Metaverse
STANDARDS FORUM™

Open Metaverse Browser Architecture

A Standards Proposal for the Metaverse Browser Engine

This document was developed by RP1, the creators of the first working metaverse browser prototype and the primary architects and maintainers of the Open Metaverse Browser Initiative (OMBI). RP1's hands-on experience designing, building, and operating a global metaverse browser prototype provides the technical foundation for the concepts, architecture, and standards described herein.

The OMBI is an open project under the Metaverse Standards Forum. Its purpose is to inspire and coordinate the development of needed open standards, and to build an open-source implementation of a metaverse browser. The goal is to enable any organization to use or build compatible metaverse browsers and servers to deploy spatial services on infrastructure they own and control. The initiative is open to all contributors and is not owned by any single company.

2026.Q2.0

Executive Summary

A metaverse browser is a native, cross-platform application that brings the browsing paradigm to three-dimensional space. It preserves the four defining properties of a browser (trust without knowledge, no per-application installation, device independence, content / display separation) while maintaining simultaneous connections to dozens of independent spatial environments and services, blending their output into a single, unified 3D scene at interactive frame rates. This document serves as the architectural blueprint for Sneeze, the core metaverse browser engine (MBE) that performs this work, analogous to Blink, Chromium's 2D web browser's engine.

The problem. When people move through an increasingly augmented world — holding up a phone, wearing AR or AI/smart glasses, or immersed in a VR headset — they will encounter new spatial content every few steps. No one will want to download and install a new application every ten steps. The metaverse browser eliminates this barrier for spatial content the same way the web browser eliminated it for documents.

Device reach. The metaverse browser runs on every class of device: phones, tablets, desktops, AR glasses, and VR headsets. The first pervasive wave of consumers will experience the metaverse through the camera on the phones they already own. AR glasses and VR headsets will follow, most likely initially driven by enterprises who incorporate them into their operations. The metaverse browser architecture does not depend on specialized hardware; it renders flexibly to a 2D screen as naturally as to a XR headset.

The operating model. The metaverse browser architecture introduces four concepts that differentiate it from today's web browser: spatial fabrics (collections of mapped content and related services that define spatial environments), services (discrete units of functionality that provide behavior within those environments), proximity (the ambient discovery mechanism that replaces URL navigation), and presence (the spatial state that establishes the user's location and enables shared experience). Together, these define a continuous streaming model where the browser connects, composites, and disconnects as the user moves, while co-present users and AI entities share the same space in real time.

The architecture. The MBE must run on multiple classes of device, with diverse hardware configurations. It insulates itself from platform fragmentation through a set of industry adopted open standards that abstract the underlying hardware:

- ANARI abstracts rendering, providing a uniform scene-description API that delegates to interchangeable rendering engines.
- SPIR-V provides portable bytecode for both custom visual effects and GPU compute programs.
- OpenXR provides a unified API for XR devices.
- WebAssembly (WASM) binary instruction format enables sandboxed third-party service logic with per-service memory isolation.

In addition, a new proposed standard called Remote Model Access Protocol (RMAP) provides access to services through its published model definition without applications needing knowledge of its protocol or wire format.

The Scene Object Model (SOM) is the central interface where all of these converge, analogous to the DOM in a web browser. It is a multi-source, hierarchical, streaming scene graph with branch ownership, access-controlled read/write APIs, and a single-copy architecture.

Supporting components address physical-world anchoring (GPS/VPS with a proposed positioning abstraction layer), identity managed by dedicated identity service providers (eliminating per-service sign-on through cryptographic credentials and zero-knowledge proofs), avatars (parametric strategies targeting kilobyte-range data for scale), and spatial audio (server-side voice mixing for privacy and scalability, with a purpose-built codec optimized for compute efficiency over compression).

Enterprise focus. The first wave of adoption will center on enterprise and industrial operations: inventory management, equipment monitoring, safety alerts, training guidance, and remote collaboration. The architecture is designed to serve these use cases at production scale.

The goal of this effort is to develop an exemplar open-source Metaverse Browser Engine (MBE): one built entirely using open standards, implementable by anyone, owned by no one. The web became universal because no single company controlled the browser. The metaverse must follow the same path.

Scope and Purpose. This document, produced by the Open Metaverse Browser Initiative (OMBI), an open project under the Metaverse Standards Forum, builds on the foundation laid in *Open Standards for the Metaverse*, which establishes the case for an open, standards-based metaverse and the role a browser plays in it. This document addresses the architectural question: given that a metaverse browser is needed, how should one be built?

To answer that question, this document details the architecture of the MBE that sits at the core of a metaverse browser application; the component that maintains the scene graph, manages connections, executes service logic, and renders the scene. The MBE is deliberately kept distinct from the browser's application shell (navigation controls, settings, user interface), which is a separate design concern.

The architecture of the MBE is intended to be implemented by any organization, on any platform, without proprietary dependencies or single-vendor control. The OMBI at the Metaverse Standards Forum will provide the venue to build consensus on the MBE architecture, enabling and encouraging cooperation between multiple standards organizations creating the standards used in its design, as it advances through community review and implementation.

The concepts, architecture, and standards described herein are presented for rigorous debate. Additional viewpoints, domain expertise, and constructive challenges are invited from industry, standard development organizations, universities, and enterprise implementers.

Table of Contents

Part 1 — Foundational Concepts

Before any architecture can be proposed, the concepts it rests on must be established. This part defines what it means to be a browser along with the four foundational properties of the metaverse browser that differentiate it from a 2D web browser — spatial fabrics, services, proximity, and presence — and explains why a browser is the right paradigm for the metaverse. These are not implementation details. They are the premises from which every architectural decision in Parts 2 and 3 follows.

1. What Defines a Browser

Before introducing the metaverse browser, it is worth examining what makes a browser a browser. The dictionary defines a browser as a native client application that retrieves, renders, and enables interaction with content defined and transmitted using open standards. However, there are four special properties that distinguish the browsing paradigm. They are so familiar that people rarely articulate them, yet so essential that removing any one would destroy the model entirely. These four properties are not features. They are the defining characteristics of the browsing paradigm itself. Any system that claims to be a browser but violates one of them is not a browser — it is just an application with a browser-shaped interface.

1.1 Trust Without Knowledge

A user visits an unknown website without fear. They have no prior relationship with the site operator, no knowledge of the code that will execute, and no guarantee of the content's intentions. The browser protects them anyway. Through sandboxing, isolation, and strict capability control, the browser ensures that untrusted content cannot harm the user's device, access their files, or interfere with other sites. This property made the web possible. Without it, only technically sophisticated users would risk visiting unknown sites. There would be no mass participation, no network effects, and no adoption at scale.

1.2 No Application Installation

Content streams to the user and executes without downloading or installing software. There is no setup wizard, no permission dialog for disk access, no permanent footprint on the device unless the user explicitly allows it. The user arrives, the content runs, and when they leave, it is gone. This is the property that makes the web frictionless. Every installation step is a barrier to entry, and the browser removes all of them.

1.3 Device Independence

The same content works on any device: desktop, laptop, tablet, phone, AR and AI/smart glasses, and VR headsets. The content author has no knowledge of which hardware or operating system the user runs, and cannot design for a specific device. The browser abstracts the device away. Content written once reaches every platform. This property is what transformed the web from a convenience into a universal platform, and it is the architectural thread that runs through nearly every major decision in this document.

1.4 Content / Display Separation

Content is defined independently of how it is displayed. The creator describes what to present. The browser decides how to render it. This separation is what makes content creation accessible to anyone. Whereas an application developer must manage their own rendering pipeline, adapt to every screen size, handle every input device, and solve every platform-specific display problem themselves, a web page author writes markup and the browser handles the rest, regardless of

whether the result appears on a desktop monitor, a phone in portrait mode, or a screen reader. The browser absorbs this entire burden, and in doing so, transforms content creation from a software engineering challenge into a publishing task.

2. The Metaverse Browser

A metaverse browser is a native, cross-platform application that preserves all four browser properties in three-dimensional space, at interactive frame rates, while executing untrusted code from hundreds of simultaneous sources. Where a web browser connects to a single website and renders a two-dimensional document, the metaverse browser maintains simultaneous connections to dozens of independent spatial environments and services, blending their output into a single, unified three-dimensional scene called the Scene Object Model (SOM). The user sees one coherent world. The browser sees dozens of independent data streams, each sandboxed, each untrusted, each contributing its piece.

2.1 Device Reach

The metaverse browser is device-agnostic. It does not assume a headset, nor does it require a phone. It does not depend on any single class of hardware. The architecture renders to whatever display the user has, adapting its input model and visual output accordingly. The devices that encompass metaverse browsing today and in the near future include:

- **Phones.** The first pervasive wave of metaverse users will experience it through the camera on the phone they already own — holding the device up to see spatial content overlaid on the real world. Billions of these devices are already deployed.
- **Tablets.** The same camera-based AR experience as phones, with a larger viewport suited to shared viewing, design review, and education.
- **Laptops and desktops.** A windowed 3D view for development, content creation, remote collaboration, and any use case where a flat screen is practical.
- **AR glasses.** Full spatial augmented reality with six-degree-of-freedom tracking and digital content potentially anchored to and overlaid on the physical world. The device form factor that may eventually replace phones for daily use.
- **AI glasses.** Camera and AI assistant with limited or no visual overlay (e.g., Meta Ray-Ban). These devices capture context and deliver audio-first spatial interaction, bridging the gap before full AR display technology matures.
- **Smart glasses.** Lightweight heads-up display devices that present notifications, navigation, and simple spatial content without full scene-scale AR rendering.
- **VR headsets.** Fully immersive devices, including mixed-reality headsets with camera passthrough that blend virtual content with a view of the physical world.

For the remainder of this document, the term "AR glasses" refers collectively to AR, AI, and smart glasses unless a distinction is specifically noted.

2.2 The 10-Step Rule

When people move through the world — holding up a phone, wearing AR glasses, or immersed in a VR headset — they will passively encounter new spatial content every few steps: a building's information overlay, a transit schedule, a safety alert, a retail display. No one will want to download a new application, create a new account, or accept a new terms-of-service agreement every ten steps.

This principle — that no per-service friction is acceptable at the pace of ambient discovery — is the fundamental justification for the metaverse browser. Every friction that the web browser can eliminate for documents (installation, registration, configuration) must be eliminated for spatial content at the same cadence. The metaverse browser eliminates these barriers the same way the web browser eliminates them for documents.

2.3 The Web Browser Parallel

The architectural analogy between the web browser and the metaverse browser is direct:

Web Browser	Metaverse Browser
HTML/CSS/JS content	3D spatial content + WASM service logic
DOM (Document Object Model)	SOM (Scene Object Model)
One origin at a time	Dozens of spatial fabrics simultaneously
URL-based navigation	Proximity-based discovery
JavaScript engine	WASM runtime
Web browser engine (Blink, Gecko)	Metaverse browser engine (Sneeze)

The mapping is not superficial. The architectural decisions that made the web browser successful (sandboxing, device abstraction, a standard object model) must also apply to the metaverse browser. They are simply harder to achieve when the content is three-dimensional, real-time, multi-source, and immersive.

2.4 Four Fundamental Differences

While the metaverse browser preserves the web browser's defining properties, here are four ways it differs from the web browser that will require a different architecture, each discussed in more detail in the sections (3-6) that follow:

1. **The SOM composites many spatial fabrics simultaneously.** A web browser renders one website at a time. The metaverse browser maintains simultaneous connections to dozens or even hundreds of spatial fabrics, blending their content into a single scene.
2. **The experience is dominated by services.** A web page is a self-contained document. A spatial fabric is a living environment populated by dozens of independent services, each running its own logic, each sandboxed.
3. **Navigation is primarily proximity-based, not URL-based.** On the web, you navigate by typing addresses or clicking links. In the metaverse, you navigate by moving through the world. Objects and services are presented to you based on their proximity to where you are located while retaining the ability to access traditional web resources local to your avatar.
4. **The user is present in the space.** On the web, you view a page from the outside — you have no location within the content. In the metaverse, you are inside the coordinate space: you have a position, an orientation, and a vantage point. Everything the browser does — what it renders, what services it activates, who it makes you visible to — follows from where you are.

2.5 Why Not Incrementally Extend the Web Browser?

The web platform has a 30-year history of absorbing new capabilities (CSS layout, XMLHttpRequest, WebSockets, WebGL), each adding a new API to the existing browser architecture. The four fundamental differences above might appear to be the next set of APIs waiting to be added. They are not. Previous extensions added capabilities within the web's existing security model (per-origin isolation). The SOM requires a fundamentally different security model: per-branch ownership within a shared 3D scene graph, where dozens of untrusted sources contribute objects to the same spatial scene simultaneously. This is not a new API — it requires a different architecture altogether.

The MBE, however, should not be viewed as a replacement for the web browser. Sneeze is a compact, self-contained engine that a web browser can embed alongside its existing rendering engine, like Blink in Chromium. The same way a web browser switches from its HTML renderer to a PDF viewer when it encounters a PDF, it would switch to the MBE when it encounters a spatial fabric. The MBE handles the SOM, ANARI, WASM services, and RMAP connections. The web browser handles everything else: the application shell, the install base, the update infrastructure, and the user's trust.

2.6 WebXR and the Web Browser

The MBE does not diminish the web browser or the standards built for it. Specifically, WebXR (the W3C standard that brings XR capabilities to web content) remains vital, and its relevance grows in direct proportion to the volume of spatial content available on the open internet.

Web browsers are the natural fit for two-dimensional devices (phones, tablets, laptops) that billions of people already carry today. Metaverse browsers are the natural fit for

three-dimensional devices (AR glasses, VR headsets) that are steadily being adopted by enterprises and consumers. The same spatial content, published to the open internet through open standards, should be accessible from both browsers, which will coexist for a very long time. WebXR will continue to play a vital role allowing web browsers to reach that content. The more spatial content the metaverse ecosystem produces, the more essential WebXR becomes.

