Universal Microservices Architecture

A Runtime-Agnostic Blueprint for Portable, Scalable Systems

Enrico Piovesan

August 2024

White Paper

Table of Contents

Abstract

The Universal Microservices Architecture (UMA) introduces a modern approach to software development that moves beyond traditional client-server models and stack-specific constraints. By enabling device-independent, highly portable capabilities, UMA supports seamless execution across web, mobile, and desktop platforms.

Each microservice within UMA is self-contained, managing its own state and communicating asynchronously with others. This design promotes fault isolation, improves scalability, and simplifies maintenance. UMA addresses inefficiencies in legacy architectures by dynamically optimizing service execution, taking into account variables such as device performance, network conditions, and user permissions.

Central to UMA is the use of WebAssembly (WASM), which compiles services into platform-agnostic binaries, ensuring high performance and broad compatibility across various platforms. The architecture also leverages edge computing to minimize latency and operational costs, while delegating more intensive processing tasks to server-side microservices as needed.

This paper outlines the core principles of UMA and provides guidance on implementation strategies, including service load balancing, environment-aware execution, and the use of stack-specific services for specialized needs. The result is a modular, performant, and flexible system that simplifies deployment, improves developer productivity, and aligns with evolving business requirements.

By adopting UMA, organizations can streamline development workflows, deliver consistent user experiences across devices, and build resilient, scalable applications ready for the demands of modern software ecosystems.

1. Introduction

The inefficiencies of traditional stack-specific architectures and rigid client-server models increasingly constrain modern software development. These legacy approaches often result in the redundant implementation of core functionalities, limiting the portability and consistency of user experiences across platforms.

The Universal Microservices Architecture (UMA) offers a transformative alternative. It introduces a paradigm centred on device independence and service modularity, enabling the creation of portable capabilities that operate seamlessly across web, mobile, desktop, and emerging environments.

At its core, UMA promotes autonomous services that manage their own data and communicate asynchronously. This design pattern not only enhances fault tolerance and scalability but also facilitates the dynamic optimization of execution paths based on factors such as device performance and user permissions.

UMA also facilitates the gradual evolution of legacy systems, allowing organizations to integrate new capabilities without sacrificing existing investments. Its modular structure supports customized workflows tailored to distinct user personas, making it adaptable to a wide range of business needs.

By compiling services into platform-independent WebAssembly (WASM) binaries, UMA ensures broad compatibility and consistent performance across environments. This combination of architectural flexibility and technical portability positions UMA as a powerful foundation for building scalable, efficient, and future-ready applications.

2. Background

2.1 Limitations of Current Development Practices

Today's software development practices often rely on target-specific silos, where capabilities are built independently for each platform. Applications are segmented by device, including desktop, mobile, and web, resulting in redundant development efforts and fragmented workflows. For example, standard functionalities like user session management are frequently reimplemented across different environments, whereas feature-specific components, such as 3D viewers, are developed using platform-dependent SDKs. This results in poor portability and inconsistent user experiences across devices.

<Figure 1>

2.2 Inefficiencies in Traditional Architectures

Business logic is typically confined to either client-side or server-side execution, with technology stacks tightly coupled to specific runtime environments. This rigid dichotomy prevents adaptive decision-making about where tasks should run, thereby limiting the ability to optimize based on contextual factors such as computational load, network latency, or user access rights. The result is increased complexity, reduced performance, and diminished agility in delivering user experiences.

2.3 Challenges in Client-Side Development

Client-side applications often require separate implementations for desktop, mobile, and web platforms. Without a shared codebase, teams must duplicate functionality across multiple stacks, introducing delays, increasing maintenance overhead, and risking inconsistencies in features and behaviour. This fragmentation not only slows delivery but also complicates testing and hinders long-term scalability.

<Figure 2>

2.4 Why a Universal Approach Is Needed

Addressing these limitations requires a shift toward a platform-agnostic strategy that enables the development of reusable, device-independent services. Such an approach must support portability across diverse environments without being constrained by specific stacks or frameworks. It should also decouple business logic from user interfaces, allowing teams to optimize execution dynamically while maintaining consistent user experiences.

The Universal Microservices Architecture (UMA) meets these requirements by introducing a service model where each microservice is autonomous, self-contained, and capable of asynchronous communication. This enables dynamic orchestration and reuse across platforms while supporting robust and scalable application design.

2.5 Key Benefits of Universal Microservices Architecture

- **Modularity**: Each service encapsulates a discrete business capability, making the system easier to develop, reason about, and evolve.
- **Scalability**: Services can be scaled independently, allowing for fine-grained resource management and efficient performance tuning.
- **Technological Flexibility**: Teams can utilize different languages or frameworks for each service, selecting the most suitable tool for the task.
- **Continuous Delivery**: UMA supports frequent updates and streamlined deployment pipelines, reducing time to market.
- **Fault Isolation**: Isolated failures improve system resilience, ensuring one failing service does not compromise the entire application.
- **Team Productivity**: Parallel development across services accelerates delivery and improves collaboration.
- **Reusability**: Services are designed to be reused across multiple applications, reducing duplication and maximizing the return on development investment.

3. Goals and Objectives

3.1 Problem Statement

As demand increases for applications that operate consistently across operating systems and device types, development teams face a range of challenges that hinder scalability, performance, and user experience. These challenges include:

- **Cross-Platform Logic Reuse**: Reusing business logic seamlessly across client-side applications and backend services, irrespective of operating system or device.
  Performance Constraints: Addressing CPU and memory limitations on client hardware to ensure responsive, high-performing applications.
- **Consistency Across Clients**: Ensuring business logic and capabilities remain consistent across platforms to provide a unified user experience.
- **Code Modularity**: Promoting reuse and reducing redundancy through modular, maintainable components.
- **Target-Specific Flexibility**: Supporting low-level implementations via device-specific SDKs or libraries without compromising reusability.
  Workflow Decomposition: Breaking complex workflows into smaller, composable services to reduce system complexity and enhance flexibility.
  Atomic Design Principles: Focusing on small, self-contained services to improve testability, reliability, and ease of automation.

3.2 Strategic Importance

Solving these challenges is crucial for building a resilient and scalable software architecture that meets the demands of modern development. The ability to reuse logic and components across platforms reduces duplication and accelerates delivery. Mitigating hardware constraints ensures optimal performance regardless of device capabilities. Maintaining consistency across client experiences strengthens usability and

brand coherence. Moreover, modularization and service decomposition support agile development practices and facilitate long-term maintainability.

3.3 Architecture Objectives

The Universal Microservices Architecture (UMA) is designed to address these needs directly. The key objectives of UMA include:

- **Logic Reuse Across Targets**: Enable the consistent reuse of business logic across clients and backends, independent of platform or operating system.
- **Hardware-Aware Optimization**: Design strategies to overcome client-side resource limitations and deliver smooth performance.
- **Consistency of Experience**: Ensure consistent implementation of capabilities and logic across devices and interfaces.
- **Reusable Modular Components**: Promote modular design and reuse to reduce development time and improve maintainability.
- **Support for Low-Level Integrations**: Enable the integration of target-specific SDKs and native libraries to achieve advanced functionality while maintaining architectural cohesion.
- **Composable Workflows**: Replace monolithic architectures with modular workflows built from smaller, interchangeable services.
- **Atomic and Testable Services**: Focus on building atomic units of functionality that support automation, improve system reliability, and simplify testing.

By fulfilling these objectives, UMA enables the creation of high-quality, scalable applications that deliver consistent performance and user experience across the full spectrum of devices.

4. Architectural Overview

The Universal Microservices Architecture (UMA) represents more than a natural progression of the Client-Side Microservices Architecture (CSMA). It is a deliberate rethinking of how modular systems can be designed to meet the demands of cross-platform, device-independent software development. While CSMA provides foundational principles, UMA expands their scope to address a broader set of challenges, especially those related to portability, dynamic execution, and runtime distribution.

In recent years, the complexity of building advanced capabilities has been significantly reduced. Modern frameworks and AI-assisted development tools are making the creation of new features faster and more accessible than ever. As AI-driven development continues to evolve, the barrier to building standalone components will continue to fall. However, the complexity of managing software at scale remains a significant burden.

Teams still spend a disproportionate amount of time maintaining lifecycles, propagating changes, rebuilding common functionality across stacks, and integrating features into legacy or monolithic systems. This effort becomes even more daunting when applications must support diverse platforms, each with its own hardware constraints and user ergonomics.

UMA directly addresses this friction by enabling portable microservices to run on a universal runtime, whether on the client or backend. It relies on asynchronous, event-driven communication to coordinate execution across the system, optimizing for performance, scalability, and modularity.

<diagram 1>

4.1 Structural Components of UMA

The UMA model cleanly separates responsibilities between the business logic and user interface layers. It introduces a three-tiered runtime structure, focused exclusively on business logic, supported by a microservice registry. The architecture is composed of the following key components:

- **Universal Runtime Application**
  This is the core environment for executing business logic on the client. It handles all service interactions, including those with the backend, and abstracts platform-specific details. By isolating business logic from presentation layers, this runtime remains agnostic to operating systems and deployment targets. Its primary role is to coordinate logic execution, enforce modular boundaries, and serve as the orchestration layer for local microservices.
- **Server-Side Low Latency Runtime**
  In situations where edge execution is insufficient due to security constraints, heavy computational demands, or policy requirements, the system can delegate specific services to a server-side runtime. This runtime instantiates temporary microservices as needed and maintains communication through a server-to-client event subscription model. The result is a flexible execution fabric that adapts to contextual demands without compromising performance or consistency.
- **Microservice Registry**
  Both the Universal Runtime Application and the Server-Side Low Latency Runtime retrieve services from a shared microservice registry. This registry provides secure, versioned APIs to fetch service binaries or metadata. It ensures that both client and server runtimes remain synchronized and capable of instantiating on-demand services, promoting dynamic and decentralized execution.
- **UI Components (Outside UMA Scope)**
  UMA explicitly decouples business logic from user interfaces. UI components are responsible solely for rendering data and capturing user input. All processing,

validation, and coordination are delegated to the universal runtime, allowing frontends to remain lightweight, reusable, and focused exclusively on delivering an optimal user experience.

<diagram 2>

4.2 Data Management

In the Universal Microservices Architecture (UMA), each microservice maintains ownership over its own data. Rather than relying on centralized state management, services independently retrieve the data they require through backend API calls and maintain a local cache during runtime. This decentralized approach enhances modularity, reduces inter-service coupling, and enables services to operate autonomously.

By isolating data responsibilities within each microservice, UMA supports greater fault tolerance and simplifies testing and debugging. Localized caching also improves performance by reducing unnecessary network requests, particularly in bandwidth-constrained environments.

4.3 Communication Model

Communication in UMA is based on an event-driven architecture that enables asynchronous interactions between microservices. This applies both within the Universal Runtime Application on the client and across the boundary between the client and server.

UMA utilizes **event brokers** to facilitate loosely coupled communication, enabling services to publish and subscribe to events without requiring direct dependencies. This model promotes scalability, resiliency, and service independence, which are essential for building complex, distributed applications.

Solutions	Description	Pros	Cons
WebSockets	A communication protocol providing full-duplex, bidirectional channels over a single TCP connection.	**Bidirectional Communication:** Enables real-time, two-way interaction between the client and server. **Low Latency:** Reduces latency compared to HTTP-based protocols. **Efficient:** Less overhead per message compared to HTTP.	**Compatibility:** Not all environments (e.g., firewalls, proxies) support WebSocket. **Security:** Requires careful handling to ensure secure communication.
gRPC	A high-performance RPC framework using HTTP/2 and Protocol Buffers for efficient communication.	**Performance:** Utilizes HTTP/2, offering improved performance and reduced latency. **Strong Typing:** Uses Protocol Buffers for defining messages, ensuring type safety. **Bidirectional Streaming:** Supports client-server and server-client streaming. **Efficient:** Smaller message sizes compared to JSON **Language Support:** Supports multiple languages.	**Browser Support:** Limited direct support in web browsers (requires a proxy layer). **Learning Curve:** Requires understanding of Protocol Buffers and gRPC specifics.
SSE (Server-Sent Events)	A protocol enabling servers to push updates to clients over a single, long-lived HTTP connection.	**Efficient:** Designed for streaming events from the server to the client. **Browser Support:** Well-supported by most modern browsers.	**Unidirectional:** Only supports server-to-client communication. **Scalability:** May not be suitable for high-load scenarios. **Compatibility:** Limited compatibility with older browsers
Long Polling	A technique where the client repeatedly requests data from the server, maintaining a continuous connection.	**Simplicity:** Easy to implement using basic HTTP. **Compatibility:** Works with most browsers without special support.	**Inefficiency:** Can be inefficient due to the repeated nature of requests.

	Each of these technologies serves different purposes and is suited to different types of event-driven applications.	**Fallback:** Can be used as a fallback for WebSocket.	**Latency:** Higher latency compared to WebSocket or SSE. **Overhead:** Increased server load due to frequent connections.

Table 1: *This table is designed to help decision-makers evaluate the suitability of various communication systems for their specific needs, considering factors such as performance, resource usage, scalability, and implementation complexity.*

Platform Considerations for Client-Server Communication

When evaluating communication technologies for iOS and Android applications, it is essential to consider factors such as platform support, available libraries, performance, and energy efficiency. Below is a summary of commonly used protocols and their respective trade-offs:

- **WebSocket**
    - *Libraries*: Well-supported by libraries such as **Starscream** for iOS and **OkHttp** for Android.
    - *Battery Consumption*: Maintaining a constant connection can impact battery life.
    - *Notes*: Ideal for real-time, bi-directional communication. Commonly used for chat systems, notifications, and live data feeds.
- **gRPC**
    - *Libraries*: Supported by **grpc-swift** on iOS and **grpc-java** on Android.
    - *Battery Consumption*: More efficient than other protocols, especially for high-frequency or structured data exchange.
    - *Notes*: A strong choice for mobile applications requiring performance, stream support, and low-overhead communication.
- **Server-Sent Events (SSE)**

    - *Libraries*: Less commonly supported in mobile environments. Alternatives like WebSocket are typically preferred.
    - *Battery Consumption*: Efficient for one-way server-to-client communication.
    - *Notes*: Suitable for streaming data such as logs or updates, but its unidirectional nature limits broader use in mobile apps.
- **Long Polling**
    - *Libraries*: Implemented using standard HTTP libraries like **URLSession** for iOS and **OkHttp** for Android.
    - *Battery Consumption*: Generally inefficient due to frequent polling and reconnection overhead.
    - *Notes*: Easy to implement and compatible with most environments, but best used as a fallback when other protocols are unavailable.

Each protocol presents unique advantages and limitations. UMA supports flexible communication layers, allowing mobile clients to adopt the protocol best suited for their platform, use case, and power profile.

4.4 Eventing Data Model

Event-driven architecture is a core enabler of responsive, interactive applications in UMA. This section outlines the UMA eventing model, which facilitates real-time communication between clients (including web, iOS, and Android) and the server-side microservices. UMA supports communication over protocols such as WebSocket and gRPC, enabling cross-platform interoperability and low-latency interactions.

To ensure consistency and seamless integration across diverse platforms and runtimes, UMA adopts a standardized data model for all event exchanges. This model is built on the CloudEvents specification, providing a common schema for structuring and interpreting event data.

4.4.1 Event Data Structure

At the heart of UMA's eventing system is a well-defined schema that describes how events are structured and transmitted. The **CloudEvents** specification defines a standard set of attributes that make event data portable, traceable, and easy to consume across services and runtimes.

This schema provides a foundation for consistent event modelling across all UMA clients and microservices. Additional guidance and implementation details can be found in PDMS-RFC-MFE Standards – Eventing and Messaging.

Attribute	Type	Required	Description	Rationale
`id`	string	✅	Unique identifier for the event	Essential for event tracking and deduplication
`type`	string	✅	Describes the type of event	Crucial for event routing and handling
`specVersion`	string	✅	Version of the CloudEvents spec	Ensures compatibility and versioning
`source`	URI	✅	Identifies the context that emitted the event	Provides origin information, using URI format for consistency
`time`	timestamp	✅	Time of the occurrence of the event	Important for event ordering and tracking
`data`	object	✅	The event payload	Contains the actual event data
`subject`	URI	❌	Describes the subject of the event	Useful for additional context, using the URI format for consistency
`dataContentType`	string	❌	Content type of the data attribute	Helps in parsing the data payload
`dataSchema`	URI	❌	The schema that the data adheres to	Useful for data validation, especially in complex scenarios

Table 2: This table defines the UMA event data structure in accordance with the CloudEvents specification. It outlines key attributes, their types, and whether they are required. Adopting CloudEvents ensures a consistent and interoperable format for event-driven communication.

```
{
```

```
  "specversion": "1.0",
  "id": "12345",
  "source": "/web/button",
  "type": "com.example.event.click",
  "time": "2025-03-25T13:40:29.097963Z",
  "data": {
    "button_id": "submit"
  },
  "subject": "user789",
  "datacontenttype": "application/json"
}
```

4.4.2 Common Interface for Event Management

To ensure consistent, real-time communication across platforms, UMA defines a **common event management interface** that operates over either **WebSocket** or **gRPC**. These protocols provide low-latency, bi-directional communication channels, making them well-suited for event-driven interactions between client applications (such as web, iOS, and Android) and the server-side microservices.

This interface supports two primary communication paths:

- **Client-side**: Within the Universal Runtime Application, enabling modular microservices to exchange events.
- **Client-server**: Between client applications and the Server-Side Low Latency Runtime, enabling cross-boundary orchestration.

By standardizing around WebSocket and gRPC, UMA enables clients to select the protocol most appropriate for their platform and workload, while maintaining a unified event structure based on the CloudEvents specification.

4.4.2.1 WebSocket Interface

WebSocket provides a persistent, full-duplex communication channel over a single TCP connection. It is particularly effective for high-frequency, real-time scenarios such as user interactions, UI state updates, and background synchronization.

The UMA WebSocket interface is composed of the following key elements:

- **Connection Establishment**

  Clients initiate and maintain a persistent WebSocket connection with the UMA server. This connection supports ongoing event exchange without repeated handshakes.
- **Event Transmission**

  Events are serialized as JSON and structured according to the CloudEvents specification. Each message transmitted through the WebSocket channel includes a full event payload with metadata and content.
- **Authentication**

  The interface supports secure, token-based authentication to ensure that only authorized clients can establish and maintain event channels.
- **Error Handling**

  The system includes mechanisms for detecting and managing connection interruptions, malformed payloads, and authentication errors. Clients receive structured error events to support graceful degradation and recovery.

This WebSocket-based interface enables UMA applications to deliver responsive, real-time user experiences while minimizing overhead and preserving compatibility across platforms.

Example WebSocket Event Transmission:

```
{
  "specversion": "1.0",
  "id": "12345",
  "source": "/web/button",
  "type": "com.example.event.click",
  "time": "2025-03-25T13:40:29.097963Z",
  "data": {
    "button_id": "submit"
  },
  "subject": "user789",
  "datacontenttype": "application/json"
}
```

4.4.2.2 gRPC Interface

gRPC is an open-source remote procedure call (RPC) framework designed for high-performance, real-time communication. It uses **Protocol Buffers (Protobuf)** for compact and efficient message serialization, making it well-suited for mobile and low-latency environments. In UMA, gRPC is particularly effective for scenarios that require **bi-directional streaming**, structured contracts, and low overhead across heterogeneous platforms.

The UMA gRPC interface includes the following key components:

- **Service Definition**
  UMA services are defined using `.proto` files, which specify event transmission methods and data structures. This provides a strongly typed, versioned contract for communication between clients and the Server-Side Low Latency Runtime.
- **Event Serialization**
  Events conforming to the CloudEvents specification are serialized using Protobuf. This ensures efficient encoding, reduced payload sizes, and cross-language interoperability while preserving event schema integrity.
- **Authentication**
  UMA supports secure communication over gRPC using industry-standard protocols, including **TLS** for encrypted transport and **OAuth 2.0** for token-based client authentication.
- **Error Handling**
  The gRPC interface includes built-in mechanisms for error propagation and structured response codes. UMA services can gracefully handle transmission failures, malformed payloads, and authorization errors, enabling robust recovery workflows on both the client-side and server-side.

