big monolithic frontend that cannot be decomposed. Particularly, there have been emerging patterns e.g. "micro-frontends" that decompose a frontend monolith into independent modules that are independently developed and deployed, while still serving as a single cohesive product.

1.1 Micro-Frontends

A micro-frontend architecture is a design architecture in which a frontend application is decomposed into individual, semi-independent micro applications that are loosely coupled [1]. The term "micro-frontend" is inspired and named after microservices. In the simplest term, micro-frontends is the concept of implementing microservices to frontend applications. The need for micro-frontends derives from the current trend of development, which is to build an application with single page frontend microservices architecture. This approach does not scale for frontend as the application gets bigger and becomes hard to maintain. That is what we call a frontend monolith. The idea behind micro-frontends is to create different teams and different deployment pipelines for set of features that are used to form a web application [2].

1.2 Monolithic Frontend

For backend teams to deliver business value – especially in the CRM business - there needs to be at least one frontend application for user interaction. In general, an API without a user interface might not bring a competitive advantage over applications with UI [3]. If an organization embraces the microservices pattern in the backend, it can be assumed that there are multiple backend teams, business domains, faster deployments, and feature implementations [4]. However, if the frontend is maintained as a monolith in the same organization, it puts pressure on the frontend application and the team who maintains it. One solution to ease the workload would be to increase the size of the frontend team or create multiple teams maintaining the application, but this is also not scalable because the frontend needs to be deployed in one go. That means the teams are dependent on each other; with a monolithic frontend it is not possible to get the flexibility to scale across the teams as promised by microservices. Moreover, regarding the scalability issues, there is also the overhead of having separate backend and frontend teams. Whenever a service's API has to be changed, the frontend must be updated, which means that both teams must work dependent on each other. If frontend teams are busy with some other implementation, then the backend changes must wait for the frontend team. Having a monolithic frontend means that the team or the teams are required to work with the same technology stack no matter how different their tasks are. They cannot introduce a new tech overhead to monolith because other teams will be affected by this change as well, resulting in a non-modernized frontend application. Teams should be independent with their technology stack to be able to deliver their tasks more effectively and efficiently. Finally, fault tolerance is another challenge with monolithic frontends. If there is a bug in one component that affects the production build of the application and prevents users from accessing the app, this means other unaffected working components will be inaccessible as well. Therefore, all the teams and customers must wait for this bug to be fixed and deployed to be able to work again.

Pipedrive recognized the scalability challenges of maintaining and implementing features into PHP-app and put an effort into missions to cut off the legacy systems from the monolith and create microservices for each.

1.3 Scope & Objectives

The objectives of this thesis are to provide information regarding micro-frontends and how Pipedrive uses them in its software development ecosystem. Then evaluating micro-frontends architecture's impact on Pipedrive's development processes. By doing so, this paper could be used as a guide for those who would like to implement micro-frontends in the enterprise software ecosystem. Thesis's scope is limited to high-level design and implementation of the micro-frontends architecture. This thesis will explain the components that form the micro-frontends architecture in Pipedrive and then evaluate its impact.

1.4 Thesis Structure

This thesis is composed of three main chapters and supplementary chapters. The first chapter gives an overview of questions to be answered in this paper. The second chapter gives a background information for understanding micro-frontends and microservices in general. More on that, this chapter also provides examples from the software industry to create better understanding how other companies implement micro-frontends to their ecosystems and what kind of patterns they use. The third chapter explains the Pipedrive's transition from frontend monolith to micro-frontends by examining the overall architecture and implementations. The fourth chapter analyses a micro-frontend and a frontend monolith from different operational aspects such as latency, deployment stability and resource usage. Fifth chapter provides insights for author's contribution to the topic. Sixth chapter summarizes the improvement areas and future works. Finally, the conclusion part summarizes the main points. Supplementary chapters include a list of acronyms, conclusion, references and appendix.

2 Background

The idea of micro-frontends is not new. It has many similarities with the self-contained systems concept. This concept is an architectural pattern that focuses on a separation of functionalities into many independent systems, making the complete logical system by composing smaller software systems [6]. This eliminates the challenges of large monoliths that grow fast and eventually become unmaintainable. When looked at from a more general perspective, micro-frontends concepts are not any different than the microservices architecture in terms of core principles they hold. The concept is getting popular among big companies like Netflix, Pipedrive, Spotify and SoundCloud. Some of the companies created their own frontend microservice frameworks and made them open source, such as Tailor.js from Zalando [7].

2.1 Micro-Frontend Principles

This section explains the core principles of micro-frontend implementations. Considering the definition of micro-frontend, the following principles are the foundation of it.

2.1.1 Modeled Around Business Domains

While scaling the frontend monolith the first step should be to identify the micro-frontends in it. To ease up this process, developers can use the domain-driven design principles to identify the bounded context and business domains. Domain-driven design suggests that each piece of software should be a representation of an organization's architecture, therefore, giving developers initiatives to design the software architecture based on domains and subdomains shaped by domain-driven design. This approach is also useful for frontend microservices and create end-to-end teams for different subdomains of the application. This can be modeled by looking at the user’s behaviors [8].

2.1.2 Automated CI/CD

If an organization uses microservices architecture, there must be a rich automation culture; otherwise any kind of micro-frontends implementation approach will be hard to implement and maintain in the long term. Since a micro-frontends project would contain many different parts in it, there must be reliable continuous integration and continuous deployment pipelines with reliable fast feedback loops. Therefore, having an automation culture set and ready for usage is crucial for adopting micro-frontends [8].

2.1.3 Code Isolation & Team Prefixes

In theory, implementation details should not affect the way micro-frontends communicate among one another. It is vital to come up with a contract among teams and other parties during the entire development processes. In this way, independent teams will be able to change the implementations without disturbing other teams unless there is an API contract change. This will ensure that each team can work independently and without any external dependencies, hence increasing the effectiveness of the integration [9].

2.1.4 Decentralization

Decentralization is an essential principle of micro-frontends and it is one of the core ideas behind microservices architecture. The idea is to decentralize teams decision not to affect other teams way of work. This ensures that the application will move away from one-size-fits-all approach and becomes more flexible by being able to use the right approach and the tools for the job. It can also be said that this principle helps to create experts in each business domain and decisions made by teams for their domains are generally more accurate. However, this does not mean that there should not be any conventions around the application, some guardrails should be provided by the executives where the team can still work independently without waiting for a central decision [9].

2.1.5 Independent Deployment Pipelines

This principle can be related to the decentralization principle. As every team should be able to make decisions on their own without disturbing the other teams, they should also be able to deploy independent artifacts without waiting external dependencies to be resolved before deploying to production. When we combine decision and deployment decentralization, it can be observed that a team might be responsible for a business domain end-to-end and have the right to make technical decisions based on the challenges faced in this specific domain [9].

2.1.6 Resilience

In case of any micro-frontends failures the application should continue business operations. Micro-frontends bring more overhead to network operations and they need to be monitored correctly. In case of a failure or shortage, there should be alternative ways to avoid impacting user experience and limit the impact of the failure just for the specific service which failed [9].

2.1.7 Observable & Monitorable

Micro-frontends and microservices eliminate many problems for fast growing companies, but they come with a cost. More microservices mean more network overhead. While communicating or routing between micro-frontends some failures might occur. It is essential to be able to log and observe the errors to understand where and why our micro-frontends fail. Tools like Sentry, DataDog and Grafana increase developer's situational awareness of their microservices [9].

2.2 Micro-Frontend Services Decisions Framework

From a technical point of view there are two possible options for defining micro-frontends; one is horizontal split and the other one is vertical split. As depicted in Figure 1, horizontal split includes multiple micro-frontends per page while vertical split aims to load one micro-frontends per page [10]. When we think about horizontal split, teams should work closely and coordinate their work as the general layout of a single view is a composite work of different teams. It is good to note that these micro-frontends might be owned by the same team as well. It is dependent on the organisational structure. Whereas, in the vertical split scenario each team is responsible for a business domain and its implementation.

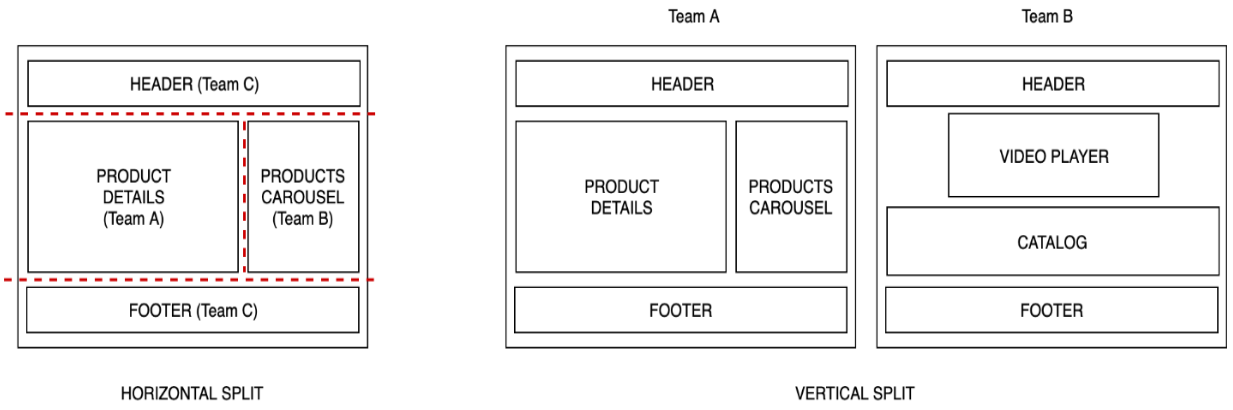


Figure 1. Horizontal vs Vertical Split **[2]**

As mentioned in the principles of micro-frontend services, business domains can be identified using DDD principles. When using the DDD approach, the first step would be to identify the parts of the application that represents a subdomain of the final application. It is possible to divide subdomains into three categories such as core, supporting and generic subdomains. Core subdomains define the focus of the application, for example, movie streaming for Netflix, music streaming for Spotify. Supporting subdomains are in relation to core subdomains to support them but they are not the key differentiators, they are not meant to deliver real value to users. Generic subdomains needed for completing the platform and companies usually prefer to outsource the applications required for these subdomains as they are not strictly related to core subdomains, it might be payment management and authentication [3].

2.3 Micro-Frontend Services Composition

There are different micro-frontends design patterns and approaches for creating micro-frontends applications which can be categorized as client-side composition, edge-side composition and server-side composition.

