Client-side Microservices Architecture

Modularizing the Web: A Runtime Architecture for Distributed Frontend Services

Enrico Piovesan

June 2023 | White Paper

Table of Contents

Table of Contents 1
Abstract 3
1. Introduction 4
2. Background 5
2.1 Traditional Microservices Architecture 5
2.2 The Legacy of Monolithic Frontends 6
2.3 Toward a Modular Client Runtime 8
3. Goals and Objectives 8
4. Architectural Overview 11
4.1 Client Application Structure 11
4.2 Core Components 12
4.3 Data Management 14
4.4 Communication Model 14
5. Components and Modules 16
5.1 Defining Client-side Microservices 16
5.2 Types of Services 16
Stateless Services 17
Subscribable Services 19
Stateful Services 20
5.3 Anatomy of a Microservice 22
5.4 Client-side Microservices Development Process 24
5.4.1 Automation 24
5.4.2 Versioning and Interface Evolution 25
5.4.3 Microservice Quality and Trust 26
5.4.4 Atomic, Portable, and Agnostic Services 26
5.5 Anatomy of the Client Runtime 27
5.5.1 The Microservice Catalogue/Registry 27
Figure 5: This illustration showcases the anatomy of a runtime implementing the Client-side Microservices Architecture, where each module interacts with the others in sequence 28
5.5.2 The Microservice Manager 28
5.5.3 The Thread Manager 29
5.5.4 The Microservice Event Manager 29
6. Implementation Details 31
6.1 Client-side Multithreading and Parallel Execution 31
6.1.1 JavaScript / Browser 31
6.1.2 iOS 31
6.1.3 Android 32
6.1.4 Trade-offs and Design Considerations 32
7. Performance and Scalability 33
7.1 Scaling 33
7.2 Concurrency and Resource Management 33

8. Security Considerations....35
8.1 What Changes in CSMA....35
8.2 Privilege Management....35
8.3 Service Isolation....36
8.4 Secure Communication....36
8.5 Monitoring and Logging....37
8.6 Dependency Management....37
8.7 Resilience and Redundancy....38
8.8 Security Testing....38
9. Conclusion....39
References....40
Glossary....42
About the Author....45

Abstract

Client-side Microservices Architecture (CSMA) is a modern approach to building modular and scalable frontend applications by organizing business logic into autonomous, independently deployable services that run in the client environment. This model draws inspiration from server-side microservices but adapts to the constraints and opportunities of the browser and mobile runtime. By leveraging asynchronous communication, service isolation, and multithreaded execution, CSMA helps reduce complexity, streamline development across teams, and improve the maintainability and performance of large-scale applications. This paper examines the fundamental principles, practical design strategies, and architectural components of CSMA, demonstrating how this approach enables teams to deliver high-quality, scalable software.

1. Introduction

Over the past few years, there has been a growing need for frontend architectures that can keep up with the rapid evolution of product features, team growth, and platform diversity. This white paper introduces an architecture tailored to meet those needs: Client-side Microservices Architecture (CSMA).

CSMA emerged as a response to common challenges faced by frontend teams working on complex enterprise applications. As products evolve, they often accumulate technical debt and interdependencies that make change difficult and slow. Moreover, modern users expect highly responsive interfaces, which demand efficient use of client resources, including multithreading and modular execution.

This architecture allows developers to break down business logic into discrete services that can be developed, tested, and deployed independently. These services interact via an event-driven model and run in isolated threads, ensuring robustness and scalability. By embracing client-side modularity, CSMA provides a foundation for building complex workflows with minimal friction across distributed teams.

In the sections that follow, we outline the foundational principles of CSMA, walk through its core components, and provide a detailed guide to implementing this architecture effectively in real-world projects.

2. Background

2.1 Traditional Microservices Architecture

Microservices architecture is a widely adopted design strategy that structures applications as collections of loosely coupled, independently deployable services. Each service encapsulates a specific business capability, communicates over lightweight protocols, and can be developed, tested, and scaled in isolation.

In backend environments, these services typically run in separate containers, have their own databases, and follow the principle of single responsibility. This isolation enables teams to adopt various technologies, iterate quickly, and minimize interdependencies that can slow down deployment cycles. Tools like Kubernetes and CI/CD pipelines help orchestrate the lifecycle of these services, reinforcing scalability and operational resilience.

These benefits, including fault tolerance, modularity, flexibility, and scalability, have made microservices the de facto choice for building distributed backend systems. However, bringing these concepts to the client side introduces new challenges. Frontend environments are constrained by runtime limitations, security models, and platform diversity. Despite these limitations, the core ideas of composability and service isolation remain highly relevant and valuable.

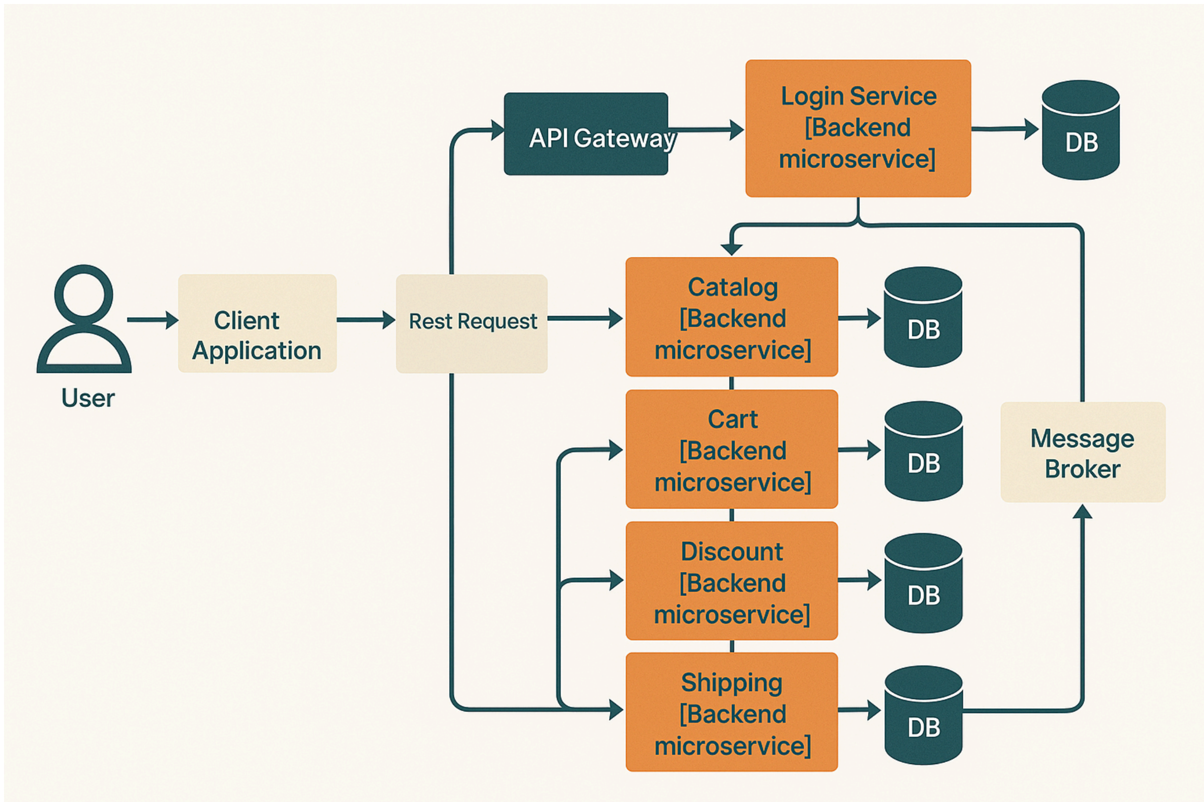

***Diagram 1**: The C4 diagram illustrates a basic e-commerce application and its backend architecture. The system comprises various microservices that communicate using the gRPC protocol. Additionally, a Message Broker is employed to manage the messages exchanged between the services.*

2.2 The Legacy of Monolithic Frontends

Before the rise of modular architectures, most frontend applications followed a monolithic structure. All business logic, UI code, and state management reside in a single, bundled application, often built using modern frameworks such as Angular, React, or Vue. This worked well in the early stages of product development, when the codebase was small and the team size was limited.

As applications scaled in both scope and team size, monolithic frontends began to show their limitations:

- **Scalability bottlenecks**: When demand increases for a specific capability, the entire application must be loaded and scaled as a single unit, even if most of it remains unused.

- **Development friction**: A growing codebase becomes harder to navigate and modify. Small changes in one area can create unintended side effects elsewhere.
- **Technological stagnation**: Adopting new tools or libraries is risky and time-consuming, as changes often require extensive refactoring across the codebase.
- **System fragility**: A bug in a single feature can bring down the whole application, increasing risk and maintenance overhead.

These issues are analogous to those faced by monolithic backend systems and have led many teams to seek more modular, service-oriented approaches for the frontend.