Example gRPC Service Definition:

```
syntax = "proto3";

service EventService {
```

```
  rpc SendEvent (CloudEvent) returns (EventResponse) {}
  rpc StreamEvents (stream CloudEvent) returns (stream EventResponse) {}
}

message CloudEvent {
  string specversion = 1;
  string id = 2;
  string source = 3;
  string type = 4;
  string time = 5;
  map<string, string> data = 6;
  string subject = 7;
  string datacontenttype = 8;
}

message EventResponse {
  string status = 1;
  string message = 2;
}
```

4.4.2 Event Processing on the Server

Once an event is received by the UMA server, it enters a structured processing pipeline designed to validate, store, route, and act upon the incoming data. This process enables UMA to respond dynamically to real-time inputs and maintain a cohesive flow across distributed microservices.

Key steps in server-side event processing include:

- Event Validation

  Incoming events are first validated against the CloudEvents schema to ensure structural integrity, required fields, and conformance to expected formats. This step is critical to prevent malformed or incomplete data from disrupting downstream workflows.

- Event Storage

  Validated events are optionally persisted in a structured datastore for auditing, replay, historical analysis, or downstream analytics. This enables traceability and supports time-series evaluation of user and system behaviour.

- Event Handling

  Based on the event type and source, UMA invokes appropriate workflows or microservices. This may include triggering business rules, updating system state, or broadcasting follow-up events to other services.

- Microservices Communication

  Events are routed internally using APIs or asynchronous messaging systems (e.g., lightweight queues or service buses). This decouples services, ensuring scalable coordination without tight dependencies.

- Analytics and Insights

  Event data is aggregated and analyzed to extract insights about application performance, user behaviour, or system health. These insights can be utilized for real-time dashboards, anomaly detection, or optimizing service execution paths.

5. Components and Modules

5.1 Universal Microservices

5.1.1 Definition

Universal microservices are modular, independently deployable units of business logic designed to run across both client-side and server-side environments. Unlike traditional microservices, which are often tied to specific operating systems, platforms, or stacks, universal microservices are inherently OS-agnostic. They rely on a runtime abstraction layer to handle platform-specific operations such as I/O, networking, or file access.

This design enables true portability and reusability, allowing the same microservice to be executed in multiple contexts—whether embedded in a web client, running in a mobile environment, or deployed in a server-side runtime.

Unlike monolithic or tightly coupled application logic, universal microservices are self-contained and autonomous. Each service can be developed, versioned, deployed, and scaled independently without requiring coordination with other services. This autonomy enhances maintainability, testing, and operational agility.

By adopting universal microservices, organizations gain a significant architectural advantage:

- Improved scalability and flexibility
- Shortened development cycles
- Increased code reuse across environments
- Enhanced system resilience through fault isolation

5.1.2.1 Stateless Services

Stateless services represent a key subclass of universal microservices. These services are designed to execute complex operations without maintaining internal state across invocations. They are particularly well-suited for asynchronous workloads and parallel

execution, and can be run interchangeably on the client or server depending on runtime conditions.

Stateless services rely on the UMA runtime's abstraction layer to perform any platform-specific operations, thereby ensuring portability across various environments.

Core characteristics:

- No retained state between executions
- Highly scalable due to parallelization potential
- Operate asynchronously without managing threads directly.
- Can offload execution to the server when client resources are constrained

Examples include:

- A microservice that calculates mathematical operations such as factorials
- An image processing service that generates thumbnails
- A fire-and-forget logger that asynchronously queues analytics or telemetry events

```
class FactorialCalculator {
    private eventDispatcher = new EventDispatcher<{result: number}>();

    public calculateFactorial = (number: number): void => {
        const factorial = (n: number): number => (n <= 1 ? 1 : n *
factorial(n - 1));
        const resultValue = factorial(number);
        this.eventDispatcher.dispatch({result: resultValue});
    };

    public subscribe(callback: (data: {result: number}) => void): void {
        this.eventDispatcher.subscribe(callback);
    }

    public unsubscribe(callback: (data: {result: number}) => void): void {
        this.eventDispatcher.unsubscribe(callback);
    }
}
```

5.1.2.2 Subscribable Services

Subscribable services are a specialized type of universal microservice designed to be discoverable and activated dynamically in response to subscription requests from other services or UI components. These services are event-aware but do not maintain internal state. Instead, they follow an aggregation pattern, collecting or computing data on demand from other services, sensors, or runtime contexts.

Like other UMA services, subscribable services rely on the runtime's abstraction layer to perform OS- or platform-specific operations, enabling seamless operation across device and system boundaries.

Key characteristics:

- On-demand activation: Instantiated only when a subscription is made by another component or service
- Abstraction-driven: Use the UMA runtime for all platform-specific interactions
- Aggregation-focused: Consolidate or transform data provided by other sources
- Stateless: Do not store or retain data between invocations

Examples include:

- A notification service that emits updates when subscribed users request event notifications
- A layout management service that adapts and publishes UI layout types based on screen size and orientation
- A connectivity service that determines application network status and broadcasts changes to subscribed components

Subscribable services are foundational to UMA's dynamic, event-driven architecture. They enable real-time adaptation to runtime conditions while maintaining modularity and decoupling. These services are especially useful in reactive interfaces, responsive design patterns, and adaptive system behaviours.

Use Case Example:

```
class NetworkStatusService {
    private eventDispatcher = new EventDispatcher<{status: string}>();
    private status: string = 'offline';

    constructor(private serviceA: NetworkServiceA, private serviceB:
NetworkServiceB) {
        this.subscribeToNetworkServices();
    }

    private subscribeToNetworkServices(): void {
        this.serviceA.subscribe(data => this.updateStatus(data));
        this.serviceB.subscribe(data => this.updateStatus(data));
    }

    private updateStatus(data: {status: string}): void {
        // Determine the status based on the data from the two services
        if (data.status === 'online') {
            this.status = 'online';
        } else if (this.serviceA.getStatus() === 'online' ||
this.serviceB.getStatus() === 'online') {
            this.status = 'online';
        } else {
            this.status = 'offline';
        }
        this.eventDispatcher.dispatch({status: this.status});
    }

    public subscribe(callback: (data: {status: string}) => void): void {
        this.eventDispatcher.subscribe(callback);
    }

    public unsubscribe(callback: (data: {status: string}) => void): void {
        this.eventDispatcher.unsubscribe(callback);
    }

    public getStatus(): string {
        return this.status;
    }
}
```

5.1.2.2 Stateful Services

Stateful services are universal microservices designed to retain information over time and maintain a defined lifecycle. Unlike stateless or subscribable services, they are

explicitly instantiated and managed by the UMA service manager, and they remain active across interactions as long as needed by the application.

These services utilize the UMA runtime's abstraction layer to perform all storage, I/O, and networking operations, ensuring they remain target-agnostic and portable across platforms. Their ability to store and manage internal state makes them essential for handling complex workflows, session management, and data persistence within client or server runtimes.

Key characteristics:

- Maintain internal state and lifecycle across invocations
- Rely on the abstraction layer for platform-specific operations and storage
- Instantiated explicitly by the service manager
- Discoverable and accessible by other services within the UMA runtime

Examples include:

- A state machine service that manages the application's current view, navigation state, or user session
  A configuration service that stores runtime values or feature flags
- A service with database-like behaviour, capable of maintaining structured records and exposing CRUD operations

Stateful services are crucial for scenarios that require persistence, coordination, or tracking across the user's journey. By isolating states within modular, discoverable units, UMA enables clear state boundaries, improved testability, and scalable design without compromising portability or flexibility.

```
import { Storage } from 'universal-runtime';
import { EventDispatcher } from './EventDispatcher'; // Assuming
EventDispatcher is implemented and imported
```

```
class StateManager {
    private eventDispatcher = new EventDispatcher<{ state: { [key: string]:
any } }>();
    private storage: Storage;

    constructor(storage: Storage) {
        this.storage = storage;
    }

    public setState(key: string, value: any): void {
        this.storage.setItem(key, value);
        this.notifySubscribers();
    }

    public getState(key: string): any {
        return this.storage.getItem(key);
    }

    private notifySubscribers(): void {
        const state: { [key: string]: any } = {};
        // Simulate reading all keys and values from storage
        state['user'] = this.storage.getItem('user');
        state['theme'] = this.storage.getItem('theme');
        this.eventDispatcher.dispatch({ state });
    }

    public subscribe(callback: (data: { state: { [key: string]: any } }) =>
void): void {
        this.eventDispatcher.subscribe(callback);
    }

    public unsubscribe(callback: (data: { state: { [key: string]: any } })
=> void): void {
        this.eventDispatcher.unsubscribe(callback);
    }
}
```

5.1.2 Anatomy

Universal microservices are built to be portable and adaptable across a wide range of platforms and environments. Understanding their internal structure is essential for developers working within the Universal Microservices Architecture (UMA).

This section provides an overview of the core elements that comprise a universal microservice, illustrating how these components contribute to its reusability, modularity, and platform independence.

<Figure 3>

5.1.2.1 Business Logic

At the heart of every universal microservice is its business logic—the core functionality the service is designed to provide. Each service should adhere to the single-responsibility principle, maintaining a clear and focused purpose. Developers must strike a balance between functionality and simplicity to ensure the service remains maintainable and reusable.

Business logic should be abstract and broadly applicable, allowing it to be reused across different areas of an application or in multiple projects. To ensure platform independence, it must avoid reliance on any OS-specific APIs or SDKs. Instead, all platform-dependent functionality should be delegated to the runtime abstraction layer.

5.1.2.2 Runtime Abstraction Layer

The runtime abstraction layer enables universal microservices to interact with their environment in a platform-agnostic way. It provides standardized APIs and utility functions that allow services to manage their own lifecycle, communicate with other services, and perform operations such as event handling, networking, and I/O.

This abstraction ensures that services remain decoupled from specific platforms or operating systems, supporting full portability and reusability across client-side and server-side runtimes. It also simplifies service development by exposing consistent interfaces for common operations.

5.1.2.3 Service Interface Collection

The service interface collection, also referred to as the service schema, defines the public API implemented by the business logic. It serves as a formal contract between the

service and its consumers, outlining the methods and properties available for integration.

By explicitly defining these interfaces, UMA ensures that services remain discoverable, predictable, and interoperable across different components within the system.

5.1.2.4 Service Metadata

Metadata plays a vital role in the lifecycle of universal microservices. Each service includes a structured metadata file that provides key information such as its name, description, author, version, dependencies, and service type.

This metadata facilitates service registration, discovery, deployment, and maintenance. It also enhances developer productivity by providing insight into how the service is intended to be used and the capabilities it offers.

5.1.2.5 Data Management

For stateful services, data management is a core responsibility. These services use the runtime abstraction layer to interact with local storage, enabling them to persist and manage their own internal state. This localized data ownership supports encapsulation while still allowing relevant information to be shared with other services when needed.

By isolating data management within the service boundary, UMA improves modularity and simplifies debugging, testing, and scaling.

5.1.2.6 Unit Testing

Robust unit testing is essential for ensuring the reliability of universal microservices. Due to the modular and distributed nature of UMA, high test coverage of business logic is critical for maintaining system stability.

Well-defined unit tests verify the correctness of individual services and help prevent regressions during updates. They also enable faster iteration and support continuous integration and deployment workflows.

5.1.3 Development Process

Developing universal microservices requires a structured and disciplined approach. This process emphasizes:

- Language-agnostic development, enabling service creation in multiple programming languages
- Independent testing and debugging of isolated components
- Use of automation for build, test, and deployment pipelines
- Implementation of semantic versioning for service lifecycle management
  Compilation into portable binaries for cross-platform execution
  Integration and contract testing to validate service compatibility and behaviour within the system

Following this process ensures that each microservice is portable, reliable, and consistent with UMA architectural standards.

<figure 4>

5.1.3.1 Multi-Language Development

Universal microservices enable developers to implement services in various programming languages, allowing teams to leverage the specific strengths, ecosystems, and efficiencies of each. This flexibility empowers teams to choose the most suitable language for each microservice, optimizing for performance, expressiveness, or familiarity.

By supporting a polyglot development model, UMA enhances agility, maintainability, and developer productivity. Teams can work with the languages they know best, which fosters collaboration, increases velocity, and improves code quality. The runtime abstraction layer ensures that services written in different languages can interoperate seamlessly, enabling cohesive execution across various environments.

5.1.3.2 Independent Testing and Debugging

To ensure reliability before integration, each microservice must be thoroughly tested and debugged in isolation. Developers use language-specific unit testing frameworks and debugging tools to verify behaviour and detect defects early in the development process.

Independent testing not only validates service correctness but also promotes better modular design by encouraging developers to think about service boundaries and failure modes in isolation.

5.1.3.3 Automation

Automation is central to modern microservice development workflows. Continuous integration (CI) and deployment (CD) pipelines should include automated steps for building, testing, linting, type checking, and validating code quality.

Tools such as **SonarQube** can be integrated to monitor code complexity and enforce quality gates. Automation also extends to documentation generation—service interfaces can be used to auto-generate accurate, up-to-date API documentation, eliminating the need for manual updates.

5.1.3.4 Semantic Versioning

Semantic versioning is essential for managing changes to microservices in a structured and predictable way. By clearly defining API contracts and assigning version numbers according to the nature of the changes—**patch**, **minor**, or **major**—developers can communicate expectations and avoid breaking dependent services.

Semantic versioning facilitates a stable evolution of the system, allowing teams to upgrade services with confidence and maintain backward compatibility as needed.

5.1.3.5 Portable Binaries

Microservices should be compiled into **portable binaries** that are independent of specific operating systems or target environments. This is fundamental to UMA's goal of device-independent execution.

WebAssembly (WASM) provides an ideal foundation for this portability. By compiling services into WASM binaries, developers can ensure cross-platform compatibility and performance consistency across client and server environments.

5.1.3.6 Integration and Contract Testing

While unit testing validates services in isolation, **integration and contract testing** verify that services interact correctly with others in the system. These tests should be part of the automated CI pipeline and executed regularly to ensure service compatibility and prevent regressions.

Contract testing ensures that services adhere to agreed-upon interfaces, thereby reducing the risk of miscommunication and enabling the safe reuse of services across teams and projects.

5.1.3.7 Quality Assurance and Trust

Maintaining high standards of quality is essential for the reliability and scalability of UMA-based systems. This includes enforcing best practices in code quality, versioning, documentation, and testing.

Establishing a culture of trust and shared accountability across teams encourages the adoption of shared services and accelerates development. Quality gates integrated into the development pipeline ensure that each service meets defined standards before being published or deployed.

5.1.3.8 Modularity and Loose Coupling

Universal microservices should be designed to be **atomic**, **modular**, and **loosely coupled**. Services must remain independent of specific frameworks, runtimes, or deployment targets.

5.2.1 Definition

The Universal Runtime is the central component of UMA. It manages and executes all business logic for client applications while maintaining portability across various targets and operating systems. The runtime is composed of several core components, each with a distinct responsibility. These components are designed to ensure performance, security, and portability. With a universal runtime in place, developers can build applications for multiple platforms using a shared set of workflows and capabilities maintained by other teams.

<figure 5>

5.2.2 Anatomy

5.2.2.1 Service Loader

The Service Loader is purpose-built for UMA and differs from traditional client-side microservices architectures. It is aware of the current runtime state as well as the hardware and capabilities of the client device. By analyzing factors such as application context, CPU load, and available memory, the algorithm can determine whether a specific service should be executed locally or delegated to the server.

This decision-making logic is configurable and can be adjusted based on the specific needs of the application. Some services may require server-side execution due to performance, security, or regional requirements. Configurability patterns allow these rules to be updated dynamically. The Service Loader can also enforce local execution for services that rely on target-specific APIs or SDKs, such as native iOS binaries using the Metal SDK or browser-dependent services utilizing web-specific APIs.

5.2.2.2 Client Service Manager

When the Service Loader decides that a service should run on the client, the Client Service Manager takes over. It performs two primary functions: fetching services from the registry on demand as the application context evolves, and managing service lifecycle operations such as initialization and disposal based on the service type, whether stateful, stateless, or subscribable.

The Service Manager also automatically handles dependencies. For example, when a "cart" service is needed, it will also load any required supporting services, such as the "user session service" and "ordering service".

5.2.2.3 Thread Manager

Multithreading is a fundamental requirement for UMA-based architectures. Client-side applications, especially in enterprise settings, can be large and complex, with multiple teams contributing asynchronously. Providing a built-in multithreading solution helps avoid fragmentation and ensures consistent performance management across services.

UMA ensures thread safety when multiple services run in parallel. Threads are isolated and atomic if a thread crashes, it is safely terminated without affecting the rest of the application. Services can be prioritized based on criticality, and threads can be dynamically boosted by allocating additional CPU resources when needed.

When designing the runtime, teams must consider target platforms, operating systems, and the types of binaries they intend to support. The multithreading layer must support both portable binaries, such as WASM, and native binaries, such as iOS SDKs or JavaScript bundles. The abstraction layer will handle stack-specific APIs without creating direct dependencies.

5.2.2.4 Client-side Microservice Event Manager

UMA uses an event-driven model for microservice communication. Each microservice acts as an independent unit that subscribes to events from other services and emits

events to its subscribers. This includes event producers that emit streams, consumers that listen for updates, and channels or brokers that route events between them.

UMA does not prescribe a specific event handling mechanism, allowing teams to choose implementations that best fit their stack and operating environment. Options may include pub/sub systems, in-memory message brokers, or other event channel models.

Strong typing of event APIs and clear contract definitions between services are critical. These reduce orchestration overhead and improve error handling. As with the Thread Manager, the abstraction layer manages platform-specific integrations while keeping implementations decoupled from each other.

5.2.2.5 Abstraction Layer

True portability for universal microservices requires complete separation from OS-specific SDKs and APIs. A reusable service that depends on a browser API, such as `window,` for networking would not function on other platforms.

The abstraction layer solves this by cleanly separating platform-specific implementations from service logic. It allows developers to write services once, while the runtime provides the appropriate underlying APIs depending on the platform. This approach enhances consistency, minimizes duplication, and reduces long-term maintenance costs.

The abstraction layer may be divided into:

- **Platform capabilities**: Reusable across applications
- **Core capabilities**: Varying by operating system or target

To fully leverage this model, teams should implement a build pipeline that can generate applications for various targets from a single runtime. During the build process, the system selects the appropriate low-level APIs and SDKs for the given platform. This process should be transparent to developers working on services or runtime logic, allowing them to focus exclusively on business logic.

Established design patterns should guide the architecture of the abstraction layer. The **Bridge pattern** is especially useful for scaling across multiple implementations. Comparing patterns may help determine the most suitable approach for specific requirements.

Pattern	Pros	Cons
Bridge	• Decouples abstraction from implementation, allowing independent changes. • Easier to extend, since new implementations can be added without modifying the abstraction. • Encourages modularity by keeping abstraction and implementation separate. • Works well for cross-platform systems (e.g., OS-specific APIs).	• Increases complexity by introducing multiple layers. • Might not be needed if implementations are simple and don't require independent changes.
Adapter	• Allows integration of existing APIs without modifying them. • Enables the reuse of legacy code by making it compatible with a new interface. • Good for third-party APIs that don't follow your system's design.	• Adds overhead since it requires extra wrapping logic. • It can become complex when adapting multiple interfaces or deep hierarchies. • Not flexible for future changes—each new incompatible API requires a new adapter.
Abstract Factory	• Encapsulates object creation, making it easy to switch between OS-specific implementations. • Provides strong separation of concerns, keeping client code independent from concrete implementations. • Good for systems needing different configurations (e.g., Windows, Linux, macOS).	• Adds complexity by requiring multiple factory classes. • Not always necessary if the number of variations is small. • Can lead to an excessive number of classes, making the codebase more difficult to manage.
Facade	• Simplifies complex APIs by providing a single entry point. • Hides OS-specific details, making the client code cleaner and more user-friendly. • Improves maintainability by reducing direct dependencies on multiple low-level APIs.	• Introduces tight coupling between the client and the facade. • Modifying underlying components less flexibly may require changes to the facade. • It can become a "God Object" if it tries to do too much.