On the left side of Figure 2, we can see the client-side composition. In this composition method micro-frontends loaded by the application using CDN or from the origin if the micro-frontends artifact is not cached at the CDN level. In the middle, by using edge-side composition, the final view is composed at CDN level and the result is delivered to the client. In the last diagram, micro-frontends composed at the origin level are converted into views, cached at CDN and served to clients.

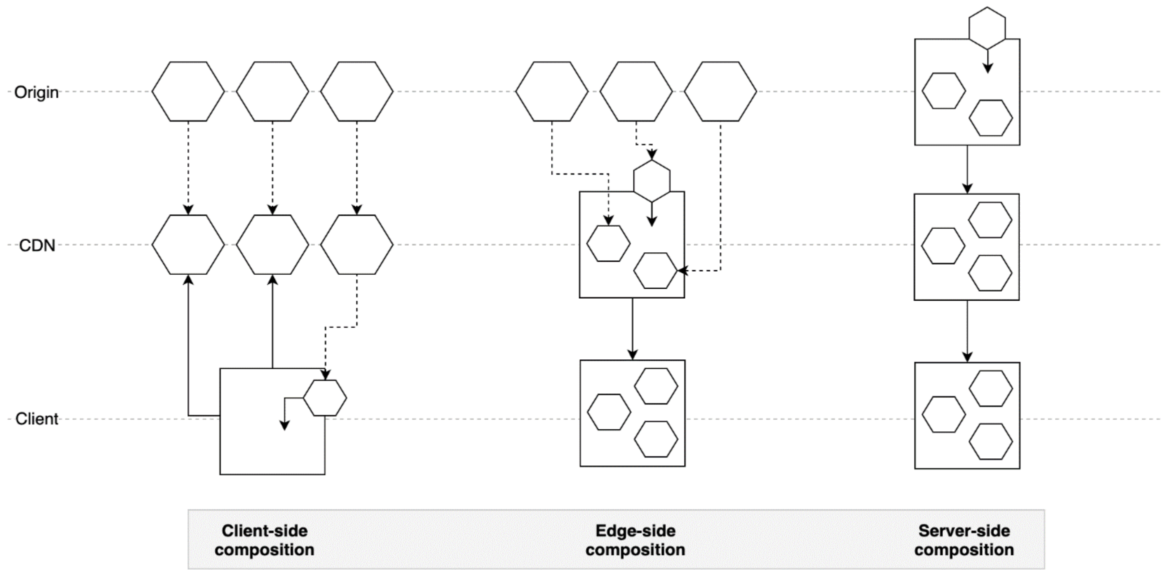


Figure 2. Micro-frontends composition diagram [2]

2.3.1 Client-Side Composition

Client-side composition can be done in several ways. First approach can be run-time integration with iframes. The HTML <iframe> element is used to embed an HTML document inside another HTML document. Iframes are useful when we need to create a standalone hosting environment and run our frontend applications independently of each other [12]. With iframes, it is possible to isolate different sub-applications from each other and render them on demand. The downside of this approach is that it creates difficulties while building integrations with other sub-applications, which means that routing, history, and deep linking become more complicated [13].

The main idea behind build-time integration of micro-frontends is publishing each micro-frontend as packages and let the container application import it in its package.json file. This approach produces a single JavaScript bundle for deployment which allows developers to eliminate redundant dependencies from various sub-applications [13]. However, there is one setback in applying this approach. If we want to introduce a change in one of the micro-frontends, we need to re-compile and release every other micro-frontend. Since the main principle of having a microservice is its ability to be developed and tested independently, it is not reasonable to introduce coupling between services during the release stage. Build-time integration is useful if you do not mind longer deployment processes and you require independent technologies and teams for specific parts of the application.

To eliminate the lockstep release process defined in the build-time integration approach, it is a good practice to integrate micro-frontends at run time. In this approach, micro-frontends are embedded on the page as a script file and exposes a global function as their entry-points. After the global functions are attached to the window object, the container application determines which service to be mounted and calls the relevant function of the service [13]. Using this approach, we can release and deploy each of the frontend microservice bundles independently.

To create a fully encapsulated and decentralized frontend architecture, it is possible to use the web-components. Most of the frontend frameworks provide a component architecture which allows us to create encapsulated, extendable, reusable and composable user interfaces. Components eliminate the need for a global logic within the application by decomposing the logic into reusable smaller pieces that can be composed in any order throughout the application depending on the needs. Using framework components is convenient within its specific ecosystem, however, it is not possible to use Angular components in a React application or the opposite [14]. If we want to have components that are framework independent, we must implement web-components that are genuinely built on web standards. This approach relies on defining custom HTML elements that can be instantiated by a container application. In contrast to JavaScript integration, this approach eliminates the need of defining global functions to be invoked by the container application [13].

2.3.2 Edge-Side Composition

In edge-side composition approach, the views are assembled at the CDN level. CDN providers like Akamai and Oracle allow developers to use XML-based markup language called Edge Side Include (ESI). ESI is used for edge level dynamic web content assembly and it aims to tackle web infrastructure scaling problems by making use of a large number of edge nodes around the world provided by a CDN network [16]. Assembly of the content can be easily scaled with this approach when compared to traditional data center approach. One of the drawbacks of this implementation is that different CDN providers implement ESI in a different way. Therefore, multi-CDN strategy might require developers to implement specific logic to each CDN [8].

2.3.3 Server-Side Composition

Server-side composition can happen at runtime or at compile time. In this type of composition, the origin server gathers up all the views from different micro-frontends and assembles the final page at the backend [15]. If the page is cacheable, then it should be served by the CDN with a long time-to-live policy to increase the performance and user experience. However, if caching is not an option and the server is getting many requests from clients then the scalability must be taken into consideration to avoid outages [8].

2.4 Routing Micro-Frontend Services

One of the important concepts with micro-frontends is how to manage routing between them. As Figure 3 depicts, routing varies for each micro-frontend architecture or composition such as origin, edge or client-side [4]. Following routing approaches are not mutually exclusive, one can combine the approaches depending on the routing needs of the application. When we are using server-side composition, we are forced to implement routing at the origin level if the page is not cached at CDN level and this approach would result in whole page to refresh since a new GET request is sent to server. Another point to be considered while using routing at origin approach is scalability. If the server is getting burst traffic with too many requests per second, server needs to scale up horizontally and keep up with the incoming traffic. Then each application server should compose the micro-frontend services needed to compose the page request by the client. CDN providers mitigate the scalability issues on the edge level by managing the load between different edge networks and its completely handled by CDN provider. When it is decided to use edge-side composition, the CDN serves the page by assembling the micro-frontends on the edge level. Therefore, all the routing happens on the CDN level and developers might not have much freedom to customize the routing.

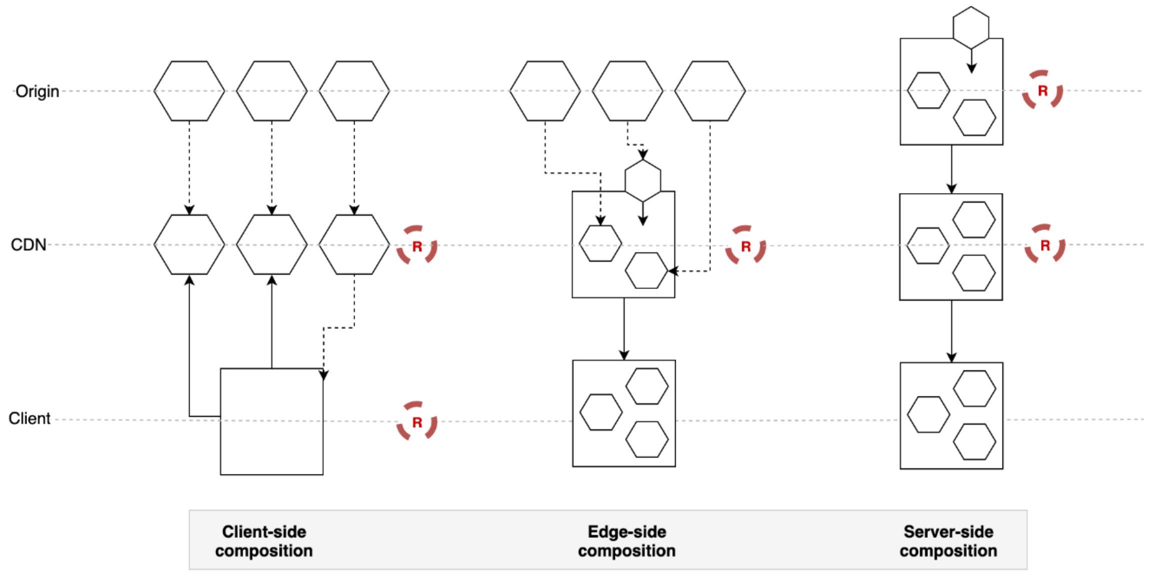


Figure 3. Micro-frontends routing diagram [5]

Another approach is to use client-side routing. In this scenario routing is done based on the public and private routes defined in the client application. This can be determined by the user roles or the states. If there is an application shell to load the micro-frontends as a single page application, the routing should be handled by the shell. At the initialization, the shell fetches the routing configuration and then decides which micro-frontends to load. On the contrary, if there is a multi-page application, micro-frontends might be loaded using client-side transclusion [8].

2.5 Communication Between Micro-Frontend Services

Based on the micro-frontend principles of decentralization and independent deployments, ideally, there should not be a need for communicating between frontends. All the micro-frontends should be self-sufficient, but reality is always different than the expected scenario and developers might need to communicate between micro-frontends. When there are more than one micro-frontends in the shell application, there is a challenge of creating consistent and coherent UI. This is also true while communicating between micro-frontends especially if they are owned by different teams. While considering the options for communications, it must be kept in mind that micro-frontends should hide implementations between each other, therefore, each micro-frontend should be unaware of each other on the same page.

After considering all the restrictions and the challenges, one solution to micro-frontends communication might be using events and observer/subscriber patterns. To be able to implement this, we use an event bus in each micro-frontend, let micro-frontend dispatch and listen to events depending on the needs [17].