The OMBI views the WebXR community as a critical partner. We share the same foundational commitment: spatial experiences delivered through open standards on the open internet, accessible to anyone with a conformant browser. Collaboration between the two communities strengthens both — ensuring that metaverse content is accessible to WebXR users on every device, and that the principles WebXR has established are faithfully upheld in the metaverse browser architecture.

3. Spatial Fabrics

A spatial fabric is to the metaverse browser what a website is to the web browser: the thing you browse. It is the most fundamental concept in the metaverse architecture, and the reader must understand it thoroughly before the technical architecture can make sense.

3.1 What a Spatial Fabric Is

A spatial fabric is a collection of mapped content and related services that together define a spatial environment. A city's street grid is a spatial fabric. A university campus is a spatial fabric. A retail store interior, a factory floor, an airport terminal, a game arena: each is a spatial fabric.

Every spatial fabric has one required component: the map service. The map service provides the spatial foundation (geometry, terrain, objects, coordinate system) and streams updates to connected browsers. It is the scaffold on which everything else in the fabric is built.

Beyond the map service, a fabric may include any number of additional services. Some are fabric-level services operated by the fabric owner: user presence, social connections, commerce, communication. Others are map-anchored services attached to specific locations in the map: an equipment status dashboard at a factory workstation, a safety zone alert near heavy machinery, a product display at a retail shelf. Map-anchored services may be operated by the fabric owner or by independent third parties.

If a website is a collection of resources served from a domain, a spatial fabric is a collection of services organized around a shared spatial context.

3.2 Simultaneous Multi-Fabric Sessions

This is the defining structural difference between the metaverse browser and the web browser.

A web browser connects to one origin at a time. The metaverse browser maintains simultaneous connections to dozens of spatial fabrics. A user walking down a city street might be connected to the street-level city fabric, the retail store fabric they just entered, a corporate campus fabric visible across the road, and a restaurant review overlay they activated ten minutes ago. All of

these fabrics contribute content to the same scene, at the same time, in the same coordinate space.

Multiple fabrics are not tabs. The fabrics share the same visual space, the same coordinate system, and the same moment in time. Each fabric contributes its own branch of the SOM, and the browser composites them into a single, unified three-dimensional scene.

3.3 Fabric Composition

A fabric's map service provides the spatial data: geometry, terrain, coordinate system, and streaming updates. Beyond the map, the fabric may include:

- Fabric-level services such as persona management, social networking, or commerce, which are part of the fabric's overall offering and not tied to specific map locations.
- Map-anchored services such as a maintenance kiosk, a transit arrival board, or an inventory tracker, which attach to specific locations in the map and generate dynamic SOM content through WASM programs communicating with their backend servers.

A Spatial Fabric can consist of → Map Service (required) + Fabric-Level Services + Map-Anchored Services.

3.4 Fabric Independence and Hierarchy

Most fabrics are independent. They are operated by different organizations, hosted on different infrastructure, and have no knowledge of each other. The city street fabric and the retail store fabric were built by different teams with different goals. The metaverse browser is the only place where they meet. This independence is what makes the SOM's multi-source composition and branch ownership model (Section 9) architecturally necessary.

But fabrics can also be hierarchical. A university campus fabric may contain sub-fabrics for each building, which in turn contain sub-fabrics for each floor or department. These are built with full knowledge of each other and typically share a common operator.

3.5 Overlays

Overlays are a distinct application of spatial fabric. Whereas a primary fabric plus its secondary fabrics attach to specific locations within the map, overlay fabrics superimpose their own mapped fabric layer on top of the main fabric layer.

A user activates an overlay, and its content appears on top of the existing world. Deactivate it, and it vanishes. Consider Yelp: a user activates the Yelp overlay and every restaurant in view displays a floating rating badge, review count, and price range above its entrance. Tap a badge and a summary of recent reviews appears. The user turns the overlay off and the information disappears. Yelp has no relationship with the restaurants' own spatial fabrics — it operates entirely in its own layer, drawing on its own data. This is the overlay model: independent content superimposed on the scene by the browser.

One hard rule governs overlays: overlays cannot embed content in other fabrics. Yelp's overlay cannot insert itself into a restaurant's own fabric uninvited. The overlay operates in its own layer, composited by the browser.

The mechanics of how overlay composition is coordinated across the SOM remain an open area for further design.

3.6 Fabric Lifecycle

Fabrics are connected, streamed, and disconnected as the user moves through the world. The browser does not "load" a fabric the way a web browser loads a page. Instead, it opens a streaming connection, receives continuous updates for as long as the fabric is relevant, and disconnects when the user has moved on. There is no "page load complete" moment. The scene is always in motion, always being composed.

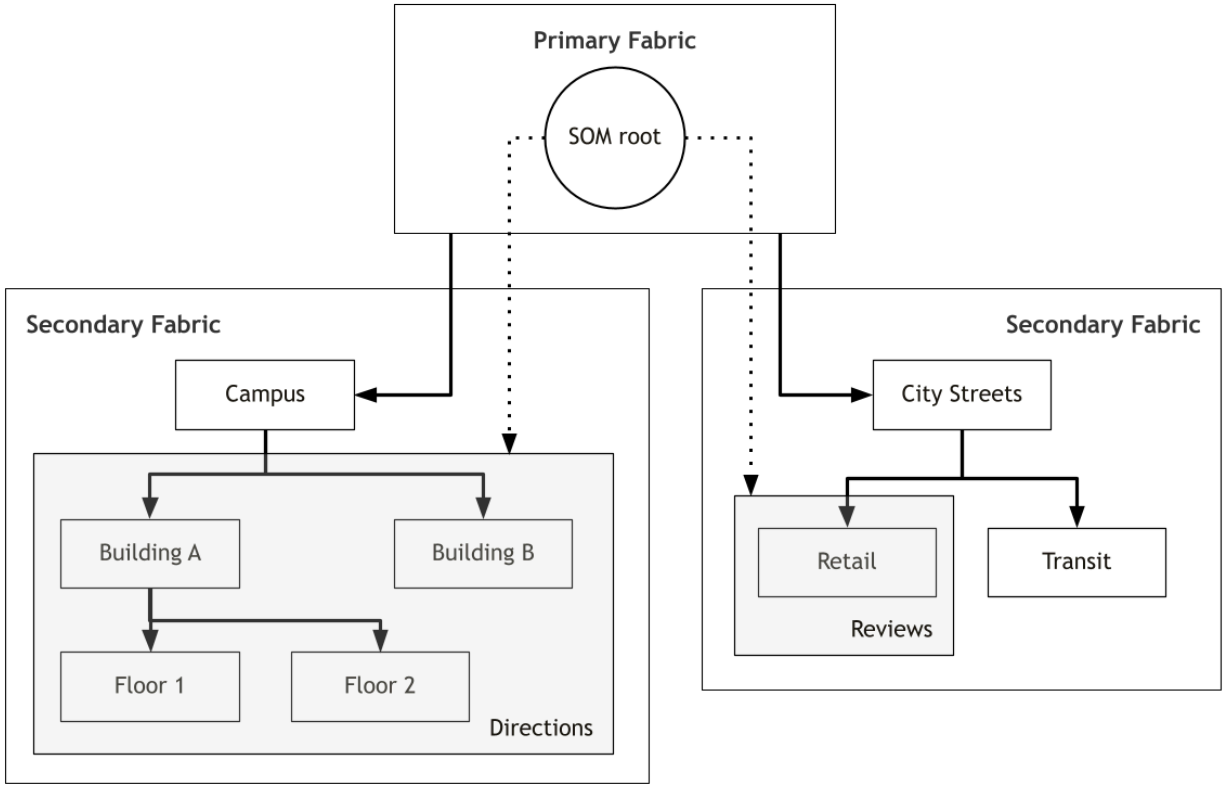


Figure 3.6: *Spatial fabric composition. Hierarchical fabrics and overlays all contribute to the user's unified scene. Solid lines represent direct fabric connections. Dashed lines represent overlay superimposition.*

4. Services

The metaverse browser is service-dominated. Services are the applications of the metaverse. They provide behavior, interactivity, and purpose to a spatial environment. Without services, a

spatial fabric is just its map — geometry with no behavior. With services, it becomes a living environment that reacts, informs, guides, and protects.

4.1 What a Service Is

A service is a discrete unit of functionality that operates within a spatial fabric. Each service has its own backend server, its own logic, and its own data. A safety zone alert on a factory floor is a service. An equipment status dashboard at a workstation is a service. A training walkthrough that guides a new technician through a maintenance procedure is a service. An AI quality inspector that analyzes camera feeds and highlights defective parts on the assembly line is a service. An IoT aggregation layer that collects temperature, vibration, and pressure readings from hundreds of sensors and visualizes them as color-coded halos around each machine is a service. A retail product display, an inventory tracker, a navigation guide: each is a service.

Services come in two flavors:

- Fabric-level services are part of the fabric's overall offering, not tied to specific map locations. User presence, social connections, and commerce are fabric-level services, operated by the fabric owner.
- Map-anchored services attach to specific locations in the map and generate dynamic content. A maintenance kiosk, a transit arrival board, and a safety zone alert are map-anchored services. These may be operated by the fabric owner or by independent third parties.

Both flavors run in sandboxed WASM instances and communicate with their backends through RMAP (Section 13).

4.2 Services vs. Web Pages

On the web, a user visits one page at a time. They choose to go there, they interact with it, and when they are done, they leave. In the metaverse, dozens of services are active simultaneously in the user's environment. The user does not "open" a service. Services are present in the space, discovered as the user moves, and remain active for as long as they are relevant. A user walking through a warehouse might have an inventory management service, a pick-and-pack routing service, an AI demand-forecasting service that flags overstock and understock in real time, an IoT environmental monitoring service tracking cold-chain temperatures, and a safety alert service all running at once, all rendering in the same scene, all independently sandboxed.

4.3 Why Services Matter: Syndication Over Destination

The current model for distributing digital functionality forces every business to build a destination (a website or an application) and drive traffic to it. The service model inverts this entirely where services are distributed to the places where your users already are. Add a service to a factory floor, a hospital corridor, a construction site, a warehouse.

The analogy is syndication. A cartoonist who publishes their own newspaper in order to reach readers has the wrong model. A cartoonist who syndicates their cartoon into every newspaper

that already has readers has the right one. Services bring functionality to where people already are, rather than asking people to come to where the functionality lives. This is a fundamentally different distribution model for information, operations, and commerce.

4.4 Common to All Devices

Services are identical across all device types. A navigation service, a safety zone alert, an equipment status dashboard: the service does not know or care whether the user is using a mobile phone on a real factory floor, wearing AR glasses, viewing a 3D model of that floor from a desktop, or immersed in a virtual training simulation. The service's WASM code runs the same way, reads the same SOM, writes to the same branch, and communicates with the same backend. This universality is a core architectural property, not an afterthought.

4.5 Service Isolation

Services come from different providers, and the browser has no prior knowledge of them. A factory fabric might include services from the building owner, the equipment manufacturer, a third-party safety monitoring company, a corporate training provider, an AI predictive-maintenance vendor, and an IoT platform aggregating sensor telemetry from the shop floor. These providers have no relationship with each other. The browser must protect each service from every other:

- One service cannot interfere with another's execution.
- One service cannot read another's data.
- One service cannot crash the browser.

This is the architectural requirement that drives the selection of WASM as the content execution runtime (Section 12). Sandboxing is not a feature of the metaverse browser; it is a defining property, inherited from the first principle established in Section 1.

4.6 Service-to-Browser Interface

Each service interacts with the browser through a controlled set of host functions:

- It reads portions of the SOM for spatial awareness (where nearby objects are, what the environment looks like).
- It writes to its own branch of the SOM to contribute visible content to the scene. The SOM is the interface between service code and rendering — services describe what should exist, and the rendering engine (through ANARI) decides how to draw it. No service has any contact with ANARI or the rendering engine, direct or indirect.
- It communicates with its backend server through RMAP, working with model objects that represent server-maintained state — properties, events, and actions defined by the service provider's class definitions.

- It queries input state — button presses, hand gestures, gaze direction — from data the browser has read from OpenXR and made available through host functions.
- It submits shaders (as SPIR-V bytecode) for dynamic content that requires runtime-generated visual effects.

The browser mediates all of these interactions. No service has direct, unmediated access to any underlying system — not the GPU, not the XR runtime, not the network, not any other service.

4.7 WASM Instances Are Per-Service

WASM instances are grouped by service, not by object. Two equipment dashboards at different locations on the factory floor that connect to the same equipment monitoring service share a single WASM instance. The service manages both dashboards from one sandbox with one backend connection. Multiple distinct services can also be configured to share a single WASM instance when appropriate.

4.8 Services in Practice

Consider a factory floor where the building fabric provides the three-dimensional geometry of the space. Layered on top of that static environment:

- A safety service marks hazard zones around heavy equipment. When a forklift begins operating, the zone turns red and expands. When it stops, the zone contracts.
- An equipment service shows real-time status of each machine: operating temperature, hours since last maintenance, fault indicators.
- A training service guides a new technician through a pump replacement procedure, highlighting each bolt in sequence and overlaying step-by-step instructions.
- An inventory service flags items due for restocking, displaying counts and reorder urgency at each storage location.
- An AI predictive-maintenance service analyzes vibration and thermal patterns from IoT sensors across the floor, placing early-warning indicators on machines likely to fault within 48 hours.
- An IoT environmental service aggregates air-quality, humidity, and temperature readings from ceiling-mounted sensors and renders them as a real-time heat map hovering above the floor.

All of these services are running simultaneously. All are sandboxed. All are writing to the same SOM. The user sees a single, coherent scene. The browser sees six independent WASM instances, each in its own memory space, each communicating with its own backend. Inter-service communication, when needed, is mediated through a controlled channel; services never share memory or bypass the browser's isolation boundaries.

5. Proximity

On the web, you type a URL or click a link. In the metaverse, you move. As a user walks through a warehouse, drives past a construction site, or steps into a hospital wing, the browser discovers spatial fabrics and services that are relevant to the user's current location. The browser connects automatically as things come into range and disconnects as they fall behind.