Key Takeaways: Selecting the Appropriate Approach

- **Bridge**: Choose this pattern when you need a scalable abstraction layer that cleanly separates service logic from platform-specific implementations. It is ideal

for supporting multiple OS-specific integrations while maintaining a consistent interface.

Adapter: Use this pattern when you need to make an existing API compatible with a new or different interface. It is best suited for integrating legacy or third-party components without changing their underlying code.

- **Abstract Factory**: This pattern is useful when you need to create OS-specific objects at runtime while keeping the client code independent of the specific implementations. It allows for greater flexibility and modularity in cross-platform scenarios.
- **Facade**: Opt for this pattern when you want to simplify interactions with complex or verbose OS-specific APIs. It provides a cleaner and more manageable interface for high-level service consumption.

5.3 Server-side Low Latency Runtime

5.3.1 Definition

In the previous section, we defined and analyzed the Universal Client Runtime, which supports reuse across multiple operating systems and client-side targets. This section focuses on the **Server-side Low low-latency runtime**, outlining its requirements, responsibilities, and essential components.

The server-side runtime plays a critical role by enabling the client to delegate the execution of microservices that encapsulate business logic not tied to a specific device or operating system. To support this functionality effectively, the server-side runtime must meet several key requirements.

First, it must be capable of retrieving the same service binaries from the registry that client applications use, ensuring that identical business logic can be executed securely in both environments. It must also expose APIs that allow the client to request the execution of specific services on the server.

The server-side runtime should support **portable binaries**, particularly WebAssembly (WASM), to ensure compatibility with the variety of platforms used by clients. It must provide a runtime abstraction layer similar to the client, offering consistent support for low-level operations such as I/O, networking, eventing, storage, and database access.

Communication between the client and server must be reliable and efficient. To support both web and mobile clients, the runtime should support **gRPC** and **WebSocket** protocols for real-time, bidirectional messaging. It must also be optimized for **low latency** and **cost efficiency** to ensure scalability and performance in production environments.

Reliability and resilience are essential characteristics. The server-side runtime must ensure a seamless experience for the end user, whether services are executed locally on the client or remotely on the server. While many features are shared with the Universal Client Runtime, it is recommended to implement a dedicated runtime on the server to accommodate the specific operational and performance needs of the server environment.

5.3.2 Anatomy

<Figure 6>

5.3.2.1 API Gateway

The API Gateway serves as the entry point for clients requesting the execution of microservices. It handles universal client runtime requests, ensuring efficient routing to the Service Loader while managing authentication, request validation, and response handling.

Clients use this API to delegate the execution of a microservice to the server, receiving an execution ID that allows tracking of the service's lifecycle asynchronously. The API also provides clients with the necessary connection details to the Event Broker, enabling event-driven communication once the service is running.

Functional Responsibilities

- Receive Execution Requests: Accept client requests to start a microservice.
- Delegate to the Service Loader: Fetch the requested microservice and initialize execution.

- Provide Execution Status: Inform the client when the service is up and running.
- Enable Event-Driven Communication: Provide connection details for the event broker (e.g., WebSockets or gRPC).
- Handle Asynchronous Workflows: Ensure low-latency responses and non-blocking service execution to optimize performance.

API Workflow & Request Lifecycle

1. Client Sends Execution Request
    - The API Gateway receives a request to execute a microservice.
    - The request includes the service ID and input parameters.
2. API Gateway Delegates Execution
    - Calls the Service Loader to fetch and start the microservice.
    - Immediately responds with an execution ID to the client.
3. Service Starts Asynchronously
    - The runtime initializes the service in a sandboxed execution environment.
    - The service registers itself with the Event Broker for communication.
4. Client Listens to Events
    - The API Gateway pushes an event once the service status changes.
5. API Gateway Provides Event Broker Connection Info
    - Once the service is running, the client receives the WebSocket or gRPC connection details to interact with the service via the Event Broker.

Handling Low Latency & Asynchronous Execution

Since service execution is asynchronous, latency is critical for a responsive system. The API Gateway must:

1. Respond Immediately to Execution Requests
    - Clients should receive an execution ID within 5-20 ms after request submission.
    - This ensures minimal delay before tracking execution progress.
2. Push Status Updates Instead of Polling
    - Instead of polling (GET /status/{execution_id}), use WebSockets/gRPC to notify clients when the service is ready.
    - This reduces unnecessary API calls and improves response time.
3. Use a High-Performance Event Broker
    - NATS, Redis Streams, or Kafka for fast, distributed event messaging.
    - WebSockets for real-time bidirectional communication with clients.

5.3.2.2 Service Loader

The Service Loader acts as an intermediary between the API Gateway and the Execution Engine, ensuring that requested microservices are fetched and queued for execution. It does not execute services but instead retrieves the necessary microservices from the registry, validates them, and places them in a queue, where they await execution by the Execution Engine.

This design ensures efficient workload management, allowing the Execution Engine to dynamically scale and process services as resources become available.

Key Responsibilities

1. Receiving Execution Requests
    - The Service Loader receives service execution requests from the API Gateway.
    - It extracts service identifiers, execution parameters, and metadata from the request.
2. Fetching Wasm Microservices
    - Queries the Microservice Registry to retrieve the associated binary with the requested service.
    - Implements caching strategies to avoid unnecessary fetch operations for frequently used services.
    - Ensures failover mechanisms in case the registry is unavailable.
3. Validating the Fetched Service
    - Performs checksum verification and integrity checks to ensure the Wasm module is secure and intact.
    - Ensures the service is compatible with the Execution Engine's runtime.
4. Queuing the Service for Execution
    - Places the validated binary service into a task queue, where it waits for execution.
    - Ensures FIFO (First-In, First-Out) order or priority-based queuing, if necessary.
5. Tracking Service Readiness
    - Notifies the API Gateway that the request has been queued for execution.
    - Exposes queue status metrics, allowing clients to track pending service executions.
6. Waiting for Execution Engine
    - The Service Loader does not run the service itself; instead, it waits for the Execution Engine to pull the next available service from the queue.
    - Once the Execution Engine picks up a service for execution, the Service Loader removes it from the queue and logs the event.

5.3.2.3 Execution Engine

The Execution Engine is responsible for executing WebAssembly microservices that the Service Loader has queued. It serves as the core runtime environment, ensuring that services run securely, efficiently, and in isolation while leveraging an abstraction layer to provide essential system capabilities.

The Execution Engine pulls execution requests from the queue, initializes services, and establishes event-driven communication between microservices and clients. Incorporating an abstraction layer enables services to interact with the underlying system in a controlled and standardized manner, ensuring portability and security.

Key Responsibilities

1. Fetching Execution Requests from the Queue
    - The Execution Engine continuously pulls the next available request from the queue managed by the Service Loader.
    - Ensures that services are executed in order or based on defined priority rules.
2. Initializing and Running Microservices
    - Loads the fetched binaries into an isolated execution environment.
    - Allocates necessary resources such as CPU, memory, and I/O permissions.
    - Ensures sandboxed execution, preventing microservices from interfering with each other.
3. Providing Core Capabilities via the Abstraction Layer
    - The Execution Engine imports and integrates an abstraction layer that offers core capabilities, including:
        - Eventing: Enables microservices to publish and subscribe to events.
        - I/O Operations: Allows controlled access to file systems and external data sources.
        - Networking: Supports secure outbound and inbound network communication.
        - Storage/Database Access: Provides structured access to databases and persistent storage.
        - Security & Authentication: Enforces identity and access controls for services.
    - The abstraction layer ensures that microservices can interact with system resources without direct low-level access, improving security and portability.
4. Establishing Event-Driven Communication

- Sets up an event broker using WebSockets, gRPC, or message queues to enable communication between services and clients.
- Provides an execution status API, allowing clients to monitor when a service is running.

5. Resource and Performance Management
    - Implements execution time limits to prevent long-running services from consuming excessive resources.
    - Supports auto-scaling by dynamically adjusting the number of running service instances.
    - Handles load balancing to distribute execution tasks efficiently across multiple instances.
6. Handling Failures and Service Termination
    - Monitors running services for failures or crashes and retries execution when necessary.
    - Ensures proper cleanup of resources after execution is completed.
    - If a service fails repeatedly, it is moved to a dead-letter queue (DLQ) for manual review and investigation.
7. Notifying Clients and Updating the API Gateway
    - Once execution begins, the API Gateway is updated with connection details.
    - If the execution is completed successfully, the client receives the final status and results.

5.3.2.4 Abstraction layer

The Capability Abstraction Layer acts as the interface between a sandboxed WebAssembly (Wasm) microservice and the host system. It securely exposes essential host-level capabilities, including file input/output (I/O), HTTP networking, local storage, and, importantly, event-driven communication with external systems through WebSockets and gRPC.

This layer also serves as a bridge to the Event Broker, allowing microservices to:

- Subscribe to events
- Publish data streams or notifications
- Maintain long-lived bidirectional sessions with clients

All capabilities are presented in a capability-based, runtime-enforced, and language-agnostic manner using the WebAssembly System Interface (WASI), WebAssembly Interface Types (WIT), and custom host functions.

Functional Responsibilities

- Expose System Capabilities (filesystem, networking, storage, etc.) via WASI or custom APIs
- Bridge Eventing via WebSocket/gRPC Broker
- Enforce Fine-Grained Access Control based on declared permissions
- Abstract Implementation Details for Multi-language Interoperability
- Maintain Real-Time Connectivity for low-latency data exchange

5.4 Microservices registry

The Microservice Registry is a component responsible for managing and serving microservices to execution runtimes. It provides an API interface for creating, reading, updating, and deleting (CRUD) microservices, as well as on-demand distribution of service binaries or bundles to the runtimes.

Designed for scalability and cloud-native deployment, the registry uses serverless architecture (e.g., AWS Lambda) for operational logic and object storage/CDNs (e.g., S3 + CloudFront) to efficiently deliver microservice artifacts with low latency and global availability.

Functional Responsibilities

- CRUD Operations
- Allow developers and administrators to register, update, and remove Wasm microservices through a dedicated API.
- On-Demand Service Delivery
- Serve the Execution Engine with the latest version of a microservice by redirecting to a cached binary on a CDN or object store.
- Versioning and Metadata Storage
- Maintain metadata for each microservice (name, version, tags, capabilities, size, checksum) in a central database.
- Security and Access Control
- Authenticate registry users and apply authorization policies for service publishing and retrieval.

6. Implementation Details

6.1 Evaluating Cross-Platform Binary Strategy

This section presents a comparative evaluation of binary formats suitable for implementing standard microservice capabilities across multiple runtimes. The analysis focuses on the **Universal Client Runtime**, which targets platforms such as the web, iOS, and Android, as well as the **Server-side Low Latency Runtime**.

The evaluation considers only those capabilities that do not require direct access to platform-specific SDKs or system-level APIs. Since UMA already provides native binary support for services requiring low-level integration, this analysis emphasizes formats that maximize **code reuse**, **portability**, and **consistency** across all UMA-supported environments.

To meet UMA's architectural goals—modularity, runtime efficiency, and streamlined deployment—the selected binary formats must be capable of portable execution while delivering reliable performance and minimizing reliance on the underlying host system.

6.1.1 Candidate Portable Binary Formats

The following binary formats were identified as potential solutions for cross-platform execution of universal microservices:

- **WebAssembly (WASM)**
- **Managed Bytecode** (e.g., JVM, .NET)
- **GraalVM Native Images**
- **Universal or Fat Binaries**
- **Containerized Executables** (e.g., Docker, OCI)

Each format was evaluated based on several criteria, including runtime compatibility, execution performance, binary size, system access control, deployment complexity, and the availability of development tooling.

6.1.2 Comparative Analysis

Characteristic	WebAssembly (WASM)	Managed Bytecode (JVM, .NET)	GraalVM Native Images	Universal/Fat Binaries	Containerized Executables
Portability (Web/iOS/Android/Server)	HIGH	Medium to HIGH	LOW	LOW	LOW
Runtime Performance	Near-native	Good (JIT-dependent)	High (AOT compiled)	Native per architecture	Moderate
Binary Size	Small	Moderate	Small	LARGE	LARGE
Startup Time	Fast	Moderate	Very Fast	Fast	Slow
Security Isolation	Strong (sandboxed)	Runtime-managed	OS-dependent	OS-dependent	OS-level namespaces
System Access (Abstraction Needed)	Indirect via WASI/host API	Indirect via runtime API	Full	Full	Full
Cross-Platform Runtime Availability	Broad and emerging	Requires VM installation	Requires platform-specific builds	LIMITED	Not feasible on mobile/web
Tooling and Ecosystem	Rapidly growing	Mature	MATURE	LIMITED	MATURE
Integration Effort (UMA)	Low to Moderate	Moderate	HIGH	HIGH	HIGH
Suitability for UMA Objectives	Strong	Conditional	Backend-specific only	POOR	POOR

Table 3: This table compares several runtime technologies based on their relevance to UMA, including WebAssembly, managed bytecode platforms, native images, universal binaries, and containerized executables. It highlights key characteristics such as portability, performance, startup time, system access, and overall suitability for UMA objectives.

6.1.3 Analysis and Trade-Offs

- **WebAssembly (WASM)**

  WASM supports sandboxed, portable execution across web browsers, mobile environments (via custom runtimes), and server platforms. Its binaries are compact, secure by default, and offer deterministic startup behaviour. While WASM does not provide direct system-level access, this limitation is addressed through the UMA abstraction layer, which enables platform-specific interactions without coupling service logic to OS APIs.

- **Managed Bytecode (e.g., JVM, .NET)**

  Managed bytecode formats offer strong cross-platform support but require the presence of language-specific virtual machines. This dependency adds complexity to web and mobile deployments, particularly in resource-constrained environments. However, these formats may still be viable in organizations with existing investments in the Java or .NET ecosystems.

  GraalVM Native Images

  GraalVM Native Images provide optimized performance and standalone execution, making them well-suited for backend microservices. However, the resulting binaries are platform-specific and not reusable across client runtimes, which conflicts with UMA's goal of universal code reuse.
- **Universal or Fat Binaries**

  These binaries are primarily useful in environments that support multi-architecture packaging, such as macOS. Despite this, they do not abstract away OS-level dependencies, and their large size and limited portability make them a poor fit for UMA.
- **Containerized Executables (e.g., Docker, OCI)**

  While highly effective for backend orchestration and isolation, containerized executables are unsuitable for mobile or web platforms due to their reliance on container engines. As a result, they are excluded from UMA's shared capability layer.

6.1.4 Recommendation

Based on the evaluation, **WebAssembly (WASM)** is recommended as the default binary format for executing non-OS-specific microservices across UMA runtimes. WASM offers an ideal balance of portability, efficiency, and security, and aligns with UMA's use of a unified abstraction layer for handling I/O, eventing, networking, and storage.

While managed bytecode formats such as JVM or .NET can serve as secondary options for server-side workloads in legacy environments, their limited support across mobile and web platforms prevents them from being used as the primary format.

The ongoing evolution of WASM, including the advancement of the WebAssembly System Interface (WASI), the component model, and language toolchain support, further solidifies its role as the standard for reusable business logic in UMA.

6.2 Possible Approaches for Multithreading Across Multiple OS and Target

To support scalable execution across diverse platforms, the **Universal Client Runtime** must allow concurrent execution of both portable WASM binaries and native platform-specific binaries. This includes JavaScript bundles in web environments, compiled frameworks in iOS, and shared objects in Android. Enabling dual execution introduces additional complexity related to thread safety, scheduling, and performance isolation.

This section evaluates three threading strategies:

- **Approach 1**: POSIX-based threading compiled to WASM
  Approach 2: Native threading via platform-specific APIs
- **Approach 3**: A hybrid model combining WASM and native execution

Each approach is evaluated within the context of UMA's abstraction layer, which decouples service logic from platform-specific APIs for I/O, event handling, networking, and storage.

6.2.1 Dual Execution Model: WASM and Native Binaries

The Universal Client Runtime supports a hybrid execution model across platforms:

- **Web applications**: Execute WASM modules alongside JavaScript services
- **iOS applications**: Execute WASM modules alongside native binaries (e.g., `.framework`, `.a`)
- **Android applications**: Execute WASM modules alongside native libraries (e.g., `.so` files or Kotlin modules)

Microservices are isolated by design. The Service Loader determines which binary format and threading strategy to apply based on runtime conditions such as device capabilities, service priority, and system load.

6.2.2 Multithreading Strategies

6.2.2.1 Approach 1: POSIX Threads in WebAssembly

This strategy compiles pthread-based C/C++ code to WASM using tools like Emscripten. Threads are mapped to Web Workers in supported environments.

Execution Model

- Threads are created from C/C++ code, compiled via Emscripten, and executed using Web Workers.
- On platforms that support `SharedArrayBuffer`, WASM threads run concurrently with shared memory access.
- The abstraction layer enables thread access to OS-specific features through host bindings.

Characteristics

- Thread scheduling and synchronization are managed at the WASM level using atomic operations and Futex-based locks.
  Thread pools may be initialized statically or dynamically
- Shared memory is supported through WASM linear memory and `SharedArrayBuffer`, if enabled

Limitations

- SharedArrayBuffer requires COOP/COEP headers, limiting availability in some browsers.
- Full POSIX thread functionality is not supported (e.g., signals, fork, or whole thread-local storage)

- Web Worker instantiation introduces thread creation overhead.

Advantages

- High code reusability, allowing the same binary to run on all platforms
- Strong sandboxing for security
- Well-suited for CPU-bound logic with minimal system interaction

Disadvantages

- Limited access to native platform services
- Increased memory use and startup latency
- Not ideal for real-time or hardware-intensive services

6.2.2.2 Approach 2: Native Threading with Platform-Specific APIs

This approach uses each platform's native threading system for both native and embedded WASM services.

Execution Model

- **Web**: Threads implemented via Web Workers and SharedArrayBuffer
- **iOS**: Threads managed via Grand Central Dispatch, NSOperationQueue, or pthreads
- **Android**: Threads managed via Java threading, ExecutorService, or Kotlin Coroutines

- WASM modules run in embedded runtimes, such as Wasmer or Wasmtime, and may interoperate with native services

Characteristics

- Lifecycle and priority are managed through platform APIs
- Native threads can directly access system components, including UI and low-level APIs

- WASM binaries operate within runtime environments that support parallelism

Advantages

- High-performance threading and efficient resource usage
- Full access to system APIs, suitable for latency-sensitive workloads
  Strong integration with platform-native services and UI

Disadvantages

- Requires duplicated implementations for each platform
- Higher maintenance costs for orchestration and testing
- Limited code reuse across environments

6.2.2.3 Approach 3: Hybrid Execution with WASM and Native Binaries

The hybrid model supports the parallel execution of WASM and native binaries, with the Service Loader and Thread Manager selecting the optimal approach for each service.

Execution Model

- WASM threads are managed via portable runtimes or Web Workers
- Native threads are scheduled using platform-specific APIs
- Runtime decisions are based on service characteristics, such as criticality or resource needs

Characteristics

- Dynamic, per-service decision-making allows selective optimization
- Thread isolation is enforced between WASM and native threads
- The abstraction layer ensures consistent access to capabilities, regardless of binary origin

Design Considerations

- The Thread Manager must handle thread lifecycle, priority, and error boundaries across WASM and native execution pools.
- Communication between WASM and native services must be handled through serialized event streams or bridges.
- Unified logging and telemetry are needed for monitoring and diagnostics across all execution contexts.

Advantages

- Maximizes both performance and portability
- Supports flexible service deployment across platforms
- Allows critical services to benefit from native execution while maintaining cross-platform reuse

Disadvantages

- Increased architectural complexity
- Requires precise orchestration to prevent contention or inconsistent behaviour
- Demands advanced scheduling and fault tolerance mechanisms.