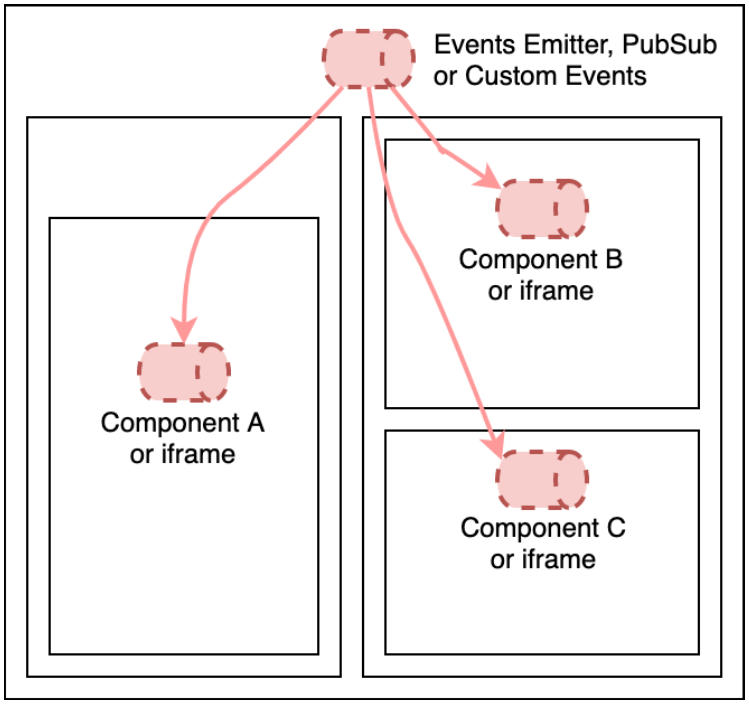


Figure 4. Event bus communication diagram [5]

Another solution is to use custom events. In this case, the events are dispatched via an object available to all micro-frontends like the window object. However, if micro-frontends are implemented using iframes then using event bus instead of sharing global window object to micro-frontends would be less challenging since each iframe has its own window object. Last but not least, one another option might be using the data query to pass data and retrieve the details to display using API. However, this approach comes with a downside of exposing sensitive data to intruders. Even though it is passed through HTTPS, it might be still possible to sniff the information, therefore, decision must be thought carefully [4].

2.6 Challenges with UI Coherency

No matter which approach is used to implement frontend microservice, visual consistency across the micro-frontends is very important. Styling is one of the first things that needs to be considered. Currently, most of the web applications are styled using cascading style sheets (CSS). CSS is a language that is inherently global and cascading, and it does not provide module system, encapsulation or name spacing. These characteristics might create problems in a micro-frontends architecture if independent teams do not agree on a standardized way of writing style classes. For example, if one team implements a micro-frontend that has a stylesheet header font bold and the other team has a different attribute for the same selector then one of the styles will override the other one. One of the solutions for this problem is to use the BEM notation to make sure that the selectors only apply where it is intended. However, this approach mostly relies on the developers. People who do not want to rely on developers only use the sassy cascading style sheet (SASS) preprocessor and use its selector nesting for names pacing purposes. Besides using the SASS preprocessor one can use CSS modules to apply styles programmatically or use shadow document object model (DOM) to isolate the style [13]. Another approach to maintain consistency between the micro-frontends is to introduce shared component libraries. Shared component libraries can reduce the number of duplicate codes inside the application by providing reusable components across the microservices. However, introducing all the reusable components in the early stages of development is not recommended. It is not possible to predict which components are going to be needed before the actual use. One practice is to let each team develop the components that they need while they are developing the application. Even though this sounds like this approach would create lots of duplicate code, allowing the patterns to emerge helps developers to see which component should be included in a shared library. The most suitable components for sharing are the ones that we call visual primitives, such as icons, buttons, labels. One can also implement components that have complex UI logic as shared components, such as dropdowns, tables, autocompleting, etc. However, these complex components should only include UI logic, not business logic. Having business logic inside the shared component libraries would create coupling with other services and it would make it hard to change the components. Ownership and governance of a shared component library is another thing to take into consideration. One of the best practices for handling shared component libraries is letting developers create shared components when it is needed if the quality is checked by some team who is responsible for maintaining the library [13].

2.7 Micro-Frontend Services Applications in the Industry

This section will give an overview of some companies that use micro-frontends architecture in their ecosystem. The first company to mention is Zalando, an e-commerce company. Zalando is currently building up an open source micro-frontends framework called Mosaic. Zalando aimed to improve scalability with smaller pieces, technology stack isolation and ease of deployments [18]. HelloFresh, the company that provides an online service for preparing and delivering ready-to-cook food boxes, implemented micro-frontends architecture by using server-side rendering. Their micro-frontends architecture includes Fragments, Particles and Tags. In simplest terms, they run their Fragment services under entry server locations and serve small SPAs [19]. AllegroTech, Polish e-retailer came up with a project called OpBox in 2016. The project allows non-technical people to combine UI components with data sources inside the web page. The Box focuses on sharing and reusing components [18].

Another approach to be mentioned is OpenTable's Open Components project which is used by Scanner and other large organizations. Open Components are using an approach in which a registry gathers available components, encapsulates data and UI layers to expose a HTML fragment which can be encapsulated in any other HTML template. This approach brings team independence, eliminates component redundancy by allowing developers to reuse components built by other teams and the option of rendering a component either on the server or on the client [8].

SAP is using iframes to build up their micro-frontends. As Zalando, SAP is also created a framework called Luigi framework which is used to create enterprise applications. Luigi framework supports popular JavaScript libraries like Angular, React and Vue [20]. For their desktop application, Spotify implemented an iframe approach by assembling multiple components into different iframes. Iframes communicating with low level implementation made with C++ the "bridge". In the early stages of the application, each SPA file of Spotify was composed of an HTML file, multiple CSS files, manifest.json and a JavaScript bundle file. All of these are loaded into an iframe to be assembled in the shell application. However, this approach abounded for the web version of Spotify because of its poor performance and they decided to go with SPA architecture like they have for TV applications [8]. Last but not least, live on-demand sports streaming platform DAZN is using an agent called bootstrap which is a combination of SPAs and components orchestrated by clients. They provide services and interfaces for smart TVs, consoles, set-top boxes and web. They rely on run time rendering of different SPAs when there is a change of business domain [8].

3 Pipedrive Case Study

Pipedrive is a cloud-based sales software with offices in Estonia, Latvia, USA, Portugal, UK, and Czech Republic. It has more than 600 employees and its CRM is used by 100,000 customers worldwide. Pipedrive aims to increase the efficiency and sales numbers of businesses through easy-to-use web and mobile CRM application [6]. To be able to compete with other CRM tools on the market, Pipedrive must be responsive to changes in an agile manner. From a technical perspective, this means that the changes in business requirements must be developed, deployed and maintained with ease. Moreover, there needs to be a scalable and flexible architecture to comply with growing codebase.

Pipedrive was initially built upon a single codebase which uses PHP, however, soon after the scalability issues, transition to microservices started. Pipedrive did not neglect the frontend application and scaled it in parallel with backend services. At first, alongside the PHP-app, Pipedrive built the first JavaScript frontend application that uses Backbone.js but this app is also treated as monolith in the early stages and later on used as front-end-root for initializing other frontend services. In the current state, PHP-app is mostly used as a presentation layer for legacy features that Pipedrive still offers to its customers.

This chapter describes Pipedrive's journey from monolithic PHP application to microservices and micro-frontends. Additionally, it explains how Pipedrive's micro-frontend architecture is engineered and what is the impact of this architecture on Pipedrive.

3.1 The Monolith (PHP-app)

Pipedrive's first application was built in 2010 using PHP as the primary programming language in one codebase. In the early stages of the company, having one codebase helped Pipedrive to grow since it is simple to test, develop, deploy, and scale horizontally [21]. The CodeIgniter framework was modified and used to provide solutions for logging, error handling, routing, and database communication purposes. Overall, it helped to keep the code clean and modular. However, as Pipedrive grew, pressure on the teams maintaining the monolith increased. As a result, developer headcount increased, different teams created for the various domain areas, and all teams started to work autonomously on their fields. However, if the application is continuously growing, scaling the monolith with team isolation based on domain areas is not feasible as testing, deployment, and resiliency of the app become problematic.

To mitigate the problems caused by the monolithic application, Pipedrive started to shift to microservices architecture. This shift helped Pipedrive to create more exact lines of business boundaries, team responsibilities, and enabled it to increase the efficiency and effectiveness of implementing new features. However, the monolith application did not disappear completely. It still serves as the core API for microservices to fetch information from Pipedrive DB, such as company data, user information, deals, contacts, etc. Moreover, Pipedrive offers a public API through a monolith application for creating integrations between Pipedrive and other tools such as Slack, Intercom, and PandaDoc.

A team is responsible for the maintenance and performance of the monolith – namely, the Core tribe – and they do not implement new features to the monolith. The aim is to get rid of it and work on a microservices architecture. To lay out the plans for the future of monolith, there is an initiative called the PHP guild, which consists of developers who deal with a monolithic application and plan out the roadmap for splitting remaining features from the monolith.

3.2 The First JavaScript Frontend Application at Pipedrive (Webapp)

Webapp, - the first JavaScript frontend application of Pipedrive – is one of the first cut-outs from monolith application. It is built on Backbone.js, and the reason behind its creation was to separate backend code from the frontend. Backbone.js is a frontend JavaScript framework that helps to abstract the data into models and DOM manipulations into views and binding them together through events. It is a cleaner approach when compared to tying the data to the DOM. If not paid attention, JavaScript applications might turn into tangled piles of callbacks and jQuery selectors to keep data in sync between the UI and JS logic [22]. With Backbone.js, data is represented as Models, which can have several operations such as create, validate, destroy, and save to the server. When a UI action triggers a change in one of the attributes of a model, the model initiates a "change" event, and all the views that are tied to the model's state respond accordingly and re-render themselves with the new information. The aim of Backbone.js is to keep business logic separated from the user interface of the application. When logic and UI loosely coupled with each other, introducing changes on the UI becomes easier.

Pipedrive improved the Webapp as business requirements change, and more resilient frontend architecture was needed. This need pushed Pipedrive to design and implement micro-frontend architecture. The views, along with their business logic, gradually cut-out from Webapp and served as micro-frontends. Meanwhile, the purpose and the philosophy of the Webapp changed from being the frontend application of Pipedrive to be a platform that all micro-frontend teams use to integrate their services with. In the current state of Webapp, it is used as a top-level initializer like a front-end-root application. It shares common components, serves, and provides information for micro-frontends. To maintain and make Webapp as a web services platform, the Core Front-End (COFE) team has been established.

3.3 Components of Micro-Frontend Services Architecture

The frontend code of Pipedrive is separated into multiple repositories. Most repositories function simply by providing static asset files that could be loaded and executed at will. Some repositories also initialize an application by providing endpoint which responds with HTML . Frontend monolith so called Webapp is one example to that. Pipedrive's micro-frontends architecture requires several mechanisms to work together. These mechanisms include service discovery, assets services and assets router, frontend components to render and ConventionUI for design consistency. This chapter explains these components in detail.