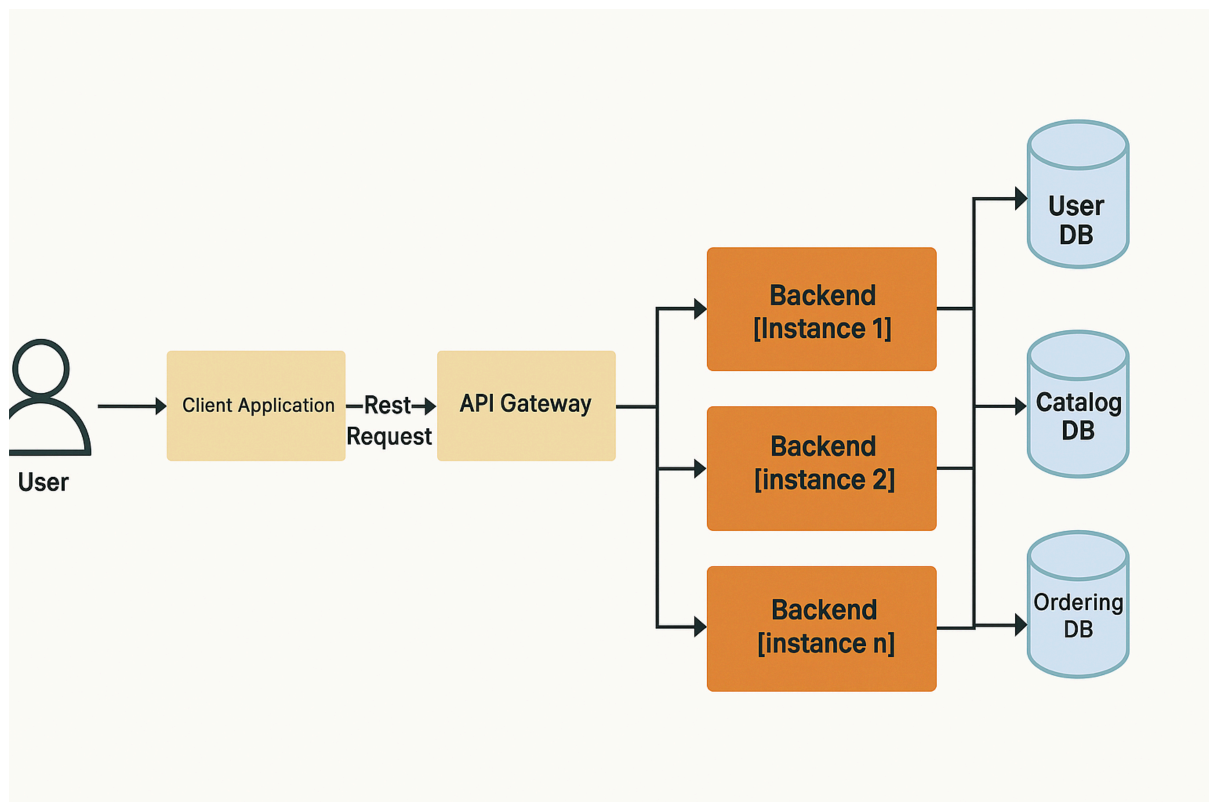

Diagram 2*: The C4 diagram illustrates a generic implementation of a monolithic backend approach. In this architecture, scalability can only be achieved by scaling the entire solution vertically.*

2.3 Toward a Modular Client Runtime

In response to the pain points of monolithic frontends, some teams adopted micro-frontend strategies, splitting the UI into independently developed and deployed components. While this improved team autonomy and deployment agility, most implementations continued to treat business logic as a monolithic layer underneath the segmented UI.

Client-side Microservices Architecture (CSMA) builds on the principles of microservices by treating business logic as a distributed set of services within the client runtime. These services are independently executable, loosely coupled, and communicate through an event-driven model. They run in isolated threads where possible, enhancing parallelism and fault isolation.

This modular runtime model enables teams to build complex applications by composing atomic units of business logic, much like how backend systems have evolved. It supports parallel development, platform diversity, and efficient runtime behaviour while reducing the entanglement typical of traditional frontend codebases.

3. Goals and Objectives

Building a client-side application today is easier than ever. High-level frameworks and tooling enable even junior developers to produce fully functional web or mobile apps quickly. This ease of entry, combined with cloud-based infrastructure and open-source tooling, has empowered startups and small teams to deliver products faster than many large organizations.

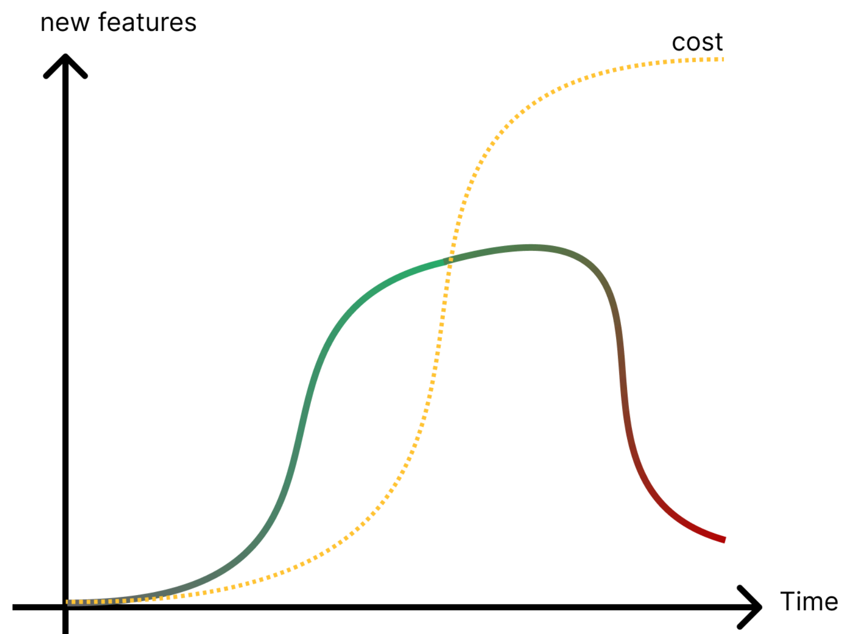

Figure 1: *This illustration aims to highlight that, generally speaking, as time progresses and the complexity of the codebase and capabilities increase, the velocity of implementing new features, after an initial spike, reaches a plateau and then decreases exponentially over time. Meanwhile, the cost of delivering new capabilities increases significantly.*

However, as products mature, they often reach a plateau. What begins as a lean feature set quickly balloons into a complex application shaped by customer feedback, shifting requirements, and fast-paced delivery cycles. Over time, several compounding challenges emerge:

- **Technical debt**: Rapid delivery often comes at the cost of quality. Code written under pressure accumulates issues that are never fully resolved, making future changes more fragile and time-consuming.
- **Development friction**: As the codebase grows, it becomes increasingly difficult to navigate, thereby increasing the cost and risk associated with each new change.
- **Changing requirements**: Constant pivots and adjustments in product direction introduce inconsistencies and fragmentation, leading to a lack of consistency.
- **Team burnout**: Sustained delivery pressure, combined with unclear ownership or complex systems, can lead to fatigue and a decline in productivity.

- **Coordination overhead**: The more teams and components involved, the harder it becomes to integrate features, manage dependencies, and maintain quality.
- **Loss of agility**: Without regular refactoring, systems become rigid and harder to extend, reducing responsiveness to customer needs.
- **Staff turnover**: Changes in team composition often lead to lost context and uneven knowledge distribution.

This trajectory is common across many organizations, especially those that have grown through acquisitions, adopted legacy systems, or expanded beyond their original scope. The cumulative effect can lead to stagnation, with delivery velocity slowing and innovation coming to a halt.

Client-side Microservices Architecture (CSMA) was designed to interrupt this pattern. By breaking apart monolithic business logic into independent services that run entirely on the client, CSMA aims to restore agility, simplify scaling across teams, and reduce the cost of change. It treats the client not just as a rendering layer, but as a programmable, distributed environment where business logic can be composed, reused, and maintained more effectively.

The goals of CSMA include:

- Enabling teams to build and evolve client applications modularly and asynchronously.
- Reducing cross-team coordination by isolating services and their lifecycles.
- Improving runtime performance through parallelism and thread isolation.
- Supporting rapid iteration and experimentation without destabilizing the broader system.
- Creating a foundation for long-term maintainability and team scalability.

By rethinking how business logic is organized and executed on the client, CSMA addresses the structural inefficiencies that slow teams down and offers a sustainable model for growth.

4. Architectural Overview

Every successful architecture begins with a clear understanding of the problem it is intended to solve. When designing Client-side Microservices Architecture (CSMA), the focus was on enabling distributed teams to build large, modular applications that remain fast, maintainable, and flexible over time. The architectural vision needed to reflect this reality: complexity increases with growth, and unchecked, it will slow delivery and raise costs.

The key insight behind CSMA is to treat business logic as a composable system of services, not a monolithic layer within the frontend. Rather than intertwining logic across UI components, state containers, and network layers, CSMA introduces a runtime that hosts modular services in isolation. These services are discoverable, independently executable, and reactive by design.

By placing this runtime at the heart of the application and separating it cleanly from the UI, CSMA enables a clean contract between presentation and behaviour. The UI becomes focused purely on rendering and interaction, while the runtime coordinates service execution, data flow, and backend communication.

This separation yields a more adaptable system. Teams can build new capabilities, swap out implementations, or update business rules without touching unrelated code. It also allows services to be reused across projects or platforms, increasing efficiency and consistency.

4.1 Client Application Structure

The CSMA client is divided into two primary parts:

- **UI Components**: These are responsible for visual presentation and user interaction. They do not contain business logic. Their role is to render data, capture user input, and delegate behaviour to the runtime.

- **Runtime Application**: This is where the application's core functionality lives. It hosts the service manager, event system, and communication interfaces. It interacts with the backend, manages service execution, and handles application state.