6.2.3 Comparative Analysis

Criteria	1 POSIX Threads in WASM	2 Native Threading (Platform-Specific APIs)	3 Hybrid Execution (WASM + Native Binaries)
Thread Model	pthread abstraction via WASM	GCD, Web Workers, Coroutines, etc.	Combined WASM + native thread pools
Binary Support	WASM only	Native and WASM via embedded runtime	Full dual support (WASM + native binaries)
Code Reusability	HIGH	LOW	MEDIUM
Performance	MEDIUM	HIGH	HIGH
Platform Integration	Limited via the abstraction layer	FULL	FULL

Startup Latency	MEDIUM to HIGH	LOW	VARIABLE
Thread Isolation	Strong (sandboxed WASM environment)	OS-level	Mixed (WASM sandbox + native thread safety)
Maintenance Overhead	LOW	HIGH	HIGH

Table 4: This table presents a comparative analysis of different threading models for UMA, including POSIX Threads in WASM, Native Threading using platform-specific APIs, and a Hybrid Execution model that combines WASM with native binaries. It evaluates key criteria, including thread model, platform integration, isolation, and suitability for various use cases.

6.2.4 Evaluation Summary

A single-threading model is not sufficient to support UMA's multi-binary execution requirements. While WASM threading offers consistent and portable behaviour, it lacks tight integration with native system capabilities and often falls short in performance-critical scenarios. In contrast, native threading models offer superior efficiency and direct hardware access but compromise portability and code reuse.

The **hybrid model** provides the best alignment with UMA's architectural goals. It allows the Universal Client Runtime to dynamically evaluate the execution context, select the appropriate binary format (WASM or native), and manage threading accordingly. The Thread Manager ensures proper isolation and prioritization, while the abstraction layer guarantees uniform access to system capabilities such as networking, eventing, and I/O. This approach supports both high-performance native workloads and portable service logic across all UMA-supported platforms.

6.3 Interface Design Patterns for Business Logic Decoupling

To support UMA's focus on runtime flexibility and long-term adaptability, interface design must extend beyond abstracting platform-specific APIs. UMA promotes interface models that serve as **strategic contracts**, enabling microservices to be composed, tested, and evolved independently of their execution environment.

Rather than binding services to fixed runtime configurations, UMA defines interfaces that allow business logic to remain fully agnostic to whether it is executed in WASM or

native code, on the client or the server. These contracts must be **stable but flexible**, expressing intent and capability without leaking platform or implementation details.

Interfaces are central to UMA's design philosophy. They drive implementation, orchestration, testing, and deployment decisions. This approach supports dynamic delegation, runtime switching, and long-term service evolution without requiring changes to the service signature.

Key design patterns that reinforce this model include:

- **Ports and Adapters (Hexagonal Architecture)**
  This pattern inverts dependencies, isolating core business logic from external concerns. In UMA, adapters for WASM and native runtimes can be swapped at build or runtime without changing the interface.
- **Bridge Pattern**
  The bridge pattern separates service interfaces from their platform-specific implementations. It allows new platforms to be supported without requiring changes to interface definitions or consuming services.
- **Strategy Pattern**
  UMA services may need to adjust behaviour based on runtime or platform characteristics. The strategy pattern enables these variations without embedding platform conditionals in the service logic.
- **Event-Driven Interfaces**
  UMA emphasizes asynchronous, event-based communication over direct procedural calls. This allows services to operate independently of the sender's execution context while maintaining responsiveness and scalability.

These patterns serve a unified architectural goal: maximize service reuse and minimize disruption when evolving runtimes, integrating third-party tools, or shifting deployment targets. By applying them at the interface layer, UMA ensures that business logic remains portable, maintainable, and adaptable across environments.

For further guidance, examples, and practical implementation techniques, refer to the companion research on platform-agnostic interface design in UMA.

6.4 Fault Tolerance, Orchestration, and Data Consistency in Distributed UMA Runtimes

UMA's distributed execution model enables microservices to run across clients, servers, and edge devices. This design improves modularity and scalability but introduces new challenges related to fault isolation, transactional integrity, and cross-runtime data consistency.

This section outlines the architectural mechanisms UMA uses to manage these complexities, including failure handling, eventual consistency, distributed workflows, and offline reconciliation.

6.4.1 Handling Failure in a Decentralized Microservice Runtime

UMA services must be resilient to partial failure. Network interruptions, hardware limitations, and device variability can all affect service availability. Microservices must be designed with failure in mind and adopt proven resilience strategies:

- **Redundancy and Load Balancing**
  Deploy multiple instances of critical services and distribute requests to eliminate single points of failure.
  Circuit Breakers
  Detect failing services early and stop sending requests until recovery is confirmed.
  Retry Logic with Backoff
  Automatically retry failed operations in case of transient errors while avoiding system overload.
  Graceful Degradation
  Deliver fallback responses or partial functionality when the full service cannot be executed.

- **Health Monitoring and Self-Healing**
  Continuously monitor service health and restart or reinitialize services when failure conditions are detected.

UMA supports these strategies through asynchronous APIs and resilient runtime interfaces that operate consistently across client and server environments.

6.4.2 Achieving Data Consistency Without Centralized Transactions

UMA follows the “database per service” model, where each microservice owns and manages its own data. This improves autonomy and scalability but makes traditional ACID transactions impractical.

UMA maintains data consistency using the following techniques:

- **Event-Driven State Propagation**
  Changes in service state are emitted as events and asynchronously consumed by downstream services.
- **Asynchronous Messaging**
  WebSocket and gRPC enable low-latency, bidirectional messaging, allowing services to coordinate state across distributed runtimes.
- **Change Data Capture (CDC) and Anti-Entropy**
  Techniques such as CDC allow services to detect and resolve data divergence across nodes or disconnected clients.
- **Idempotent Operations**
  Services must ensure that repeated or duplicate requests do not result in data corruption.
- **Conflict Resolution**
  UMA supports strategies like Last-Write-Wins, vector clocks, and Conflict-free Replicated Data Types (CRDTs) to manage conflicting updates.

6.4.3 Distributed Transactions: The Saga Pattern in UMA

To orchestrate long-running, multi-step workflows without centralized transactions, UMA implements the **Saga pattern**. A saga breaks a distributed process into independent, compensatable steps.

There are two coordination styles:

- **Orchestration**
  A central coordinator, typically on the server, controls the execution of each step and invokes compensating actions when needed.
- **Choreography**
  Each service listens for events and responds independently, triggering the next step in the workflow. This decentralized model aligns well with UMA's platform-agnostic execution.

UMA allows services to declare compensating actions as part of their interface. The runtime can invoke these handlers if downstream services fail or become unresponsive.

6.4.4 Dealing with Offline Execution and State Reconciliation

Because UMA supports execution on edge or mobile devices, services must be designed to handle intermittent connectivity and reconcile state later.

Key practices include:

- **Local Data Storage with Deferred Sync**
  Allow services to perform operations offline and sync changes once reconnected.
- **Versioned or Timestamped State Models**
  Use time-based conflict resolution when merging state updates from multiple sources to ensure consistency.
- **Delta Sync and Polling**
  Transmit only changed data segments to minimize bandwidth and latency.

- **Reconciliation Patterns**
  Use patterns like Book Keeper, which track local changes and request server acknowledgment to confirm synchronization.

These strategies ensure eventual convergence of the distributed state while preserving service responsiveness during offline periods.

6.4.5 Architectural Recommendations

To achieve robust cross-runtime execution in UMA, the following best practices are recommended:

- Use event brokers to decouple communication between runtimes
  Favour stateless, asynchronous service patterns.
  Employ lightweight state machines to coordinate workflows across services
- Ensure every service publishes events and handles idempotent requests
- Implement consistent logging and tracing mechanisms to monitor service flows
- Support rollback and fallback logic within both client and server runtimes.

6.5 Evaluation of Technologies for Implementing the Server Low Latency Runtime

The Server Low Latency Runtime is a critical component of the Universal Microservices Architecture (UMA). Its primary role is to execute portable microservices, compiled as platform-independent binaries, in an environment that offers low latency, high performance, and efficient resource usage. This runtime is tailored for services that do not require direct interaction with platform-specific APIs but still demand fast response times, secure isolation, and seamless integration into client-server communication patterns.

This section reviews potential technologies for building the Server Low Latency Runtime, with a particular emphasis on WebAssembly (WASM)-based solutions. WASM has been selected as UMA's default binary format due to its secure, sandboxed execution model,

compact binary size, rapid startup, and cross-platform compatibility across both client and server environments.

6.5.1 Evaluation Criteria

To meet the requirements of UMA, a candidate runtime must satisfy the following criteria:

- **Performance and Startup Time**
  The runtime must execute services with low cold-start latency and deliver consistent performance under varying loads.
- **Scalability and Concurrency**
  Support for concurrent execution is essential. This can be achieved through lightweight threading models or compatibility with container orchestration systems that enable scalable deployments.
- **WASI and Abstraction Layer Integration**
  Full support for the WebAssembly System Interface (WASI) is required. The runtime must also integrate with UMA's abstraction layer to securely expose networking, I/O, storage, and other system capabilities needed by the microservices.
- **Embeddability and Extensibility**
  The runtime should be lightweight and modular, making it easy to embed within UMA's service loader and extend to support custom execution logic or telemetry hooks.
- **Stability and Ecosystem**
  The chosen technology must be production-ready, actively maintained, and backed by a robust ecosystem. Compatibility with standard toolchains and developer workflows is also essential.
- **Cost Efficiency**
  Efficient use of memory and CPU resources is crucial to support high-density service execution. The runtime must enable cost-effective scaling, whether deployed in cloud environments or at the edge.

6.5.2 Comparative Analysis of Candidate Runtimes

The following runtimes were evaluated based on the criteria above:

Runtime	Compilation Mode	WASI Support	Performance	Startup Time	Embeddability	Maturity and Stability	Cost Efficiency	Pros	Cons
Wasmtime	JIT, AOT (Cranelift)	Yes	High (optimized)	Fast	High	Mature	High	Optimized with Cranelift; good WASI support; cross-platform	Slightly higher memory use than minimal runtimes
WasmEdge	AOT	Yes	**Very High**	**Very Fast**	High	Stable, CNCF-backed	Very High	Cloud-native, fast for serverless/microservices; integrates with K8S	Limited debugging tools and community size vs Wasmtime
Wasmer	JIT, AOT	Yes	High	Moderate-Fast	High	Active, extensible	High	Flexible compiler backends; large language support	Slightly larger runtime footprint; more configurat

									ion comp lexity
WAM R	JIT, AOT, Inter prete r	Yes	Mode rate	Very Fast	Very High	Light weig ht focus	Very High	Small footp rint; highl y embe ddabl e, confi gura ble	Perfo rman ce lower for comp ute-h eavy micro servic es
Was m3	Inter prete r	Parti al	Low-Mode rate	Very Fast	Very High	Exper iment al/lig htwei ght	Very High	Ultra-fast start up; tiny binar y size	Poor perfo rman ce for heav y work loads; limite d WASI supp ort
Extis m	JIT	Yes	High	Fast	High	Appli catio n plugi n focus	Mode rate	Simpl e integ ratio n; rich SDK supp ort, good for plugi ns	Less contr ol over raw exec ution /runti me inter nals
waze ro	Inter prete r	Yes	Mode rate	Fast	High	Go-n ative, zero deps	Very High	Writt en in Go; good	Go-o nly use case;

								for Go ecosystems	limited optimization for cross-language needs
Lunatic	Interpreter	Yes	Moderate	Fast	High	Niche	High	Actor-model concurrency; Erlang-style fault tolerance	Immature ecosystem; use-case specific

Table 5: This table evaluates various WebAssembly runtimes based on criteria such as compilation mode, WASI support, performance, startup time, embeddability, and cost efficiency. It highlights each runtime's strengths and trade-offs to help determine the most suitable option for UMA use cases.

6.5.3 Recommendation

Based on the comparative evaluation, **WasmEdge** emerges as the most suitable technology for implementing the UMA Server Low Latency Runtime.

WasmEdge is a high-performance WebAssembly runtime specifically optimized for cloud-native and edge environments. It supports ahead-of-time (AOT) compilation, delivers fast startup times, and offers full compatibility with the WebAssembly System Interface (WASI). These features make it well aligned with UMA's needs for serverless, on-demand, and latency-sensitive execution. WasmEdge also integrates natively with container platforms and Kubernetes, making it ideal for dynamic orchestration and elastic scaling of server-side microservices. Its adherence to WebAssembly standards

and growing adoption within the Cloud Native Computing Foundation (CNCF) ecosystem reinforce its readiness for production use.

As a strong alternative, **Wasmtime** provides a mature execution environment with both just-in-time (JIT) and AOT compilation support. It has a robust and flexible architecture, full WASI compliance, and tight integration with Rust-based toolchains. Wasmtime is particularly well suited for scenarios that require detailed runtime introspection, broader language support, or development environments already centred on Rust.

For deployments constrained by memory or compute resources, the **WebAssembly Micro Runtime (WAMR)** offers a viable option. Its lightweight design and rapid initialization make it ideal for embedded systems and low-power edge nodes. However, these benefits come at the cost of reduced performance, limiting their applicability for compute-intensive microservices.

6.5.4 Integration Considerations

Regardless of the selected runtime, seamless integration with UMA's abstraction layer is essential. This layer provides access to platform capabilities, including file I/O, networking, storage, and event dispatch, while maintaining isolation between the microservices and the underlying operating system.

Runtimes that offer native WASI extensions or support for pluggable host APIs, such as WasmEdge and Wasmtime, are particularly well-suited for this integration. Embedding the runtime within the UMA Execution Engine enables microservices to operate in secure sandboxes while leveraging shared infrastructure components such as service discovery, event brokers, and observability systems.

To meet UMA's operational goals, the Server Low Latency Runtime must also support configurable execution controls. These include timeouts, concurrency limits, resource quotas, and event-driven activation. Together, these features ensure that UMA services remain performant, cost-efficient, and resilient at scale.

7 Performance and Scalability

The Universal Microservices Architecture (UMA) is designed to optimize responsiveness, efficiency, and scalability across both client-side and server-side runtimes. By executing business logic as portable microservices compiled into WebAssembly or native binaries, UMA enables intelligent workload delegation based on device capabilities, resource availability, and security constraints. This section outlines UMA's strategies for addressing performance and scalability throughout its distributed runtime model.

7.1 Client-Side Runtime Performance

Client-side performance is critical for UMA applications targeting web, iOS, and Android platforms. UMA enables portable binaries to execute microservices directly within client environments, minimizing network latency and improving responsiveness when local execution is feasible.

7.1.1 Web (Browser)

In modern browsers, UMA executes WebAssembly modules compiled with POSIX thread support via Emscripten. Multithreading is achieved using Web Workers and SharedArrayBuffer, though constraints such as COOP/COEP headers and Web Worker startup latency present limitations. UMA mitigates these issues through pre-allocated worker pools and efficient memory allocators, such as mimalloc. Although browser-based WASM threading does not yet match native performance, UMA delivers fast startup times and improved runtime efficiency by reducing blocking and reusing threads.

7.1.2 iOS

On Apple devices, UMA supports both WASM execution via embedded runtimes and delegation to native microservices compiled for iOS. Concurrency is managed using

Grand Central Dispatch and OperationQueue, enabling efficient use of system threads. The UMA runtime abstraction layer classifies workloads and delegates execution to native threads when platform-optimized performance is required. This hybrid model preserves portability while leveraging native execution for critical paths.

7.1.3 Android

UMA achieves concurrency on Android using Kotlin Coroutines, Java Executors, and native pthreads, depending on the binary format. WASM binaries run within embedded runtimes that support parallel execution, while native services use Android's built-in threading frameworks. UMA dynamically evaluates runtime context, device capabilities, and service criticality to allocate threads with minimal latency. Microservices are isolated and executed in parallel, ensuring UI responsiveness under high load. The service loader and thread manager collaborate to determine whether a service is run locally or delegated to a backend node.

7.2 Server-Side Runtime Performance

Server-side runtimes in UMA are optimized for low-latency execution, scalable concurrency, and efficient resource utilization. UMA relies on WASM runtimes such as WasmEdge and Wasmtime to provide fast startup, secure isolation, and high-throughput execution.

7.2.1 Startup Latency

With ahead-of-time (AOT) compilation, WASM runtimes deliver startup times between 1 and 5 milliseconds. This rapid cold start supports short-lived, on-demand service execution, eliminating the overhead typically associated with container orchestration.

7.2.2 Concurrency and Throughput

UMA supports high concurrency through lightweight sandboxed microservices and event-driven orchestration. Execution is parallelized using thread pools, and the runtime abstraction layer ensures efficient scheduling of resource-intensive workloads.

Support for multithreading varies by runtime, with WasmEdge offering WASI-Threads. UMA's thread manager coordinates OS-level and runtime-specific concurrency mechanisms to maximize throughput.

7.2.3 Scalability

UMA enables elastic horizontal scaling by decoupling microservice execution from fixed infrastructure. Services are retrieved from a registry and instantiated on demand. Integration with content delivery networks (CDNs) enables binaries to be served from edge nodes, thereby reducing latency and enhancing geographic performance.

7.2.4 Resource Isolation

WebAssembly runtimes provide strong sandboxing with minimal overhead. UMA can run thousands of concurrent microservices per node, making it ideal for dense, distributed deployments that do not rely on heavyweight virtual machines or containers.

7.3 Dynamic Execution and Load Distribution

A defining feature of UMA is its dynamic execution model. The runtime continuously evaluates the optimal location to execute each microservice. The service loader assesses factors such as CPU availability, memory pressure, power constraints, and execution priority. If a client device lacks the necessary resources, the service is delegated to a server instance via the UMA event broker.

This adaptive load distribution reduces perceived latency, preserves client responsiveness, and ensures backend resource efficiency. UMA maintains thread isolation and enforces service prioritization to avoid overload and performance degradation, even in low-connectivity or resource-constrained environments.

7.4 Summary

UMA delivers scalable, high-performance execution across heterogeneous environments. By combining portable binaries, multithreading, and dynamic runtime heuristics, UMA achieves near-native responsiveness on both client and server systems. Its modular design enables fine-grained scaling, rapid cold starts, and cost-effective resource usage, making it well-suited for distributed applications that demand performance without compromising portability.

8 Security Considerations

Universal Microservices Architecture introduces a distributed execution model in which services can run dynamically on either client or server devices. These microservices are delivered as portable binaries and interact asynchronously through event-driven protocols. This architecture introduces unique security challenges, including code integrity, isolation, access control, and data protection across diverse devices and networks.

This section outlines the core strategies and practices required to maintain a secure runtime environment within UMA.

8.1 Threat Surface and Execution Boundaries

Microservices in UMA may run in untrusted environments such as browsers, mobile devices, or edge nodes, as well as in centralized backends. Each environment introduces specific risks:

- **Client-side execution** is vulnerable to tampering, reverse engineering, and privilege escalation.
- **Server-side runtimes** must enforce secure isolation across concurrently executing services.
- **Dynamic service loading** over networks introduces risks of binary injection, unauthorized code execution, and man-in-the-middle attacks.

To mitigate these risks, UMA enforces strict binary validation, transport-layer encryption, and layered runtime protections across all deployment environments.

8.2 Client-Side Risk Mitigation

Executing microservices on client devices increases exposure to local threats, including runtime inspection, memory access, and storage attacks. UMA employs platform-specific strategies to mitigate and manage these risks.

8.2.1 Web (Browser)

UMA leverages WebAssembly's sandboxed execution model, which restricts access to host APIs by default. When combined with WASI, the runtime follows a capability-based access model, ensuring microservices are granted only the minimal permissions required.

Still, web browsers remain susceptible to runtime inspection tools and memory probes through SharedArrayBuffer. UMA limits the exposure of sensitive logic by dynamically assessing risk and offloading critical tasks to secure backends when needed.

8.2.2 iOS and Android

On mobile devices, UMA executes either native microservices or WASM modules via embedded runtimes. To harden these runtimes:

- Binaries are obfuscated and signed.
- Run-time integrity checks prevent tampering or sideloading.
- Critical services may be delegated to the server when local execution poses a security risk.