3.3.1 Service Discovery & Diplomat

As a result of embracing micro-frontends architecture, service discovery becomes a crucial part of the application. In other words, service discovery is a process of acquiring the endpoint of a running and healthy service instance [22].

As there can be more than one healthy and running service instance, service discovery is also closely related to how to do load balancing. All services and service consumers need to use the Diplomat library to carry out service discovery for both internal and external services. Diplomat is an inhouse library that is used by service developers in order to communicate to Consul (central services and configuration registry). Diplomat provides handy functions for service registration, discovery, load balancing and configuration management.

For inhouse service-to-service calls, Pipedrive uses a client-side discovery pattern. As depicted in Figure 5, in the client-side discovery pattern, the clients gather the service location by asking a service registry that holds the location of all services [23]. From the end-user perspective, who has loaded Pipedrive web application into the specific browser and is sending requests to Pipedrive backend, server-side discovery pattern is used. As shown in Figure 6, in server-side discovery pattern, client's request goes through load balancer (router) then the load balancer queries the service registry and forwards to available service instances [23]. In the latter case, the web application that sends the requests to the backend does not have any information on which services are going to fulfill these requests. The service discovery (URL-to-service-endpoint translation) is made by Barista (the API gateway).

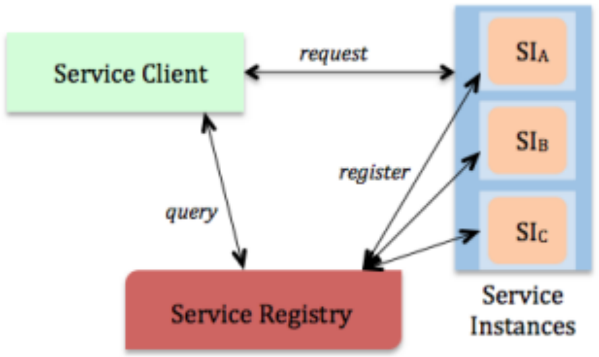


Figure 5. Client-side discovery pattern [7]

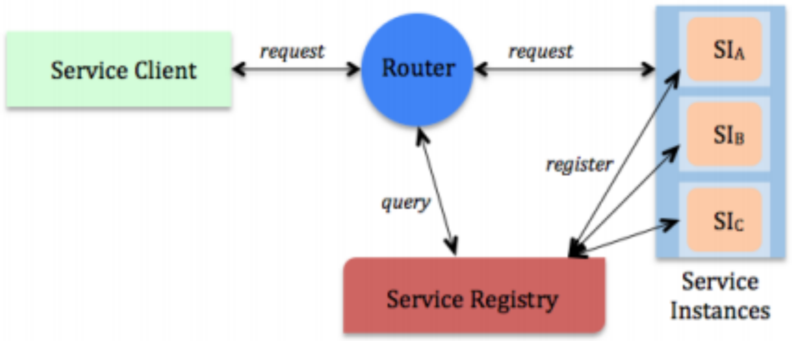


Figure 6. Server-side discovery pattern [7]

Let us presume, there is a service in “machine 1” that wants to call a target service, that runs on “machine 2” in Figure 7. In order to complete the service call, one needs to obtain the reference to the instance of the target service. One may ask Diplomat for the list of all the instances or just let Diplomat randomly pick up one instance. In most cases, upon the request, Diplomat returns the list of service instances from the local cache (Diplomat caches all the service references for 10 seconds). If it does not have them in the cache, it asks the local consul agent, which executes the RPC call against the central Consul. Consul is a service mesh solution providing a full featured control plane with service discovery, configuration, and segmentation functionality [23]. Local consul agents are practically stateless, which means it does very little caching or no caching at all. So, sending a query against the local consul agent is expensive because it almost always triggers a network call to the consul server agent. That is why caching is introduced inside the Diplomat.

Usually, calling the function for getting service instances from Diplomat is fast as it hits the cache; if more than half of cache expiry time (5 sec) has bypassed, it will return stale data, but executes an asynchronous call that refreshes the data in the cache. Meaning that the first service discovery call might be a bit expensive, but under average load, it is swift.

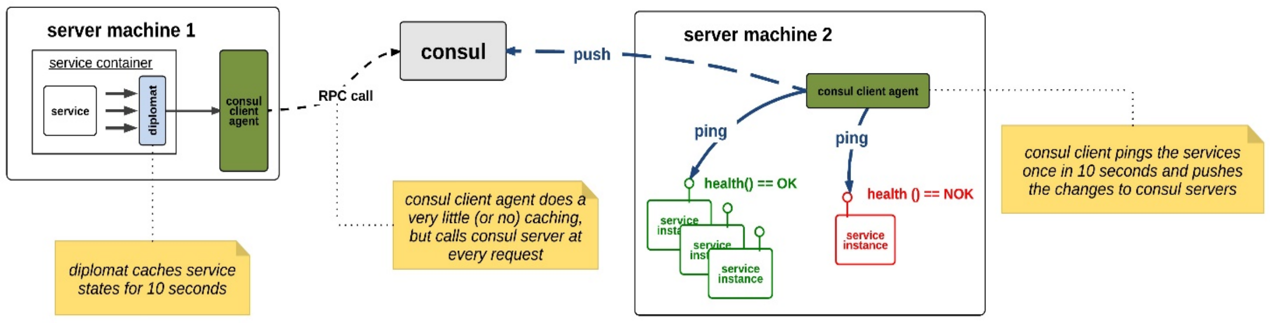

server machine 1
service container
service
diplomat
consul client agent
RPC call
consul
push
server machine 2
consul client agent
ping
ping
health() == OK
health () == NOK
service instance
service instance
service instance
service instance
diplomat caches service states for 10 seconds
consul client agent does a very little (or no) caching, but calls consul server at every request
consul client pings the services once in 10 seconds and pushes the changes to consul servers

Figure 7. Diplomat service discovery

3.3.2 Jura (Assets Router Service)

In the very beginning of Pipedrive's microservices architecture, both assets and service loading requests were going through one API gateway called Barista. Barista is Pipedrive's web traffic and API gateway that implements service discovery, authentication, authorization, rate limiting, multi data center routing, and monitoring of the incoming service calls. Most of the inbound traffic goes through Barista. However, Barista was not scalable enough for both assets and all other service domains as there were so many requests coming through. That is why Pipedrive decided to separate them. The reasons behind this decision were mostly related to reducing Barista workload, mitigating security risks, reducing requests size for assets, and allowing the use of CDN.

Barista was dealing with complex authentication issues and putting too much workload on it created problems. By having the separation of asset service domains, one can route assets' requests directly to the service and bypass Barista. Moreover, security risks are reduced by not exposing unwanted endpoints to unauthorized requests or processing additional XSS vectors that may come with assets requests. With different domains, cookies set on pipedrive.com will not be sent to pipedriveassets.com. This means that Pipedrive saves up to 8kb of data transfer with each asset request at a minimum. It may sound insignificant, but with all the applications to fetch assets, they add up to a tremendous amount of overhead. Mostly, without the separate domain, the request for a 1kb file would be larger than the returned data by 800%.

As stated above, using an assets router service allowed Pipedrive to implement CDN. A Content Delivery Network, or Content Distribution Network, is a geographically distributed network of proxy servers and their data centers [24]. CDN is a GeoIP based proxy network that finds the closest proxy server to the end-user and proxy the requests to Pipedrive. When assets already exist in the proxy cache, the proxy returns the data immediately and does not forward the request. When an asset is missing, it fetches the asset from the origin which is Pipedrive’s asset container and caches it based on the caching headers set by the asset service.

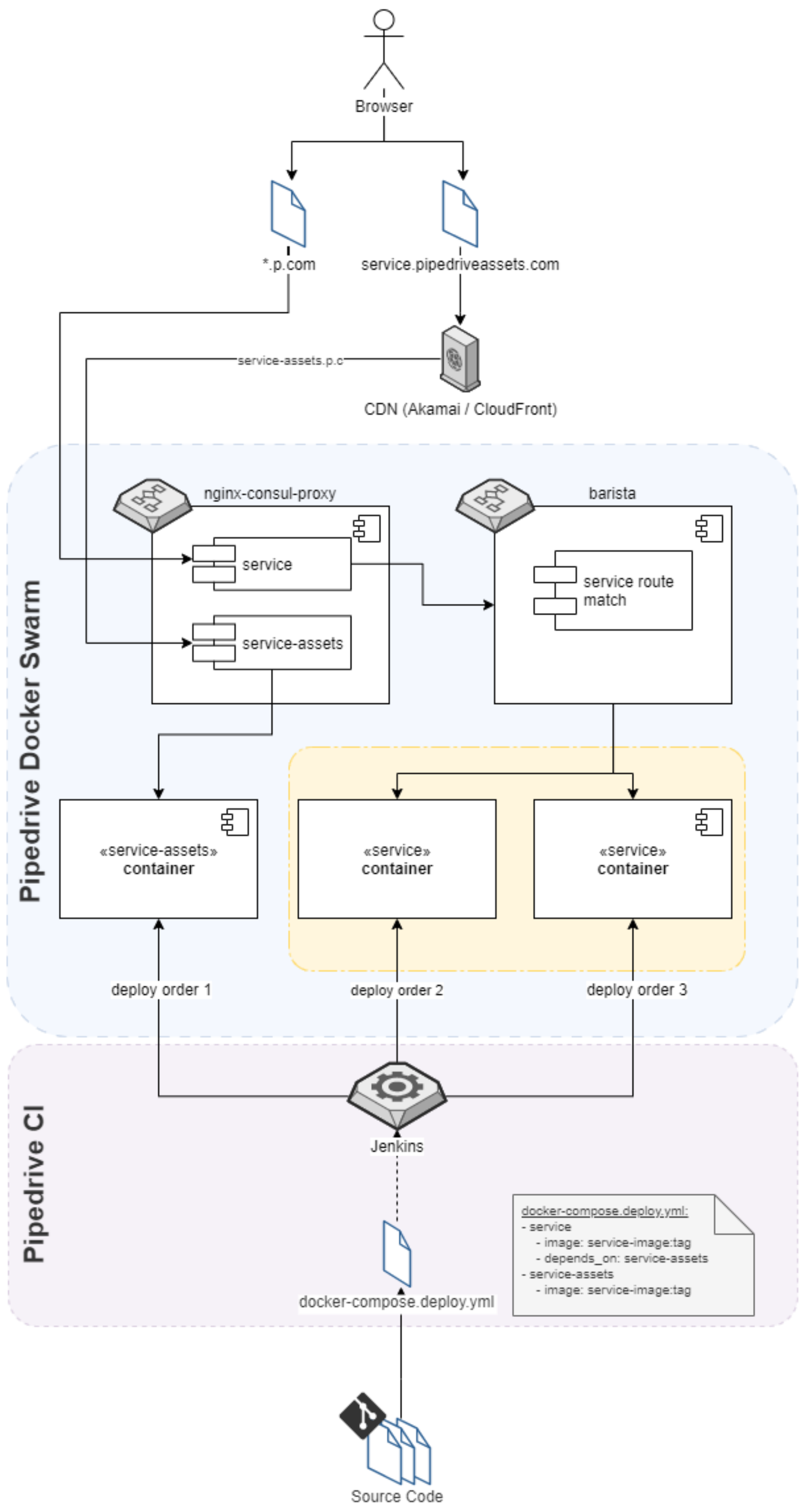
It is also worth mentioning that the CDN caches the full request and the full response. As a CDN provider, Akamai CDN is being used at Pipedrive, it has the most significant amount of edge nodes available around the world, so using this service made Pipedrive’s site and services much faster.