This is a fundamental shift in the relationship between user and content. On the web, discovery is intentional. In the metaverse, discovery is ambient. The content comes to you because you are near it. But ambient discovery does not mean forced acceptance: the user must retain control over what activates. The browser connects automatically, but a filtering layer — governed by user preferences, provider reputation, and in enterprise settings, organizational policy — determines what actually reaches the scene. The mechanisms for this filtering are addressed in Section 22.

5.1 Positioning Varies by Environment

The mechanism that determines "where the user is" varies by setting:

- Outdoors, GPS (global positioning system) provides coarse positioning to determine which fabrics are nearby. VPS (visual positioning system) provides fine-grained anchoring to the physical world at centimeter-level accuracy.
- Indoors (factories, warehouses, hospitals), indoor positioning systems serve the same role, using technologies appropriate to the environment.

The mechanism varies. The principle is the same: the user's physical location determines what the browser connects to. The technical details of positioning systems are covered in Section 17; this section addresses only the concept.

5.2 The Streaming Paradigm

Proximity drives a continuous streaming model. Objects and services appear and disappear as the user moves. The browser is always loading and unloading, always connecting and disconnecting. There is no "page load complete" moment. The SOM is always in constant flux.

This is comparable to a radio receiver scanning frequencies as it moves: the stations do not change, but which ones are in range does. The browser's streaming paradigm ensures that the user always sees the content that is relevant to where they are, without ever waiting for a "page" to finish loading.

6. Presence

On the web, "visiting" a website is a metaphor. The user is not anywhere. Two people reading the same article have no spatial relationship to each other and no awareness of each other's existence. In the metaverse, presence is literal. The user has a position, an orientation, and a field of view. They are there — and everything that follows (what they see, what they hear, who they

encounter, which services activate) flows from where they are. Without presence, there is nothing to be proximal to, no services to discover, and no experience to have.

6.1 What Presence Means

Presence in the metaverse means occupying a specific location within one or more spatial fabrics, with a specific vantage point, in real time. The browser knows where the user is (via GPS/VPS positioning and device tracking), what direction they are facing, and what falls within their field of view. This spatial state is the input that drives every other system in the architecture:

- Proximity uses the user's position to determine which fabrics and services to connect to and disconnect from.
- Rendering uses the user's vantage point to determine what to draw and from what perspective.
- Spatial audio uses the user's head position to spatialize sound sources relative to the listener.
- Services use the user's location to deliver context-appropriate content — a safety alert near machinery, a product display near a shelf, a transit schedule at a bus stop.

Presence is not a feature layered on top of the architecture. It is the spatial state from which all other runtime behavior derives.

6.2 Presence and Proximity

Presence is the prerequisite for proximity. In order to be proximal to something, you must first be present in a space. Your presence — your location, orientation, and movement — determines what you are near, what comes into range, and what falls behind. Proximity (Section 5) describes the discovery mechanism; presence is the spatial state that proximity operates on.

The distinction matters architecturally. Proximity is a process: the browser continuously evaluates what to connect to and disconnect from. Presence is a state: the user's current position and vantage point at any given moment. The process depends on the state.

6.3 Co-Presence

When two or more entities are present in the same spatial fabric at the same time, they are co-present. Co-presence is not a separate system — it is the natural consequence of multiple users being present in the same space. But it introduces architectural requirements that single-user presence does not:

- **Visibility.** Each user must see the others. Avatar data must be retrieved, rendered, and animated in real time for every co-present entity.
- **Voice.** Co-present users need to hear each other. Spatial audio must spatially mix each user's voice relative to every other user's position.

- **Position broadcasting.** Each user's position must be shared with the primary fabric so that other connected browsers can render their avatar at the correct location. The primary fabric manages this: it tracks who is present and broadcasts position updates to all connected clients.
- **Interactivity.** Co-present users must be able to act on the same objects and services together — picking up a shared model, manipulating a control panel, or collaborating on a spatial model in real time.

Co-presence is what transforms the metaverse from a solitary content viewer into a shared environment. Enterprise use cases depend on it directly: remote collaboration, field assistance (an expert guides a technician through a repair while seeing what they see), training (an instructor works alongside trainees in a shared spatial environment), and conferencing (participants occupy a shared space with spatial voice and visible body language). Consumer use cases are equally dependent: social gatherings, family time across distances, shared entertainment, and community events.

6.4 Non-Human Presence

Humans are not the only entities present in the metaverse. The architecture must support multiple types of present entities:

- AI agents operate autonomously within spatial fabrics. A factory floor might have an AI safety monitor that moves through the space, an AI maintenance advisor that appears at equipment stations, or an AI guide that walks new employees through orientation.
- Robots with AR/MR connectivity bridge the physical and virtual. A warehouse robot's physical position is reflected in the metaverse; a remote operator sees the robot's perspective and controls it through the spatial interface.
- NPCs (non-player characters) are fabric-controlled entities that populate environments: a virtual receptionist, a training scenario character, a crowd in a stadium simulation.
- Virtual companions are user-owned entities that persist across fabrics, with autonomous behavior patterns.

Every present entity — human or otherwise — has a position, an avatar, and the ability to interact. The identity architecture addresses this through entity types: every avatar carries a required entity type (human, AI agent, or NPC) that the protocol enforces. A user always knows whether they are interacting with a real person, an AI, or a scripted character. Avatars are addressed in more detail in Section 19.

6.5 Presence at Scale

The architectural challenge of presence is scale. A quiet office might have ten co-present users. A factory floor might have hundreds. A concert or sporting event might have tens of thousands. The architecture must handle the full spectrum without fundamental redesign.

At small scale, every co-present entity is individually visible: full avatar rendering, individual voice streams, precise position tracking. At large scale, the architecture must degrade gracefully: distant users become simplified avatars (level-of-detail reduction), voice mixing shifts to crowd ambience for distant groups, and position updates are batched by spatial region. The avatar LOD system and server-side spatial audio mixing are designed with this scaling model in mind.

Part 2 — Core Architecture

Part 1 asked what the metaverse browser must do. Part 2 answers how. The sections that follow present the architectural decisions with the strongest consensus: the principles that govern every trade-off, the scene data structure at the center of the system, the hardware abstractions that ensure device independence, the content execution sandbox, and the service connectivity layer. These decisions are internally consistent and interdependent — they reinforce each other by design.

7. Architectural Principles

Part 1 established the four differentiating properties of the metaverse browser: spatial fabrics, services, proximity, and presence. The architectural decisions that follow are shaped by these properties and governed by the principles below. They explain why the architecture takes the shape it does, and in a transparent manner.

7.1 Abstraction Over Direct Access

The browser never couples to a specific vendor's API, protocol, or hardware. Every layer interfaces through an abstraction. The rendering layer does not call a graphics API such as Vulkan directly; it calls ANARI, which delegates to whatever rendering engine is available. The networking layer does not implement a specific wire protocol; it calls RMAP, which delegates to whatever transport the service requires. This principle is the architectural expression of the browser's defining property of device independence (Section 1).

7.2 Assume All Content is Hostile

Security is enforced by architecture, not by policy. Sandboxing, memory isolation, and capability control are structural guarantees, not guidelines that depend on good behavior. App store review, terms of service, and developer agreements are not security mechanisms. The browser runs untrusted code from unknown providers discovered passively through proximity. The architecture must make it structurally impossible for that code to harm the user, the device, or other services. There is no other option.

7.3 Nostalgia is Not a Viable Reason

Technologies are evaluated on present merit and future trajectory, not on past popularity or developer familiarity. A technology that was transformative ten years ago may be architecturally incompatible today. Evaluations in this document consider what a standard can do now, how it is evolving, and whether it can serve the requirements of a system that must operate for decades. Installed base and developer sentiment are acknowledged but are not architectural arguments.

7.4 The Architecture Must Accommodate Future Expansion

Standards that are frozen, deprecated, or unable to evolve are rejected regardless of current adoption. Every interface point in the architecture is designed to accept new backends, new protocols, and new capabilities without structural change. When a new graphics API emerges, the rendering engine adapts or ANARI gains a new backend; the browser does not change. Extensibility is not a feature of the architecture; it is a structural requirement.

7.5 Performance at Scale

Components that are too slow, too heavyweight, or too disruptive to operate at the required scale are not viable, regardless of other merits. A metaverse browser rendering at 90 frames per

second has a frame budget of 11.1 milliseconds. It executes code from a hundred simultaneous services. It composites content from dozens of spatial fabrics in real time. Every component in the stack must be able to operate within these constraints. A component that causes unpredictable pauses, consumes excessive memory, or cannot scale to the required number of concurrent instances is architecturally disqualifying.

7.6 One Copy of the Truth

Data exists in exactly one place. Maintaining multiple copies of the same data creates synchronization overhead, memory waste, and consistency bugs. At 90 frames per second with dozens of concurrent writers, duplicated data structures become intractable. The scene exists once. Everything that needs the scene reads from or writes to that one copy through controlled interfaces. Anything that requires duplication of the central data structure is architecturally disqualifying.

7.7 Notwithstanding 7.1–7.6

These principles and the architectural decisions that follow from them are presented for rigorous debate. Additional viewpoints, domain expertise, and constructive challenges are welcomed. The architecture improves through scrutiny.

8. System Overview

The architecture described in this document is being built as *Sneeze*, the OMBI's reference implementation of the MBE. Sneeze is to the metaverse browser what Blink is to the web browser — the core engine that maintains the scene graph, manages connections, executes service logic, and renders the scene.

The MBE's architecture is organized into five layers, from hardware at the bottom to content creation at the top. The boundaries between layers are the architecture. Each boundary exists for a specific reason, and the section below explains each layer's role and why the boundary between it and its neighbors is drawn where it is.

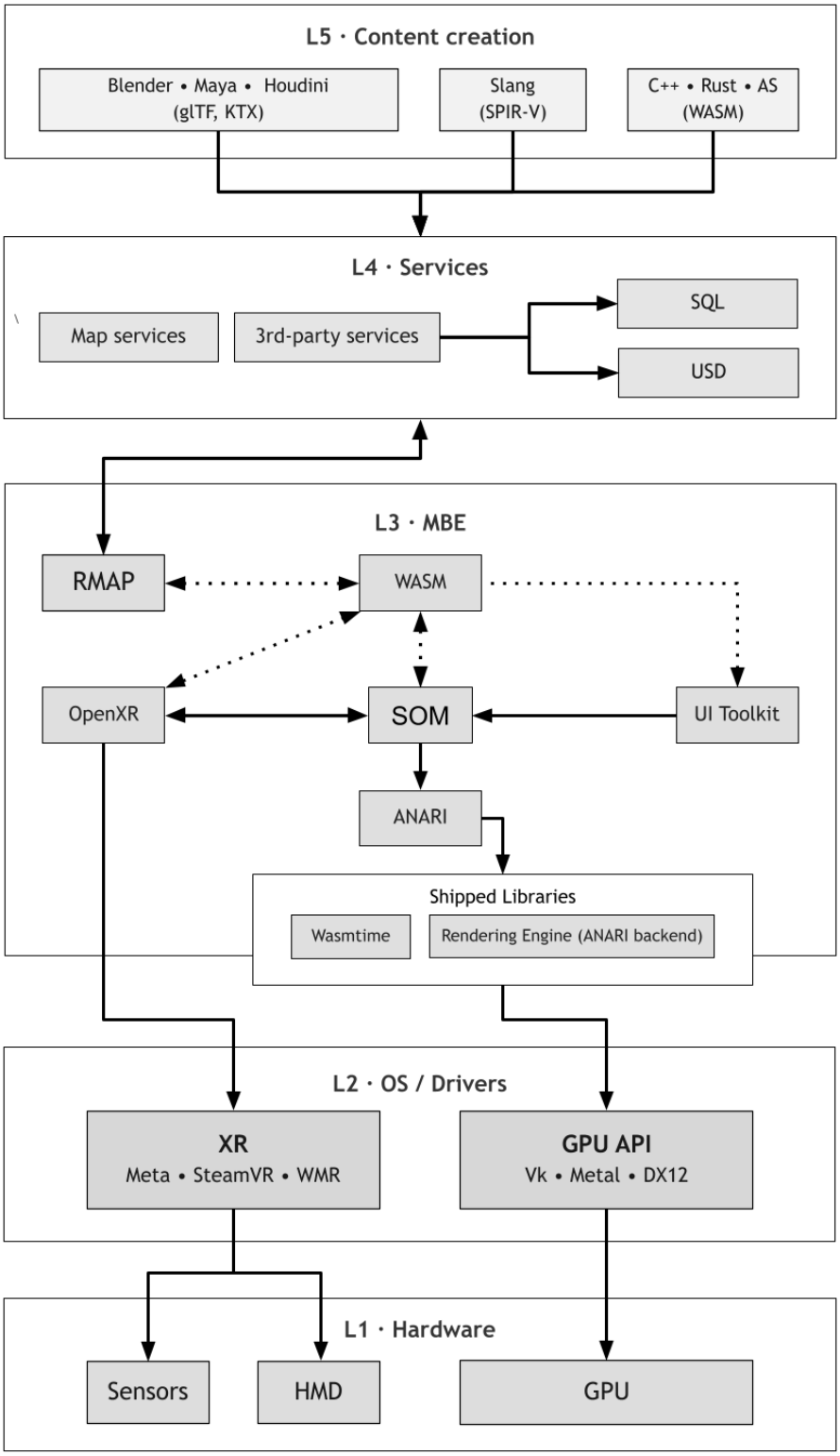


Figure 8: *Five-layer system architecture. The SOM is the central API through which all other subsystems interact. WASM's dotted lines represent indirect access through browser-mediated host functions. Wasmtime and a rendering engine ships with each vendor's browser in Layer 3.*

8.1 Layer 1: Hardware

The MBE has no direct relationship with Layer 1, consisting of GPUs, CPUs, headsets, cameras, and other sensors. Hardware diversity is the entire reason the abstraction layers in Layer 3 exist. If every device ran the same hardware, the abstractions would be unnecessary. They are necessary precisely because the hardware landscape is fragmented and will remain so.