By offloading sensitive logic and enforcing binary integrity, UMA reduces attack vectors on mobile clients.

8.3 Server-Side Isolation and Control

UMA enforces strict sandboxing for server-side execution using hardened WASM runtimes. Isolation is achieved through the following mechanisms:

- Services are declared and restricted to specific WASI capabilities.
- Each microservice runs in a separate process or virtual runtime, isolated from others.
- Resource quotas and execution timeouts prevent resource exhaustion or abuse.

Compared to containers or virtual machines, WASM sandboxes offer lower overhead and stronger execution determinism. The UMA Execution Engine monitors all services for anomalous behaviour and enforces boundary policies in real time.

8.4 Secure Binary Delivery and Verification

UMA microservices are distributed via a trusted pipeline using a combination of cryptographic and transport-level protections:

- **Code Signing:** All binaries are cryptographically signed before being published. The runtime verifies signatures before loading any service.
- **Immutable Registry Entries:** Each version of a service is hashed and indexed to prevent tampering.
- **Transport Security:** All binaries are delivered over TLS, with optional mutual authentication.
- **Runtime Attestation:** For high-security use cases, UMA can incorporate runtime attestation to verify the environment before execution.
  These mechanisms ensure that only authenticated, untampered binaries are executed in any UMA runtime.

8.5 Authentication and Access Control

UMA separates identity from execution and supports unified authentication and authorization across runtimes:

- Clients authenticate using OAuth2 or JWT tokens before invoking services.
  The runtime verifies identity claims and enforces capability restrictions based on predefined roles.
  Events are scoped by policy, limiting which services or clients can publish or consume them.

This layered access control model ensures tight privilege management and auditability, regardless of where the microservice is executed.

8.6 Securing Event-Driven Communication

UMA microservices interact through asynchronous events that must be protected against forgery and tampering:

- **Authenticated Channels:** Event messages are signed or transmitted over gRPC and WebSocket channels with mTLS.
- **Schema Validation:** Runtime brokers enforce schema validation to prevent the transmission of malformed or untrusted payloads.
- **Scoped Subscriptions:** Each service can only subscribe to event streams explicitly allowed by policy.

By securing message transmission and enforcing runtime checks, UMA prevents unauthorized access or manipulation of event traffic.

8.7 Supply Chain and Dependency Security

UMA microservices often include external libraries or dependencies. To secure the supply chain:

- **SBOMs (Software Bill of Materials):** Generated automatically and stored with each service release.
- **Dependency Scanning:** Integrated into CI/CD pipelines to detect known vulnerabilities.
- **Version Pinning and Audit Trails:** All microservices are versioned and traceable to their source and build configuration.

These practices ensure that UMA services meet consistent security standards and remain free from known vulnerabilities and exploits.

8.8 Summary

Security in UMA is built on architectural rigour and runtime enforcement. By applying strict isolation, transport security, service verification, and dynamic delegation, UMA

ensures that microservices remain secure across devices, platforms, and networks. Its approach strikes a balance between performance, modularity, and trust, enabling distributed services to operate safely without compromising execution flexibility or scalability.

9 Conclusion

Universal Microservices Architecture (UMA) presents a robust and forward-looking blueprint for building modular, scalable, and portable software systems that operate seamlessly across client, server, and edge environments. By decoupling business logic from platform-specific concerns and encapsulating it within atomic microservices, UMA enables a high degree of flexibility, maintainability, and reuse across modern application architectures.

Through technologies such as WebAssembly, UMA ensures consistent behaviour and strong isolation across runtimes, while its event-driven communication model supports asynchronous, reactive workflows. Runtime-level delegation, based on device capabilities, resource availability, and security posture, enables UMA to deliver context-aware execution, optimizing performance and cost across heterogeneous environments.

The architecture empowers development teams to:

- Reuse business logic across multiple platforms without duplication.
- Dynamically scale services based on execution context and demand.
- Accelerate delivery by composing systems from reusable microservice units.

These characteristics help reduce operational complexity, improve time-to-market, and support long-term maintainability. Whether executing on the edge, in the browser, or within server-side WASM sandboxes, UMA strikes a practical balance between portability and performance.

Looking ahead, UMA is well-positioned for AI-assisted software development. Its granular service model and well-defined interfaces offer a natural fit for systems where services can be discovered, composed, or even generated autonomously. As AI tools evolve to understand code semantics and developer intent, UMA provides the architectural foundation for automating microservice creation, testing, and deployment with minimal human intervention. This enables faster prototyping, personalized application behaviour, and continuous system adaptation.

However, adopting UMA introduces architectural trade-offs. Distributed, multi-binary runtimes require careful orchestration, robust interface contracts, and consistent event schema management. WebAssembly's portability brings execution efficiency and security, but may require hybrid strategies in environments with limited multithreading or system-level access. UMA's abstraction layers help mitigate these gaps, albeit at the expense of some overhead for consistency.

Despite these challenges, the long-term benefits are compelling. UMA enables teams to build once and deploy anywhere, adapt execution to runtime conditions, and deliver reliable, secure, and high-performance user experiences. As support for WebAssembly, edge computing, and AI-native development continues to expand, UMA offers a resilient, scalable, and future-proof foundation for the next generation of distributed applications.

Universal Microservices Architecture: Reference and Glossary

I. Introduction

This document serves as a comprehensive supplement to the white paper titled "Universal Microservices Architecture" (UMA) by Enrico Piovesan. Its primary objective is to enhance the white paper's academic rigor and usability by providing a meticulously curated "Reference Section" and a "Glossary of Terms." The "Reference Section" lists authoritative sources for all external technologies, standards, and concepts mentioned, ensuring verifiability and offering readers pathways for deeper exploration. The "Glossary of Terms" defines key technical vocabulary and acronyms, promoting clarity and consistency for a diverse technical audience.

The "Universal Microservices Architecture" white paper introduces a novel approach to microservices, emphasizing platform-agnostic execution across client, server, and edge environments. It leverages WebAssembly (WASM) as a core technology for portable binaries and integrates various modern distributed system patterns and communication protocols to achieve high performance, scalability, and resilience. This report directly supports the UMA white paper by formalizing its external dependencies and terminology, thereby elevating its standing as a definitive resource in the domain of distributed systems and microservices architecture.

II. Reference Section

This section provides a categorized list of all external technologies, standards, and concepts referenced in the 'Universal Microservices Architecture' white paper. Each entry includes its official source, a brief explanation of its relevance to UMA, and an indication of its authoritative documentation.

A. Standards and Specifications

WebAssembly (WASM) Core Specification
The WebAssembly Core Specification is foundational to UMA, enabling the compilation of services into platform-agnostic binaries for high performance and broad compatibility across web, mobile, and server environments.1 This core specification defines the semantics of WebAssembly modules independently from concrete embeddings.2
The inherent design of WebAssembly, particularly its portable binary format and the WebAssembly System Interface (WASI) standard, directly addresses the core

challenge of achieving true "universal" execution across heterogeneous environments. This is a critical enabler for UMA, allowing a single service binary to run efficiently on diverse client and server platforms. This contrasts sharply with traditional containerization or Java Virtual Machine (JVM) /.NET-based approaches, which are typically tied to specific host environments or require heavy runtimes.[1] WASM's unique position as the enabling technology underpins UMA's core value proposition of ubiquitous microservices.

- **Reference:** WebAssembly. (n.d.). *Specifications*. Retrieved from https://webassembly.org/specs/ [2]

WebAssembly System Interface (WASI)
WASI is a standard that securely exposes essential host-level capabilities to sandboxed WebAssembly microservices, including file I/O, HTTP networking, local storage, and event-driven communication. This is crucial for WASM modules to operate effectively outside web browsers.1 WASI defines a modular system interface that provides POSIX-like features, such as file I/O, constrained by capability-based security.2
WASI is not merely an API; it is a critical component that transforms WASM from a pure computation engine into a viable application runtime for non-browser environments. Without WASI, WASM's "universal" promise would be severely limited, as services would lack the means to interact with the underlying system. Its capability-based security model also reinforces UMA's emphasis on "sandboxing" for secure execution, a key aspect of distributed microservices.[1]

- **Reference:** WebAssembly. (n.d.). *Specifications - WASI API*. Retrieved from https://webassembly.org/specs/ [2]

WebAssembly Interface Types (WIT)
WIT is used in conjunction with WASI for defining language-agnostic interfaces for capabilities. This ensures seamless interoperability between WASM modules and their host environments, facilitating the integration of diverse programming languages within the UMA ecosystem.1

- **Reference:** WebAssembly. (n.d.). *Specifications - Tool Conventions*. Retrieved from https://webassembly.org/specs/ (Implied by WASM ecosystem)

HTTP/2
HTTP/2 is a vital protocol utilized by gRPC for improved performance and reduced latency in communication between UMA microservices.1 The official specification for HTTP/2 is defined by RFC 9113, which collectively obsoletes all preceding RFCs defining HTTP.4

- **Reference:** IETF. (2023). *RFC 9113: HTTP/2*. Retrieved from https://httpwg.org/specs/ [4]

WebSocket Protocol (RFC 6455)
The WebSocket Protocol, standardized by IETF as RFC 6455, provides full-duplex,

bidirectional communication channels over a single TCP connection. This enables real-time, low-latency interaction within UMA, particularly for client-server communication.1 RFC 6455 details a technology that enables two-way communication between a client and a remote host, designed to work over HTTP ports 80 and 443 while supporting HTTP proxies and intermediaries.5

The choice of WebSockets as a primary communication protocol for real-time interaction in UMA reflects a clear architectural decision to move beyond older, less efficient HTTP-based polling mechanisms. RFC 6455 explicitly states that WebSockets "supersede existing bidirectional communication technologies that use HTTP as a transport layer" and addresses issues such as "high overhead" and the need for "multiple underlying TCP connections" associated with HTTP polling.[6] While Server-Sent Events (SSE) and Long Polling are mentioned in the white paper, their inclusion likely serves as a recognition of their historical role or as potential graceful degradation strategies, rather than core real-time mechanisms. This indicates a trend towards more efficient, dedicated protocols for real-time needs in modern distributed architectures like UMA, optimizing for latency and resource utilization.

- **Reference:** IETF. (2011). *RFC 6455: The WebSocket Protocol*. Retrieved from https://www.tech-invite.com/y60/tinv-ietf-rfc-6455.html [5]

Server-Sent Events (SSE) (HTML Living Standard)

SSE is a server push technology enabling servers to push updates to clients over a single, long-lived HTTP connection. It is efficient for streaming events from server to client and is well-supported by most modern browsers.1 The HTML Living Standard for Server-Sent Events introduces the

EventSource interface to enable this functionality.[8]

- **Reference:** WHATWG. (n.d.). *HTML Living Standard: 9.2 Server-sent events*. Retrieved from https://html.spec.whatwg.org/multipage/server-sent-events.html [8]

Protocol Buffers (Protobuf)

Protocol Buffers are used by gRPC for compact and efficient message serialization, ensuring type safety and cross-language interoperability across UMA components.1 Protobuf is a language-neutral, platform-neutral, and extensible mechanism for serializing structured data, offering benefits such as compact data storage, fast parsing, and availability in many programming languages.11

- **Reference:** Google. (n.d.). *Protocol Buffers Overview*. Retrieved from https://protobuf.dev/overview/ [11]

Transport Layer Security (TLS) 1.3 (RFC 8446)

TLS 1.3 is used for encrypted transport in gRPC communication and mutual TLS (mTLS) for securing event messages, ensuring confidentiality and integrity in UMA's distributed

environment.1 RFC 8446 specifies TLS version 1.3, designed to enable client/server applications to communicate securely over the Internet, preventing eavesdropping, tampering, and message forgery.13

- **Reference:** IETF. (2018). *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3*. Retrieved from https://www.rfc-editor.org/rfc/rfc8446 [13]

OAuth 2.0 (RFC 6749)

OAuth 2.0 is used for token-based client authentication in gRPC communication and for clients to authenticate before invoking services within UMA.1 RFC 6749 defines the OAuth 2.0 authorization framework, which allows a third-party application to obtain limited access to an HTTP service on behalf of a resource owner without sharing credentials directly.16

The adoption of OAuth 2.0 in UMA represents a strategic choice for managing secure access in a highly distributed and potentially multi-tenant environment. It moves away from traditional, less secure authentication models, such as direct credential sharing, towards a token-based, delegated authorization model. This is critical for maintaining a "zero-trust security" posture, especially as UMA supports diverse clients (web, mobile) and potentially external third-party integrations, ensuring that access is granular, revocable, and does not expose sensitive user credentials. The combination of OAuth 2.0 with mTLS further strengthens inter-service communication security.

- **Reference:** IETF. (2012). *RFC 6749: The OAuth 2.0 Authorization Framework*. Retrieved from https://www.rfc-editor.org/rfc/rfc6749 [16]

POSIX Threads (pthreads) (IEEE Std 1003.1)

POSIX Threads, or pthreads, is a standard for threads used as a basis for multithreading strategies in WASM and native environments (iOS, Android) within UMA.1 The IEEE Std 1003.1™-2024 Edition defines a standard operating system interface and environment to support applications portability at the source code level.19

- **Reference:** The Open Group. (n.d.). *IEEE Std 1003.1™-2024 Edition (POSIX.1-2024)*. Retrieved from https://pubs.opengroup.org/onlinepubs/9799919799/ [19]

CloudEvents Specification

The CloudEvents specification provides a standardized data model for all event exchanges in UMA, offering a common schema for structuring and interpreting event data. This is crucial for UMA's event-driven architecture, ensuring interoperability between disparate services that communicate via events.1

- **Reference:** CloudEvents. (n.d.). *Specification*. Retrieved from https://cloudevents.io/ (External Standard)

Open Container Initiative (OCI) Specifications

The Open Container Initiative (OCI) defines standards for container images, runtimes, and

distribution.21 While UMA moves beyond traditional containers for client-side and edge deployments, OCI's mention in the white paper highlights that Docker/OCI are effective for backend orchestration and isolation, but unsuitable for mobile or web platforms due to their reliance on container engines.1 OCI includes the Runtime Specification, Image Specification, and Distribution Specification, unifying the building, distribution, and running of containers.21

The limitation of OCI-compliant containers to server-side environments, due to their reliance on container engines, is a direct factor in the emergence of architectures like UMA. UMA does not seek to replace containers entirely but addresses a critical gap where containerization falls short: universal client-side and edge execution. This positions UMA as a complementary, rather than strictly competitive, technology in the broader cloud-native landscape, specifically carving out a niche for truly ubiquitous microservices.

- **Reference:** Open Container Initiative. (n.d.). *Specifications*. Retrieved from https://opencontainers.org/ [21]

Semantic Versioning

Semantic Versioning is essential for managing changes to microservices in UMA in a structured and predictable way. It defines API contracts and assigns version numbers (patch, minor, major) to indicate the nature of changes, which is critical for maintaining compatibility and managing dependencies in a distributed system.1

- **Reference:** Semantic Versioning 2.0.0. (n.d.). *SemVer*. Retrieved from https://semver.org/ (External Standard)

Software Bill of Materials (SBOMs)

SBOMs are automatically generated and stored with each service release in UMA to secure the supply chain. This practice is a critical aspect of UMA's security posture, providing transparency into the components of each microservice and enabling proactive vulnerability management.1

- **Reference:** NTIA. (n.d.). *Software Bill of Materials (SBOM)*. Retrieved from https://www.ntia.gov/SBOM (External Standard)

B. Core Technologies and Frameworks

gRPC

gRPC is a high-performance Remote Procedure Call (RPC) framework that utilizes HTTP/2 and Protocol Buffers for efficient communication. It offers improved performance, strong typing, bidirectional streaming, and support for multiple languages, making it a key communication mechanism for UMA microservices.1 gRPC is an open-source framework designed to run in any environment, efficiently connecting services within and across data

centers with pluggable support for load balancing, tracing, health checking, and authentication.26

- **Reference:** gRPC. (n.d.). *gRPC Official Website*. Retrieved from https://grpc.io/ [26]

WebSockets

WebSockets, as a communication protocol, provide full-duplex, bidirectional channels over a single TCP connection. This enables real-time, two-way interaction with low latency, making it a suitable choice for dynamic client-server communication in UMA.1 The WHATWG WebSockets standard is the living specification for web applications using this protocol.5

- **Reference:** WHATWG. (n.d.). *WebSockets (HTML Living Standard)*. Retrieved from https://html.spec.whatwg.org/multipage/web-sockets.html [5]

Long Polling

Long Polling is a technique where the client repeatedly requests data from the server, maintaining a continuous connection. It is simple to implement using basic HTTP and compatible with most browsers, often serving as a fallback for WebSockets in UMA.1 The server responds only if new messages are available or a timeout is reached, emulating a real-time server push.28

- **Reference:** Educative. (n.d.). *What is HTTP Long Polling?*. Retrieved from https://www.educative.io/answers/what-is-http-long-polling [28]

NATS

NATS is a high-performance event broker recommended for fast, distributed event messaging on the server-side within UMA.1 It is a cloud-native, open-source messaging technology designed for adaptive edge and distributed systems, offering a single platform for streaming, Key-Value, Object store, and PubSub functionalities.29

- **Reference:** NATS.io. (n.d.). *Cloud Native, Open Source, High-performance Messaging*. Retrieved from https://nats.io/ [29]

Redis Streams

Redis Streams are a data structure functioning as an append-only log with operations for real-time event recording and syndication. They are recommended for fast, distributed event messaging on the server-side in UMA.1 Redis Streams support various trimming and consumption strategies, including consumer groups, and offer efficient inserts and reads through radix trees.31

- **Reference:** Redis. (n.d.). *Redis Streams*. Retrieved from https://redis.io/docs/latest/develop/data-types/streams/ [31]

Apache Kafka

Apache Kafka is an open-source distributed event streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. It is recommended for fast, distributed event messaging on

the server-side in UMA due to its high throughput, scalability, permanent storage, and high availability.1

The inclusion of NATS, Redis Streams, and Kafka as recommended event brokers suggests that UMA's server-side eventing layer is highly flexible and adaptable to specific contexts. While all three offer high performance, they present different architectural trade-offs: NATS excels in simplicity and edge-to-cloud connectivity, Redis Streams provides lightweight, real-time event sourcing with robust consumer group capabilities, and Kafka offers a highly scalable, durable, and feature-rich platform for stream processing. This architectural flexibility allows implementers to select the most appropriate event backbone based on specific workload characteristics, data durability requirements, and integration needs, rather than prescribing a single solution. This adaptability is a key strength for a "universal" architecture.