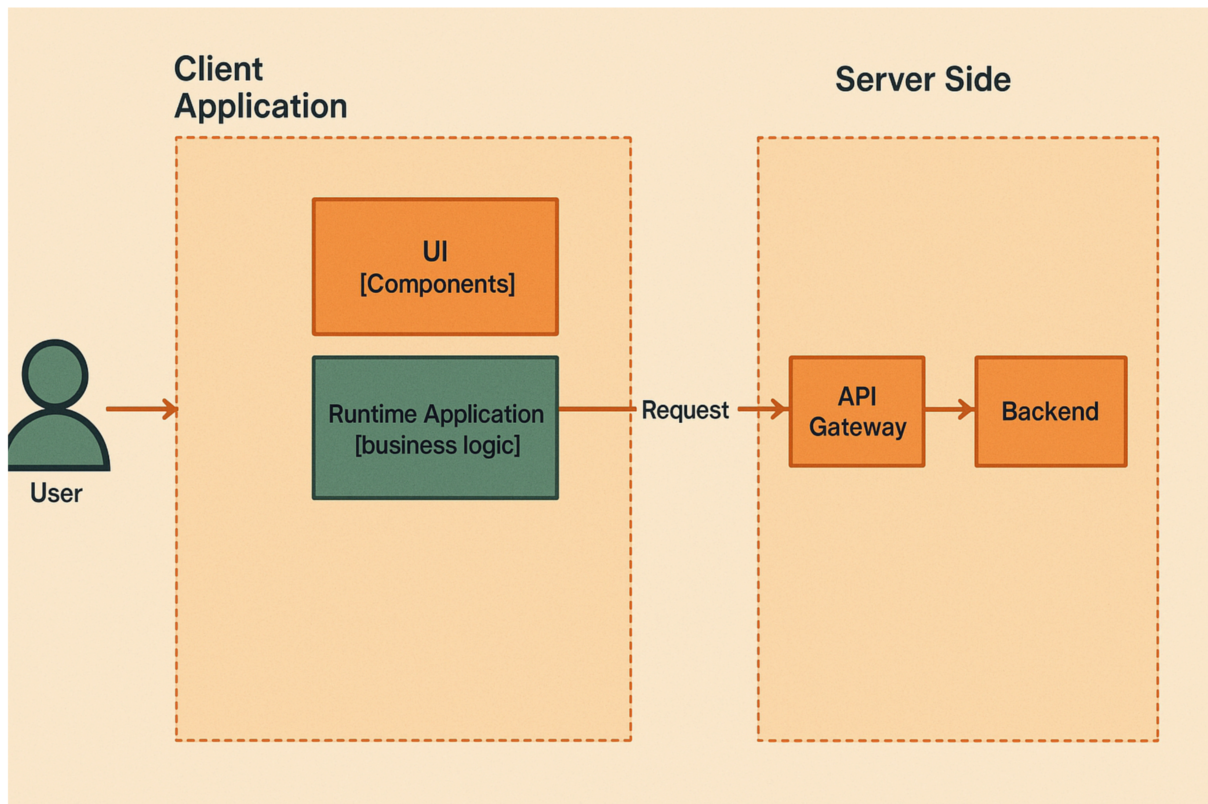

***Diagram 3**: The C4 diagram presents an overview of the Client-side Microservices Architecture. It depicts a user interacting with a client-side application, which is split into two sections: the UI components, containing no business logic, and the runtime, which hosts the core capabilities. The diagram also clarifies that the runtime is responsible for communication with the backend, rather than the UI components.*

This structure makes business logic explicit and composable, allowing for clear and concise implementation. The runtime becomes the programmable engine of the app, while the UI focuses solely on providing a seamless user experience.

4.2 Core Components

The CSMA runtime is made up of several foundational components:

- **Service Manager**: Dynamically loads services based on context and demand. It manages the lifecycle of each service, including initialization, disposal, and dependency resolution.
- **Thread Manager**: Executes services in isolated threads to improve concurrency, fault isolation, and performance. This is especially useful in large applications where multiple tasks must run in parallel without blocking the main thread.
- **Event Manager**: Coordinates communication between services through an event-driven model. It enables asynchronous interaction and decouples producers from consumers.
- **Service Registry**: Maintains metadata and availability information for all services in the system. This allows services to be discovered and queried as needed.

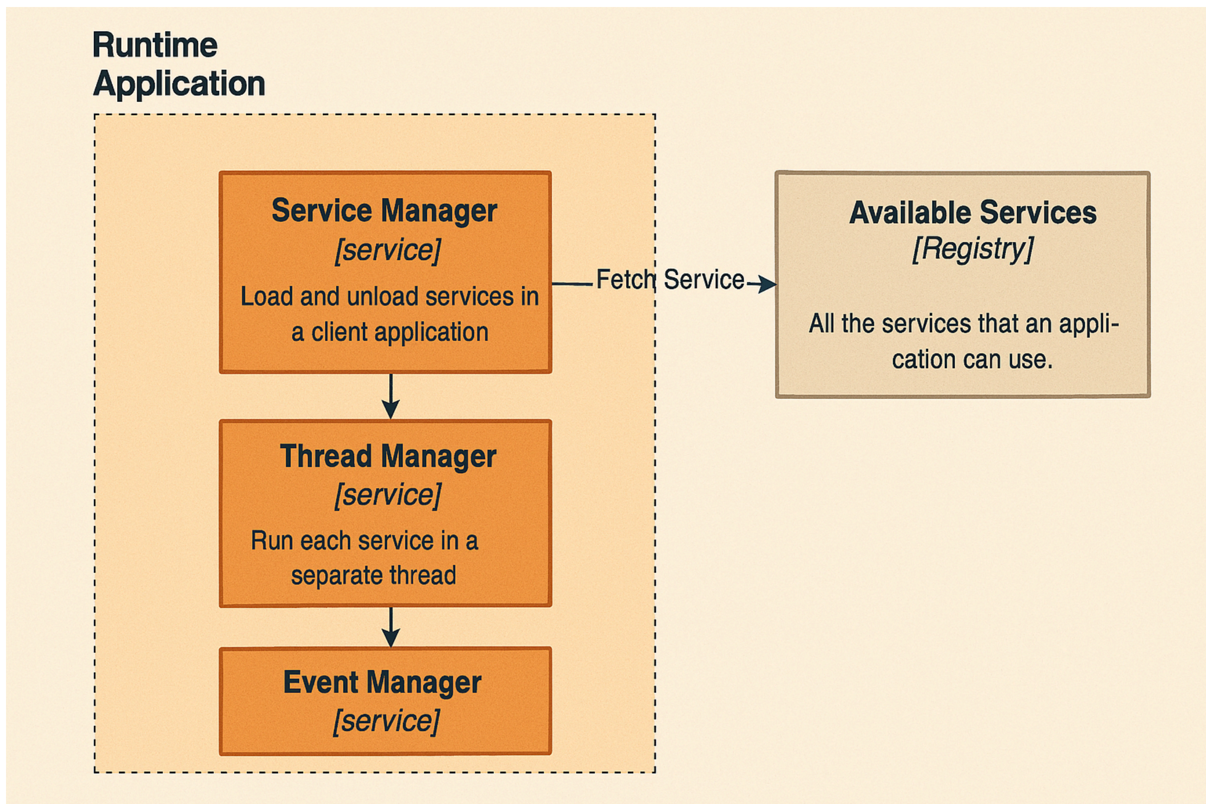

Diagram 4: *This diagram highlights the essential components necessary for the basic implementation of the Client-side Microservices Architecture (CSMA). The Service Manager fetches services on demand for initialization, the Thread Manager is responsible for running microservices in separate threads, and the Event Manager facilitates communication between microservices using the eventing system.*

Together, these components allow the runtime to function as a lightweight orchestrator for distributed client-side services.

4.3 Data Management

Each service in CSMA is responsible for its data. Services fetch data from backend APIs and maintain their runtime cache. This decentralized approach reduces coupling between features, making services more portable and testable.

Because services do not rely on shared global state, they can operate independently, which improves modularity and supports parallel development.