The first step to setting up the service with assets is to split the service into two, which means having a service container and asset container in the same docker-compose file. If the service is configured properly, one container serves the assets and the other the API. While it is possible to separate repositories and Docker images for the two container types such as assets and API container, it is easier to have just one repository that runs on different modes based on the environment variables (serving the assets or the API). In order to standardize the service names, a standardized convention is established. According to convention, service domain names created as *<service>.pipedrive.com* OR *<company-domain>.pipedrive.com/your/barista/url* and assets domain as *<service>.pipedrive assets.com.*

The main idea of having assets services and CDN is to make all the new assets available before the requests are sent to the containers. This way, if an asset is missing from the CDN, it will turn to assets-origin and fetch the new file. CDN itself should retain old caches for about a week, so if there are mixed requests to old and new versions of the app, it does not cache old content to new files accidentally. To differentiate versions, a specific versioning system is used, different build will have different value for version parameter in the GET request and they are cached separately.

The Figure 8, describes the simplified request types from the browser (top-down) and the source code to deploy mechanism (bottom-up). Both flows meet in the middle, in the running containers.

From the end-user perspective, it is possible to request two different domains, such as service and assets. If the user makes a service request, the request goes through nginx-consul-proxy, which is a set of NGINX server configurations that dynamically forwards specific domains to their service cluster, goes to Barista if applicable, and then to the service container. On the other hand, if the request is an asset request, it goes to CDN to nginx-consul-proxy and, finally, service assets container. In this case, Barista should not be used and must be bypassed.



```
docker-compose.deploy.yml:
- service
   - image: service-image:tag
   - depends_on: service-assets
- service-assets
   - image: service-image:tag
```

Figure 8. Services with assets and CDN

From the deployment and development perspective, the process starts with creating and configuring a docker-compose.deploy.yml file, which will be used for starting the containers.

Inside the docker-compose.deploy.yml, there should be two services. The first one is the standard service with service-specific scaling configuration and the second one is the service for assets. Service assets can be deployed using the same image as the service itself. But the name must be different in Consul, and this can be described with environment variables for the second service.

CDN optimized asset delivery systems works well, but the set-up complexity makes it harder for developers to maintain the overall architecture. With Jura improvement, overall architecture stayed the same, but the set-up for services themselves changed a little bit. On the server-side, the change allowed the service to register its assets' path via the Diplomat library. On the public side, Jura the asset router configured to make two lookups from Consul. First, it compares request paths' first segment to match a healthy service, then retrieves the configured assets path for that service and then assembles forward proxy URI. By addressing the complexity problem, Jura removed many obstacles in the agile development flow:

- Increased overall project delivery speed as there needs to be no involvement from other teams to set-up and configure anything.
- Reduced the workload for the infrastructure team as there is no need to re-configure Akamai, DNS, or NCP for each asset's domain.
- Streamlined overall architecture of the asset's domains' setup by providing single domain for all the services and automatic mapping paths to the services.

The Jura workflow shown in Figure 9 consists of the following steps:

1. A request is made to CDN.
2. If the requested assets are not cached the request is done to service otherwise CDN makes a request to Jura that looks like this:

   https://cdn-assets.useast1.pipedrive.com/filters-menu/app.js?v=86d7ec6fca_46

   The request consists of the following parts:

   - cdn-assets: Jura subdomain
   - .us-east-1: region

- .pipedrive.com: Pipedrive domain
- /filters-menu: service name which is used to search the service instance in Consul
- /app.js: file name
- ?v=86d7ec6fca_46: asset version

3. Jura asks Consul for a healthy asset's instance based on the request path's first segment (/filters-menu).

4. In case a healthy service instance is found, Jura assembles the forward proxy URL *http://${service.ip}:${service.port}/${originPath}/${assetPath}*.

   - *originPath*: The path to the assets in the service.
   - registerServices: This is necessary because in case a service has both API and assets, by moving the assets to a separate folder and providing the path to Jura, the API is never exposed.
   - *assetPath*: The path of the assets in the service found from parsing the request URL.

5. Requested service responds with assets file which is returned to Jura.

6. Jura cleans the headers and adds CORS and cache headers to the file and forwards it to NCP. NCP forwards it to CDN and CDN forwards it to the browser and the file is served.

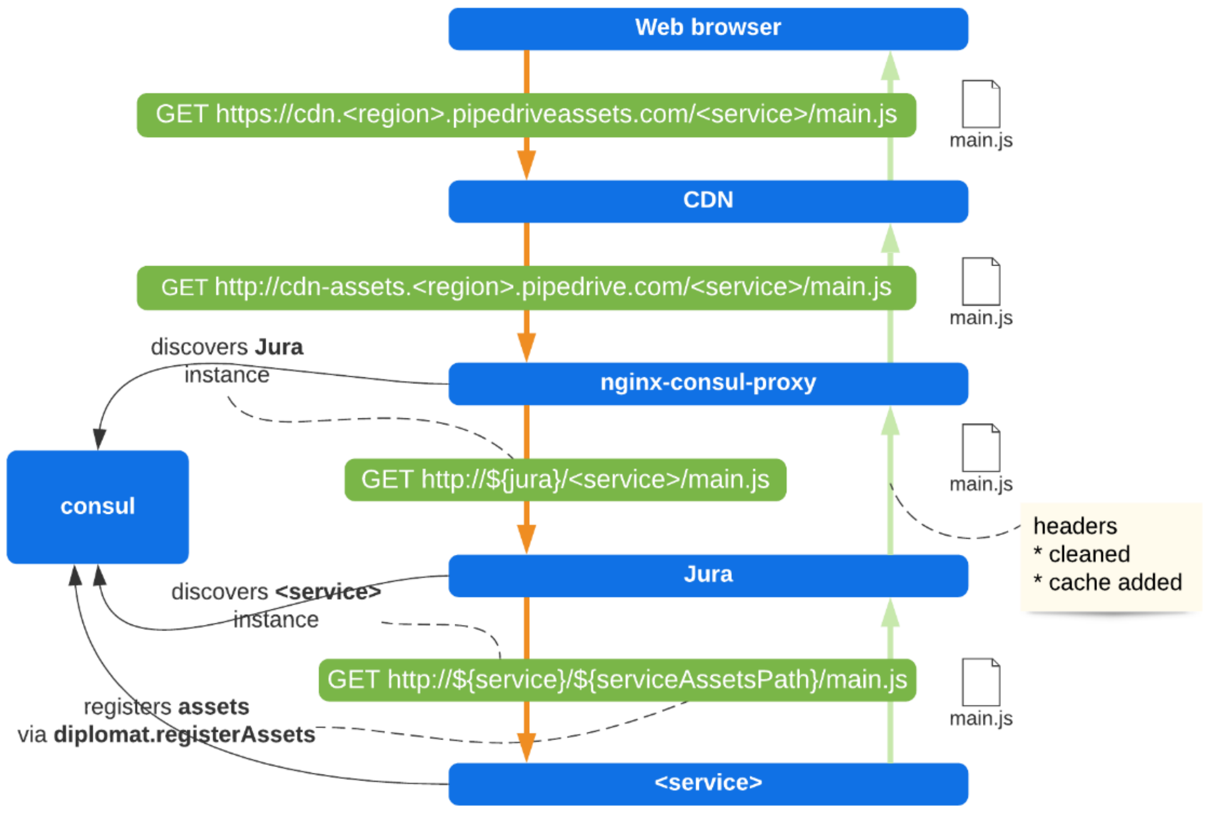

Web browser
GET https://cdn.<region>.pipedriveassets.com/<service>/main.js
main.js
CDN
GET http://cdn-assets.<region>.pipedrive.com/<service>/main.js
main.js
discovers Jura instance
nginx-consul-proxy
consul
GET http://${jura}/<service>/main.js
main.js
headers
* cleaned
* cache added
Jura
discovers <service> instance
GET http://${service}/${serviceAssetsPath}/main.js
main.js
registers assets via diplomat.registerAssets
<service>

Figure 9. Jura workflow

3.3.3 Micro-Frontend Services Components

A component can be any JS code that can be exported in any service. In most cases, it is a micro-frontend that allows us to render a part of our application independently. Each component exposed has a unique name that others would know to load it by. Each frontend service usually presents at least one element with the services name but can expose as many as it needs. The naming convention is generally like this *serviceName:componentName*. Each service that provides a component needs to register and serve the assets. For this, a service needs to call the *diplomat.registerAssets* method to define which parts are available.

In Figure 10, *diplomat.registerAssets* method has two different components. One is called *default.js*, and the other one is *componentName.js*.

```
await diplomat.registerAssets({
  version: process.env.DOCKER_TAG,
  originPath: '/assets/',
  components: {
    '': {
      js: 'default.js',
    },
    'componentName': {
      js: 'componentName.js',
      css: 'componetName.css',
      loadWithComponentLoader: true,
    }
  }
})


export default {
  greeter: (person) => console.log(`Hello ${person}`);
}


export default async (componentLoader) => {
  const myDependency = componentLoader.load('serviceName:componentName');

  return myDependency.greeter('Martin');
}
```

Figure 10. Diplomat component registration

It is possible to export a function or an object or a HTML page as a component, but there is a case where we define a component with *loadWithComponentLoader: true*. In that case, the component is expected to be a function that accepts componentLoader itself as an argument. This way, we can easily load other components within our components. The components that are exposed as promises will be resolved automatically and loaded whenever the promise is resolved. Components can be loaded using an instance of ComponentLoader. Those instances are usually accessible in system such as Webapp. Components are loaded asynchronously and are cached throughout the system.

Besides component exporting and loading, frontend applications must have a place to start from that does all the important top-level initialization of global tools, rendering the menus, rendering global messages and rendering micro-frontend components into the main area.