- **Reference:** Apache Kafka. (n.d.). *Official Website*. Retrieved from https://kafka.apache.org/ [33]

Java Virtual Machine (JVM)

The JVM is a managed bytecode platform considered as a candidate portable binary format for UMA. It offers strong cross-platform support but requires language-specific virtual machines for execution, which contrasts with WASM's more lightweight runtime model.1

- **Reference:** Oracle. (n.d.). *The Java Virtual Machine Specification*. Retrieved from https://archive.org/details/the-java-virtual-machine-specification-java-se-16-edition [34]

.NET (Common Language Runtime - CLR)

Similar to the JVM,.NET is a managed bytecode platform considered as a candidate portable binary format for UMA. It provides cross-platform support but necessitates language-specific virtual machines, presenting a different operational footprint compared to WASM.1 Microsoft's.NET documentation provides comprehensive resources for developing applications across various platforms.35

- **Reference:** Microsoft. (n.d.). *.NET documentation*. Retrieved from https://learn.microsoft.com/en-us/dotnet/ [35]

GraalVM Native Images

GraalVM Native Images provide optimized performance and standalone execution, making them well-suited for backend microservices. However, this technology produces platform-specific binaries, which stands in contrast to WASM's universality in UMA's context.1 Native Image compiles Java code ahead-of-time into a native executable, offering faster startup and lower resource consumption.37

- **Reference:** Oracle. (n.d.). *GraalVM Native Image Reference Manual*. Retrieved

from https://www.graalvm.org/latest/reference-manual/native-image/ [37]

Docker

Docker provides containerized executables that are highly effective for backend orchestration and isolation. However, they are deemed unsuitable for mobile or web platforms within UMA due to their reliance on container engines.1 Docker Docs offers comprehensive resources for installing and using Docker products.39

- **Reference:** Docker. (n.d.). *Docker Docs*. Retrieved from https://docs.docker.com/ [39]

Emscripten

Emscripten is a compiler toolchain used to compile pthread-based C/C++ code to WASM. This enables existing native codebases to be integrated into UMA, leveraging the performance benefits of WASM while maintaining compatibility with established programming paradigms.1 Emscripten focuses on speed, size, and Web platform compatibility.42

- **Reference:** Emscripten. (n.d.). *Official Website*. Retrieved from https://emscripten.org/ [42]

Web Workers

Web Workers are used in web environments to execute WASM threads and manage concurrency, allowing UMA services to run in background threads without interfering with the user interface.1 MDN documentation provides a detailed introduction to using web workers, which run scripts in a global context different from the main window.44

- **Reference:** Mozilla. (n.d.). *Using Web Workers*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers [44]

SharedArrayBuffer

SharedArrayBuffer is used in web environments for WASM threads to run concurrently with shared memory access. Its functionality requires specific COOP/COEP headers due to security considerations.1 This object represents a raw binary data buffer that allows views on shared memory across different agents, such as the main web page and its web workers.46

The combined use of Web Workers and SharedArrayBuffer directly addresses the inherent limitations of JavaScript's single-threaded nature in browsers for computationally intensive tasks. While Web Workers enable background execution, the overhead of data copying via postMessage() can be significant. SharedArrayBuffer provides a crucial mechanism for true shared-memory concurrency, which is vital for high-performance WASM threads. The mention of mimalloc to mitigate issues with Web Worker startup latency and memory use further emphasizes the intricate performance considerations and the need for specialized tools to optimize WASM's multithreading capabilities within the

browser's security model. This demonstrates UMA's commitment to pushing the boundaries of client-side performance.

- **Reference:** Mozilla. (n.d.). *SharedArrayBuffer*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer [46]

Grand Central Dispatch (GCD)
GCD is a native threading system used on iOS for managing concurrency within UMA's mobile applications.1 It is a low-level API for managing operations asynchronously or synchronously, built on top of multiple threads and managing a shared thread pool.48

- **Reference:** Apple. (n.d.). *Dispatch | Apple Developer Documentation*. Retrieved from https://developer.apple.com/documentation/dispatch [48]

NSOperationQueue
NSOperationQueue is another native threading system used on iOS for managing concurrency within UMA's mobile applications.1 It organizes and invokes operations according to their readiness, priority level, and interoperation dependencies, executing them on separate threads.53

- **Reference:** Apple. (n.d.). *OperationQueue | Apple Developer Documentation*. Retrieved from https://developer.apple.com/documentation/foundation/nsoperationqueue [53]

Java threading / ExecutorService
Java threading and the ExecutorService API provide native threading systems used on Android for managing concurrency in UMA. The java.util.concurrent package offers high-level APIs for concurrent task execution and thread pool management.1 ExecutorService extends the Executor interface, providing methods to manage task termination and produce Future objects for tracking asynchronous tasks.[60]

- **Reference:** Oracle. (n.d.). *Package java.util.concurrent*. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html [56]

Kotlin Coroutines
Kotlin Coroutines are lightweight alternatives to threads used on Android for managing concurrency in UMA, enabling asynchronous code to be written in a natural, sequential style.1 The
kotlinx.coroutines library provides tools for launching coroutines, handling concurrency, and working with asynchronous streams.[61]

The white paper's explicit mention of these diverse, platform-specific concurrency mechanisms, despite its focus on WebAssembly's cross-platform nature, indicates a sophisticated understanding of practical implementation. It suggests that while

WASM provides the portable binary format, the UMA runtime must still leverage native platform capabilities for optimal concurrency and performance. This is a crucial design decision, acknowledging that a "universal" architecture does not imply a "one-size-fits-all" concurrency model, but rather an intelligent adaptation to the underlying operating system and runtime environment for peak efficiency. This also implies that UMA's "Abstraction Layer" is responsible for mapping WASM's multi-threading needs to these diverse native threading systems.

- **Reference:** Kotlin. (n.d.). *Coroutines | Kotlin Documentation*. Retrieved from https://kotlinlang.org/docs/coroutines-overview.html [61]

Wasmer
Wasmer is an embedded runtime for WASM modules on mobile platforms and is also considered for the Server-side Low Latency Runtime in UMA.1 It is a container technology powered by WebAssembly that allows users to run programs securely, quickly, and at scale, locally or in the cloud, simplifying affordable scaling.63

- **Reference:** Wasmer. (n.d.). *Official Website*. Retrieved from https://wasmer.io/ [63]

Wasmtime
Wasmtime is an embedded runtime for WASM modules on mobile platforms and is also considered for the Server-side Low Latency Runtime in UMA. It offers Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation, full WASI compliance, and integration with Rust.1 Wasmtime is a fast and secure runtime for WebAssembly, built on the Cranelift optimizing code generator for quick generation of high-quality machine code.64

- **Reference:** Wasmtime. (n.d.). *Official Website*. Retrieved from https://wasmtime.dev/ [64]

WasmEdge
WasmEdge is a high-performance WebAssembly runtime specifically optimized for cloud-native and edge environments. It is recommended for the UMA Server Low Latency Runtime, supporting AOT compilation, fast startup, and full WASI compatibility.1 WasmEdge is lightweight, high-performance, and extensible, powering serverless apps, embedded functions, microservices, smart contracts, and IoT devices.66
The detailed evaluation and specific recommendation of WasmEdge (and consideration of Wasmtime) for the server-side runtime reveals a critical focus on performance characteristics crucial for UMA's dynamic execution and load distribution. The comparison to Linux containers, where WasmEdge is significantly faster and smaller, highlights a key advantage of WASM in cloud-native and edge contexts where resource efficiency and rapid startup are paramount. The existence of many runtimes, such as WAMR for embedded systems and Wasm3 for interpreters, suggests a nuanced understanding of different deployment scenarios and the need to select the right tool for the right job, even within the WASM

ecosystem. This reinforces UMA's commitment to highly optimized, context-aware runtime environments.

- **Reference:** WasmEdge. (n.d.). *Official Website*. Retrieved from https://wasmedge.org/ [66]

WebAssembly Micro Runtime (WAMR)
WAMR is a lightweight WASM runtime suitable for embedded systems and low-power edge nodes. It offers rapid initialization but with reduced performance compared to other runtimes, making it ideal for resource-constrained environments within UMA.1

- **Reference:** Bytecode Alliance. (n.d.). *WAMR: A Lightweight WebAssembly Runtime*. Retrieved from https://bytecodealliance.org/articles/wamr-a-lightweight-webassembly-runtime [67]

Wasm3
Wasm3 is a fast WebAssembly interpreter and a universal WASM runtime. It is designed to be efficient in terms of runtime executable size, memory usage, and startup latency, offering an alternative to faster JIT compilers in scenarios where speed is not the primary concern.1 It is particularly useful on platforms like iOS and WebAssembly itself, where generating executable code pages at runtime is not possible.68

- **Reference:** Wasm3. (n.d.). *GitHub Repository*. Retrieved from https://github.com/wasm3/wasm3 [68]

Extism
Extism is another WebAssembly runtime evaluated for UMA, each with specific characteristics regarding performance, startup time, and use cases, contributing to the flexibility of runtime choices within the architecture.1

- **Reference:** Extism. (n.d.). *Official Website*. Retrieved from https://extism.org/ [69]

Wazero
Wazero is a WebAssembly runtime developed by Tetrate.io, Inc., designed for Go developers. It is notable for having "zero dependencies" and not relying on CGO, which allows Go applications to be extended with code written in other languages, offering another flexible runtime option for UMA.1

- **Reference:** Tetrate.io. (n.d.). *Wazero GitHub Repository*. Retrieved from https://github.com/tetratelabs/wazero [70]

Lunatic
Lunatic is another WebAssembly runtime evaluated for UMA, each with specific characteristics regarding performance, startup time, and use cases, contributing to the comprehensive consideration of runtime environments for the architecture.1

- **Reference:** Lunatic. (n.d.). *Official Website*. Retrieved from https://lunatic.io/ [71]

mimalloc

Mimalloc is an efficient memory allocator used in UMA to mitigate issues with Web Worker startup latency and memory use in browser-based WASM threading.1 It is a general-purpose memory allocator designed for excellent performance and is a compact library, utilizing simple and consistent data structures.72

- **Reference:** Microsoft. (n.d.). *mimalloc GitHub Repository*. Retrieved from https://github.com/microsoft/mimalloc [72]

Kubernetes (K8s)

Kubernetes is a container orchestration system that WasmEdge integrates with natively, making it ideal for dynamic orchestration and elastic scaling of server-side microservices within UMA.1 K8s is an open-source system designed for automating the deployment, scaling, and management of containerized applications.73

- **Reference:** Kubernetes. (n.d.). *Official Website*. Retrieved from https://kubernetes.io/ [73]

AWS Lambda

AWS Lambda is a serverless computing service mentioned as an example for the Microservice Registry's operational logic in UMA, showcasing a potential deployment model that eliminates the need for provisioning or managing servers.1 Lambda handles underlying infrastructure management, allowing developers to focus solely on their code.74

- **Reference:** Amazon Web Services. (n.d.). *AWS Lambda*. Retrieved from https://aws.amazon.com/lambda/ [74]

Amazon S3

Amazon S3 (Simple Storage Service) is an example of object storage utilized by the Microservice Registry to efficiently deliver microservice artifacts in UMA.1 S3 offers industry-leading scalability, data availability, security, and performance for storing and managing any amount of data.75

- **Reference:** Amazon Web Services. (n.d.). *Amazon S3*. Retrieved from https://aws.amazon.com/s3/ [75]

Amazon CloudFront

Amazon CloudFront is an example of a Content Delivery Network (CDN) used by the Microservice Registry to efficiently deliver microservice artifacts with low latency and global availability in UMA.1 CloudFront is a low-latency CDN that securely delivers content through over 600 globally dispersed Points of Presence (PoPs).76

The selection of AWS Lambda, Amazon S3, and Amazon CloudFront for the Microservice Registry highlights UMA's reliance on a robust, scalable, and globally distributed cloud infrastructure. Lambda ensures that the registry's operational logic is cost-efficient and scales automatically with demand. S3 provides highly durable and scalable storage for the microservice binaries themselves. Crucially,

CloudFront ensures that these binaries are delivered to diverse client and edge runtimes with minimal latency, regardless of geographic location. This infrastructure choice directly supports UMA's dynamic execution and load distribution by making microservice artifacts readily available worldwide, underpinning the "universal" aspect of the architecture from a deployment perspective.

- **Reference:** Amazon Web Services. (n.d.). *Amazon CloudFront*. Retrieved from https://aws.amazon.com/cloudfront/ [76]

SonarQube
SonarQube is a self-managed static analysis tool designed for continuous codebase inspection, automating code quality and security reviews. It can be integrated into CI/CD pipelines to monitor code complexity and enforce quality gates within UMA's development processes.1 SonarQube provides actionable code intelligence and integrates with popular CI/CD platforms.77

- **Reference:** SonarSource. (n.d.). *SonarQube Official Website*. Retrieved from https://www.sonarsource.com/products/sonarqube/ [77]

Metal SDK
Metal SDK is a device-specific SDK mentioned for native iOS binaries, indicating UMA's ability to integrate with platform-specific optimizations where necessary to achieve peak performance on Apple devices.1 It allows rendering advanced 3D graphics and computing data in parallel with graphics processors.78

- **Reference:** Apple. (n.d.). *Metal | Apple Developer Documentation*. Retrieved from https://developer.apple.com/documentation/metal [78]

JavaScript bundles, Kotlin modules
These are mentioned as examples of native binaries for web and Android environments, respectively. Their inclusion showcases the diverse output formats UMA considers and integrates to achieve its universal compatibility across various platforms.1

- **Reference:** (General programming language/platform documentation, e.g., MDN Web Docs for JavaScript, Kotlinlang.org for Kotlin)

C. Architectural Concepts and Patterns

Client-Side Microservices Architecture (CSMA)
UMA is presented as a progression and rethinking of CSMA, which provides foundational principles for modular, service-oriented design on the frontend. This conceptual lineage highlights UMA's evolution from established patterns in client-side development.1

- **Reference:** (No specific external reference provided in snippets)

Single-responsibility principle
This is a fundamental design principle stating that each service should have a clear and

focused purpose. Adherence to this principle is crucial for effective microservices design within UMA, promoting modularity, testability, and maintainability.1

- **Reference:** (No specific external reference provided in snippets)

ACID transactions
ACID (Atomicity, Consistency, Isolation, Durability) refers to traditional transactional properties that are impractical in UMA's distributed model. The white paper acknowledges the limitations of these properties in highly distributed environments, leading to the adoption of alternative consistency models.1

- **Reference:** (No specific external reference provided in snippets)

Saga pattern
The Saga pattern is an architectural pattern implemented in UMA to orchestrate long-running, multi-step workflows without centralized transactions. It breaks processes into independent, compensatable steps, managing distributed transactions through either orchestration or choreography styles.1
This architectural choice highlights a fundamental trade-off inherent in highly distributed systems like UMA: sacrificing strong, immediate consistency (as provided by ACID transactions) for increased availability, partition tolerance, and scalability. By embracing patterns like Saga, UMA acknowledges the realities of network latency and partial failures in a distributed environment, prioritizing resilience and performance over strict transactional guarantees. This is a common and necessary design choice in modern cloud-native architectures, and its explicit mention signals UMA's adherence to these pragmatic distributed system principles.

- **Reference:** (No specific external reference provided in snippets)

Last-Write-Wins, vector clocks, Conflict-free Replicated Data Types (CRDTs)
These are strategies supported by UMA to manage conflicting updates and achieve data consistency in a decentralized environment. They represent approaches to eventual consistency, which is often necessary in highly distributed systems where strong consistency is difficult to maintain.1

- **Reference:** (No specific external reference provided in snippets)

Book Keeper pattern
The Book Keeper pattern is a reconciliation pattern used in UMA to track local changes and request server acknowledgment to confirm synchronization during offline execution. This pattern is critical for enabling robust offline capabilities and ensuring data integrity when connectivity is intermittent.1

- **Reference:** (No specific external reference provided in snippets)

Ports and Adapters (Hexagonal Architecture)
Ports and Adapters, also known as Hexagonal Architecture, is an interface design pattern

used in UMA that inverts dependencies, isolating core business logic from external concerns. This separation is crucial for maintaining the platform-agnostic nature of UMA's microservices.1

- **Reference:** (No specific external reference provided in snippets)

Bridge Pattern, Strategy Pattern, Abstract Factory Pattern, Facade Pattern
These are interface design patterns employed in UMA to manage complexity and promote adaptability. The Bridge Pattern separates service interfaces from their platform-specific implementations, allowing independent changes and easier extension. The Strategy Pattern enables UMA services to adjust behavior based on runtime or platform characteristics without embedding platform conditionals in the service logic. The Abstract Factory Pattern encapsulates object creation, making it easy to switch between OS-specific implementations. The Facade Pattern simplifies complex APIs by providing a single entry point.1
The deliberate application of these specific design patterns is a direct architectural response to the complexity introduced by UMA's "universal" ambition. UMA's nature implies supporting diverse platforms and dynamic execution, which inherently introduces significant complexity in managing platform-specific implementations and runtime adaptations. These patterns serve as the "glue" that allows the platform-agnostic WASM binaries to interact seamlessly and optimally with their diverse native environments, showcasing a mature architectural approach to achieving universality.

- **Reference:** (No specific external reference provided in snippets)

Content Delivery Networks (CDNs)
CDNs are distributed networks used by the Microservice Registry in UMA to efficiently deliver microservice artifacts with low latency and global availability. This ensures that microservices can be quickly accessed and deployed to diverse client and edge environments.1 Amazon CloudFront is a specific example of a CDN used for this purpose.76

- **Reference:** (General concept, but CloudFront [76] is a specific example)

Cloud Native Computing Foundation (CNCF)
The CNCF is an ecosystem that WasmEdge is part of, reinforcing its readiness for production use in cloud-native environments. This affiliation signals WasmEdge's alignment with broader industry trends and best practices in cloud-native development.1

- **Reference:** CNCF. (n.d.). *Official Website*. Retrieved from https://cncf.io/ (External Organization)

COOP/COEP headers (Cross-Origin-Opener-Policy / Cross-Origin-Embedder-Policy)
These HTTP headers are required for SharedArrayBuffer to function in some browsers, which can present limitations for WASM threading in web environments. Their necessity highlights the security considerations involved in enabling advanced web capabilities like

shared memory.1

- **Reference:** Mozilla. (n.d.). *Cross-Origin-Opener-Policy (COOP)*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy [46]

Event-driven architecture
Event-driven architecture is a core concept in UMA, enabling asynchronous interactions between microservices within the Universal Runtime Application and across the client-server boundary. This architectural style promotes loose coupling and scalability.1

- **Reference:** (No specific external reference provided in snippets)

Edge computing
Edge computing is a computing paradigm leveraged by UMA to minimize latency and operational costs by allowing services to run on edge devices. This approach brings computation closer to the data source, improving responsiveness and reducing bandwidth usage.1

- **Reference:** (No specific external reference provided in snippets)

Distributed transactions
Distributed transactions are managed in UMA using patterns like Saga due to the decentralized nature of data management. This approach addresses the challenges of maintaining consistency across multiple independent services without relying on traditional, often impractical, two-phase commit protocols.1

- **Reference:** (No specific external reference provided in snippets)

Change Data Capture (CDC) and Anti-Entropy
These are techniques used in UMA to detect and resolve data divergence across nodes or disconnected clients, supporting UMA's data consistency model. CDC captures changes to data sources, while anti-entropy mechanisms reconcile inconsistencies, ensuring data eventual consistency in a distributed environment.1

- **Reference:** (No specific external reference provided in snippets)

Idempotent Operations
Idempotent operations are a design principle where services must ensure that repeated or duplicate requests do not result in data corruption. This is critical for fault tolerance and data consistency in distributed systems, especially in the presence of retries or network issues.1

- **Reference:** (No specific external reference provided in snippets)

Continuous Integration (CI) and Deployment (CD)
CI/CD refers to automated pipelines that include steps for building, testing, linting, type checking, and validating code quality. These practices are essential for UMA's rapid development and release cycles, ensuring consistent quality and efficient delivery of

microservices.1

- **Reference:** (No specific external reference provided in snippets)

Microservice Registry
The Microservice Registry is a central component in UMA responsible for managing and serving microservices to execution runtimes. It provides an API for CRUD (Create, Read, Update, Delete) operations and on-demand distribution of service binaries, acting as a central catalog for available services.1

- **Reference:** (No specific external reference provided in snippets)

API Gateway
The API Gateway serves as the entry point for clients requesting the execution of microservices in UMA. It handles routing, authentication, and response handling, providing a unified interface to the underlying distributed services.1

- **Reference:** (No specific external reference provided in snippets)

Service Loader
The Service Loader is a UMA component that determines whether a specific service should be executed locally or delegated to the server. This decision is based on runtime state, hardware capabilities, and client capabilities, enabling UMA's dynamic execution model.1

- **Reference:** (No specific external reference provided in snippets)

Execution Engine
The Execution Engine in UMA is responsible for executing WebAssembly microservices that the Service Loader has queued. It serves as the core runtime environment for WASM binaries across diverse platforms.1

- **Reference:** (No specific external reference provided in snippets)

Abstraction Layer
The Abstraction Layer enables universal microservices to interact with their environment in a platform-agnostic way. It provides standardized APIs and utility functions, decoupling the core service logic from platform-specific details and contributing to UMA's portability.1

- **Reference:** (No specific external reference provided in snippets)

Multithreading
Multithreading is a fundamental requirement for UMA-based architectures to support scalable execution across diverse platforms by running multiple parts of a program concurrently. This is essential for maximizing resource utilization and responsiveness.1

- **Reference:** (No specific external reference provided in snippets)

Fault Tolerance
Fault tolerance refers to the ability of UMA services to be resilient to partial failure. This is achieved through strategies like redundancy, circuit breakers, retry logic, and graceful degradation, ensuring the system remains operational despite component failures.1

- **Reference:** (No specific external reference provided in snippets)

Data Consistency
Data consistency in UMA is maintained through event-driven state propagation, asynchronous messaging, change data capture, idempotent operations, and conflict resolution strategies. This acknowledges the challenges of maintaining strong consistency in a distributed system and opts for models like eventual consistency.1

- **Reference:** (No specific external reference provided in snippets)

Supply Chain Security
Supply chain security encompasses practices to secure external libraries or dependencies included in UMA microservices. This includes generating SBOMs, performing dependency scanning, and employing version pinning to mitigate risks from third-party components.1

- **Reference:** (No specific external reference provided in snippets)

Atomic Design Principles
Atomic Design Principles focus on creating small, self-contained services. In UMA, this approach improves testability, reliability, and ease of automation, contributing to the overall agility of the development process.1

- **Reference:** (No specific external reference provided in snippets)

Decentralized Microservice Runtime
The Decentralized Microservice Runtime is UMA's core model, where services can run across clients, servers, and edge devices. This distributed execution environment is central to UMA's promise of universality and flexibility.1

- **Reference:** (No specific external reference provided in snippets)

Event-Driven State Propagation
In UMA, Event-Driven State Propagation is a mechanism where changes in service state are emitted as events and asynchronously consumed by downstream services to maintain data consistency. This asynchronous communication pattern supports scalability and resilience in a distributed environment.1

- **Reference:** (No specific external reference provided in snippets)

Offline Execution and State Reconciliation
UMA services are designed with the capability to handle intermittent connectivity and reconcile state later. This is achieved using local data storage with deferred synchronization, versioned state models, delta synchronization, and polling mechanisms, ensuring functionality even when disconnected.1

- **Reference:** (No specific external reference provided in snippets)

Dynamic Execution and Load Distribution
Dynamic Execution and Load Distribution is a defining feature of UMA, where the runtime continuously evaluates the optimal location to execute each microservice. This evaluation is based on factors such as CPU availability, memory pressure, and power constraints.1

The ability to dynamically shift microservice execution between client, edge, and server based on real-time metrics (CPU, memory, power) is the direct mechanism by which UMA delivers its promise of efficiency and responsiveness. This is not merely about portability through WASM; it involves intelligent orchestration of portable components. This feature fundamentally differentiates UMA from traditional microservices or even client-side microservices architectures. It suggests a highly adaptive and self-optimizing system, where the runtime itself acts as a distributed orchestrator. This has significant implications for operational cost reduction, improved user experience through lower latency, and enhanced resilience in variable network and device conditions, moving beyond static deployment decisions to truly dynamic resource allocation.