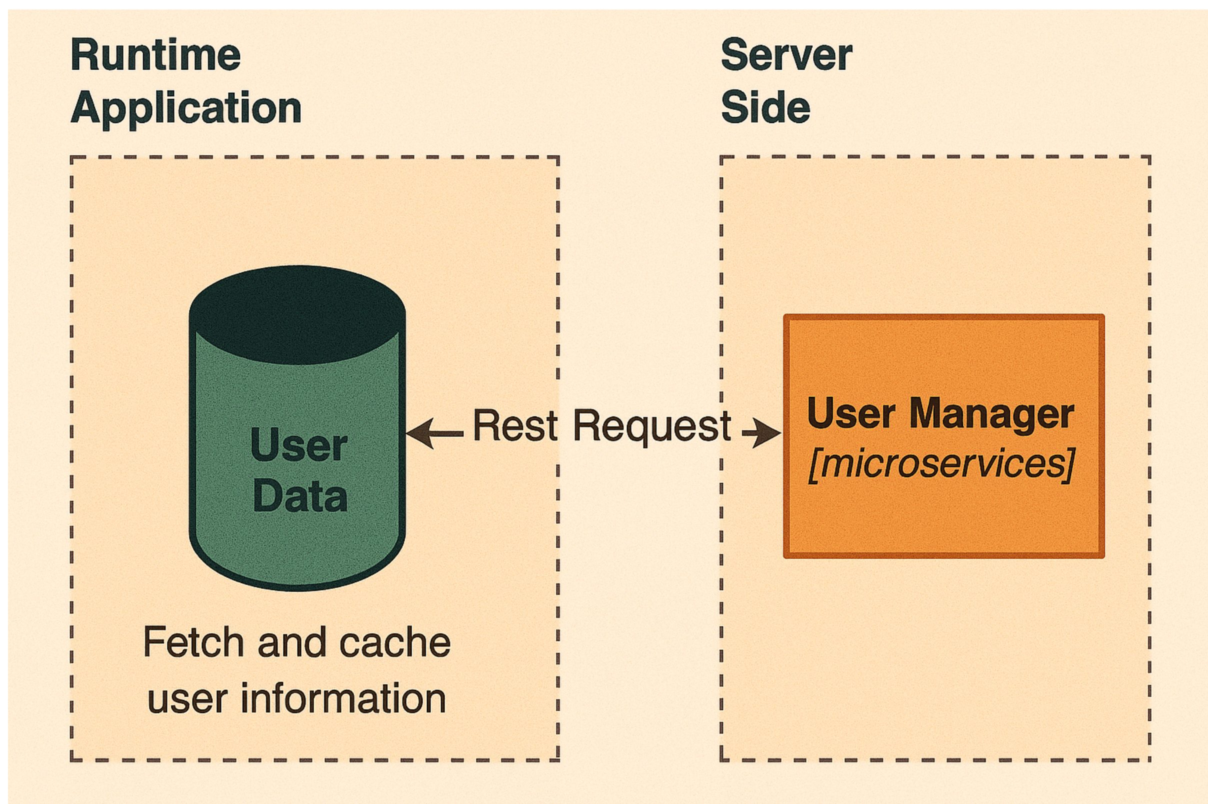

Diagram 5: *This diagram illustrates how each client-side microservice owns its data and maintains a copy at runtime after fetching the content using a REST request to the backend.*

4.4 Communication Model

Services in CSMA do not call each other directly. Instead, they communicate through events. Each service can publish events to a shared event bus and subscribe to specific

topics relevant to its function. This enables loose coupling, allowing services to respond reactively to changes elsewhere in the system.

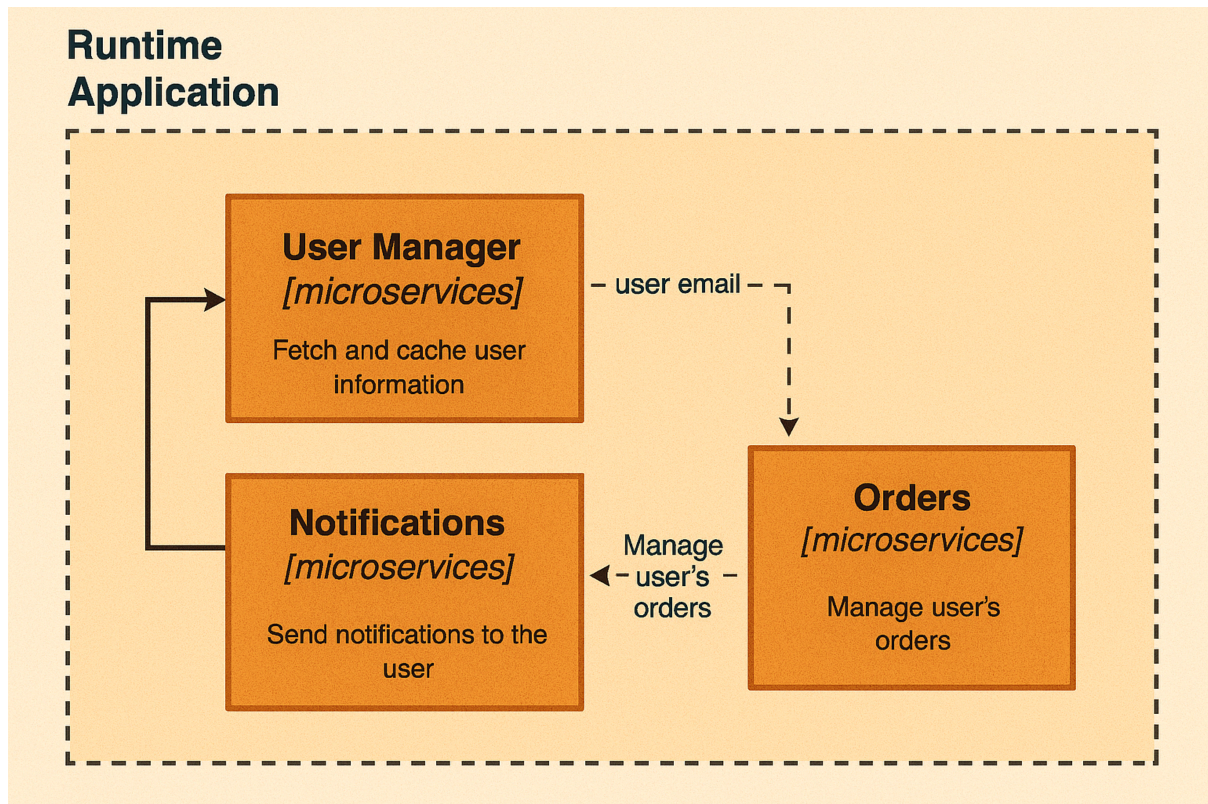

Diagram 6: *This diagram showcases how multiple services independently communicate with each other through an event broker by publishing and subscribing to events*.

This model supports scalability and makes the system more adaptable to change. New services can be introduced without refactoring existing ones, and the flow of data and behavior remains traceable and testable.

5. Components and Modules

5.1 Defining Client-side Microservices

In CSMA, business logic is structured as modular services that run within the client runtime. These services are not just functions or helper classes. They are fully encapsulated units with defined lifecycles, scoped data, and clear contracts.

Client-side microservices are:

- **Isolated**: Each service executes independently and has minimal external dependencies.
- **Self-contained**: Services manage their own state, configuration, and data retrieval logic.
- **Reusable**: They can be composed into workflows or plugged into other applications.
- **Event-driven**: They publish and subscribe to events to communicate, avoiding direct coupling.
- **Lifecycle-aware**: Services are instantiated, activated, and disposed of according to their context and needs.
- **Platform-agnostic**: While implementation details vary by runtime, the architectural pattern is consistent across web and native clients.

These services can be versioned, cached, tested, and even deployed independently, allowing teams to iterate quickly and safely.

5.2 Types of Services

Client-side microservices fall into three primary categories, each with distinct behaviour and use cases:

	STATELESS SERVICE	SUBSCRIBABLE SERVICE	STATEFUL SERVICE
Discoverable	✓	✓	✓
Has a lifecycle	✗	(instantiated upon the first subscription)	(instantiated explicitly)
Persists information (contains a Store)	✗	✓	✓
Subscribable	✗	✓	✓

Table 1: *This Table represents the different lifecycle of each service type and its capacity for persisting data*

Stateless Services

```
class TextAnalyzer extends AutoDisposable implements ITextAnalyzer {
    resultStats = new Observable<AnalysisResult>({});

    public analyze = (...documents: string[]): void => {
        const wordCounts: Record<string, number> = {};
        let totalWords = 0;
        let totalSentences = 0;

        for (const doc of documents) {
            const sentences = doc.split(/[.!?]+/).filter(Boolean);
            totalSentences += sentences.length;

            const words = doc.match(/\b\w+\b/g) || [];
            totalWords += words.length;

            for (const word of words) {
                const normalized = word.toLowerCase();
                wordCounts[normalized] = (wordCounts[normalized] || 0) + 1;
            }
        }

        const avgSentenceLength = totalWords / (totalSentences || 1);
        const topKeywords = Object.entries(wordCounts)
            .sort((a, b) => b[1] - a[1])
            .slice(0, 5);

        this.resultStats.publish({
            totalWords,
            totalSentences,
            avgSentenceLength,
            topKeywords,
        });

        this.dispose();
    };
}

interface ITextAnalyzer {
    analyze(...documents: string[]): void;
}

interface AnalysisResult {
    totalWords: number;
    totalSentences: number;
    avgSentenceLength: number;
    topKeywords: [string, number][];
}
```

- **Execution**: Called directly, no retained state.
- **Purpose**: Perform quick computations, formatting, or fire-and-forget actions.

  Use cases: Analytics logging, data transformation, batch computations.

Subscribable Services

```
class LiveTemperatureMonitor extends AutoDisposable implements ITemperatureMonitor {
    currentTemperature = new Observable<TemperatureStats>({
        average: 0,
        highest: -Infinity,
        sensorCount: 0
    });

    private sensors: Observable<number>[] = [];

    public registerSensor(sensor: Observable<number>) {
        this.sensors.push(sensor);
        sensor.subscribe(this.recalculate);
    }

    private recalculate = () => {
        const readings = this.sensors.map(s => s.value());
        const count = readings.length;
        const sum = readings.reduce((acc, t) => acc + t, 0);
        const max = Math.max(...readings);

        this.currentTemperature.publish({
            average: sum/count,
            highest: max,
            sensorCount: count
        });
    };

    public unsubscribeAll() {
        this.sensors.forEach(sensor => sensor.unsubscribe(this.recalculate));
        this.sensors = [];
        this.dispose();
    }
}

interface ITemperatureMonitor {
    registerSensor(sensor: Observable<number>): void;
    unsubscribeAll(): void;
}

interface TemperatureStats {
    average: number;
    highest: number;
    sensorCount: number;
}
```

- **Execution**: Instantiated upon first subscription, disposed when no longer needed.
- **Purpose**: React to events or listen to external sources without maintaining state.
- **Use cases**: Network status monitors, responsive layout logic, notification listeners.