We can categorize components into two categories: micro-frontend components and shared components. The ones that render a page are called micro-frontend components and shared components are the ones which are mostly used as utility components. In the case of micro-frontends components, it is expected to return a component that returns an object. The object should include mount, update and unmount functions in it.

This approach provides developers with great flexibility by allowing them to use their own React version. Each of the functions receives an object as an argument. This object also shown in Figure 11 contains:

- el: DOM node to render the micro-frontend into.
- props: Properties passed by the renderer to the micro-frontend. Always includes the visible prop and it is not passed to the unmount.
- prevProps: Previous version of the props passed. Only used in the update function.

The mount and update functions are often similar, but the mount function will only be called once and update function is called every time when parent component updates the child component.

Another thing to pay attention to in Figure 11 is *visible* property. The visible prop is used because, in many cases, the micro-frontend component is not unmounted when it is not visible anymore. In frontend-roots such as Webapp and Froot, we use a logic named view stack. View stack allows us to keep a view rendered on the background when displaying another view. This way switching back to the view that was already rendered before is lightning fast. When the view is not visible, then it should simply keep at the state it was on before, and when it becomes visible again, one should react to any change that may have happened in the meantime.

```
export default (componentLoader) => {
  mount: async ({ el, props }) => {
    const user = await componentLoader.load('myComponent'); //
Let's imagine myComponent exposes a user object

    if (visible) {
      document.title = 'My page';
    }

    el.innerHTML = `Hello ${user.name}`;
  },
  update: async ({ el, props }) => {
    const user = await componentLoader.load('myComponent'); //
Let's imagine myComponent exposes a user object

    if (visible) {
      document.title = 'My page';
    }

    el.innerHTML = `Hello ${user.name}`;
  },
  unmount: async ({ el }) => {
    el.innerHTML = '';
  }
};
```

Figure 11. Example micro-frontend component

To render a micro-frontend component, one would need a special React component called *MicroFEComponent*. This component is accessible via *componentLoader* in Webapp as *froot:MicroFEComponent*.

froot:MicroFEComponent accepts the following props:

- componentName: Name of the component to be loaded with *componentLoader*.
- onLoad: Callback function what to do when the component is mounted.
- componentProps: Props passed to the component.

3.3.4 ConventionUI

Pipedrive is operating in a competitive global market where user experience is a key differentiator. We have a growing number of designers who produce digital or physical designs together with PMs, developers, or external agencies. Most of them, distributed to separate locations and tribes without the set-up of centralized design governance. Without design standards in place, every new hire slows down the process, and inconsistencies grow throughout the channels, platforms, and within the product itself. Throughout time, different teams have documented various design-related resources, but today most of them have become obsolete and misleading. Currently, there is no single easy-to-use destination for accessing current Pipedrive brand-aligned design systems that would cover the needs of committed production teams and what would define our principles for the excellent user experience. As for the development perspective of Pipedrive's design convention, it uses a set of components called ConventionUI. Elements of ConventionUI are designed by product designers and developed by developers and added to the system. There are certain sets of rules while developing a ConventionUI component to keep the library maintainable. The process starts with the design team defining the specifications of a component. This definition must contain pixel-perfect design delivered via Figma. It should also define the behavioral logic. All the definitions must be done well enough to be understood in a single manner. When the design has been properly defined, development for the component can start. ConventionUI includes UI components such as buttons, dialogues, modals, text and color schemas, select items, panels and spacing schemas. The library can be imported into JSX and postcss files and used when needed.

4 Author’s Contribution

Although micro-frontends concept is around since 2016, there are not many academic resources on the topic. Many sources are consisting of blog posts and conference videos where companies share their way of implementation of micro-frontends considering this fact, the author aimed to provide an up-to-date source of information on implementing and designing micro-frontend architecture. For that, related work from all around the world, and Pipedrive’s documentations have been examined, filtered, and organized by the author. Moreover, analysis and outcomes of Pipedrive’s implementation of micro-frontends are presented to provide insights for both companies and the developers who wish to implement such architecture into their development environment. The author also contributed to Pipedrive by providing extensive information on micro-frontends which could be used by Pipedrive developers to get deep understanding of the topic.

Since Pipedrive is a big company with nearly 300+ developers, most of the improvements and developments are made with a teamwork. Different tribes and teams are involved in creating micro-frontend architecture of Pipedrive. The author is part of Tartu tribe which is responsible for security, billing, and authentication services. As stated in the thesis, the micro-frontends architecture of Pipedrive consists of different components. Although the author did not contribute to creation of each component directly, he delivered three projects and contributed heavily on Pipedrive’s billing micro-frontend project. Figure 12 depicts the author’s contribution. Each project and billing micro-frontend have direct connection with micro-frontend architecture.

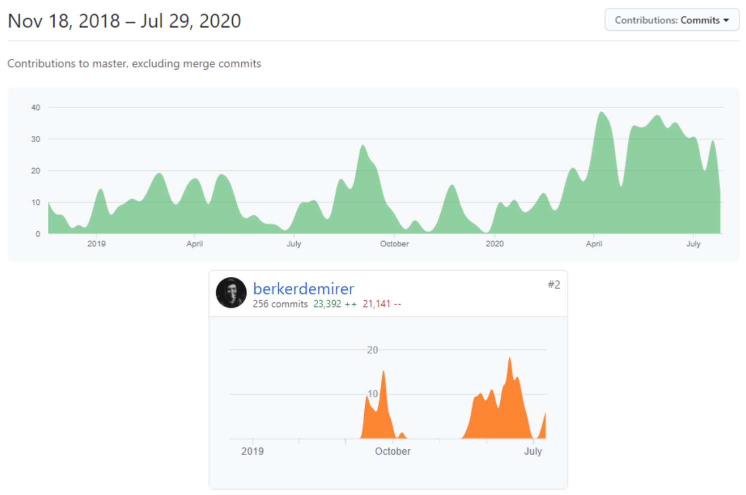
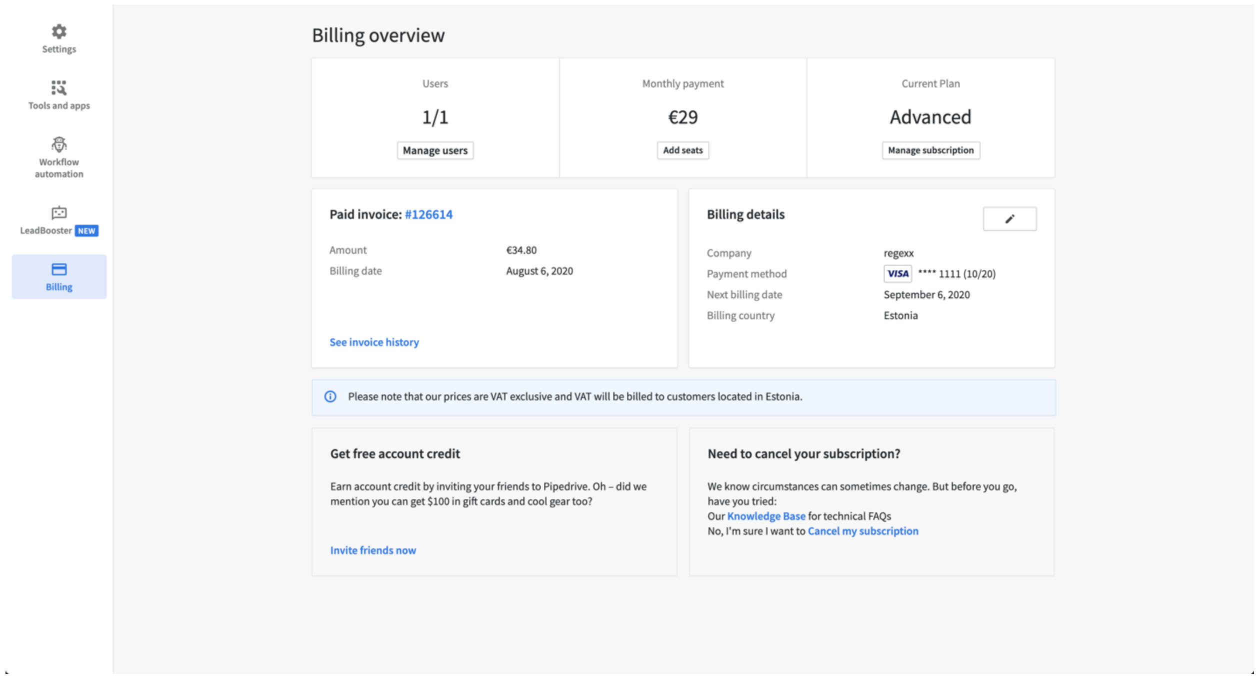


Figure 12. Billing micro-frontend commit counts

Billing micro-frontend service registers itself as a micro-frontend through Diplomat and it is discoverable by the other services. Settings page, which acts as a container service to load different micro-frontends, embeds the billing micro-fronted to its viewport when it is requested by the client. Billing micro-frontend is also working as a container application for another micro-frontend which is used for churn management. Figure 13 depicts the settings wrapper and billing micro-frontend entry component. Settings page wrapper is placed left side of the page and billing micro-frontend is rendered on the center. It consists of different components and shows the billing related information of the customer.



Billing overview

Users	Monthly payment	Current Plan
1/1	€29	Advanced
Manage users	Add seats	Manage subscription

Paid invoice: #126614

Amount	€34.80
Billing date	August 6, 2020

See invoice history

Billing details

Company	regexx
Payment method	VISA **** 1111 (10/20)
Next billing date	September 6, 2020
Billing country	Estonia

Please note that our prices are VAT exclusive and VAT will be billed to customers located in Estonia.

Get free account credit

Earn account credit by inviting your friends to Pipedrive. Oh – did we mention you can get $100 in gift cards and cool gear too?

Invite friends now

Need to cancel your subscription?

We know circumstances can sometimes change. But before you go, have you tried:
Our Knowledge Base for technical FAQs
No, I'm sure I want to Cancel my subscription

Figure 13. Billing micro-frontend screenshot

5 Analysis

This chapter evaluates Pipedrive's micro-frontend implementation by comparing operational metrics such as latency, deployment stability, reliability, and resource usage between micro-frontend and monolith frontend applications. Billing frontend is selected as a micro-frontend as the author has direct contribution to the repository. Historical data is gathered using Pipedrive's Grafana boards.