- **Reference:** (No specific external reference provided in snippets)

Sandboxing
Sandboxing is a security mechanism used in WASM runtimes within UMA to isolate microservices and restrict their access to host APIs. This confinement helps prevent malicious or faulty code from affecting the entire system.1

- **Reference:** (No specific external reference provided in snippets)

Code Signing
Code signing is a security measure in UMA where all binaries are cryptographically signed before being published. The runtime verifies these signatures before loading any service, ensuring supply chain security and preventing unauthorized code execution.1

- **Reference:** (No specific external reference provided in snippets)

Runtime Attestation
For high-security use cases, UMA can incorporate runtime attestation to verify the integrity and trustworthiness of the execution environment before running a service. This provides an additional layer of security by ensuring that the underlying platform is uncompromised.1

- **Reference:** (No specific external reference provided in snippets)

JWT tokens (JSON Web Tokens)
JWT tokens are used by clients for authentication before invoking services in UMA, often in conjunction with OAuth2. These tokens provide a compact and self-contained way for securely transmitting information between parties as a JSON object.1

- **Reference:** (No specific external reference provided in snippets)

mTLS (mutual TLS)
mTLS is used for securing event messages transmitted over gRPC and WebSocket channels in UMA. This protocol ensures strong authentication and encryption between services, where both the client and server authenticate each other, enhancing overall system security.1

- **Reference:** (No specific external reference provided in snippets)

AI-assisted software development
UMA is positioned to benefit from AI-assisted software development, with its granular service model and well-defined interfaces fitting naturally for systems where services can be discovered, composed, or generated autonomously. This aligns UMA with emerging trends in software engineering.1

- **Reference:** (No specific external reference provided in snippets)

Table 1: Categorized References

This table provides a structured list of all external sources referenced, categorized by their type, along with their full titles, authors/organizations, publication/last updated dates (if available), and URLs. This organized presentation enhances the white paper's credibility, usability, and facilitates further research into the foundational components of UMA.

Category	Name of Technology/ Standard/Concept	Full Title of Reference Document/Website	Author/Organization	Publication/Last Updated Date	URL
Standards & Specs	WebAssembly (WASM) Core Spec	Specifications	WebAssembly	n.d.	https://webassembly.org/specs/
Standards & Specs	WebAssembly System Interface (WASI)	Specifications - WASI API	WebAssembly	n.d.	https://webassembly.org/specs/
Standards & Specs	WebAssembly Interface Types (WIT)	Specifications - Tool Conventions	WebAssembly	n.d.	https://webassembly.org/specs/
Standards & Specs	HTTP/2	RFC 9113: HTTP/2	IETF	2023	https://httpwg.org/specs/
Standards & Specs	WebSocket Protocol (RFC 6455)	RFC 6455: The WebSocket	IETF	December 2011	https://www.tech-invite.com/y60/tinv-i

		Protocol			etf-rfc-6455.html
Standards & Specs	Server-Sent Events (SSE)	HTML Living Standard: 9.2 Server-sent events	WHATWG	29 June 2025	https://html.spec.whatwg.org/multipage/server-sent-events.html
Standards & Specs	Protocol Buffers (Protobuf)	Protocol Buffers Overview	Google	n.d.	https://protobuf.dev/overview/
Standards & Specs	TLS 1.3 (RFC 8446)	RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3	IETF	August 2018	https://www.rfc-editor.org/rfc/rfc8446
Standards & Specs	OAuth 2.0 (RFC 6749)	RFC 6749: The OAuth 2.0 Authorization Framework	IETF	October 2012	https://www.rfc-editor.org/rfc/rfc6749
Standards & Specs	POSIX Threads (IEEE Std 1003.1)	IEEE Std 1003.1™-2024 Edition (POSIX.1-2024)	The Open Group	n.d.	https://pubs.opengroup.org/onlinepubs/9799919799/
Standards & Specs	CloudEvents Specification	Specification	CloudEvents	n.d.	https://cloudevents.io/
Standards & Specs	Open Container Initiative (OCI)	Specifications	Open Container Initiative	n.d.	https://opencontainers.org/
Standards & Specs	Semantic Versioning	SemVer	Semantic Versioning 2.0.0	n.d.	https://semver.org/
Standards &	Software Bill	Software Bill	NTIA	n.d.	https://www.

Specs	of Materials (SBOMs)	of Materials (SBOM)			ntia.gov/SBOM
Core Tech & Frameworks	gRPC	gRPC Official Website	gRPC	2025	https://grpc.io/
Core Tech & Frameworks	Long Polling	What is HTTP Long Polling?	Educative	n.d.	https://www.educative.io/answers/what-is-http-long-polling
Core Tech & Frameworks	NATS	Cloud Native, Open Source, High-performance Messaging	NATS.io	n.d.	https://nats.io/
Core Tech & Frameworks	Redis Streams	Redis Streams	Redis	n.d.	https://redis.io/docs/latest/develop/data-types/streams/
Core Tech & Frameworks	Apache Kafka	Official Website	Apache Kafka	n.d.	https://kafka.apache.org/
Core Tech & Frameworks	Java Virtual Machine (JVM)	The Java Virtual Machine Specification, Java SE 16 Edition	Oracle	2024-01-04	https://archive.org/details/the-java-virtual-machine-specification-java-se-16-edition
Core Tech & Frameworks	.NET (CLR)	.NET documentation	Microsoft	n.d.	https://learn.microsoft.com/en-us/dotnet/
Core Tech & Frameworks	GraalVM Native Images	GraalVM Native Image Reference Manual	Oracle	2025	https://www.graalvm.org/latest/reference-manual/native-image/
Core Tech &	Docker	Docker Docs	Docker	n.d.	https://docs.

Frameworks					docker.com/
Core Tech & Frameworks	Emscripten	Official Website	Emscripten	n.d.	https://emscripten.org/
Core Tech & Frameworks	Web Workers	Using Web Workers	Mozilla	Jun 19, 2025	https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
Core Tech & Frameworks	SharedArrayBuffer	SharedArrayBuffer	Mozilla	Mar 10, 2025	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
Core Tech & Frameworks	Grand Central Dispatch (GCD)	Dispatch | Apple Developer Documentation	Apple	n.d.	https://developer.apple.com/documentation/dispatch
Core Tech & Frameworks	NSOperationQueue	OperationQueue | Apple Developer Documentation	Apple	n.d.	https://developer.apple.com/documentation/foundation/nsoperationqueue
Core Tech & Frameworks	Java threading / ExecutorService	Package java.util.concurrent	Oracle	n.d.	https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
Core Tech & Frameworks	Kotlin Coroutines	Coroutines | Kotlin Documentati	Kotlin	20 June 2025	https://kotlinlang.org/docs/coroutines-

		on			overview.html
Core Tech & Frameworks	Wasmer	Official Website	Wasmer	n.d.	https://wasmer.io/
Core Tech & Frameworks	Wasmtime	Official Website	Wasmtime	n.d.	https://wasmtime.dev/
Core Tech & Frameworks	WasmEdge	Official Website	WasmEdge	n.d.	https://wasmedge.org/
Core Tech & Frameworks	WebAssembly Micro Runtime (WAMR)	WAMR: A Lightweight WebAssembly Runtime	Bytecode Alliance	n.d.	https://bytecodealliance.org/articles/wamr-a-lightweight-webassembly-runtime
Core Tech & Frameworks	Wasm3	GitHub Repository	Wasm3	n.d.	https://github.com/wasm3/wasm3
Core Tech & Frameworks	Extism	Official Website	Extism	n.d.	https://extism.org/
Core Tech & Frameworks	Wazero	Wazero GitHub Repository	Tetrate.io	n.d.	https://github.com/tetratelabs/wazero
Core Tech & Frameworks	Lunatic	Official Website	Lunatic	n.d.	https://lunatic.io/
Core Tech & Frameworks	mimalloc	mimalloc GitHub Repository	Microsoft	n.d.	https://github.com/microsoft/mimalloc
Core Tech & Frameworks	Kubernetes (K8s)	Official Website	Kubernetes	n.d.	https://kubernetes.io/
Core Tech & Frameworks	AWS Lambda	AWS Lambda	Amazon Web Services	n.d.	https://aws.amazon.com/lambda/
Core Tech & Frameworks	Amazon S3	Amazon S3	Amazon Web Services	n.d.	https://aws.amazon.com/s

					3/
Core Tech & Frameworks	Amazon CloudFront	Amazon CloudFront	Amazon Web Services	n.d.	https://aws.amazon.com/cloudfront/
Core Tech & Frameworks	SonarQube	SonarQube Official Website	SonarSource	n.d.	https://www.sonarsource.com/products/sonarqube/
Core Tech & Frameworks	Metal SDK	Metal | Apple Developer Documentation	Apple	n.d.	https://developer.apple.com/documentation/metal
Architectural Concepts	Client-Side Microservices Arch. (CSMA)	(Concept)	(General Concept)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Single-responsibility principle	(Principle)	(General Principle)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	ACID transactions	(Concept)	(General Concept)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Saga pattern	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Last-Write-Wins	(Strategy)	(General Strategy)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Vector Clocks	(Strategy)	(General Strategy)	n.d.	(No specific external

					reference provided in snippets)
Architectural Concepts	Conflict-free Replicated Data Types (CRDTs)	(Strategy)	(General Strategy)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Book Keeper pattern	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Ports and Adapters (Hexagonal Arch.)	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Bridge Pattern	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Strategy Pattern	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Abstract Factory Pattern	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Facade Pattern	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Cloud Native Computing	Official Website	CNCF	n.d.	https://cncf.io/

	Foundation (CNCF)				
Architectural Concepts	COOP/COEP headers	Cross-Origin-Opener-Policy (COOP)	Mozilla	n.d.	https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy
Architectural Concepts	Event-driven architecture	(Architectural Style)	(General Style)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Edge computing	(Computing Paradigm)	(General Paradigm)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Distributed transactions	(Concept)	(General Concept)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Change Data Capture (CDC)	(Technique)	(General Technique)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Anti-Entropy	(Technique)	(General Technique)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Idempotent Operations	(Principle)	(General Principle)	n.d.	(No specific external reference provided in snippets)

Architectural Concepts	Continuous Integration (CI)	(Practice)	(General Practice)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Continuous Deployment (CD)	(Practice)	(General Practice)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Microservice Registry	(Component)	(UMA Component)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	API Gateway	(Pattern)	(General Pattern)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Service Loader	(Component)	(UMA Component)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Execution Engine	(Component)	(UMA Component)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Abstraction Layer	(Component)	(UMA Component)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Multithreading	(Concept)	(General Concept)	n.d.	(No specific external reference provided in snippets)

Architectural Concepts	Fault Tolerance	(Property)	(General Property)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Data Consistency	(Property)	(General Property)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Supply Chain Security	(Practice)	(General Practice)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Atomic Design Principles	(Methodology)	(General Methodology)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Decentralized Microservice Runtime	(Concept)	(UMA Concept)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Event-Driven State Propagation	(Mechanism)	(UMA Mechanism)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Offline Execution & State Rec.	(Capability)	(UMA Capability)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Dynamic Execution & Load Dist.	(Capability)	(UMA Capability)	n.d.	(No specific external reference provided in snippets)

Architectural Concepts	Sandboxing	(Security Mechanism)	(General Mechanism)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Code Signing	(Security Practice)	(General Practice)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	Runtime Attestation	(Security Concept)	(General Concept)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	JWT tokens	(Token Format)	(General Format)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	mTLS (mutual TLS)	(Security Protocol)	(General Protocol)	n.d.	(No specific external reference provided in snippets)
Architectural Concepts	AI-assisted software development	(Emerging Trend)	(Emerging Trend)	n.d.	(No specific external reference provided in snippets)

III. Glossary of Terms

This section provides clear and concise definitions for key technical terms and acronyms used throughout the 'Universal Microservices Architecture' white paper.

A. Key Technical Terms and Concepts

Abstraction Layer: A component in UMA that enables universal microservices to interact with their environment in a platform-agnostic way, providing standardized APIs and utility functions.[1]

ACID Transactions: Traditional transactional properties (Atomicity, Consistency, Isolation, Durability) that are impractical in UMA's distributed model due to the inherent complexities of coordinating state across multiple independent services.[1]

AI-assisted Software Development: The emerging trend where UMA's granular service model and well-defined interfaces can facilitate autonomous discovery, composition, or generation of services, aligning with advancements in artificial intelligence to enhance development processes.[1]

API Gateway: In UMA, this serves as the entry point for clients requesting microservice execution, handling routing, authentication, and response. It provides a single, unified interface to a potentially complex backend of microservices.[1]

Atomic Design Principles: A design philosophy in UMA focusing on small, self-contained services to improve testability, reliability, and ease of automation. This promotes a modular and manageable codebase.[1]

Book Keeper Pattern: A reconciliation pattern used in UMA to track local changes and request server acknowledgment to confirm synchronization during offline execution. This is crucial for maintaining data consistency when clients operate with intermittent connectivity.[1]

Change Data Capture (CDC): A technique used in UMA to detect and resolve data divergence across nodes or disconnected clients. CDC captures changes made to data in a source system and propagates them to other systems, supporting data consistency in distributed environments.[1]

Client-Side Microservices Architecture (CSMA): A foundational concept for modular, service-oriented design on the frontend, which UMA progresses from. CSMA involves breaking down monolithic frontend applications into smaller, independent services.[1]

Code Signing: A security measure in UMA where all binaries are cryptographically signed before publication, and signatures are verified before loading any service. This ensures the authenticity and integrity of microservice artifacts, enhancing supply chain security.[1]

Conflict-free Replicated Data Types (CRDTs): Data structures that can be replicated across multiple servers, allowing concurrent updates without requiring

complex synchronization mechanisms. UMA supports these strategies to manage conflicting updates and achieve data consistency in a decentralized environment.[1]

Content Delivery Networks (CDNs): Distributed networks used by UMA's Microservice Registry to efficiently deliver microservice artifacts with low latency and global availability. CDNs cache content closer to users, reducing load times and improving user experience.[1]

Data Consistency: In UMA, this is maintained through event-driven state propagation, asynchronous messaging, change data capture, idempotent operations, and conflict resolution strategies. It refers to the state where all copies of data are the same or converge over time in a distributed system.[1]

Decentralized Microservice Runtime: UMA's core model where services can run across clients, servers, and edge devices. This distributed runtime environment enables flexible deployment and execution based on dynamic conditions.[1]

Distributed Transactions: Transactions managed across multiple independent nodes in UMA, typically using patterns like Saga due to the decentralized nature of data. Unlike traditional ACID transactions, these often involve eventual consistency.[1]

Dynamic Execution and Load Distribution: A defining feature of UMA where the runtime continuously evaluates the optimal location to execute each microservice based on factors like CPU availability, memory pressure, and power constraints. This adaptive capability optimizes performance and resource utilization.[1]

Edge Computing: A computing paradigm leveraged by UMA to minimize latency and operational costs by allowing services to run on edge devices, closer to the data sources or end-users. This reduces the need to send all data to a central cloud.[1]

Event-Driven Architecture: A core concept in UMA, enabling asynchronous interactions between microservices within the Universal Runtime Application and across the client-server boundary. Services communicate by producing and consuming events.[1]

Event-Driven State Propagation: A mechanism in UMA where changes in service state are emitted as events and asynchronously consumed by downstream services to maintain data consistency. This promotes loose coupling and scalability by avoiding direct synchronous calls.[1]

Execution Engine: In UMA, this is the component responsible for executing

WebAssembly microservices that the Service Loader has queued, serving as the core runtime environment. It translates WASM bytecode into native machine instructions.[1]

Fault Tolerance: The ability of UMA services to be resilient to partial failure, achieved through strategies like redundancy, circuit breakers, retry logic, and graceful degradation. This ensures the system can continue operating even if some components fail.[1]

Idempotent Operations: Operations in UMA services designed such that repeated or duplicate requests do not result in data corruption. This property is crucial for reliability in distributed systems, especially when network failures or retries are common.[1]

Last-Write-Wins: A conflict resolution strategy supported by UMA to manage conflicting updates and achieve data consistency in a decentralized environment. When multiple updates occur to the same data, the last one received is considered the authoritative version.[1]

Long Polling: A communication technique where the client maintains a continuous connection by repeatedly requesting data from the server. The server holds the request open until new data is available or a timeout occurs. It is often used as a fallback for WebSockets for real-time updates.[1]

Microservice Registry: A central component in UMA responsible for managing and serving microservices to execution runtimes. It provides an API for CRUD (Create, Read, Update, Delete) operations and on-demand distribution of service binaries, acting as a dynamic catalog for available services.[1]

Multithreading: A fundamental requirement for UMA-based architectures to support scalable execution across diverse platforms by running multiple parts of a program concurrently within a single process. This improves responsiveness and resource utilization.[1]

Offline Execution and State Reconciliation: UMA services' capability to handle intermittent connectivity by storing local data with deferred synchronization, versioned state models, delta synchronization, and polling for later reconciliation. This ensures applications remain functional even without a persistent network connection.[1]

Ports and Adapters (Hexagonal Architecture): An interface design pattern used in UMA that inverts dependencies, isolating core business logic from external

concerns such as databases, user interfaces, or external services. This promotes modularity and testability.[1]

Runtime Attestation: For high-security use cases, UMA can incorporate this to cryptographically verify the integrity and trustworthiness of the execution environment before running a service. This ensures that the underlying platform has not been tampered with.[1]

Saga Pattern: An architectural pattern implemented in UMA to orchestrate long-running, multi-step workflows without centralized transactions. It breaks processes into independent, compensatable steps, ensuring business consistency in distributed systems where ACID transactions are impractical.[1]

Sandboxing: A security mechanism used in WASM runtimes within UMA to isolate microservices and restrict their access to host APIs. This creates a secure, controlled environment for code execution, preventing unauthorized operations.[1]

Semantic Versioning: A structured and predictable way of assigning version numbers (MAJOR.MINOR.PATCH) to software releases, essential for managing changes to UMA microservices and defining clear API contracts.[1]

Server-Sent Events (SSE): A protocol enabling servers to push updates to clients over a single, long-lived HTTP connection. It is efficient for streaming events from server to client, typically used for one-way data flows.[1]

Service Loader: A UMA component that determines whether a specific service should be executed locally or delegated to the server. This decision is based on runtime state, hardware capabilities (like CPU availability and memory pressure), and client capabilities, enabling dynamic load distribution.[1]

Single-Responsibility Principle: A design principle stating that each UMA service should have a clear and focused purpose, responsible for only one area of functionality. This improves modularity and maintainability.[1]

Software Bill of Materials (SBOMs): A formal, machine-readable list of ingredients that make up software components. In UMA, SBOMs are automatically generated and stored with each service release to secure the supply chain by providing transparency into dependencies.[1]

Supply Chain Security: Practices in UMA to secure external libraries or dependencies included in microservices, such as generating SBOMs, performing dependency scanning, and version pinning. This aims to protect against vulnerabilities introduced through third-party components.[1]

Vector Clocks: A strategy supported by UMA to manage conflicting updates and achieve data consistency in a decentralized environment. Vector clocks are used to determine the partial ordering of events in a distributed system.[1]

WebAssembly (WASM): A core technology for UMA, enabling compilation of services into platform-agnostic binaries for high performance and broad compatibility across web, mobile, and server environments. WASM provides a portable, compact binary instruction format.[1]

Web Workers: Used in web environments to execute WASM threads and manage concurrency, allowing UMA services to run in background threads without blocking the main user interface thread. This improves application responsiveness.[1]

B. Acronyms and Abbreviations

This table lists all acronyms and abbreviations identified in the white paper, providing their full spellings and a brief, context-specific explanation where beneficial. This centralized reference improves comprehension and consistency for readers.