The browser ships with at least one rendering engine (ANARI backend) that targets the platform's GPU API. The browser's application code is identical regardless of which rendering engine or GPU API is running underneath. What changes per platform is the rendering engine the browser ships with and the installed drivers.

8.2 Layer 2: OS Drivers and Runtimes

The boundary between Layer 1 and Layer 2 is the driver-to-hardware interface. The architecture is deliberately ignorant of how drivers communicate with hardware. Layer 2 contains the platform-provided APIs and vendor-supplied drivers that the browser's linked libraries call down to:

- **GPU APIs:** Vulkan, Metal, DirectX (DX12). These are the low-level interfaces that rendering engines use to communicate with GPU hardware. Each provides both graphics and compute capabilities. The GPU vendor's driver implements these APIs for their specific hardware.
- **XR runtimes:** Meta's OpenXR runtime, SteamVR. These are OpenXR backends that interface with headset hardware.

8.3 Layer 3: The Browser

The boundary between Layer 2 and Layer 3 is the abstraction boundary. The browser's application code never calls a platform-specific API directly. It calls ANARI (not a GPU API) and OpenXR (not a vendor's proprietary SDK). This boundary is the expression of the first architectural principle in Section 7.

Layer 3 is the MBE itself. Its components, from top to bottom within the browser:

- RMAP sits at the top of the browser, managing all external communication. Every connection to every spatial fabric and service flows through RMAP.
- The SOM sits below RMAP. It is the browser's central interface — the API through which every other subsystem accesses the scene. It holds the unified scene state composed from all connected fabrics and services, and enforces ownership, access control, and structural integrity at the API layer.
- WASM, ANARI, OpenXR, and SPIR-V sit below the SOM. Each abstracts a specific capability: content execution, rendering, XR device interaction, and portable GPU programs. WASM has a direct connection back up to RMAP because each service's WASM module communicates with its own backend.

- Browser-shipped libraries are dependencies that ship with the browser rather than being provided by the operating system. They include the WASM runtime (Wasmtime) and the rendering engine (ANARI backend). The decision to link libraries statically or dynamically is left to each browser vendor — what matters is that these are browser-owned dependencies, not platform-provided ones.

8.4 Layer 4: Services

The boundary between Layer 3 and Layer 4 is the network. Content above this line lives on servers operated by fabric and service providers. Content below this line lives on the user's device. RMAP is the interface that bridges them.

Services are remote. Map services store and stream spatial data (using SQL, USD, or other storage mediums). Third-party services host application logic and real-time data. Everything in Layer 4 is accessed over the network through RMAP.

8.5 Layer 5: Content Creation

The boundary between Layer 4 and Layer 5 is the compilation step. Content providers author in whatever tools and languages they prefer. What crosses the boundary is compiled output: glTF assets, KTX textures, SPIR-V shaders, and .wasm service modules.

Content creation is remote. Authoring tools (Blender, Maya, Houdini, Revit, and others) produce three-dimensional glTF assets and scenes. Slang compiles custom shaders to SPIR-V portable bytecode. C++, Rust, and AssemblyScript compile service logic to WASM modules. These tools produce the artifacts that services host and distribute. The browser never sees these tools; it sees only their output.

9. The Scene Object Model (SOM)

The SOM is to the metaverse browser what the DOM is to the web browser: the central interface that every other subsystem talks to. It is presented first among the core subsystems for the same reason a paper on web browser architecture would lead with the DOM: every other component reads from it, writes to it, or both.

The SOM is a robust API that enforces ownership, mediates access, and maintains structural integrity under concurrent use. It does not fetch data, manage connections, or initiate logic on its own. But it is far more than passive storage. Every interaction with scene state flows through the SOM's controlled interface: WASM services read and write through host functions, ANARI reads scene data for rendering, the browser's internal systems manage object lifecycle and computational updates. The SOM enforces branch ownership at the API layer, restricts read access per branch, and guarantees node-level atomicity for concurrent writers. It is the single point of contact between every subsystem in the browser and the scene it renders.

9.1 Why "Scene Object Model" and Not "Scene Graph"

The underlying data structure is a scene graph: a tree of nodes with transforms, where parent transforms cascade to children. But calling it a "scene graph" undersells the model by an order of magnitude. The SOM is an object model layered on top of a scene graph, and the distinction is the same one that separates the DOM from "a tree of HTML elements."

Several properties distinguish the SOM from a conventional scene graph:

- **Multi-source composition.** Dozens of independent sources (map services, fabric-level services) write to the same tree simultaneously. No single author controls the scene.
- **Ownership and access control.** Every branch of the tree has a designated owner that can write only to its own subtree. Read access spans the full tree but can be restricted per branch.
- **Live, streaming, and mutable.** The SOM is never "loaded." It is continuously composed, updated, and pruned as fabrics connect, stream, and disconnect.
- **API contract.** The SOM exposes a controlled interface to WASM service modules, not raw memory access. The browser mediates all interaction.

These properties are not features bolted onto a scene graph. They are the defining characteristics of the SOM, and the reason it is an object model rather than merely a data structure.

9.2 Multi-Source Composition

A web browser's DOM has a single author: the website's JavaScript. The SOM has dozens of simultaneous authors: every connected spatial fabric's map service streams map objects into its own branch, and every active service may write additional content into the tree.

Map services stream typed map nodes into their SOM branches, organized in a three-tier spatial hierarchy. Each tier groups objects by their functional role: celestial nodes (galaxies, stars, planets, moons), terrestrial nodes (continents, countries, cities, parcels), and physical nodes (buildings, trees, furniture, vehicles, equipment).

The hierarchy nests naturally — physical objects sit within terrestrial regions, which sit on celestial bodies. Each node carries its own transform and bounding volume. Some nodes reference glTF assets or other properties that give them a visual appearance. Other nodes may have no visual representation at all, carrying properties for animations, 3D transforms, or attachment points for services or other spatial fabrics.

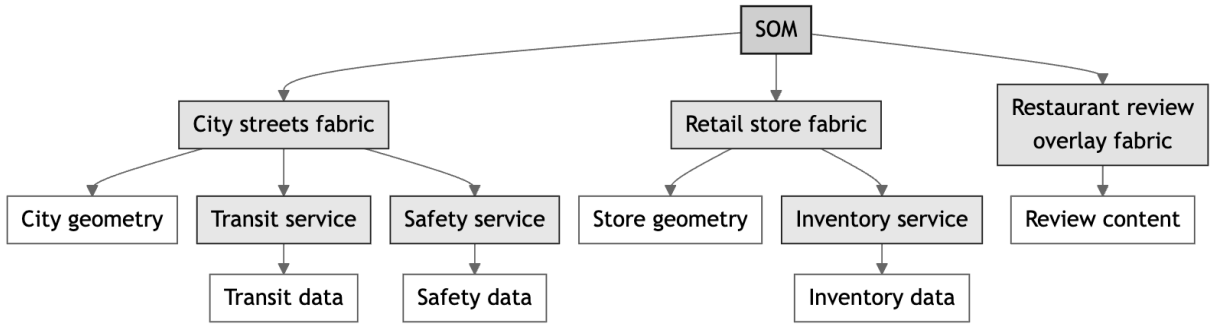
These nodes are never loaded all at once. There is no "page load complete" equivalent. The SOM is always in flux — map nodes appear as fabrics and services come into range, and they disappear as they leave.

9.3 Branch Ownership and Access Control

Dozens of independent, untrusted authors write to the same scene simultaneously. A city fabric, a retail fabric, a transit service, and a restaurant review overlay all contribute objects to the same SOM. Without ownership boundaries, any service could overwrite, corrupt, or impersonate another's content — a transit service could alter a retailer's store geometry, or a malicious overlay could inject fake safety alerts into a city fabric. The SOM needs a structural mechanism that confines each author to its own territory.

Each spatial fabric's map service and each service anchored within a spatial fabric owns a branch of the SOM tree. Ownership is hierarchical and strictly enforced.

Each spatial fabric and each service owns a branch of the SOM tree. Ownership is hierarchical and strictly enforced.



WRITE: each owner can modify only its own subtree. READ: all services can read the full tree (subject to per-branch permissions).

Figure 9.3: *Branch ownership in the SOM. Each fabric and service owns its subtree. Write access is confined to owned branches; read access spans the full tree subject to per-branch permissions.*

A service can only write to its own branch. The transit service cannot modify the retail store's geometry. The restaurant review overlay cannot inject content into the city streets fabric. The browser enforces these boundaries at the API level. Ownership is over branches of the tree, not regions of three-dimensional space. Two services whose visual content overlaps spatially still own separate branches. The SOM's logical structure and the scene's spatial layout are independent.

By default, services can read broadly across the full SOM for spatial awareness. But some data is private — consider a multi-player card game where each player's hand must be hidden from others. Two layers of protection address this:

- Server-side filtering. The service's backend (via RMAP) sends each player only the data they are authorized to see. Private data never crosses the wire. This is the primary defense.
- Browser-side read permissions. Even for data that has reached the browser, the SOM enforces read restrictions per branch. A WASM service module sees the SOM minus any branches it is not authorized to read. This prevents a compromised or malicious service module from accessing data belonging to other services.

Together, write confinement, read permissions, and server-side filtering form a three-layer security model. WASM sandboxing (Section 12) adds a fourth layer that protects code isolation — one service cannot crash or interfere with another.

9.4 Single-Copy Architecture

The SOM must exist as a single authoritative copy in application memory.

Maintaining multiple copies of the scene state would consume memory proportionally to the number of copies, introduce synchronization overhead between them, and create consistency bugs when copies diverge. At 90 frames per second with dozens of concurrent writers, keeping multiple copies consistent is not merely expensive; it is intractable. Every synchronization point is a potential frame-time violation, and every divergence is a potential rendering artifact.

The architecture achieves single-copy access through two mechanisms:

- **ANARI shared arrays.** The SOM's geometry, transforms, and material data are stored in memory layouts that ANARI (and the rendering engine behind it) can read directly, without copying into a separate rendering buffer. The rendering engine reads the committed scene state in place through ANARI's shared data interface.
- **WASM host functions.** WASM modules access the SOM through host functions exposed by the browser, not by copying scene data into their linear memory. The host functions provide a controlled view of the SOM, filtered by the module's read permissions, without duplicating the underlying data.

Any runtime, language, or execution model that requires its own copy of the scene data is architecturally incompatible with this requirement. The implications for content execution are addressed in Section 12.

9.5 Concurrent Read/Write at Frame Rate

The SOM must support simultaneous writes from fabric streams and WASM modules while the rendering engine reads it (through ANARI) for rendering, all at 90 frames per second (11.1 milliseconds per frame).

The SOM is a continuously mutating interface. Dozens of services, fabric streams, and browser subsystems write to it concurrently. The rendering engine reads from it every frame. There is no moment when the SOM is "complete" — it is always in motion, always being written to. Nor is it ever "partial" in any meaningful sense. It perpetually exists only in its current state. Next frame, it will be a different current state.

This is by design. Like audio samples or avatar positions, a late or missed update is not worth waiting for — the next frame will overwrite it anyway. The architecture does not synchronize all writers to a frame boundary. It does not require the SOM to reach a globally consistent state before a read. The rendering engine (through ANARI) reads whatever is there right now, and what it reads is good enough because it will read again 11 milliseconds later.

The only hard constraint is structural integrity: individual reads and writes to the data structure must be atomic at the node level. A read must never see a half-written node (some properties from the old state, some from the new). A write must never leave the tree in a structurally corrupt state. This is a standard concurrent data structure problem, addressed through lock-free techniques and atomic operations at the node or branch level — not through global locks, buffering, or duplication.

9.6 Spatial Indexing

The SOM's logical hierarchy (the tree of branches described above) organizes objects by ownership. But rendering, physics, and proximity queries need to find objects by position: what is in front of the camera, what is within arm's reach, what just entered the user's vicinity.

A spatial index (such as a bounding volume hierarchy, an octree, or a k-d tree) is maintained alongside the logical hierarchy. It provides:

- **Frustum culling for rendering:** which objects are potentially visible to the camera.
- **LOD selection:** which level of detail to use based on distance from the viewer.
- **Proximity queries:** which services and fabrics are within interaction range.
- **Collision and interaction detection:** what the user's hands or controllers are touching.

The spatial index is a view of the same data, not a copy. It is updated incrementally as objects are added, moved, or removed from the SOM.

9.7 Asynchronous Writers and the Render Loop

The SOM is not updated in a single synchronous pipeline. It is continuously mutated by multiple independent, asynchronous processes, and periodically read by the renderer. These two categories of activity operate on different schedules.

Asynchronous SOM writers (not tied to the frame rate):

- **RMAP data arrival.** Data from fabric streams and service backends arrives asynchronously over the network. Incoming events are dispatched to listeners that modify the SOM as needed — adding objects, updating properties, removing content that has gone out of range. This is event-driven, not frame-driven.
- **WASM service execution.** Each active service runs on its own thread, reading the SOM (filtered by its permissions) and writing to its own branch. Services may respond to incoming RMAP events, react to changes elsewhere in the SOM, or conjure objects procedurally. They modify the SOM whenever their logic requires, not at frame boundaries.
- **Browser-managed computation.** Animations, transform cascading, and other internal browser operations also update the SOM. The internal simulation rate for these computations is 1/64 of a second (one tick, ≈ 15.6ms) — a different cadence than the

display frame rate. Whether tick-driven updates execute independently or are folded into the render loop is an open design decision.