Stateful Services

```typescript
class UserSessionManager extends AutoDisposable implements ISessionService {
    session = new Observable<UserSession | null>(null);

    public login(userId: string): void {
        const now = new Date();
        const newSession: UserSession = {
            userId,
            loginTime: now,
            lastActivity: now
        };
        this.session.publish(newSession);
        this.startActivityTracking();
    }

    public updateActivity(): void {
        const current = this.session.value();
        if (current) {
            current.lastActivity = new Date();
            this.session.publish({ ...current });
        }
    }

    public logout(): void {
        this.session.publish(null);
        this.dispose();
    }

    private startActivityTracking(): void {
        const id = setInterval(() => {
            const current = this.session.value();
            if (current) {
                const now = new Date().getTime();
                const last = current.lastActivity.getTime();
                const minutesInactive = (now - last) / 1000 / 60;
                if (minutesInactive > 15) {
                    this.logout();
                }
            }
        }, 60000); // check every minute

        this.autoDispose(() => clearInterval(id));
    }
}

interface ISessionService {
    login(userId: string): void;
    logout(): void;
    updateActivity(): void;
}

interface UserSession {
    userId: string;
    loginTime: Date;
    lastActivity: Date;
}
```

- **Execution**: Explicitly instantiated, retains local state during its lifecycle.
  Purpose: Manage shared application state or track session-specific logic.
- **Use cases**: Authentication handlers, shopping carts, state machines.

Each type plays a specific role within the runtime and should be used deliberately based on the intended behaviour and data management strategy.

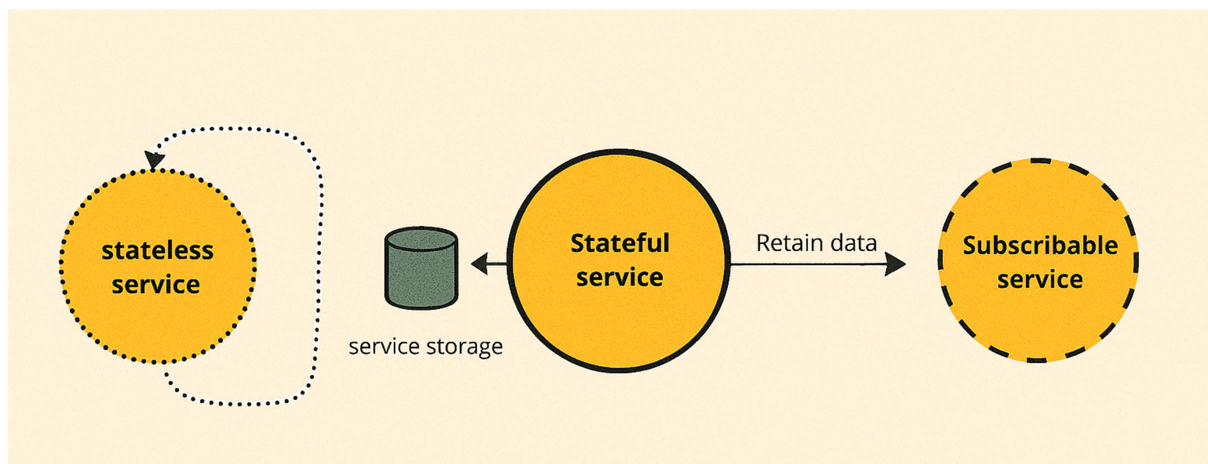

Figure 2: *This illustration showcases the different anatomy and lifecycle of the three types of services.*

5.3 Anatomy of a Microservice

A well-structured microservice typically includes the following elements:

- **Business Logic Core**: The main function or stateful object that defines what the service does.
- **Lifecycle Hooks**: Setup and teardown logic to manage memory, side effects, or observers.
- **Interface Definition**: A strongly typed contract describing public APIs and event types.
- **Metadata**: Versioning info, ownership tags, dependencies, and configuration descriptors.
- **Event Bindings**: Subscriptions and publishers tied to the runtime event system.

- **Optional Local Store**: In the case of stateful services, an isolated in-memory cache or persistence layer.

This structure supports reusability, clarity, and consistency across teams and services.

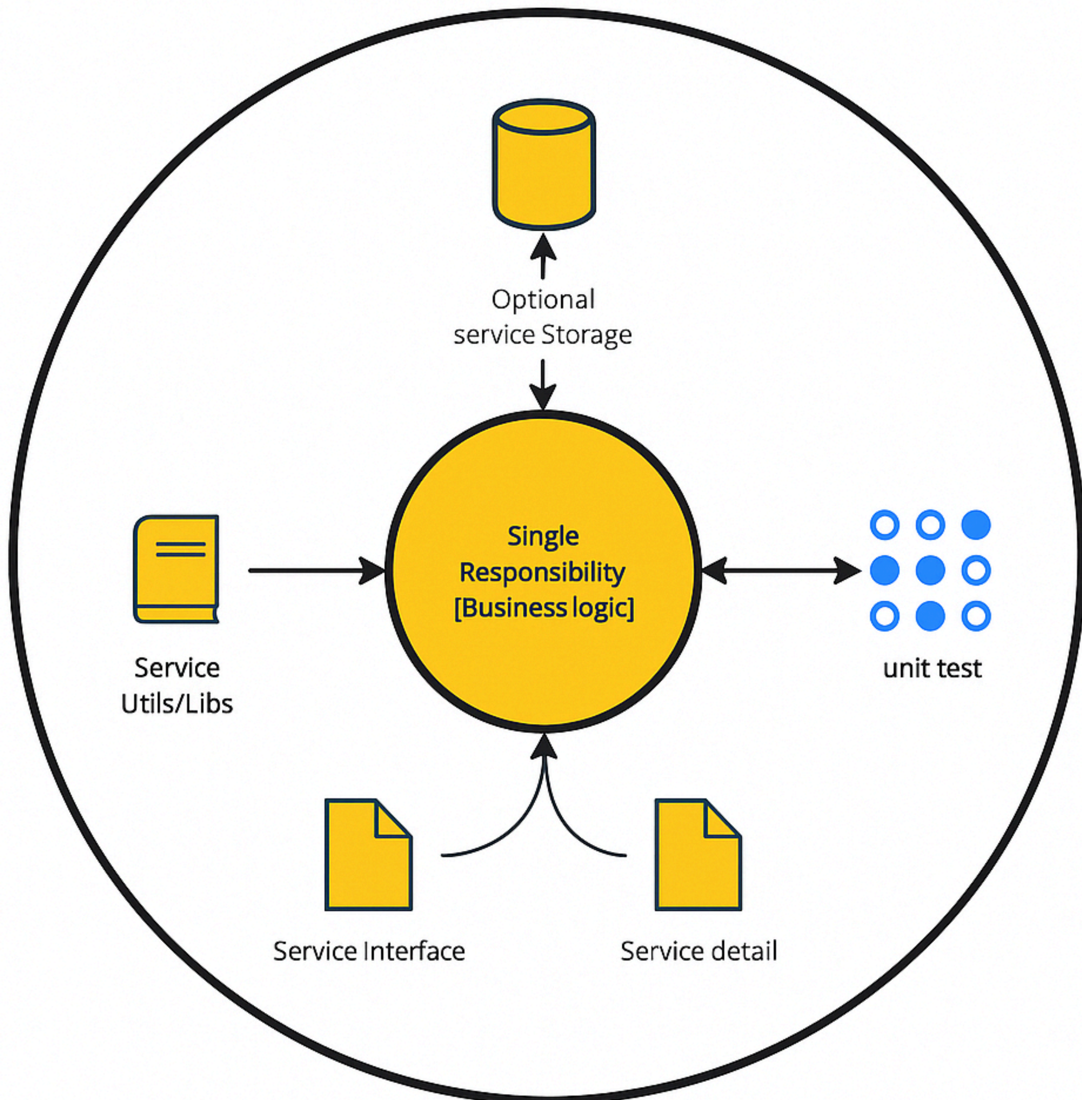

Figure 3: *This graphical representation illustrates the logical grouping of all components that make up a generic client-side service.*

5.4 Client-side Microservices Development Process

Once a microservice is defined and structured, the next step is turning that definition into a reliable, testable, and reusable unit of execution. CSMA emphasizes a disciplined approach to the development pipeline, balancing flexibility with quality and consistency.

There are four key principles to follow throughout the process: **automation**, **versioning**, **trust**, and **portability**.