Table 2: Glossary of Acronyms

Acronym	Full Spelling	Brief Context/Explanation
ACID	Atomicity, Consistency, Isolation, Durability	Traditional transactional properties, impractical in UMA's distributed model.
AI	Artificial Intelligence	Broad field of computer science that UMA is positioned to benefit from.
API	Application Programming Interface	Set of definitions and protocols for building and integrating application software.
AOT	Ahead-Of-Time (compilation)	Compilation of code into machine code before execution, used by some WASM runtimes.

AWS	Amazon Web Services	Cloud computing platform, examples like Lambda, S3, CloudFront are used.
CDC	Change Data Capture	Technique to detect and resolve data divergence across nodes.
CDN	Content Delivery Network	Distributed network used to deliver content efficiently with low latency.
CI/CD	Continuous Integration/Continuous Deployment	Automated pipelines for building, testing, and deploying software.
CLR	Common Language Runtime	The virtual machine component of Microsoft's.NET framework.
CNCF	Cloud Native Computing Foundation	Ecosystem that WasmEdge is part of, reinforcing production readiness.
COEP	Cross-Origin-Embedder-Polic y	HTTP header required for SharedArrayBuffer to function securely.
COOP	Cross-Origin-Opener-Policy	HTTP header required for SharedArrayBuffer to function securely.
CPU	Central Processing Unit	The electronic circuitry that executes instructions of a computer program.
CRDTs	Conflict-free Replicated Data Types	Strategies for managing conflicting updates in decentralized environments.
CSMA	Client-Side Microservices Architecture	Foundational concept for modular, service-oriented design on the frontend.
Dapr	Distributed Application Runtime	Portable, event-driven runtime for building microservices on cloud and edge.

DDoS	Distributed Denial of Service	Cyber-attack where multiple compromised computer systems attack a target.
DOM	Document Object Model	Programming interface for HTML and XML documents, not directly manipulated by Web Workers.
GCD	Grand Central Dispatch	Native threading system used on iOS for managing concurrency.
gRPC	gRPC (Remote Procedure Call)	High-performance RPC framework using HTTP/2 and Protocol Buffers.
GUID	Globally Unique Identifier	A 128-bit number used as a unique identifier.
HTML	Hypertext Markup Language	Standard markup language for documents designed to be displayed in a web browser.
HTTP	Hypertext Transfer Protocol	Application protocol for distributed, collaborative, hypermedia information systems.
HTTP/2	Hypertext Transfer Protocol Version 2	Major revision of HTTP, used by gRPC for improved performance.
IANA	Internet Assigned Numbers Authority	Organization responsible for global coordination of IP addressing, DNS, and protocol parameters.
IaC	Infrastructure as Code	Managing and provisioning computer data centers through machine-readable definition files.
IETF	Internet Engineering Task Force	Organization that develops and promotes Internet standards.
IESG	Internet Engineering Steering	Body responsible for technical

	Group	management of IETF activities.
IoT	Internet of Things	Network of physical objects embedded with sensors and other technologies.
JIT	Just-In-Time (compilation)	Compilation of code during program execution.
JWT	JSON Web Tokens	Compact, URL-safe means of representing claims to be transferred between two parties.
JVM	Java Virtual Machine	Managed bytecode platform, candidate portable binary format in UMA.
K8s	Kubernetes	Open-source system for automating deployment, scaling, and management of containerized applications.
KVC	Key-Value Coding	Mechanism for accessing object properties indirectly by name.
KVO	Key-Value Observing	Mechanism that allows objects to observe changes to properties of other objects.
LF	Line Feed	Control character that advances the paper in a printer or the cursor on a display one line.
LLMs	Large Language Models	Advanced AI models capable of generating human-like text.
LIFO	Last-In, First-Out	Data structure where the last element added is the first one removed.
mTLS	mutual Transport Layer Security	Security protocol where both client and server authenticate each other.

NIST SSDF	National Institute of Standards and Technology Secure Software Development Framework	Framework for secure software development practices.
OCI	Open Container Initiative	Linux Foundation project to design open standards for containers.
OMA	Open Mobile Alliance	Standards body for mobile service enablers.
OTA	Over-The-Air	Method of delivering software updates or configurations wirelessly.
OWASP	Open Web Application Security Project	Online community that produces freely available articles, methodologies, documentation, and tools in web application security.
PoPs	Points of Presence	Data centers or facilities that house servers and networking equipment.
POSIX	Portable Operating System Interface	Family of standards specified by the IEEE for maintaining compatibility between operating systems.
Protobuf	Protocol Buffers	Language-neutral, platform-neutral, extensible mechanism for serializing structured data.
QoS	Quality of Service	Description or measurement of the overall performance of a service.
RAM	Random Access Memory	Form of computer memory that can be read and changed in any order.
RAG	Retrieval Augmented Generation	AI technique that combines information retrieval with text generation.

RFC	Request for Comments	Publication in a series of memoranda for Internet standards.
RPC	Remote Procedure Call	Protocol that allows a program to request a service from a program located on another computer.
S3	Simple Storage Service	Amazon's object storage service.
SBOMs	Software Bill of Materials	Formal, machine-readable list of ingredients that make up software components.
SDK	Software Development Kit	Set of software development tools that allows the creation of applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar development platform.
SHA-1	Secure Hash Algorithm 1	Cryptographic hash function.
SIMD	Single Instruction, Multiple Data	Class of parallel computers that can perform the same operation on multiple data points simultaneously.
SQL	Structured Query Language	Domain-specific language used in programming and designed for managing data held in a relational database management system.
SSE	Server-Sent Events	Protocol enabling servers to push updates to clients over a single, long-lived HTTP connection.
STIG	Security Technical Implementation Guides	Cybersecurity methodologies for standardizing security protocols.

TCP	Transmission Control Protocol	One of the main protocols of the Internet protocol suite.
TLS	Transport Layer Security	Cryptographic protocol designed to provide communications security over a computer network.
UMA	Universal Microservices Architecture	The subject of the white paper, emphasizing platform-agnostic execution.
URI	Uniform Resource Identifier	String of characters used to identify a name or a web resource.
VM	Virtual Machine	Emulation of a computer system.
WASI	WebAssembly System Interface	Standard that securely exposes host-level capabilities to sandboxed WebAssembly.
WASM	WebAssembly	Core technology for UMA, enabling compilation of services into platform-agnostic binaries.
WHATWG	Web Hypertext Application Technology Working Group	Community that maintains the HTML Living Standard.
W3C	World Wide Web Consortium	International community that develops open standards to ensure the long-term growth of the Web.
WIT	WebAssembly Interface Types	Used with WASI for defining language-agnostic interfaces for capabilities.
XML	Extensible Markup Language	Markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

IV. Conclusion

This supplementary report provides a meticulously structured "Reference Section" and "Glossary of Terms" for the 'Universal Microservices Architecture' white paper. The "Reference Section" meticulously lists authoritative sources for all external technologies and standards, categorized for ease of navigation and verification. The "Glossary of Terms" comprehensively defines key technical concepts and acronyms, ensuring clarity and consistency for readers.

Through this detailed work, the white paper is significantly augmented, offering readers not only a deeper understanding of UMA's intricate design but also the necessary tools to explore its foundational components and terminology with confidence and precision. This enhancement elevates the white paper's standing as a definitive resource in the domain of distributed systems and microservices architecture, providing a robust foundation for further academic inquiry and practical implementation.

Works cited

1. UMA - White Paper.pdf
2. Specifications - WebAssembly, accessed June 30, 2025, https://webassembly.org/specs/
3. WebAssembly - Wikipedia, accessed June 30, 2025, https://en.wikipedia.org/wiki/WebAssembly
4. HTTP Documentation - IETF HTTP Working Group, accessed June 30, 2025, https://httpwg.org/specs/
5. WebSocket - Wikipedia, accessed June 30, 2025, https://en.wikipedia.org/wiki/WebSocket
6. RFC 6455 (Dec 2011, Proposed STD, 71 pages): 1 of 4, p. 1 to 14, accessed June 30, 2025, https://www.tech-invite.com/y60/tinv-ietf-rfc-6455.html
7. RFC 6455: The WebSocket Protocol, accessed June 30, 2025, https://www.rfc-editor.org/rfc/rfc6455
8. Server-sent events - Wikipedia, accessed June 30, 2025, https://en.wikipedia.org/wiki/Server-sent_events
9. Using server-sent events - Web APIs | MDN, accessed June 30, 2025, https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events
10. 9.2 Server-sent events - HTML Standard - whatwg, accessed June 30, 2025, https://html.spec.whatwg.org/multipage/server-sent-events.html
11. Overview | Protocol Buffers Documentation, accessed June 30, 2025, https://protobuf.dev/overview/
12. protobuf - Documentation - Google Cloud, accessed June 30, 2025, https://googleapis.dev/nodejs/scheduler/1.1.3/google.protobuf.html
13. IETF RFC 8446 TLS - ARC-IT, accessed June 30, 2025, https://www.arc-it.net/html/standards/standard1099.html

14. The Transport Layer Security (TLS) Protocol Version 1.3 - Tech-invite, accessed June 30, 2025, https://www.tech-invite.com/y80/tinv-ietf-rfc-8446.html
15. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3, accessed June 30, 2025, https://www.rfc-editor.org/rfc/rfc8446
16. Authorization Code Flow - Auth0, accessed June 30, 2025, https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow
17. Map of OAuth 2.0 Specs, accessed June 30, 2025, https://www.oauth.com/oauth2-servers/map-oauth-2-0-specs/
18. RFC 6749: The OAuth 2.0 Authorization Framework, accessed June 30, 2025, https://www.rfc-editor.org/rfc/rfc6749
19. The Open Group Base Specifications Issue 8, accessed June 30, 2025, https://pubs.opengroup.org/onlinepubs/9799919799/
20. accessed December 31, 1969, https://standards.ieee.org/standard/1003_1-2017.html
21. Open Container Initiative, accessed June 30, 2025, https://opencontainers.org/
22. OCI Introduction: The Full Journey from Code to Container in a Kubernetes Environment, accessed June 30, 2025, https://medium.com/@rifewang/oci-introduction-the-full-journey-from-code-to-container-in-a-kubernetes-environment-1e36d2890ca5
23. opencontainers/runtime-spec: OCI Runtime Specification - GitHub, accessed June 30, 2025, https://github.com/opencontainers/runtime-spec
24. Open Container Initiative - Wikipedia, accessed June 30, 2025, https://en.wikipedia.org/wiki/Open_Container_Initiative
25. OCI Certified - Open Container Initiative, accessed June 30, 2025, https://opencontainers.org/community/certified/
26. grpc package - google.golang.org/grpc - Go Packages, accessed June 30, 2025, https://pkg.go.dev/google.golang.org/grpc
27. gRPC, accessed June 30, 2025, https://grpc.io/
28. What is HTTP Long Polling ? - Educative.io, accessed June 30, 2025, https://www.educative.io/answers/what-is-http-long-polling
29. NATS.io – Cloud Native, Open Source, High-performance Messaging, accessed June 30, 2025, https://nats.io/
30. Download - NATS.io, accessed June 30, 2025, https://nats.io/download/
31. Redis Streams | Docs, accessed June 30, 2025, https://redis.io/docs/latest/develop/data-types/streams/
32. Open Source | Docs - Redis, accessed June 30, 2025, https://redis.io/docs/latest/get-started/
33. Apache Kafka, accessed June 30, 2025, https://kafka.apache.org/
34. The Java Virtual Machine Specification, Java SE 16 Edition : Oracle America, Inc. : Free Download, Borrow, and Streaming - Internet Archive, accessed June 30, 2025, https://archive.org/details/the-java-virtual-machine-specification-java-se-16-edition
35. .NET documentation | Microsoft Learn, accessed June 30, 2025, https://learn.microsoft.com/en-us/dotnet/

36. NET fundamentals documentation - Learn Microsoft, accessed June 30, 2025, https://learn.microsoft.com/en-us/dotnet/fundamentals/
37. Native Image - GraalVM, accessed June 30, 2025, https://www.graalvm.org/latest/reference-manual/native-image/
38. Oracle GraalVM - Get Started, accessed June 30, 2025, https://docs.oracle.com/en/graalvm/
39. docker-library/docs: Documentation for Docker Official Images in docker-library - GitHub, accessed June 30, 2025, https://github.com/docker-library/docs
40. Docker: Accelerated Container Application Development, accessed June 30, 2025, https://www.docker.com/
41. Docker Docs, accessed June 30, 2025, https://docs.docker.com/
42. Main — Emscripten 4.0.11-git (dev) documentation, accessed June 30, 2025, https://emscripten.org/
43. About this site — Emscripten 4.0.11-git (dev) documentation, accessed June 30, 2025, https://emscripten.org/docs/site/about.html
44. Web Workers basic example, accessed June 30, 2025, https://mdn.github.io/dom-examples/web-workers/simple-web-worker/
45. Using Web Workers - Web APIs | MDN, accessed June 30, 2025, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
46. SharedArrayBuffer - JavaScript - MDN Web Docs, accessed June 30, 2025, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
47. SharedArrayBuffer[Symbol.species] - JavaScript - MDN Web Docs, accessed June 30, 2025, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer/Symbol.species
48. Concurrent Programming With GCD in Swift 3 - WWDC16 - Videos - Apple Developer, accessed June 30, 2025, https://developer.apple.com/videos/play/wwdc2016/720/
49. Modernizing Grand Central Dispatch Usage - WWDC17 - Videos - Apple Developer, accessed June 30, 2025, https://developer.apple.com/videos/play/wwdc2017/706/
50. mrhappyasthma/PyCentralDispatch: A Grand Central Dispatch (GCD) inspired API for python. Not optimal, mostly just for convenience. - GitHub, accessed June 30, 2025, https://github.com/mrhappyasthma/PyCentralDispatch
51. (Swift) Grand Central Dispatch (GCD) | by Kenta Kodashima - Medium, accessed June 30, 2025, https://kentakodashima.medium.com/swift-grand-central-dispatch-gcd-80bcb16a147f
52. Dispatch | Apple Developer Documentation, accessed June 30, 2025, https://developer.apple.com/documentation/dispatch
53. NSOperationQueue | Apple Developer Documentation, accessed June 30, 2025, https://developer.apple.com/documentation/foundation/operationqueue?langu

age=objc
54. Operation | Apple Developer Documentation, accessed June 30, 2025, https://developer.apple.com/documentation/foundation/operation
55. OperationQueue | Apple Developer Documentation, accessed June 30, 2025, https://developer.apple.com/documentation/foundation/nsoperationqueue
56. Lesson: Concurrency - The Java Tutorials, accessed June 30, 2025, https://docs.oracle.com/javase/tutorial/essential/concurrency/
57. java.util.concurrent (Java Platform SE 8) - Oracle Help Center, accessed June 30, 2025, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
58. ExecutorService Java 8 API Documentation - shutdown() - Coderanch, accessed June 30, 2025, https://coderanch.com/t/661526/certification/ExecutorService-Java-API-Documentation-shutdown
59. Custom executor service is a cool feature - Java SDK - Couchbase Forums, accessed June 30, 2025, https://www.couchbase.com/forums/t/custom-executor-service-is-a-cool-feature/878
60. ExecutorService (Java Platform SE 8) - Oracle Help Center, accessed June 30, 2025, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html
61. kotlinx.coroutines/coroutines-guide.md at master · Kotlin/kotlinx.coroutines · GitHub, accessed June 30, 2025, https://github.com/kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md
62. Coroutines | Kotlin Documentation, accessed June 30, 2025, https://kotlinlang.org/docs/coroutines-overview.html
63. Wasmer: Universal applications using WebAssembly, accessed June 30, 2025, https://wasmer.io/
64. Wasmtime - Anaconda.org, accessed June 30, 2025, https://anaconda.org/anaconda/wasmtime
65. Wasmtime, accessed June 30, 2025, https://wasmtime.dev/
66. WasmEdge, accessed June 30, 2025, https://wasmedge.org/
67. accessed December 31, 1969, https://bytecodealliance.org/articles/wamr-a-lightweight-webassembly-runtime
68. wasm3/wasm3: A fast WebAssembly interpreter and the ... - GitHub, accessed June 30, 2025, https://github.com/wasm3/wasm3
69. accessed December 31, 1969, https://extism.org/
70. tetratelabs/wazero: wazero: the zero dependency ... - GitHub, accessed June 30, 2025, https://github.com/tetratelabs/wazero
71. accessed December 31, 1969, https://lunatic.io/
72. microsoft/mimalloc: mimalloc is a compact general purpose ... - GitHub, accessed June 30, 2025, https://github.com/microsoft/mimalloc
73. Kubernetes, accessed June 30, 2025, https://kubernetes.io/

74. Serverless Computing - AWS Lambda - Amazon Web Services, accessed June 30, 2025, https://aws.amazon.com/lambda/
75. Cloud Object Storage – Amazon S3 – Amazon Web Services - AWS, accessed June 30, 2025, https://aws.amazon.com/s3/
76. Low-Latency Content Delivery Network (CDN) - Amazon CloudFront ..., accessed June 30, 2025, https://aws.amazon.com/cloudfront/
77. Code Quality, Security & Static Analysis Tool with SonarQube | Sonar, accessed June 30, 2025, https://www.sonarsource.com/products/sonarqube/
78. Metal | Apple Developer Documentation, accessed June 30, 2025, https://developer.apple.com/documentation/metal?changes=_1_5
79. Metal | Apple Developer Documentation, accessed June 30, 2025, https://developer.apple.com/documentation/metal

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.