The render loop (at frame rate, 90fps / 11.1ms):

- **OpenXR provides tracking data.** Head pose, hand pose, and controller input are read from the XR runtime.
- **The view position is computed.** The user's physical-world position (from GPS/VPS) and head tracking (from OpenXR) are combined to position the virtual camera relative to the SOM's origin. The SOM's content does not move; the viewpoint does.
- **The scene is rendered through ANARI.** ANARI reads the SOM in whatever state it happens to be in at this moment and delegates to the rendering engine, which translates the scene into draw calls against the platform's GPU API (Vulkan, Metal, DX12) to produce the rendered image.
- **The frame is submitted to the compositor.** The rendered frame is handed to the OpenXR compositor (or to the display, on non-XR devices) for presentation.

The render loop does not wait for writers to finish. It does not coordinate with RMAP, WASM, or the browser. It reads the SOM as it is in the given moment, renders, and submits. The asynchronous writers continue mutating the SOM while the frame is being rendered, and the next frame will reflect whatever has changed in the interim.

10. Rendering Abstraction (ANARI)

ANARI (Analytic Rendering Interface for Data Visualization) is a Khronos Group standard that provides a high-level, scene-graph-level rendering API. The application works with abstract objects (meshes, materials, lights, cameras, volumes) and ANARI delegates the rendering to whichever rendering engine is loaded.

The MBE must render a complex, multi-source 3D scene to the screen every frame. But the GPU rendering API differs on every platform: Vulkan on Windows, Linux, and Android; Metal on Apple; DirectX 12 on Windows; proprietary APIs on specialized hardware. Calling any one of these directly would couple the browser to a specific vendor's ecosystem and destroy the device independence established in Section 1. The browser needs to describe *what* the scene looks like and let something else figure out *how* to draw it on the local hardware.

10.1 What ANARI is — and what it is not

ANARI is a very thin abstraction layer — essentially a standard of standards. It defines a uniform API for describing scenes (meshes, materials, lights, cameras) and delegates the actual work to an interchangeable rendering engine behind it. ANARI itself performs no rendering, no scene traversal, no GPU memory management, and imposes no measurable performance overhead. If you can do it today without ANARI, you can do it under ANARI — it will not stand in your way. Its sole purpose is to guarantee that the rendering engine behind it can be swapped out without changing the application code above it.

10.2 Downstream Components

ANARI composites output from two sources that reside between ANARI and the GPU:

- **The rendering engine.** A substantial piece of software that translates ANARI's high-level scene descriptions into low-level draw calls. The rendering engine handles scene traversal, frustum culling, draw call batching, GPU memory management, LOD selection, and shader compilation. It calls the platform's GPU API (Vulkan, Metal, DX12) to communicate with the hardware. All performance characteristics (frame rate, memory consumption, concurrency behavior, scalability limits) are properties of the rendering engine, not of ANARI itself. The selection of this engine for the exemplar MBE is a significant architectural decision, analyzed in detail in Section 14.
- **The UI toolkit.** A separate component that renders 2D interface content (text, buttons, layout, input primitives) to GPU textures. Those textures are applied to SOM objects and rendered by ANARI alongside scene geometry. The UI toolkit and the rendering engine never call each other; the browser mediates between them through textures and SOM objects. The UI toolkit provides the browser's own interface as well as a service-facing toolkit that WASM services use to create UI without building their own rendering pipeline. The full analysis of the UI toolkit's architecture, candidate libraries, and open design questions is presented in Section 15.

To draw an analogy, ANARI is to rendering what a database abstraction layer is to SQL dialects. The application writes queries (scene descriptions) and the rendering engine translates them to whatever the GPU speaks. Swapping the rendering or UI toolkit is like swapping a database driver — the application's queries do not change.

10.3 glTF PBR alignment

ANARI's material model aligns with glTF's Physically Based Rendering (PBR) parameters. Standard materials — the vast majority of surfaces in the metaverse — are expressed as glTF PBR parameter sets (base color, metallic, roughness, normal, emissive) on SOM objects and flow directly through ANARI to the rendering engine without any shader compilation. Custom visual effects beyond PBR require SPIR-V shaders (Section 23.4). This alignment is expected to be formalised by Q3 2026.

11. XR Device Abstraction (OpenXR)

OpenXR is a Khronos Group standard that provides a unified API for XR hardware interaction. It abstracts headset tracking, controller input, hand tracking, and rendering submission across every vendor's device into a single interface the browser codes against once.

The MBE must interact with XR hardware — headsets, controllers, hand tracking, eye tracking, spatial anchors — across every vendor's device. Before OpenXR, every headset required its own proprietary SDK: separate code paths for Oculus, separate code paths for SteamVR, etc. and no path at all for future devices that did not yet exist. For a browser that must run on any device

without modification, this per-vendor coupling is untenable. OpenXR eliminates it by placing a single standard interface between the browser and every XR device on the market.

Note that mobile phone-based AR uses ARKit on iOS and ARCore on Android devices that provide a different set of functionality than OpenXR, which primarily targets headsets. The proposed solution (Section 16.2) is an OpenXR-compatible wrapper around these libraries, so the browser always talks to an OpenXR interface regardless of platform.

11.1 Primary Functions

OpenXR's capabilities divide into five areas:

- **Device discovery and session management.** Finding the headset, establishing a session, managing its lifecycle. Sessions have states (idle, ready, synchronized, visible, focused), and the application responds to transitions.
- **Tracking.** Head pose, controller pose, hand pose, spatial anchors. Everything is expressed in coordinate spaces, and the runtime handles sensor fusion. The application asks "where is the user's head?" and gets a position and orientation.
- **Input (action system).** Instead of coding to specific hardware buttons, the application defines abstract actions ("grab," "teleport," "menu") and binds them to hardware through interaction profiles. New controllers work without code changes.
- **Rendering submission.** Frame timing, view configuration (mono, stereo), swapchain management (the texture chains the application renders into), and composition layers (how rendered frames are composited for the display).
- **Extensions.** Hand tracking, eye tracking, face tracking, passthrough (AR/MR), scene understanding, spatial anchors. These capabilities are provided through OpenXR extensions without breaking the core specification.

11.2 Key Concepts for the MBE

Several OpenXR concepts have direct architectural implications for the browser:

- **Reference spaces.** OpenXR defines view space (head-relative), local space (seated origin), and stage space (room-scale floor origin). The mapping between OpenXR's reference spaces and each spatial fabric's coordinate system is an architectural decision that requires careful design.
- **Composition layers.** The application submits layers that the runtime composites: projection layers (the main 3D scene), quad layers (2D UI panels), and environment layers. This maps naturally to how the MBE composes output: the main SOM rendered as a projection layer, with UI overlays as quad layers.
- **Multi-vendor runtime model.** An application targeting OpenXR runs unchanged on Meta Quest, SteamVR, Android XR, Monado, and every other conformant runtime.

11.3 What OpenXR does not do

OpenXR does not render anything. It has no concept of meshes, materials, or scenes. It provides tracking data (where things are), a place to submit rendered frames (swapchains), and input events (what the user is doing). Rendering is the domain of ANARI and the rendering engine behind it.

11.4 How services access input

The browser is the sole consumer of OpenXR. Services never interact with OpenXR directly. The browser reads tracking, input, and extension data each frame and exposes relevant subsets to WASM services through host functions (Section 12.3). Sensitive input channels — eye tracking, face tracking, and any camera-derived data — are gated by the user consent model (Section 22). A service receives only the input it has been granted permission to see.

12. Content Execution (WASM)

The MBE executes code from hundreds of simultaneous, untrusted sources. Each service (a safety alert, an inventory tracker, a training walkthrough) runs its own logic every frame: animations, data processing, user interaction, backend communication. The browser has no prior knowledge of any of this code and no reason to trust it. On the web, JavaScript runs in a single-threaded, shared-heap environment because there is typically one origin active at a time. In the MBE, however, a hundred independent code modules from a hundred unrelated providers execute concurrently in the same process, all writing to the same scene. The execution runtime must isolate each one completely, enforce per-service resource limits, prevent any one from degrading the others, and do all of this fast enough to fit within an 11.1-millisecond frame budget.

WebAssembly (WASM) is a binary instruction format for a stack-based virtual machine designed to meet exactly these constraints. It is not a programming language; it is a compilation target. Developers write in C, C++, Rust, AssemblyScript, or other languages and compile to .wasm bytecode. A WASM runtime executes that bytecode in a sandboxed environment relatively close to native speed — typically 50-80% depending on the workload — which is the best performance achievable within a sandbox. The performance cost is the price of isolation.

WASM was originally designed for use in web browsers, but the specification has no dependency on the browser. Standalone WASM runtimes (Wasmtime, Wasmer, WasmEdge) are embeddable C/C++ libraries. The MBE links a WASM runtime directly, with no browser engine, no DOM, and no JavaScript engine.

12.1 Why WASM

The issues stated above create four non-negotiable requirements:

- **Per-service memory isolation.** A buggy or malicious service must not be able to read or corrupt another service's state. WASM enforces this at the specification level. Every module instance gets its own linear memory, a contiguous byte array the module can

grow within limits but cannot escape. There are no raw pointers, no ability to address memory outside the module's own allocation. This is a fundamental constraint of the instruction set, not a runtime check.

- **Per-service threading.** Services must not share a single thread. The MBE controls how WASM modules execute. Each service runs on its own thread (or one from a managed pool) with its own memory sandbox. A hundred services means a hundred independent instances executing concurrently.
- **CPU budget enforcement.** One poorly performing service must not degrade the entire experience. WASM runtimes provide fuel metering (every instruction burns fuel; when exhausted, execution traps gracefully) and epoch-based interruption (the MBE increments a counter; modules yield when the epoch changes). The MBE can also cap each module's maximum memory allocation.
- **Best-available sandboxed performance.** WASM bytecode compiled from C++ or Rust executes relatively close to native speed (typically 50-80%, depending on the workload), due to sandbox overhead such as bounds checking and indirect call validation. This is measurably slower than unsandboxed native code, but it is the best performance achievable within an isolation boundary — and isolation is non-negotiable. With a hundred services executing every frame, the cumulative CPU cost of content execution remains a critical engineering constraint.

12.2 Execution Model

Aspect	How It Works
Runtime	Standalone WASM runtime (e.g., Wasmtime) embedded in the MBE's native codebase
Module loading	.wasm binaries received from services, compiled once, cached
Instance per service	Each service gets its own module instance with isolated memory and execution state
Threading	Each instance on its own OS thread or from a managed pool
Memory isolation	Each instance's linear memory is inaccessible to all others (instruction-set enforced)
Memory caps	Maximum linear memory size per instance, set by the MBE
CPU budgets	Fuel metering or epoch-based interruption per frame/tick
Capability control	WASI controls host resource access; services get only granted capabilities
Failure handling	Traps caught by the MBE; failing service does not affect others

Aspect	How It Works
API contract	WASM imports (MBE → module) and exports (module → MBE), typed and verified at load

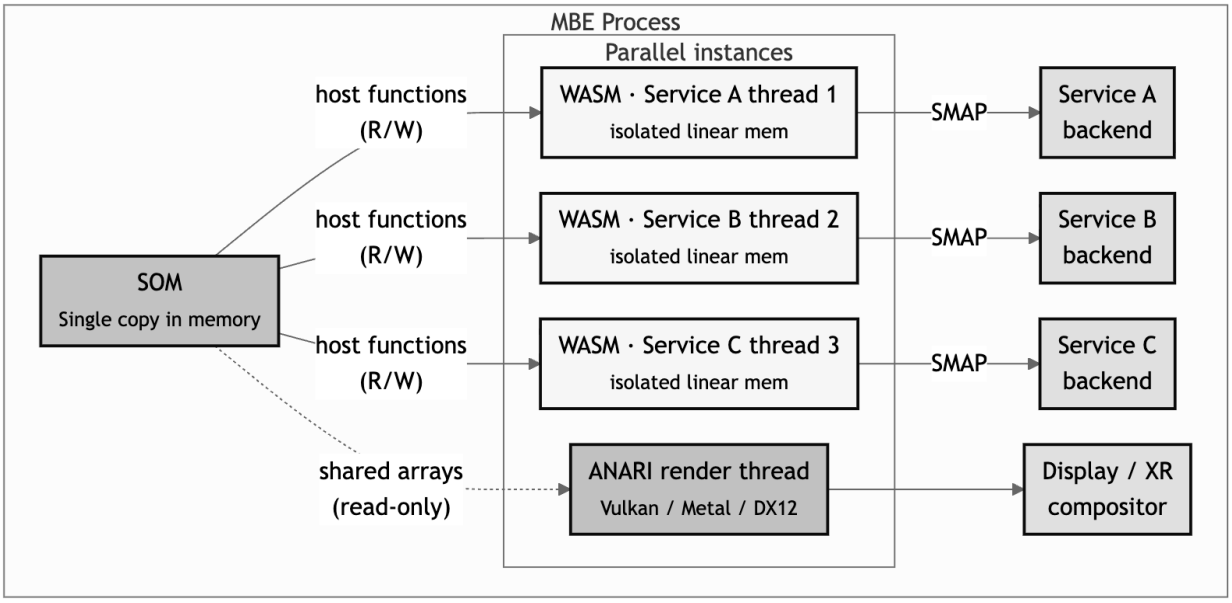


Figure 12.2: *MBE runtime architecture. WASM service instances access the SOM through host functions (read/write); ANARI reads the SOM through shared arrays (read only) and delegates to the rendering engine, which issues draw calls to the platform's GPU API. Each WASM instance runs on its own thread with isolated linear memory. RMAP connects each service to its backend.*

12.3 Host Function Surface

WASM services interact with the MBE exclusively through browser-provided host functions. The host function API is broader than the SOM and RMAP alone:

- **SOM read/write.** Services read the scene for spatial awareness and write to their own branch to contribute content. The SOM is the sole interface between service code and rendering — services describe what should exist in the scene, and the rendering engine (through ANARI) handles how to draw it. No service has any contact with ANARI or the rendering engine, direct or indirect.
- **RMAP model access.** Services communicate with their backend servers through RMAP, working with model objects that represent server-maintained state.
- **Input state.** Services query user input (button presses, hand gestures, gaze direction, controller state) through host functions that expose data the browser has read from OpenXR. Sensitive input (such as eye tracking) is subject to permission controls.