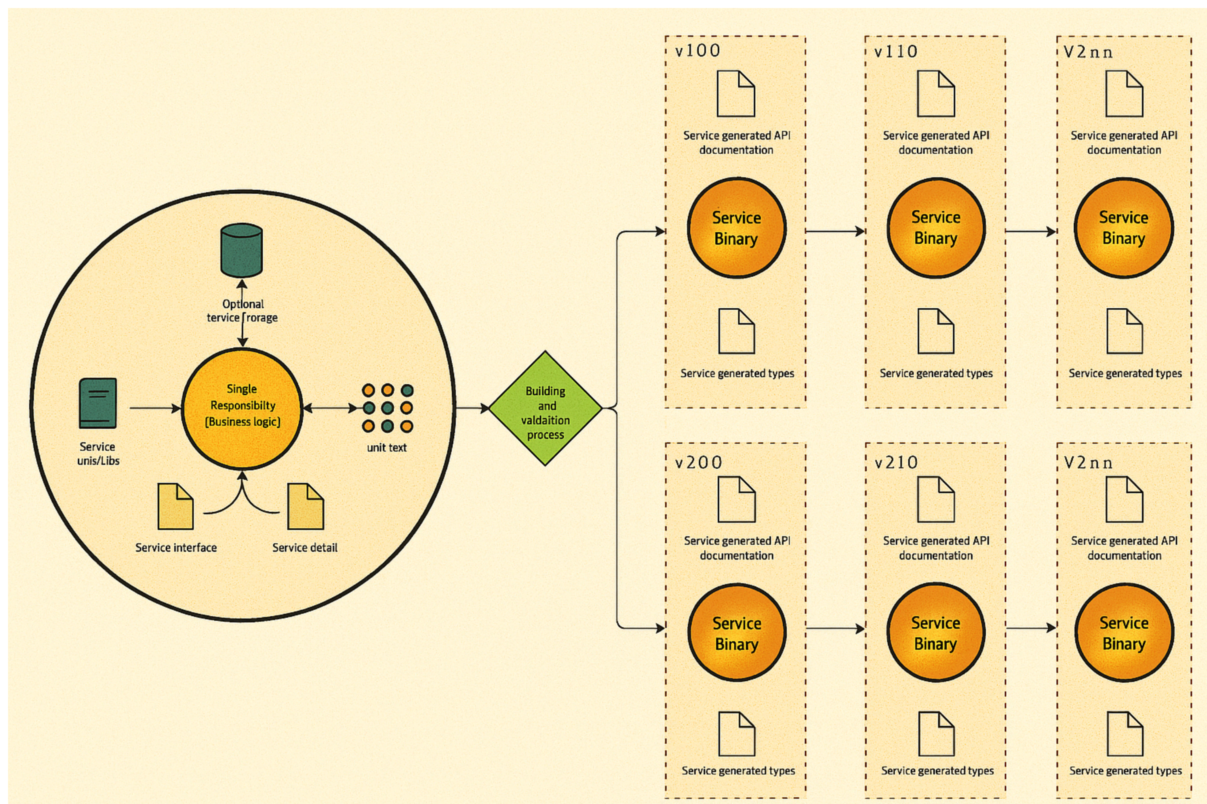

Figure 4: *Illustration of the microservice development process, highlighting the development environment, pipelines and automation for building and validating services, and the versioning and archiving of service builds.*

5.4.1 Automation

Automation is essential from the first line of code to final deployment. The goal is to catch issues early and enforce architectural standards without relying on manual checks.

- **Linting and typing**: Set up strict lint rules and static typing to detect bugs and reduce ambiguity.
- **Pre-commit hooks**: Run lightweight tests and formatters before code is committed.
- **CI pipelines**: Automate unit tests, build validation, and bundling for multiple targets.
- **Quality analysis**: Tools like SonarQube or similar static analyzers help enforce cognitive complexity and modularity goals.
- **API documentation**: Auto-generate documentation from interface definitions to streamline onboarding and integration.

Automation reduces friction, increases confidence, and reinforces architectural discipline.

5.4.2 Versioning and Interface Evolution

Services evolve, but breaking changes must be handled carefully. Versioning ensures that consumers of a service are not disrupted by updates.

- **Interfaces as contracts**: All services define a strict interface that describes their input, output, and event behaviour.
- **Semantic versioning**: Use version numbers to indicate patch, minor, or major changes.
- **Diff-based automation**: Compare interface diffs across versions to determine whether changes are backward compatible.
- **Dependency tracking**: Consumers should specify service versions and optionally test compatibility using integration test suites.

A predictable versioning strategy builds trust and makes service upgrades more manageable across teams and applications.

5.4.3 Microservice Quality and Trust

Modularity brings power but also requires trust. Teams often reuse existing services without knowing their internals, so quality standards must be enforced at the infrastructure level.

- **High test coverage**: Every service should ship with a suite of unit tests covering core logic and expected edge cases.
- **Runtime validation**: Optionally, services can include runtime guards or schema checks to validate inputs and outputs.
- **Error isolation**: Services must be resilient to failures, avoiding side effects or leaks that affect the broader runtime.
- **Observability hooks**: Built-in logging or debug flags help teams trace issues during development or in production.

Establishing trust in a distributed system starts with consistent quality across each atomic unit.

5.4.4 Atomic, Portable, and Agnostic Services

Microservices should be designed to live beyond a single app or platform. Portability ensures long-term value and reuse.

- **Runtime-agnostic code**: Avoid coupling business logic to UI frameworks or platform-specific APIs.
- **Decoupled artifacts**: Build services as independent modules, separate from the host app's build system.
- **Cross-platform builds**: Use compilers or transpilers to generate binaries or bundles for different targets (e.g., browser, mobile).
- **Minimal dependencies**: Keep service dependencies lean and abstract to reduce lock-in.

This allows the same business logic to run in web, native, or embedded environments with minimal changes.

5.5 Anatomy of the Client Runtime

The client runtime, along with the microservices, are the core elements of the Client-side Microservices Architecture. In this section, we will identify the four essential components that any client-side runtime must implement to utilize this software architecture style. It's important to note that the following components are logical; depending on the needs or characteristics of the selected stack for developing your application, the number of components or their subdivisions may vary. The recommendation is to build individual components as atomically as possible to facilitate modification, enhancement, and adaptation over time.

5.5.1 The Microservice Catalogue/Registry

The microservice catalogue is responsible for registering and listing all the service bundles or binaries that an application can use during its lifetime. When designing your runtime, you have three primary approaches for your catalogue based on your requirements:

1. All microservices are contained within the application bundle and available offline.
2. The registry is a server-side component that the application accesses via APIs to download services on demand.
3. A hybrid solution where some essential and common services are part of the application bundle, while others are available on demand through an API.

Each approach has its pros and cons, so it's crucial to evaluate all trade-offs. For example, a web client with services separated from the application can be very modular and allow almost real-time service association, but it may face latency and resilience issues. Conversely, a mobile application might benefit from a hybrid solution, including foundational services within the initial bundle and additional functionalities on demand. An embedded application might prefer all services in the initial bundle for offline capabilities and to avoid latency, though this approach may limit flexibility in updating capabilities and workflows.

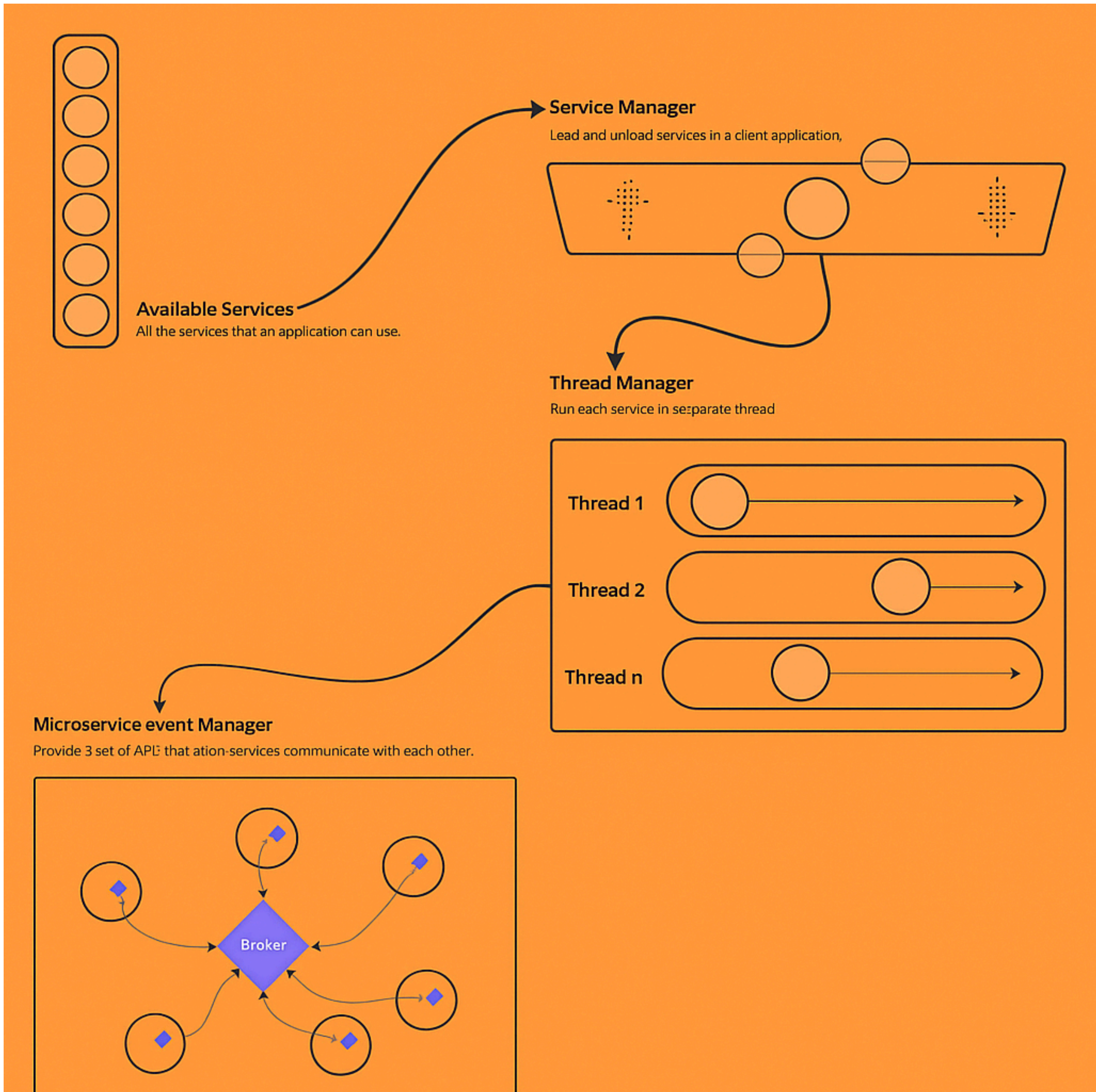

Figure 5: *This illustration showcases the anatomy of a runtime implementing the Client-side Microservices Architecture, where each module interacts with the others in sequence.*

5.5.2 The Microservice Manager

The service manager is responsible for two primary tasks: fetching services on demand when the application context changes and managing the service lifecycle (initialization and disposal) based on the service type (stateful, stateless, and subscribable). This component automatically handles the initialization and disposal of service dependencies. For instance, if the "cart" service needs to load when displaying the cart widget, the service manager will also load dependencies like the "user session service" and the "ordering service."