5.1 Latency

Figure 13 shows the latency data of two services where Webapp – monolith frontend and billing micro-frontend – a micro-frontend in the last thirty days period. Average latency of the micro-frontend is higher than the frontend monolith. While Webapp is able to respond a request in average of 11ms, billing micro-frontend service takes 65ms to return a response. The reason behind that billing micro-frontend is to make external calls to load necessary data such as billing subscriptions, available plans, and promotions whereas in the frontend monolith calls are mostly local. Although 65ms is within acceptable limits, the response time can be improved by reducing call chain length and keeping data as local as possible.

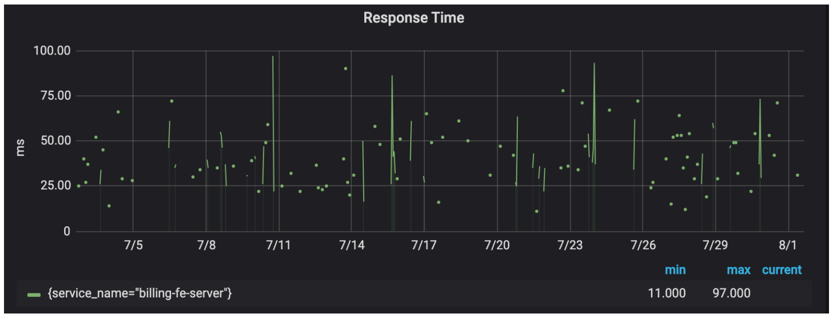


Figure 12. Billing micro-frontend response time

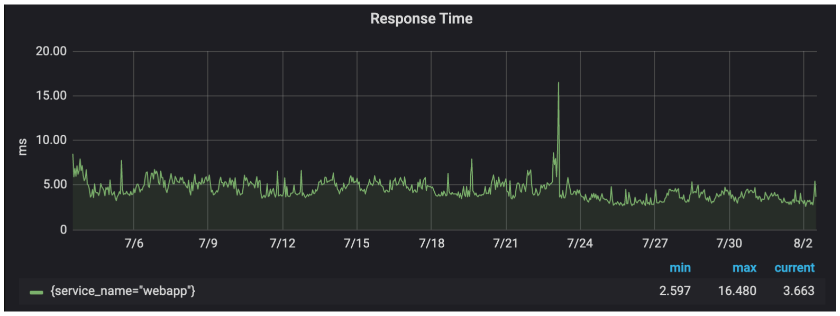
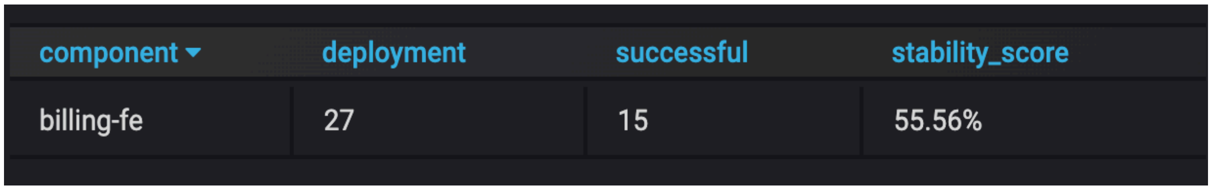


Figure 13. Webapp response time

5.2 Deployment Stability and Time

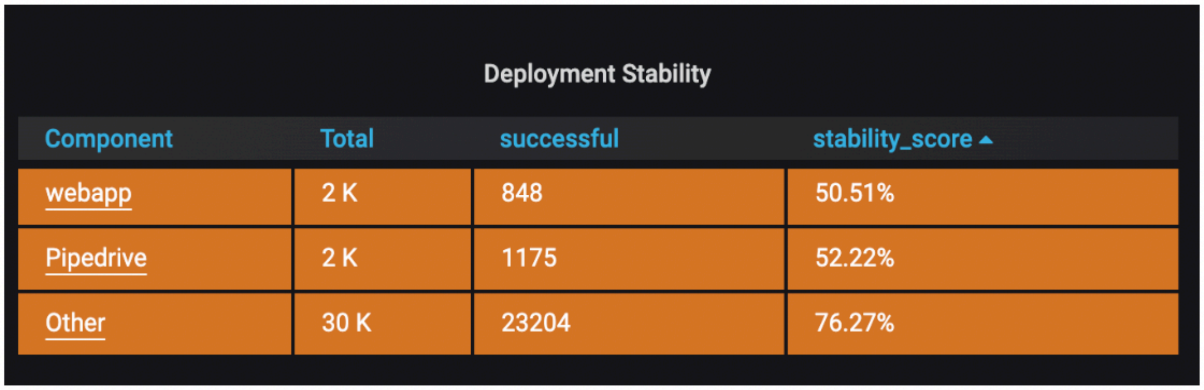
Deployment stability is one of the most impactful issues of monolith applications. As it can be seen from Figure 14, two monoliths of Pipedrive have the stability score of ~50%, which means that half of the deployments failed during the deployment pipeline. The reason behind the failures is mostly related to failing tests and timeouts. This creates a bottleneck in development as developers need to wait for the changes to be deployed before they implement a new feature.

On the contrary, testing is much more easy with micro-frontends as there are less dependencies compared to monoliths. Pipedrive should focus on getting rid of monolith repositories as there is nothing much to do to mitigate the problems. It is unrealistic to put an effort to increase the maintainability of the monolith applications as they have low priority on development projects. However, bad testing and programming practices can also hinder the stability of the micro-frontend. As it can be seen from the Figure 14, billing micro-frontend is slightly better than the monoliths. The reason behind the low stability score is mainly due to end-to-end testing issues.



component	deployment	successful	stability_score
billing-fe	27	15	55.56%

Figure 14. Billing micro-frontend stability score



Deployment Stability

Component	Total	successful	stability_score
webapp	2 K	848	50.51%
Pipedrive	2 K	1175	52.22%
Other	30 K	23204	76.27%

Figure 15. Webapp stability score

Deployment time is also an important point to investigate. When we take a look at Webapp's queue and deployment durations it can be seen that monolith app takes longer to deploy a change to live environment. Average deployment time of the Webapp is 21 minutes and queue time is 37 minutes.

When billing micro-frontend was analyzed, it can be seen that in terms of deployment time and queue time it is much faster than the monolith. Average deployment time is 13 minutes. and queue time is 2 minutes. This also proves that micro-frontends are faster to deploy.

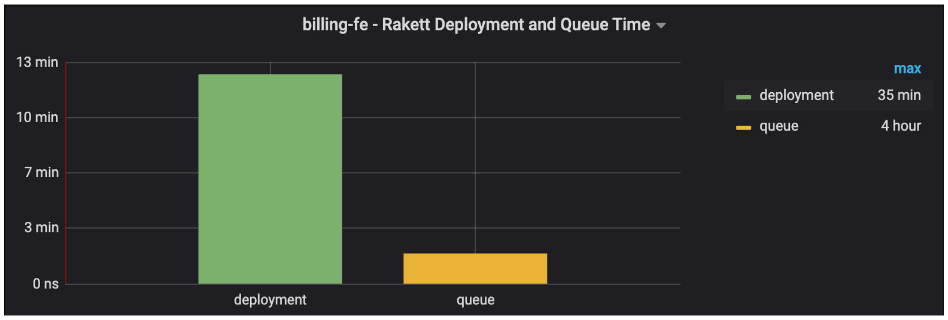


Figure 16. Billing micro-frontend deployment and queue time

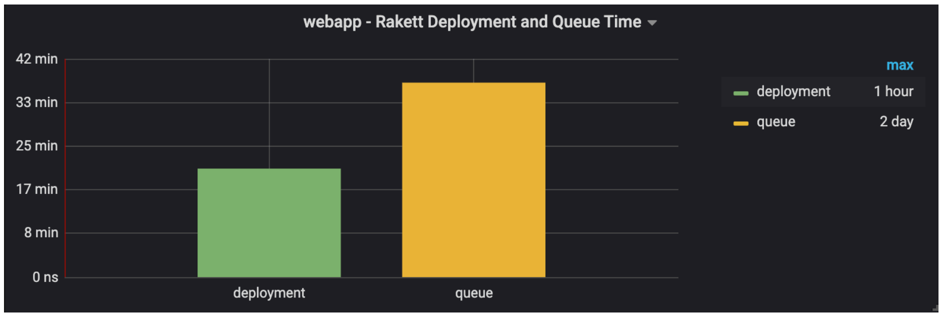


Figure 17. Webapp deployment and queue time

5.3 Reliability

Reliability is another metric to consider when evaluating the performance of the service. By their nature, micro-frontends are expected to make more external calls than the monoliths because a microservice might need to communicate with other microservices. When this is the case, a micro-frontend's reliability also depends on other services. Assume a micro-frontend calls another service with a reliability of 99.9%. This means that out of thousand calls, one will fail and for a call chain with depth of 10, we are down to 99% reliability. However, if services are configured correctly, this error margin can be reduced. In Pipdrive's case the differences vary between services, and it is hard to conclude that Pipedrive's frontend monolith is more reliable over the micro-frontends. For example, according to logs, billing micro-frontend uptime is higher than Webapp's. While Webapp had 99.85% uptime, billing frontend had 100% uptime over the 30 days period.

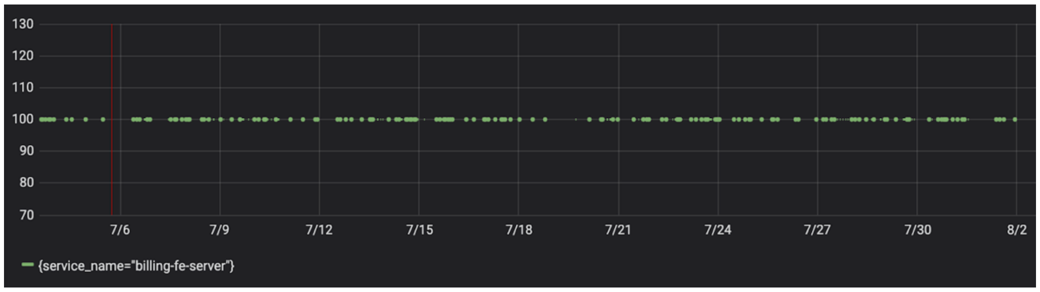


Figure 16. Billing micro-frontend reliability score

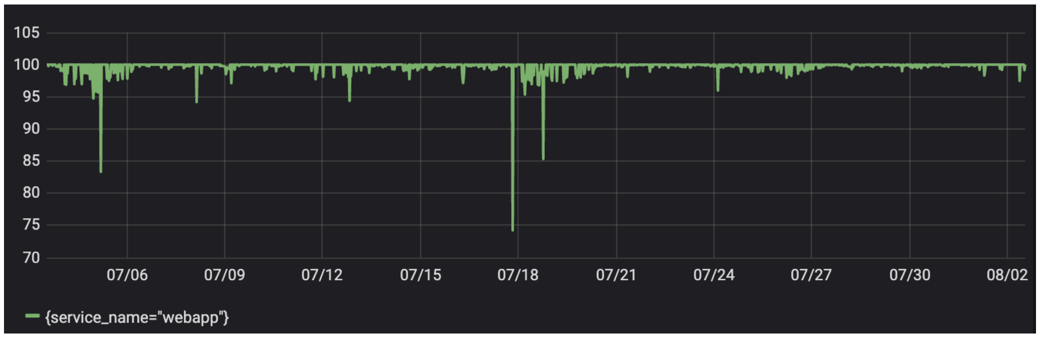


Figure 17. Webapp reliability score

5.4 Resource Usage

Resource usage is an important metric for a service. As the number of services increase or there is a bad engineered architecture, infrastructure of the application might need to work under heavy load, thus, using more resources and leading to costly operations. When we investigate the resource usage between monoliths and micro-frontends, we discovered that monoliths might outperform micro-frontends in edge cases like a network call that transfers a large data, but this case is not likely to happen if the system is designed properly. In most cases, the resource usage is lower with micro-frontend. With monolith, number of instances must be scaled even if one functionality is under high load while the other functionality might perform well. On the other hand, microservices are isolated to specific functionality and easier to scale. However, for micro-frontends this might be not relevant, since they are just static servers.