- **Shader submission.** Services submit SPIR-V bytecode for dynamically generated visual effects. These are graphics shaders (fragment, vertex) for custom rendering effects that glTF PBR parameters cannot express. Service-submitted GPU compute programs are not currently supported (see Section 23.4).
- **Spatial queries.** Services query the SOM's spatial index for ray casting, proximity queries, collision detection, and coordinate transforms through host functions. The browser performs the query against its internal spatial data structures and returns results, keeping the spatial index and its implementation private.

Every capability is mediated by the browser. The host function boundary is the security perimeter: services cannot reach past it to access the GPU, the XR runtime, the network stack, or any other service's memory. What a service can do is determined entirely by what host functions the MBE exposes to it.

12.4 The Case Against JavaScript

JavaScript will be the loudest-demanded alternative to WASM. The argument will be: "JavaScript is the most popular language on earth, every developer knows it, it powers the entire web — therefore you cannot exclude it from the metaverse." This argument confuses authoring convenience with runtime suitability. JavaScript's runtime design is architecturally incompatible with the MBE's requirements in every dimension that matters.

- **Single-threaded by design.** JavaScript's execution model is fundamentally single-threaded with a cooperative event loop. The MBE requires each service on its own thread. Web Workers exist, but each is a heavyweight separate engine instance with serialization overhead on every message.
- **No memory isolation.** All JavaScript in a single execution context shares the same memory heap. Achieving isolation requires separate engine instances (V8 Isolates), each heavyweight. WASM provides memory isolation by specification, enforced by the instruction set.
- **No CPU budget enforcement.** There is no built-in mechanism to limit how much CPU time a JavaScript execution context consumes. A runaway loop blocks its thread indefinitely. The only recourse is termination, not graceful degradation.
- **Performance gap.** JavaScript is dynamically typed, interpreted (then JIT-compiled after warmup), and garbage-collected. For compute-heavy, type-stable workloads, WASM from C++ or Rust is typically 2-5x faster than well-optimized JavaScript after JIT warmup — and the gap widens for workloads that stress allocation, garbage collection, or polymorphic call sites. This is not the strongest argument against JavaScript in this architecture; the other points are more damning, but it compounds them.
- **Garbage collection is incompatible with real-time rendering.** JavaScript's garbage collector can pause execution at unpredictable intervals (typically 1-10ms, sometimes higher). In a 90fps VR application with an 11.1ms frame budget, a single GC pause can

cause a dropped frame. Dropped frames in VR cause motion sickness. WASM has no garbage collector; memory management is deterministic.

- **Memory overhead per context.** Isolating services in JavaScript requires separate V8 Isolates, each carrying megabytes of fixed overhead (heap, JIT state, garbage collector, built-in prototypes). A hundred V8 Isolates would consume hundreds of megabytes to gigabytes before any service code runs. On memory-constrained XR devices, this is disqualifying. WASM instances are lightweight: a linear memory buffer sized to the service's needs plus shared compiled code.
- **Dynamic typing is wasted overhead.** Every operation carries runtime type checks. Polymorphic call sites, hidden class transitions, and deoptimization bailouts degrade performance unpredictably. WASM operates on fixed-width types determined at compile time.
- **Attack surface.** JavaScript engines are among the most complex and most targeted software in existence, with hundreds of Common Vulnerabilities and Exposures (CVEs). Embedding one in the MBE would embed one of the largest attack surfaces in computing into a native application running untrusted code. WASM runtimes are simpler by orders of magnitude, purpose-built with security as a primary design goal.
- **Engine dependency.** Running JavaScript requires embedding V8 or SpiderMonkey — millions of lines of code with their own release cycles, bugs, and technical debt. WASM runtimes are smaller and purpose-built for embedding.
- **Single-copy SOM incompatibility.** The MBE maintains one authoritative copy of the SOM in C++ memory. The rendering engine (through ANARI) references it without copying (shared arrays). WASM modules access it through host functions with no serialization. JavaScript cannot participate in this architecture. V8's garbage-collected heap is a separate memory space. Giving JavaScript access to the SOM requires either full duplication into JS objects (synchronized on every update) or expensive proxy wrappers with marshalling overhead. With a hundred services, this means a hundred partial copies or a hundred heavyweight bridge layers. The single-copy SOM is achievable specifically because of the ANARI/rendering-engine + WASM architecture; adding JavaScript would break it.
- **"Support both" is the worst option.** Embedding both WASM and a JavaScript engine doubles the attack surface, doubles the maintenance burden, doubles the API surface, and doubles the testing. The experience still degrades whenever any JavaScript service is in the scene, because all of the above problems apply. The MBE is only as good as its worst-supported runtime.

There is historical precedence. WASM was created because JavaScript's performance ceiling is too low for demanding workloads. Google Earth, Figma, AutoCAD, Adobe Photoshop, and Doom 3 all moved to WASM in the browser. The metaverse browser's performance requirements are categorically higher than the web browser's. If JavaScript already cannot keep up with

single-application demands in a web browser, it cannot serve as the execution runtime for a hundred simultaneous services in a 90fps VR application.

12.5 Developer Access

Excluding JavaScript from the runtime does not exclude JavaScript developers from the metaverse. The authoring language is the developer's choice:

Language	Path to WASM	Audience
C / C++	Emscripten, WASI SDK	Systems programmers, game developers
Rust	Native WASM target (first-class)	Systems programmers, safety-focused developers
AssemblyScript	Native WASM compiler	JavaScript/TypeScript developers (familiar syntax)
Go	TinyGo → WASM	Go developers
C#	Blazor / NativeAOT → WASM	.NET developers
Swift	SwiftWasm	Apple ecosystem developers
Kotlin	Kotlin/Wasm	Android/JVM developers

AssemblyScript deserves particular attention: it is a strict subset of TypeScript designed specifically to compile to efficient WASM. For JavaScript/TypeScript developers, the syntax is nearly identical, the learning curve is minimal, and the output is WASM bytecode with all the performance and isolation guarantees the MBE requires.

13. Service Connectivity (RMAP)

In the metaverse, the MBE connects simultaneously to dozens of services it has never seen before, each built by a different provider, each with its own protocol, its own data model, and its own real-time update patterns. A safety alert service might use WebSockets. An equipment monitoring service might use a custom binary protocol optimized for sensor telemetry. A retail inventory service might use gRPC. Others use REST APIs that require consumers to read documentation and write integration code for each specific service. The browser cannot write custom integration code for each one — it discovers these services passively through proximity and must connect on the spot, with no developer in the loop.That model is impossible when the browser encounters new services every ten steps. The MBE needs a standard way to connect to any unknown service as routinely as it renders an unknown mesh.

Remote Model Access Protocol (RMAP) is the MBE's answer. RMAP was developed by Metaversal Corporation (the company behind RP1) and has been in production use for several years. It is not yet a formal standard; the OMBI intends to advance it as an open standard through an

appropriate standards body. It is to networked services what ANARI is to rendering: a standard interface that decouples the browser from the underlying implementation.

13.1 Model Access, Not Data Transfer

RMAP is not a transport protocol and it is not a data interchange format. It is a model access protocol. The service provider defines a model — a class definition describing the shape of server-maintained objects, their properties, events, and actions. RMAP provides a standardized way for the client to view and manipulate instances of that model. The client works with the model interface. How data actually moves between server and client — the encoding, the transport, the update strategy — is entirely the service provider's concern, implemented in their source adapter, and invisible to the browser.

This is a deliberate inversion of how the web works. On the web, the interchange format is the standard: HTML specifies every tag, attribute, and behavior in hundreds of pages of specification. RMAP specifies no wire format at all. The standard is the model contract — the class definition — not the payload. Service providers are free and encouraged to use established data formats where appropriate, but for real-time telemetry or high-frequency updates, a compact binary protocol can be orders of magnitude smaller and faster than structured text. The choice is the provider's, and they can change their implementation at any time without the client needing to know.

13.2 Service Driver Inversion

This is the core architectural concept. The service provider creates, distributes, and maintains the client-side integration (the source adapter and model definitions). The browser does not build integrations for each service. The service ships its own driver. When a new service appears, its provider publishes a RMAP package. The browser loads that package, and now it can connect.

This inverts the traditional client-server relationship. In the traditional model, the client developer writes code to integrate with each service's API. In the RMAP model, the service developer writes the client-side adapter once, and every conformant browser can use it.

13.3 Three-Layer Architecture

RMAP organizes client-side communication into three layers:

- **Service layer.** Manages the connection to a remote service: discovery, client connection, session identification (login), authentication, and session lifecycle. Handles both stateless and real-time connections transparently — the service provider chooses the connection model, and the browser adapts without needing to know the difference.
- **Model layer.** Represents the objects the service exposes, defined by the service provider's class definitions. The browser works with models — observable representations of server-maintained objects, with typed properties, subscribable events, and invocable actions. Models update in real time as the server pushes changes. The model definition is the contract; the browser never sees below it.

- **Source layer.** Translates between the model and whatever wire protocol the service actually uses. This is the adapter. It is the reason the browser never sees or cares about the underlying transport. It also allows one model to connect to different services that utilize different protocols.

13.4 Connection Lifecycle

Service providers publish RMAP packages (source adapters and/or model definitions). When the browser encounters a new service, it downloads the appropriate package, loads it, and connects. The mechanism is analogous to how a web browser downloads and executes JavaScript, except the package is compiled WASM with defined capabilities.

Each WASM service module has its own RMAP instance connecting to its backend. The WASM sandbox isolates the service's code. RMAP connects each isolated module to its own backend server. They are complementary: WASM provides execution isolation, RMAP provides connectivity.

Connection lifecycle (browser's perspective):

1. A spatial fabric or service comes into proximity range.
2. The browser discovers the service and identifies its RMAP package(s).
3. Each package is downloaded (if not cached) and loaded.
4. RMAP establishes a connection via the service layer.
5. Authentication occurs through a session object in the service layer (see also Section 18).
6. Models are opened and begin streaming real-time updates into the WASM module.
7. The WASM module's logic processes incoming data and updates the SOM.
8. When the service leaves proximity range, the connection is closed and resources are released.

14. Rendering Engine

The rendering engine is the single most performance-critical component in the browser. Every frame, it walks the SOM, determines what is visible, translates scene descriptions into GPU draw calls, binds shaders, manages GPU memory, and submits the result — all within an 11.1-millisecond budget for 90 fps XR. The user never sees the rendering engine, but they see nothing without it. Visual fidelity, frame rate, battery life on mobile, and the upper limit on scene complexity are all properties of this one component.

ANARI guarantees that the rendering engine is interchangeable — the browser's application code never calls it directly. But interchangeable does not mean interchangeable with anything. The metaverse browser's requirements eliminate most existing renderers. No existing ANARI-compatible engine meets all of the MBE’s requirements today. Closing this gap is the most significant near-term challenge in the MBE's rendering stack.

14.1 Requirements

The MBE's rendering engine must meet a specific set of constraints that differ from typical scientific visualization or offline rendering:

- **Real-time frame rates.** 90 fps for VR headsets, 60-120 fps for AR glasses, 60 fps for flat screens. The frame budget is 11.1 ms or less. Every frame, the engine must traverse the SOM, cull invisible geometry, batch draw calls, bind shaders, and submit to the GPU within that budget.
- **ANARI-compatible.** The engine must expose an ANARI interface so the browser's application code never calls a GPU API directly. This is non-negotiable — it is the architectural expression of the first principle established in Section 1.
- **Rasterization-first.** The mass-market devices for the metaverse are phones, AR glasses, and standalone headsets. These devices do not have the GPU power for real-time ray tracing at 90 fps. The rendering engine must be rasterization-first, with ray tracing as an optional enhancement on capable hardware (desktop GPUs, workstation-class devices).
- **Multi-source scene.** The SOM is composed from dozens of simultaneous fabrics and services. The engine must handle a scene graph that is continuously mutating — objects appearing, disappearing, and updating every frame from multiple asynchronous writers — without stalling.
- **Cross-platform GPU API support.** The engine must target Vulkan (Windows, Linux, Android), Metal (Apple), and DirectX 12 (Windows). A single engine that can compile against all three, or a clean abstraction that allows per-platform engine variants, is required.
- **SPIR-V shader consumption.** Custom shaders arrive as SPIR-V bytecode. The engine must accept SPIR-V and bind the resulting shaders into its pipeline.
- **Embeddable.** The engine ships as a browser dependency (Layer 3), not as a standalone application. It must be embeddable as a library (statically linked or dynamically loaded) without pulling in its own windowing, event loop, or application framework.

14.2 Current Landscape

Existing ANARI-compatible rendering engines:

- **OSPRay (Intel).** Mature, production-quality ray tracer. CPU-focused with GPU acceleration. Excellent for scientific visualization. Not designed for real-time rasterization at 90 fps on mobile hardware.
- **VisRTX (NVIDIA).** GPU ray tracer built on OptiX/RTX. ANARI-native — the first production renderer that speaks ANARI. High visual quality. Requires NVIDIA GPU hardware and ray tracing cores. Positioned as a development reference and high-end enterprise tool, not a mass-market consumer engine.

- **VisGL**. Rasterization-based ANARI backend. OpenGL-based. The closest existing option to a rasterization-first ANARI engine, but built on a legacy graphics API (OpenGL) rather than modern Vulkan/Metal/DX12. Performance and feature set are limited compared to what the MBE requires.
- **Halogen.** An ANARI compatible wrapper around Google's Filament, a physically-based rasterization renderer designed for mobile and desktop (Vulkan, Metal, DX12). It is lightweight, embeddable, and targets the platforms the MBE needs. Halogen+Filament is the rendering engine currently in use by Sneeze. Its cross-platform reach and strong glTF PBR material support make it well suited to the MBE's requirements.

The gap is clear: no existing ANARI backend is a rasterization-first, cross-platform, modern-GPU-API engine suitable for real-time XR at scale. This is the most significant near-term gap in the MBE's rendering stack.