This component can be split into multiple sub-components and be as modular as needed. Implementation can vary depending on whether the services are pre-bundled or fetched from a server-side registry.

5.5.3 The Thread Manager

Multithreading is central to this architecture for several reasons. Modern client-side applications can become very large and complex, especially in the enterprise world, where multiple teams develop features asynchronously. Offering an out-of-the-box solution for multithreading is advantageous for development teams, as it avoids the need to solve this problem individually and provides powerful tools to manage application performance based on specific needs.

Multithreading ensures thread safety when running multiple services in parallel. Threads are atomic instances, meaning if a thread encounters an exception and crashes, the operating system generally terminates just the involved thread, preventing a complete application crash. In widely distributed codebases, services can be organized into categories with special rules for critical application features. The multithreading solution can also enhance specific services by temporarily allocating more CPU resources.

When designing your runtime, consider your target environment, operating system, project requirements, and the types of binaries you wish to support.

5.5.4 The Microservice Event Manager

As discussed in the high-level architecture overview, the client-side microservice architecture relies on event-driven communication between microservices. This consists of event producers generating streams of events, event consumers listening for these events, and event channels transferring events from producers to consumers.

At this point, you understand what a client-side microservice is and how the runtime uses multithreading to run multiple tasks in parallel. Services, as atomic standalone units, need to communicate with each other by subscribing to events from other services and emitting events to their subscribers. Each event can potentially subscribe to and listen for changes from any other service while emitting information for other events to consume.

The Client-side Microservices Architecture does not prescribe a specific implementation for eventing. Depending on the stack and operating system used, you can select the technology that best suits your needs, such as publish/subscribe or observable patterns, message brokers, or event channels. While a particular implementation is not recommended, using strict types for the

event API and defining interfaces and contracts between microservices is critical. This reduces or eliminates the level of orchestration needed between events and services.

Figure 6: *This illustration showcases a generic implementation of a shopping application, highlighting how each microservice communicates with others. It is important to note the data and information flow, as well as the different types of services involved. (The design of the application is not meant as a recommendation but is an oversimplified version to explain the concept only.)*

6. Implementation Details

One of the most powerful features of CSMA is its ability to support the concurrent execution of client-side services. As applications grow, parallel processing becomes essential for maintaining responsiveness, reducing bottlenecks, and isolating faults. This section outlines how different platforms handle concurrency and how CSMA leverages those capabilities through the Thread Manager.

6.1 Client-side Multithreading and Parallel Execution

While the principles of service isolation and parallelism are consistent, the technical details vary by platform. Below is a breakdown of multithreading tools and strategies across major environments.

6.1.1 JavaScript / Browser

- **Main Thread**: JavaScript executes in a single-threaded event loop by default. UI rendering and user interactions share this thread.
- **Web Workers**: Used to run background scripts independently. They are ideal for offloading CPU-intensive tasks without blocking the UI.
- **Service Workers**: Specialized background workers for handling caching, network requests, and offline support.
- **Promises and Async/Await**: While these manage asynchronous logic, they do not introduce true multithreading. They help with non-blocking operations but still run on the main thread unless paired with workers.

6.1.2 iOS

- **Grand Central Dispatch (GCD)**: A powerful tool that enables tasks to be executed asynchronously on multiple cores. It provides a lightweight abstraction over raw threads.
- **Dispatch Queues**: Tasks are assigned to either serial or concurrent queues, offering control over execution order and concurrency.

- **Operation Queues**: A higher-level API built on GCD, offering dependency management between tasks and more predictable scheduling.
- **Direct Threading**: While possible, it's rarely needed due to GCD's efficiency and safety.

6.1.3 Android

- **AsyncTask**: Historically used for lightweight background tasks, but now largely deprecated in favour of more modern options.
- **Handlers and Loopers**: Core tools for posting messages between threads and managing concurrency in UI components.
- **Executors**: Provide managed thread pools for executing background tasks.
- **Coroutines (Kotlin)**: The preferred modern solution for writing asynchronous code that is readable, efficient, and lifecycle-aware.

6.1.4 Trade-offs and Design Considerations

Each platform comes with trade-offs:

- **JavaScript**: Easy to reason about, but constrained by a single-threaded model unless explicitly using Web Workers. Communication between threads can be verbose.
- **iOS**: Offers high performance and excellent tooling, but requires careful attention to avoid deadlocks or race conditions when using concurrent queues.
- **Android**: Flexible and powerful, but managing thread lifecycles and memory leaks can be tricky without a strong architecture.

To build a robust runtime, CSMA must abstract these platform-specific details and offer a unified interface to developers. The goal is to allow developers to focus on business logic, while the runtime handles the complexity of execution.

A centralized thread manager, aware of platform constraints and performance profiles, ensures that services are executed efficiently and safely.

7. Performance and Scalability

Client-side Microservices Architecture (CSMA) is designed to scale on both the technical and organizational levels. It aims to support responsive, high-performance applications by making smart use of client-side concurrency and modular service boundaries. While scalability in backend systems is often addressed through horizontal infrastructure, the client side requires different strategies focused on resource constraints, execution parallelism, and runtime efficiency.

7.1 Scaling

Unlike server-side architectures, client applications must operate within the hardware limitations of end-user devices. This puts a hard ceiling on memory, CPU, and thread availability. CSMA addresses these limits with targeted strategies:

- **Thread-aware service execution**: Services run in isolated threads where available, preventing UI-blocking operations and isolating faults.
- **Lazy loading**: Services are not loaded until they are needed. This reduces initial bundle size and improves startup time.
- **Context-based service activation**: Runtime conditions (user actions, feature toggles, screen size) determine which services are initialized and when.
- **Hybrid service packaging**: Critical services can be bundled with the client, while others are fetched dynamically as needed. This reduces update friction and allows fine-tuned resource usage.

The runtime’s ability to manage these decisions dynamically makes it easier to deliver performant applications even as complexity increases.

7.2 Concurrency and Resource Management

Efficient use of threads and I/O is key to maintaining performance, especially on devices with limited resources. CSMA introduces centralized mechanisms for managing concurrency in critical areas:

- **Queued I/O execution**: Rather than allowing services to make direct network or storage calls, requests are funnelled through a centralized queue. This prevents collisions, reduces race conditions, and prevents the device from being overwhelmed.
- **Adaptive thread pooling**: The thread manager adjusts the number of concurrent threads based on the device's capabilities and the priority of each service.
- **Service throttling**: Low-priority services can be deferred or paused when system load is high.
- **Non-blocking architecture**: All service-to-service communication uses events, ensuring asynchronous interaction without thread locks or shared memory contention.

These strategies allow CSMA to deliver consistent performance across a wide range of devices and operating conditions.

8. Security Considerations

Security in Client-side Microservices Architecture (CSMA) requires a shift in mindset. Unlike traditional server-side systems, where logic is centralized and guarded, CSMA distributes business logic across the client. This introduces new surfaces for attack and demands careful consideration of how services execute, store data, and communicate.

This section highlights the core areas of concern when building secure client-side applications using CSMA.

8.1 What Changes in CSMA

Client-side services differ from monolithic frontends and traditional microservices in two major ways:

- **Distributed logic**: Instead of being centralized on the server, critical logic is split across client-executed services, which may be developed by different teams.
- **Runtime orchestration**: Services communicate using client-side event buses and run in isolated threads, which introduces unique constraints on permission handling and communication integrity.

These characteristics make it essential to treat each service as a potential point of exposure and enforce boundaries at the runtime level.

8.2 Privilege Management

Not all services should have access to the same data or capabilities. The runtime must support scoped privilege levels to prevent unauthorized access.

Examples by platform:

- **iOS**: Use the Security Framework and Keychain to manage secure data access and enforce trust policies.

- **Android**: Leverage the Android Keystore System to manage credentials and limit access to sensitive operations.
- **Web**: Use browser-native tools like the BroadcastChannel API with strict origin checks, message signing, and validation logic to ensure safe cross-context communication.

Service-level policies should be enforced by the runtime itself, rather than relying on the service implementation to behave correctly.

8.3 Service Isolation

Service isolation improves both reliability and security. The thread manager plays a key role in enforcing this boundary.

Key practices:

- **Immutable objects**: Avoid shared mutable state by default. Favour message passing and functional data flows.
- **Thread-local storage**: Keep service-specific data in local memory contexts, avoiding cross-service bleed
- **Atomic operations**: Use safe methods for updating shared values, even if they are accessed indirectly.
- **Avoid global state**: Reduce reliance on singletons or browser-wide objects.

Proper isolation reduces the risk of cascading failures or accidental access to unauthorized information.

8.4 Secure Communication

Services in CSMA rely heavily on event-based messaging. This makes communication hygiene a top priority.

Best practices include:

- **Message validation**: Ensure each message conforms to a defined contract and reject anything malformed or unexpected
- **Event typing**: Use strict typing for all emitted and consumed events to prevent misuse or injection.
- **Scoped event channels**: Prevent services from listening to every event by default. Only expose what is necessary for functionality.
- **API gateways**: For external communication, proxy all requests through secured endpoints with proper authentication and rate limiting.