Using Docker and virtual machines adds overhead to resource usage of the service but also help to allocate resources in a smarter way. Moreover, log aggregation, monitoring, and image orchestration also increase the resource usage.

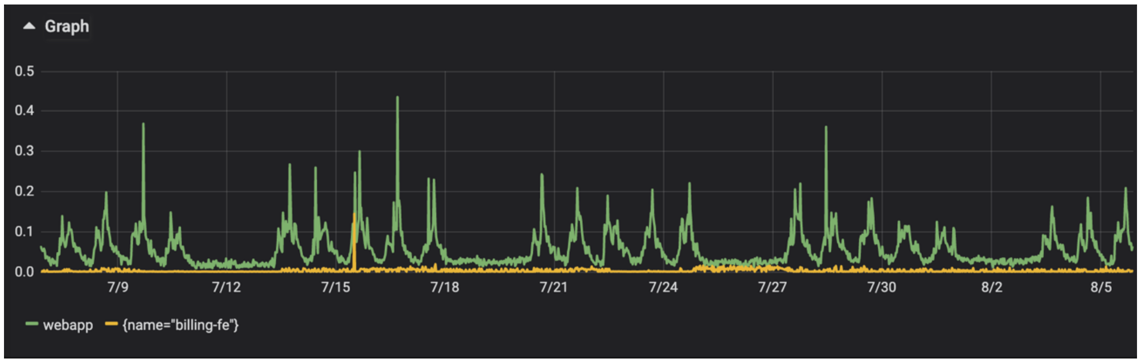


Figure 18. Resource usage graph

As seen in Figure 16, when we ran the queries for Kubernetes cluster memory usages for Webapp and billing micro-frontend, we discovered that Webapp's CPU usage is higher than billing micro-frontend. The values might differ for different clusters, but monolith consumes more CPU than the micro-frontend in general.

6 Conclusion

As web browsers get increasingly powerful and new frontend frameworks emerge, frontend applications have started to handle more business logic than ever. In this kind of situation, many companies faced with scalability problems with their frontend applications as their frontend teams and business requirements grew. Since maintaining the monolith application is a costly business, developers started to search for different approaches to solve the bottleneck. Implementing the microservices approach to frontend monoliths proved to be an effective method for many companies. However, micro-frontend is not the "silver bullet" to any scalability issues. Small companies might benefit more from monolith architecture than micro-frontend services.

The case study discussed how Pipedrive made its transition from having a frontend monolith to micro-frontends architecture. The thesis also explained the key components and concepts that form Pipedrive's micro-frontends architecture such as Diplomat library, Jura, Micro-frontends components and ConventionUI. This study provides insights for Pipedrive's historical improvements on the frontend application.

In conclusion, by providing quantitative metrics, the study helped to reveal Pipedrive's performance with micro-frontend services and monoliths. The study showed that micro-frontend does better in terms of deployment stability, resource usage and reliability. On the other hand, monoliths are better with latency. Last but not least, presence of the legacy code and architectural limitations, which are still present in the ecosystem, Pipedrive's switch to micro-frontend architecture is not yet perfect. Challenges and improvement areas are still current for Pipedrive.

6.1 Future Work

As stated in the conclusion part, Pipedrive's micro-frontend architecture has improvement areas to be addressed. First of all, there are still some views working inside the monolith, and maintaining those views requires an effort as they are legacy code. A small number of people are familiar with it, and monolith deployments are far from being acceptable. Both for performance and organizational concerns, Pipedrive should get rid of views in the monolith parts.

Creating a new micro-frontend service and connecting it to Pipedrive's development and deployment ecosystem takes too much time and requires lots of configurations. There are several boilerplate codes for creating the service, but they are not unified, and this creates maintainability and inconsistency problems across the frontend services. Setting up a new micro-frontend service should be simplified and supported with up-to-date documentation.

The loading time of the services is not so fast because currently unchanged vendors, libraries, and bundles are downloaded after each deploy since the version of the container changes, but not the actual code. Moreover, there is no standardized optimization for the bundle size of the services. Because of that, some services have a bigger bundle size than others, and it slows down the loading time.

Asset registration is bound to the Diplomat library, which forces developers to upgrade all the services if there is a bug fix or some other change in the library. This problem has already created an incident; therefore, asset registration should be decoupled from the diplomat library.

7 References

[1] B. Myers, "The Strengths and Benefits of Micro Frontends," [Online]. Available: https://www.toptal.com/front-end/micro-frontends-strengths-benefits. [Accessed 15 March 2020].

[2] L. Mezzalira, "Micro-frontends decisions framework," 22 December 2019. [Online]. Available: https://lucamezzalira.com/2019/12/22/micro-frontends-decisions-framework/. [Accessed 20 May 2020].

[3] L. Mezzalira, "Identifying micro-frontends in our applications," 21 May 2019. [Online]. Available: https://medium.com/dazn-tech/identifying-micro-frontends-in-our-applications-4b4995f39257. [Accessed 27 March 2020].

[4] L. Mezzalira, "Micro-frontends in context," 13 May 2020. [Online]. Available: https://increment.com/frontend/micro-frontends-in-context/. [Accessed 28 July 2020].

[5] L. Mezzalira, Building Micro-Frontends, O'Reilly, 2020.

[6] Pipedrive, "Pipedrive rated #1 CRM in two leading industry quadrants," 11 May 2019. [Online]. Available: https://www.pipedrive.com/en/newsroom/pipedrive-rated-1-crm-in-two-leading-industry-quadrants. [Accessed 20 March 2020].

[7] M. Klusch, "Service Discovery," in *Network Data Collected via Web*, Springer, Editors: R. Alhajj, J. Rokne, 2014.

[8] L. Mezzalira, S. Peltonen and D. Taibi, Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review, 2020/07/01.

[9] S. Newman, Building Microservices: Designing Fine-Grained Systems. O’Reilly Media, 1st edition, O’Reilly Media, 2015.

[10] S. Newman, The Principles of Microservices, O'Reilly Media, Inc, 2015.

[11] J. Bogard, "Composite UIs for Microservices: Vertical Slice APIs," 15 May 2019. [Online]. Available: https://jimmybogard.com/composite-uis-for-microservices-vertical-slice-apis/. [Accessed 09 June 2020].

[12] J. Colin, "Front-End Micro Services," 06 December 2018. [Online]. Available: https://engineering.zalando.com/posts/2018/12/front-end-micro-services.html. [Accessed 10 July 2020].

[13] R. Gaur, "Breaking down the last Monolith-Micro Frontends," 24 August 2019. [Online]. Available: https://dev.to/aregee/breaking-down-the-last-monolith-micro-frontends-hd4. [Accessed 12 April 2020].

[14] M. Geers, "Micro Frontends - extending the microservice idea to frontend development," 2019. [Online]. Available: https://micro-frontends.org/. [Accessed 20 May 2020].

[15] P. Huang, "Micro-Frontend Architecture in Action with Six Ways," The DEV Community, 2019 June 2019. [Online]. Available: https://dev.to/phodal/micro-frontend-architecture-in-action-4n60. [Accessed 10 June 2020].

[16] C. Jackson, "Micro Frontends," 2019. [Online]. Available: https://martinfowler.com/articles/micro-frontends.html. [Accessed 20 May 2020].

[17] Luigi, "A better answer to distributed UI development - secure and infinitely scalable.," SAP, [Online]. Available: https://luigi-project.io/about. [Accessed 21 July 2020].

[18] L. Revill, "Why Web Components Are so Important," Medium, 10 June 2016. [Online]. Available: https://blog.revillweb.com/why-web-components-are-so-important-66ad0bd4807a. [Accessed 21 March 2020].

[19] P. Senders, "Front-end Microservices at HelloFresh," 15 August 2017. [Online]. Available: Front-end Microservices at HelloFresh. [Accessed 20 July 2020].

[20] Z. Tech, "Project Mosaic | Microservices for the Frontend," Zalando, 2016. [Online]. Available: https://www.mosaic9.org/. [Accessed 15 March 2020].

[21] ThoughtWorks, "Micro Frontends," 2016. [Online]. Available: https://www.thoughtworks.com/radar/techniques/micro-frontends. [Accessed 20 March 2020].

[22] M. Tsimelzon, B. Bill Weihl, J. Chung, D. Frantz, J. Basso, C. Newton, M. Hale, L. Jacobs and C. O'Connell, "ESI Language Specification 1.0," 04 August 2001. [Online]. Available: https://www.w3.org/TR/esi-lang/. [Accessed 12 March 2020].

[23] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter and E. Murphy-Hill, "Advantages and Disadvantages of a Monolithic Repository," in *2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018.

[24] A. Messina, R. Rizzo, P. Storniolo and A. Urso, "A Simplified Database Pattern for the," in *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications*, 2016.

[25] S. Munira, A. Y. Nageye and A. F. Haque, "Content Delivery Network Architecture and Load," in *ICERIE 2017*, Bangladesh, 2017.

[26] "Getting Started," [Online]. Available: https://backbonejs.org/. [Accessed 25 July 2020].

[27] "Introduction to Consul," [Online]. Available: https://www.consul.io/intro. [Accessed 25 July 2020].

8 Appendix

Non-exclusive licence to reproduce thesis and make thesis public