14.3 Alternative Paths

ANARI's entire purpose is to make the rendering engine swappable. The browser can ship with one engine today and replace it with a better one tomorrow — the application code above ANARI will not change. Halogen+Filament fills the rasterization gap today, but other engines may offer advantages for specific platforms, use cases, or performance profiles in the future:

- **Adapt an open-source game engine renderer.** Engines like O3DE (Amazon, Apache 2.0) and Godot (MIT) contain mature rasterization pipelines built on Vulkan. Their rendering subsystems could potentially be extracted or adapted to expose an ANARI interface. The rendering subsystem of O3DE (Atom) is architecturally modular and has been explored as a candidate. The challenge is decoupling the renderer from the engine's application framework, scene graph, and asset pipeline — producing a standalone library rather than an embedded component of a larger system.
- **Partner with ANARI backend developers.** The Khronos ANARI working group includes organizations actively developing backends. A partnership to build or sponsor additional rasterization backends could leverage existing ANARI expertise and ensure specification conformance from the start.
- **Purpose-built engine.** Build a new rendering engine from scratch, designed specifically for the MBE's requirements: ANARI-native, rasterization-first, cross-platform (Vulkan/Metal/DX12), optimized for multi-source streaming scenes. Maximum architectural alignment, but the largest engineering investment.

Different browser vendors or different browser compilations from the same vendor may ship different rendering engines, just as different web browsers use different JavaScript engines. The user sees the same scene either way. Competition at the rendering engine level, with ANARI guaranteeing interoperability above it, is a healthy outcome.

15. Spatial User Interface

On the web, the browser provides a rich UI framework that every website builds on — text rendering, font rasterization, layout, scrolling, input focus, cursor handling, clipboard, virtual keyboards, hit testing, and accessibility — all delivered automatically through HTML and CSS. Web developers take this for granted. No one writes a font rasterizer to display a login form.

In a 3D spatial environment, none of this exists unless the browser provides it. A WASM service that wants to display a settings panel with a slider and a text field must solve an extraordinary number of problems: rendering text onto a surface in 3D space, placing that surface so the user can see and interact with it, routing input from whatever device the user has (gaze, hand ray, controller pointer, touch screen), managing focus across panels from multiple services visible simultaneously, handling virtual keyboard invocation on headsets, and ensuring the UI is accessible to users with disabilities. Expecting every service to solve all of this independently is unrealistic. It guarantees a fragmented, inconsistent user experience and an impossibly high barrier to entry for service developers.

The MBE must provide two layers of UI capability: the browser's own application shell and a service-facing spatial UI toolkit.

15.1 Rendering Integration

The UI library renders its content (text, buttons, layout) to an offscreen GPU texture. That texture is applied as the material on a SOM object — a flat rectangular surface in 3D space. To ANARI, it is just another textured quad in the scene. ANARI renders it alongside every other mesh, with correct depth sorting, occlusion, and lighting. Each system operates in its own domain without crossing into the other's.

For input, the browser ray-casts from the user's pointer (gaze, hand ray, controller, mouse, touch) against the panel's geometry. If it hits, the 3D intersection point is transformed to 2D coordinates within the texture, and the browser forwards that as a UI input event.

Not all UI elements should be occluded by the scene. The browser manages at three rendering tiers with different depth behavior:

- **Scene geometry.** Normal depth-tested rendering. Fabrics, objects, avatars — everything occludes and is occluded naturally.
- **World-space UI panels.** Service panels anchored in 3D space. These participate in depth testing like any other scene object — if a wall is between the user and a panel, the wall occludes the panel. This is the correct behavior for UI that belongs to a specific location in the scene.
- **HUD / overlay elements.** Head-locked controls, always-visible indicators (hamburger menu, notifications, navigation). These render on top of everything with depth testing disabled so that no scene object can cover them. In XR, these are composited as an

overlay layer by the XR compositor (the OpenXR-native approach). On flat screens, they draw after the scene pass.

This model adapts to both display modes:

- **XR.** World-space service panels are SOM objects that participate in the scene naturally. HUD elements use the XR compositor's overlay layer, guaranteeing visibility regardless of scene content. Head-locked panels stay at a fixed distance in front of the user.
- **Flat screens.** The 3D scene renders via ANARI first. Service panels can appear as world-space objects within the 3D viewport or as 2D overlays, depending on context. HUD elements draw after the scene pass.

This architecture means the choice of UI library is a contained decision. The UI library never calls ANARI. ANARI never calls the UI library. The browser mediates between them through textures and SOM objects — a clean boundary that allows either component to be replaced independently.

15.2 Application Shell

The browser needs its own interface for identity management (sign-in, persona selection), service filtering and consent, settings and preferences, content rating warnings, avatar selection and customization, audio controls, notifications, and navigation. The application shell is built using the platform's native UI framework — the operating system's own windowing, menus, toolbars, and controls. On desktop, this means standard OS widgets. On mobile, native UIKit or Android Views. In XR, the platform's compositor overlay mechanisms. The browser application owns this layer entirely; the MBE engine does not prescribe it. This keeps the engine focused on the 3D viewport while allowing each browser vendor to build a shell that feels native to the platform.

15.3 Service-Facing UI Toolkit

The architectural parallel to the web's HTML form elements and DOM API. The browser exposes host functions that let WASM services create and manage UI without building a rendering pipeline:

- **Panel creation and placement.** Services request a UI panel; the browser renders it as a texture surface in the scene. Placement modes include world-anchored (fixed in space), view-anchored (follows the user's gaze at a fixed distance), and hand-anchored (attached to the user's hand or controller). The browser manages panel placement to prevent collisions and ensure readability.
- **Text rendering.** Font rasterization, layout, line wrapping, and scaling handled by the browser. Services provide text content and styling parameters; the browser renders it. Text must remain legible across viewing distances, which means automatic scaling or LOD for text surfaces in 3D.
- **Input primitives.** Buttons, text fields, sliders, toggles, dropdowns, scrollable lists, and radio groups — the spatial equivalents of HTML form elements. The browser renders them,

handles device-appropriate input (touch, gaze-and-dwell, hand pinch, controller click, mouse), and returns events to the service.

- **Virtual keyboard.** On headsets and AR glasses, text input requires a virtual keyboard. The browser provides and manages it. Services that create text fields get keyboard support automatically — they do not implement their own.
- **Focus management.** Multiple services may have visible panels simultaneously. The browser manages input focus: which panel is active, how focus transfers between panels, and how to prevent one service's UI from capturing input intended for another.
- **Layout.** A layout system for arranging elements within a panel (row, column, grid, and flexible spacing). Services describe their UI structure; the browser computes the layout. This is the spatial equivalent of CSS Flexbox.
- **Styling.** A constrained styling model for colors, fonts, spacing, borders, and opacity. Services can customize the look of their panels within the browser's framework, similar to how web form elements accept CSS but maintain baseline behavior.
- **Accessibility.** Screen reader hooks, alternative input routing, and semantic element descriptions built into the toolkit from the start — not retrofitted. A button exposed through the toolkit is automatically accessible to assistive technology.

15.4 Candidate Libraries

The service-facing toolkit needs an underlying UI rendering implementation that runs inside the 3D viewport. The application shell is handled by the platform's native UI (Section 15.2) and is not a factor here. The candidates for the in-viewport toolkit:

- **Dear ImGui.** Immediate-mode C++ library (MIT license). Extremely lightweight, trivial to integrate with any rendering backend, and massively adopted in game engines and 3D tools. Fast to prototype. However, it looks like developer tooling, not a consumer product — no retained widget tree, limited layout, and weak text rendering for rich content. Excellent for development overlays, debug tools, and performance dashboards. Not suitable for shipping consumer UI or the service-facing toolkit.
- **RmlUi.** HTML/CSS-like markup with retained-mode rendering (MIT license). Familiar authoring model (an HTML/CSS subset), renders through a pluggable backend, supports data binding, and provides good text and layout. Designed specifically for games and 3D applications. Smaller community than ImGui or Qt, and not a full web engine (no JavaScript, no DOM) — which is a strength here but limits what designers already know. Provides the layout and styling power of web UI without the web engine dependency. Open source, embeddable, actively maintained.
- **Qt / QML.** Full widget toolkit (LGPL or commercial license). Mature, feature-rich, excellent text rendering, strong accessibility support, large ecosystem. But it is a massive dependency with licensing complexity (LGPL requires dynamic linking or a commercial license). Integrating with a custom rendering pipeline is nontrivial — Qt pulls in its own

event loop, windowing, and rendering, all of which the MBE already owns. Viable but heavyweight. The licensing and integration overhead make it a difficult fit for an open-source browser engine.

- **Ultralight.** Lightweight HTML/CSS renderer (proprietary license). Near-web-quality rendering without a full browser engine, familiar authoring model. But the proprietary license contradicts the open standards philosophy. Ruled out on licensing grounds.
- **Custom implementation.** Built from scratch on ANARI (no external dependency). Maximum control over rendering, XR integration, and API design. But the engineering effort is enormous — text rendering, layout, input handling, accessibility, and virtual keyboard are each substantial projects on their own. Reserved for specific spatial interaction patterns (gaze targeting, hand panel placement) that no existing library handles. Not viable as the entire UI stack.

The recommended approach: RmlUi as the UI rendering layer for the service-facing toolkit inside the 3D viewport. Custom spatial interaction code handles the 3D-specific layer (panel placement in world space, gaze/hand input routing to the 2D UI surface, and XR-specific behaviors) while RmlUi handles everything within the panel surface.

15.5 Open Design Questions

- **Toolkit API shape.** What host functions does the browser expose to WASM services? A declarative model (services describe UI structure, browser renders it) versus an imperative model (services create and manipulate individual elements) versus a markup model (services pass a UI document, browser renders it).
- **Theming and fabric-level styling.** Can fabrics define a visual theme that service panels adopt? A retail fabric might want a consistent look across all services within it, while still allowing each service to customize.
- **3D-native elements.** Beyond flat panels, are there primitives for 3D UI — radial menus, spatial sliders, volume controls that exist as objects in the scene rather than on a panel surface?
- **Panel lifecycle.** How does a service's UI panel persist when the service goes out of range? Does it fade, collapse to a notification, or vanish? What about services that want persistent UI (a music player, a navigation overlay)?
- **Input priority.** When a user's hand ray intersects both a UI panel and a scene object behind it, which gets the input? The panel capture and passthrough model needs definition.

16. Platform Landscape

This section details the present landscape of the core browser components: which backends are available on which platforms, where the gaps are, and how the architecture handles them. The five abstraction layers defined in Section 8, governed by the first architectural principle in Section

7, are not theoretical — they exist to absorb a specific, measurable fragmentation problem in the hardware and software landscape.

No two major platforms share the same combination of GPU API, compute API, and XR runtime. Windows offers Vulkan and DirectX 12; Apple offers only Metal; Android and Linux offer Vulkan; Quest runs on Android with its own OpenXR runtime. A browser that must run on all of these platforms cannot afford to assume any single API will be available everywhere. The tables below map the concrete backend availability that the abstractions must cover.

16.1 Rendering (ANARI, Rendering Engines, and GPU APIs)

Section 10 defines the ANARI abstraction and the role of the rendering engine behind it. This section maps those components to the concrete platform landscape.

Rendering engines (ANARI backends):

Engine	Type	Underlying GPU APIs
OSPRay	Intel ray tracing library	Embree
VisRTX	NVIDIA GPU ray tracing	OptiX, CUDA - NVIDIA only
VisGL	OpenGL rasterizer	OpenGL
Halogen+Filament (Section 14)	Rasterization-first engine, ported to ANARI and currently in use by Sneeze	Vulkan, Metal, DX12

The rasterization gap and candidate engines are discussed in Section 14.

GPU APIs (used by rendering engines):

GPU API	Type	Platforms
Vulkan	Cross-platform GPU rasterization and ray tracing	Windows, Linux, Android, Quest, Apple (via KosmicKrisp)
Metal	Apple's native GPU API	macOS, iOS, visionOS
DirectX 12	Microsoft's native GPU API	Windows

At least one GPU API is available on every target platform, and at least one rendering engine can target each GPU API.

16.2 XR Runtimes (OpenXR)

Section 11 defines the OpenXR abstraction as the browser's primary XR path. The table below maps the major XR runtimes and frameworks, indicating which are OpenXR compatible.

Runtime / framework	OpenXR	Device class	Platforms / devices
Meta OpenXR runtime	Yes	MR HMD	Horizon OS (Quest 3, 3S, Pro)
Meta Link OpenXR runtime	Yes	MR HMD	Windows (Quest via Link / Air Link)
Google Android XR OpenXR runtime	Yes	MR HMD, AR glasses	Android XR (Samsung Galaxy XR and partner devices)
SteamVR	Yes	VR HMD	Windows, Linux (Valve Index, Bigscreen Beyond, Pimax, tethered PC VR)
Pico OpenXR runtime	Yes (1.1 conformant)	MR HMD	Pico OS (Neo 3, Pico 4, Pico 4 Ultra)
HTC VIVE OpenXR runtime	Yes	MR HMD	VIVE Focus 3, XR Elite, Focus Vision (standalone); Windows (PC VR)
Varjo OpenXR runtime	Yes	MR HMD	Windows (Varjo XR-4, VR-3, Aero)
Magic Leap OpenXR runtime	Yes	AR glasses	Magic Leap 2
Snapdragon Spaces	Yes	AR glasses	Android (tethered — Lenovo ThinkReality, XREAL, TCL RayNeo)
Monado	Yes	VR HMD, MR HMD, AR glasses	Linux (open-source reference runtime; experimental Windows, Android)
visionOS (ARKit / RealityKit)	No	MR HMD	Apple Vision Pro
ARKit	No	AR phone	iOS (phone-based AR)
ARCore	No	AR phone	Android (phone-based AR)
Native 2D rendering	N/A	Flat screen	Laptops, desktops, phones used as flat screens

Device class legend

VR HMD — opaque head-mounted display, no passthrough.