8.5 Monitoring and Logging

Visibility is crucial for spotting misuse or unexpected behaviour in a distributed architecture.

Considerations:

- **Centralized logging**: Collect runtime logs across services for analysis and correlation.
- **Audit trails**: Record access to sensitive service APIs and data changes.
- **Anomaly detection**: Watch for patterns that suggest abuse, such as rapid event emissions, repeated access denials, or unauthorized service activations.

8.6 Dependency Management

Every service depends on libraries and frameworks, which introduce supply chain risk.

Key practices:

- **Automated scanning**: Use tools to detect known vulnerabilities in open-source dependencies.
- **Version pinning**: Lock dependency versions to prevent unvetted updates.
- **Trusted sources**: Only use libraries with active maintenance and a strong security track record.

8.7 Resilience and Redundancy

Security also involves planning for failure. Services should degrade gracefully in the event of an attack or fault condition.

Guidelines:

- **Fail safe, not open**: Services should reject risky actions by default when in doubt.
- **Fallback behaviour**: Provide limited, read-only, or cached modes when full functionality is unavailable.
- **Service redundancy**: For critical logic, allow runtime fallback to a backup implementation when the primary fails.

8.8 Security Testing

Integrate security into every step of the development and deployment process.

- **Static analysis**: Detect insecure code patterns during development.
- **Penetration testing**: Simulate real-world attacks to identify weaknesses.
- **CI/CD integration**: Run security checks on every build to catch regressions early.

9. Conclusion

Client-side Microservices Architecture (CSMA) represents a major shift in how frontend applications can be designed, scaled, and maintained. By decomposing business logic into independently deployable services and running them within a modular client runtime, CSMA brings the benefits of backend microservices to the user-facing layer.

This approach enables distributed teams to work in parallel without constantly stepping on each other's toes. It reduces the risks that come with monolithic frontend architectures and makes it easier to isolate failures, adopt new technologies, and evolve applications incrementally.

CSMA is not just a pattern for structuring code; it is a framework for how teams collaborate and how products grow. It supports asynchronous development, encourages testability, and enables consistent patterns across platforms and projects.

While this architecture introduces new complexities such as service discovery, runtime orchestration, and event-driven communication, it offers a long-term return on investment by improving agility, fault tolerance, and developer experience.

Organizations adopting CSMA can expect to deliver new features more quickly, maintain higher quality standards, and adapt to change with less friction. In an environment where responsiveness, flexibility, and scale are increasingly essential, CSMA offers a path forward that is both practical and future-proof.

References

- Bellemare, A. (2020). *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media.
- Dodds, K. C. (2022). *Why I Don't Like Micro-Frontends*. https://kentcdodds.com/blog/why-i-dont-like-micro-frontends
- Fowler, M. (2019). *Micro Frontends*. https://martinfowler.com/articles/micro-frontends.html
- Google Web Dev. (2022). *Web Performance Optimization*. https://web.dev/fast/
- Google Chrome Dev Summit. (2022). *Designing Scalable Web Applications*. https://developer.chrome.com/blog/dev-summit-2022-scalability/
- Herrington, J. (2023). *Architecting Web Applications with Module Federation* [Video]. https://www.youtube.com/watch?v=3zAaR3c1gB0
- Jackson, C. (2019). *Micro Frontends Trade-offs*. https://martinfowler.com/articles/micro-frontends.html#TradeOffs
- Khalid, U. (2023). *Securing Micro-Frontends*. https://dev.to/umairkhalid/securing-micro-frontends-3mj6
- LogRocket. (2021). *Multithreading in JavaScript with Web Workers*. https://blog.logrocket.com/multithreading-javascript-web-workers/
- LogRocket. (2022). *Implementing the Event Bus Pattern in JavaScript*. https://blog.logrocket.com/implementing-event-bus-pattern-javascript/
- MDN Web Docs. (n.d.). *Web Workers API*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API
- MDN Web Docs. (n.d.). *BroadcastChannel API*. https://developer.mozilla.org/en-US/docs/Web/API/BroadcastChannel
- MDN Web Docs. (n.d.). *Content Security Policy (CSP)*. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
- Module Federation (n.d.). *Webpack Module Federation*. https://webpack.js.org/concepts/module-federation/

- NPM Docs. (n.d.). *About Semantic Versioning*. https://docs.npmjs.com/about-semantic-versioning
- OWASP. (2024). *Frontend Security Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Frontend_Security_Cheat_Sheet.html
- Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media.
- Single-spa Docs. (n.d.). *Single-spa: JavaScript Microfrontend Framework*. https://single-spa.js.org/
- Spotify Engineering. (2020). *Scaling Frontend Development at Spotify*. https://engineering.atspotify.com/2020/09/scaling-frontend-development/
- ThoughtWorks. (2023). *Micro-Frontends: Revolutionizing Front-End Development*. https://www.thoughtworks.com/radar/techniques/micro-frontends
- Vanilla Extract. (n.d.). *Type-safe Styling for Modular Frontends*. https://vanilla-extract.style/
- Vercel Engineering. (2022). *Web Workers and Parallelization in Frontend Apps*. https://vercel.com/blog/web-workers-and-parallelization
- Webpack. (n.d.). *Module Federation Examples Repository*. https://github.com/module-federation/module-federation-examples

Glossary

- **Application Shell**: A minimal HTML/JS/CSS structure that loads on first visit and provides the foundation for dynamically rendered services.
- **BroadcastChannel API**: A browser API that allows messaging between browser contexts (e.g., tabs, iframes, workers) on the same origin using a publish/subscribe pattern.
- **Client-side Microservice**: A modular, self-contained unit of logic that runs within the browser or client runtime with its dependencies and lifecycle.
- **Concurrency**: The ability to perform multiple tasks simultaneously, often via asynchronous programming or Web Workers.
- **Contract**: A formal interface definition specifying inputs, outputs, and behaviours of a service or module, enabling interoperability.
- **CSMA (Client-side Microservices Architecture)**: A frontend architecture that breaks applications into modular services executed in the client, supporting isolation, scalability, and autonomy.
- **Event Bus**: A centralized messaging mechanism for services to emit and listen to events in a decoupled way.
- **Event-Driven Architecture**: A software design where components interact by emitting and responding to events instead of direct method calls.
- **Hybrid Packaging**: A deployment model that mixes pre-bundled and on-demand (lazy) loading of services.
- **Import Maps**: A browser-native way to control module resolution at runtime, allowing dynamic loading and aliasing of dependencies.
- **Isolation**: The practice of separating service execution to prevent shared state and contain failures.
- **Lazy Loading**: The deferral of loading services or components until they are required by the user or flow.
- **Lifecycle Hooks**: Functions triggered at specific points during a service's lifespan (e.g., start, stop, cleanup).

- **Micro-Frontend**: A UI module that is independently developed, versioned, and deployed as part of a larger application.
- **Module Federation**: A Webpack capability that allows JavaScript modules to be loaded remotely at runtime for building modular applications.
- **Orchestrator**: A runtime component responsible for discovering, loading, and managing the lifecycle and interactions of client-side services.
- **Privilege Management**: Restricting service access to only the permissions and APIs required, improving security.
- **Publish/Subscribe (Pub/Sub)**: A messaging model where senders publish events and listeners subscribe to relevant event types without tight coupling.
- **Runtime Context**: The specific environment in which a service executes, such as the main thread or a Web Worker.
- **Scoped Services**: Services that are conditionally loaded and executed only for specific routes, states, or flows.
- **Semantic Versioning (SemVer)**: A versioning convention using MAJOR.MINOR.PATCH to signal compatibility and the nature of changes
- **Service Registry**: A runtime directory of all registered client-side services and their exposed interfaces.
- **SharedWorker**: A Web API that allows multiple browser contexts to share the same worker thread for state or caching.
- **Single-spa**: A JavaScript framework that enables the composition of multiple micro-frontends into a unified application.
- **Stateful Service**: A service that maintains its internal state across requests or usage sessions.
- **Stateless Service**: A service that operates only on input and produces output without retaining any data between calls.
- **Thread Manager**: A runtime utility that maps services to execution contexts such as threads or Web Workers, for optimized performance.
- **Version Pinning**: The strategy of locking a dependency to a specific version to prevent regressions from automatic upgrades.

- **Web Worker**: A browser feature that enables running JavaScript in a background thread to avoid blocking the main UI thread.

About the Author

Enrico Piovesan is a Platform Software Architect at Autodesk with over 20 years of experience designing and building cloud-native, event-driven, and modular systems. He specializes in scalable, high-performance architecture and developer-first platform solutions.

Enrico has pioneered several architecture patterns, including:

- **Client-Side Microservices Architecture (CSMA)** – Bringing modular, service-oriented design to the frontend
- **Universal Microservices Architecture (UMA)** – Enabling portable WebAssembly-first services that run across browsers, edge, mobile, and cloud

He is also a serial founder, having launched startups in education, travel, and payment systems. His work blends innovation and pragmatism, always with a focus on autonomy, discoverability, and long-term architectural integrity.

Enrico actively shares his thinking through a five-day blog series:

- **Rethinking the Client** – Modular frontends and the evolution of client platforms
- **Designing for Intelligence** – Software architecture in the age of AI
- **WASM Radar** – Weekly signals and insights from the WebAssembly ecosystem
- **Explain Me Like I'm Code** – Technical concepts explained with stories, humour, and metaphor
- **The Rise of Device-Independent Architecture** – Portable systems, microservices, and universal runtimes

Outside of work, Enrico is a certified ski instructor, a proud father of two, and someone who believes that fresh powder beats screen time any day.