MR HMD — head-mounted display with video or optical passthrough for mixed reality.
AR glasses — see-through optical display worn as eyewear.
AR phone — handheld device using camera passthrough.
Flat screen — conventional 2D display with no XR runtime.

OpenXR is the browser's path for VR HMDs, MR HMDs, and AR glasses, with Apple Vision Pro as the significant exception.

Phone-based AR uses platform-native frameworks — ARCore on Android or ARKit on iOS.

Laptops, desktops, and phones used as flat screens require no XR runtime at all.

Mobile phone support. OpenXR is not available as a native runtime on Android or iOS phones, where ARCore and ARKit are the widely-used tracking APIs. OMBI proposes to support mobile phones through a dedicated backend path alongside OpenXR-conformant devices, following the MoltenVK pattern of bridging a proprietary platform API to an open standard.

On Android, this could be realized by extending Collabora's Monado — already the core of the Android XR OpenXR runtime — with an ARCore-backed driver delivered via the Khronos Runtime Broker app. On iOS, the equivalent ARKit shim would ship as an in-app library, since Apple does not permit user-installable system runtimes. Both paths could be streamlined by an OpenXR handheld AR device profile which will be proposed to the Khronos OpenXR Working Group, giving ARCore and ARKit a standards-aligned participation path without requiring headset-class runtime support from either vendor.

Legacy / deprecated. Windows Mixed Reality was OpenXR compatible but was removed from Windows 11 starting with 24H2 (2024); community drivers such as Oasis keep HP Reverb G2 and similar devices working by exposing them as SteamVR devices. Microsoft HoloLens 2 production ended in 2024, though its OpenXR runtime remains available for deployed hardware.

16.3 Support for Apple

Apple does not support Vulkan or OpenXR on any of its platforms. This is a deliberate ecosystem strategy, not a technical limitation.

For rendering, the abstraction architecture handles Apple transparently. ANARI delegates to a rendering engine that targets Metal on Apple platforms. No browser code changes. This is precisely why the first architectural principle in Section 7 (abstraction over direct access) exists.

For SPIR-V shader ingestion, Vulkan-on-Metal translation layers (KosmicKrisp, MoltenVK) provide the bridge. These layers present a Vulkan API surface on Apple hardware, translating Vulkan calls to Metal internally, including SPIR-V shader consumption. The browser ships the translation layer as an additional dependency on Apple targets. The rendering engine and SPIR-V pipeline operate identically to any other Vulkan platform.

For XR device interaction, the situation is different. Apple provides its own AR/VR frameworks (ARKit, RealityKit) with no OpenXR compatibility. The MBE would require an OpenXR translation layer for ARKit and RealityKit so that the browser always talks to an OpenXR interface regardless

of platform (see section 16.2). This layer would translate OpenXR API calls into the equivalent ARKit/RealityKit calls. Building this compatibility layer is proposed as a dedicated working group effort. If Apple adopts OpenXR natively in the future, the layer becomes unnecessary and the browser benefits automatically.

16.4 Browser-Shipped Libraries

Some components ship with the browser rather than being provided by the operating system. Whether statically linked into the executable or distributed as dynamic libraries alongside it is an implementation decision for each browser vendor; architecturally, what matters is that these are browser-owned dependencies:

- WASM runtimes (Wasmtime, Wasmer, or WasmEdge) for executing service logic.
- The rendering engine (ANARI backend) for translating scene descriptions into GPU draw calls.

These libraries do not depend on the user's OS version or installed drivers. On Apple platforms, the browser also ships a Vulkan-on-Metal translation layer (KosmicKrisp or MoltenVK) to provide Vulkan API support and SPIR-V ingestion.

16.5 Per-Platform Availability

Vulkan is available on every target platform — natively on Windows, Linux, Android, and Quest; via KosmicKrisp (Khronos-conformant, Apple Silicon macOS, iOS support planned in 2026) or MoltenVK on other Apple platforms. Halogen+Filament targets Vulkan, Metal, and DX12 natively and reaches every platform on this basis. Future alternative engines (Section 14.4) can target Vulkan and achieve the same reach. DX12 remains available as a native option on Windows.

OpenXR is available on every headset target except Apple Vision Pro — Meta's runtime on Quest, Android XR on Samsung Galaxy XR and partner devices, SteamVR and Monado on Linux, and multiple runtimes on Windows PC VR. Apple visionOS is the sole headset platform without an OpenXR runtime, requiring a compatibility wrapper. Phone-based AR on Android and iOS uses ARCore and ARKit respectively and is outside OpenXR's scope.

Platform	Vulkan	Other GPU APIs	OpenXR Runtime	Non-OpenXR XR
Windows	✓ native	DX12	SteamVR, Meta Link, Varjo, VIVE, Pico Link, Magic Leap	—
Linux	✓ native	—	SteamVR, Monado	—
macOS	✓ via KosmicKrisp / MoltenVK	Metal	Not needed as flat screen	—

Platform	Vulkan	Other GPU APIs	OpenXR Runtime	Non-OpenXR XR
Android (phone)	✓ native	—	OpenXR Wrapper required	ARCore (phone AR)
Android XR (headset)	✓ native	—	Android XR runtime	—
Quest	✓ native	—	Meta OpenXR runtime	—
iOS (phone)	✓ via MoltenVK	Metal	OpenXR Wrapper required	ARKit (phone AR)
visionOS	✓ via MoltenVK	Metal	OpenXR Wrapper required	visionOS native (ARKit/RealityKit)

On desktop and phone-as-screen, no XR runtime is needed — the browser renders to the display through ANARI's standard rendering output path.

Part 3 — Supporting Component Systems

The core architecture defined in Part 2 does not operate in isolation. It depends on a variety of supporting systems, each of which presents its own design challenges. Where Part 2 addresses problems whose solutions align clearly with the core architectural principles, the sections that follow represent these supported systems along with their open questions. Some present a recommended direction with alternatives acknowledged. Others frame the problem and outline potential options without declaring a winner. All require input from the broader community: standards bodies, domain experts, enterprise implementers, and the working groups best positioned to solve each problem. The OMBI invites that collaboration.

17. Physical-World Anchoring (GPS/VPS)

GPS/VPS is the bridge between the physical world and the spatial fabric. It is what makes the AR side of the metaverse work, and it is the technical foundation of the proximity model described in Section 5. In VR, the user's position is virtual. In AR, the user's position is physical, and virtual objects must be overlaid at correct physical positions. For AR and VR users to share the same spatial fabric, the AR user's physical position must be mapped into the fabric's coordinate system with convincing precision.

17.1 Two Layers of Positioning

GPS (coarse, meters). Standard GPS provides 3-5 meter accuracy; assisted GPS roughly 1 meter. This is not precise enough to anchor virtual objects to real-world surfaces, but it drives the streaming paradigm: the browser uses GPS to determine which spatial fabrics are relevant (which neighborhood, which buildings), which services to connect to, and which content to load or unload. GPS is the trigger for "what should I be connected to right now?"

GPS does not work indoors. For factories, warehouses, hospitals, and other indoor environments, indoor positioning systems serve the same role.

VPS (precise, centimeters). Visual Positioning Systems use the device's camera to recognize the physical environment and return centimeter-level position and orientation. The device captures what it sees, compares against a database of known environments, and resolves exactly where the user is standing and which direction they are facing. VPS is what allows a virtual sign to float in front of a real building at exactly the right spot, or a safety zone alert to mark the precise boundary of a hazardous area on a factory floor.

17.2 Integration with OpenXR

OpenXR provides local tracking: the user's head and hand positions relative to where they started. It does not know where in the world "where they started" is. GPS/VPS provides the absolute anchor.

OpenXR tracking drifts over time. In small room-scale areas, current headsets achieve sub-centimeter accuracy. But as users move through larger spaces (buildings, city streets), the SLAM algorithms that inside-out tracking relies on accumulated error. Environmental factors (low light, featureless walls, reflective surfaces, repetitive patterns) degrade accuracy significantly.

GPS/VPS provides periodic re-anchoring, snapping OpenXR's drifted local space back to ground truth. The frequency of re-anchoring depends on the environment and acceptable error margin.

17.3 Camera Positioning

GPS/VPS positions the camera — the user's viewpoint — relative to the SOM's origin. Fabric content is positioned independently within the SOM by the scene graph hierarchy. The two are decoupled: GPS/VPS does not move or transform fabric content, and fabric placement does not

depend on the user's location. Because both the camera and the content share the same SOM origin as their reference frame, the viewport shows the correct spatial relationship without any explicit coordination between them.

17.4 VPS Data Ownership

Today's dominant VPS providers operate centralized scan databases. Google ARCore Geospatial relies on Google's Street View coverage. Niantic maintains over a million pre-mapped locations in its own cloud. If the metaverse depends on a single vendor's scan database for positioning, that vendor effectively controls where AR works — which spaces are mapped, who gets access, and what data policies govern the scans. An aerospace manufacturer's factory floor, a hospital's operating suite, a private home — all would need to be scanned into a third party's cloud to be AR-capable. For many enterprise and consumer environments, this is unacceptable. It also contradicts the decentralized, open architecture this document proposes.

The architecture recommends that VPS data reside with the fabric. A retailer scans their stores and hosts the VPS data on their own servers. A factory operator scans their facility. A homeowner scans their home. The browser requests VPS data from whatever fabric it is connected to and localizes against it. No central VPS authority, no uploading private spaces to a third party. Each fabric is sovereign over its own space — the same principle that governs content, services, and identity.

This model is viable because open standards for scan data interchange already exist. Fabric owners are not locked into a single vendor's scanning hardware or proprietary format. They scan with whatever equipment they choose, store in an open format, and serve it to any conformant browser.

17.5 VPS Data and Positioning Standards

Several open standards support the decentralized VPS model by ensuring that scan data and positioning output are interoperable across vendors and devices.

Scan data interchange.

Standard	Body	Purpose
E57 (ASTM E2807)	ASTM International	Vendor-neutral format for 3D point cloud and LiDAR scan data. Supported by all major scanning hardware (FARO, Leica, Trimble, NavVis, Matterport). Hybrid XML + binary architecture with lossless compression. The established interchange format for scan data.

Positioning output.

Standard	Body	Purpose
GeoPose 1.0	Open Geospatial Consortium (OGC)	Standard encoding for real-world position and orientation (6DOF) relative to a geographic reference frame (WGS84). Defines how to express where something is and which way it faces in a machine-readable, interoperable format. Already adopted by OpenVPS and other open implementations.

Platform-level protocols (emerging).

Standard	Body	Purpose
OSCP	Open AR Cloud	Open Spatial Computing Platform. A suite of protocols for spatial content discovery, VPS localization, and spatial data synchronization, built on decentralized principles with no central cloud dependency.
SpatialDDS	Open AR Cloud	Protocol specification (2026) for spatial data discovery, distribution, and synchronization across devices and services. Defines how spatial data is found and shared at the network level.

The existence of E57 and GeoPose means that fabric owners can scan with any hardware, store in an open format, and serve positioning data to any browser that speaks the standard. No proprietary lock-in is required at any layer. The OSCP and SpatialDDS protocols, while still emerging, point toward a fully open spatial infrastructure that aligns with the MBE's architectural principles.

17.6 Security and Privacy

Server-side VPS processing. Some VPS solutions require the device to send camera images to a server for localization. This means real-world imagery of the user's surroundings, potentially including people, private spaces, and confidential facilities, is transmitted to a third party. For an aerospace factory or a private home, this is a non-starter.

On-device VPS processing. The scan data is downloaded to the device and localization happens locally. No images leave the device. But the fabric owner's proprietary scan data is now on someone else's hardware.

A possible middle ground. The fabric sends a compressed, feature-only representation sufficient for the device to localize against, but not sufficient to reconstruct the full space. Full geometry stays server-side. Some VPS systems already work this way, transmitting feature maps rather than complete point clouds.

Location privacy. GPS/VPS data exposes precise physical location and therefore extremely sensitive. The fabric needs the user's position to serve content, but precise real-world location

should not be exposed to other users or services without explicit consent. This ties directly into the identity architecture (Section 18).

17.7 Technological Gaps

Darkness and low light. Camera-based VPS depends on visible features. In darkness, fog, or heavily occluded environments, performance degrades significantly. Mitigations include multi-sensor fusion (accelerometers, magnetometers, LiDAR) and deep learning models more robust to lighting variation. Darkness remains a recognized limitation, not a solved problem.

Extreme precision. Standard camera-based VPS provides centimeter accuracy. Some environments demand more:

Precision Need	Example	Required Accuracy
Consumer AR (street, retail)	Virtual signs, navigation, shopping overlays	1-5 cm
Professional AR (construction)	BIM overlay on a construction site	1-2 cm
Industrial (manufacturing)	Guided assembly of components	Sub-centimeter
Aerospace manufacturing	Fuselage panel positioning	Sub-millimeter (0.1mm)

Sub-millimeter accuracy requires specialized Real-Time Location Systems using infrared, ultrasonic, or acoustic sensors with installed infrastructure, not camera-based VPS.

Changing environments. Physical spaces change. A retail store reconfigures its floor layout. Products are moved to different shelves. Construction progresses on a building site. When the environment changes, VPS reference data goes stale and localization fails or produces incorrect positions. The frequency of re-scanning and distributing updated scan data is an operational challenge, not a technical one, but the architecture must accommodate it.

Environment transitions. As the user moves between fabrics, the positioning technology may change: outdoor sidewalk (GPS + camera VPS), retail interior (the store's camera VPS scan), aerospace factory (infrared RTLS), dark concert venue (LiDAR + inertial fallback). The transition must be seamless.

17.8 Positioning Abstraction Layer

The browser needs a positioning abstraction layer following the same pattern as ANARI (rendering) and OpenXR (device interaction). The browser asks "where am I?" and receives a position, orientation, and confidence level, regardless of whether the answer came from GPS, camera VPS, ultrasonic RTLS, LiDAR, or inertial dead-reckoning.

Each fabric provides a positioning backend appropriate to its environment. The browser interfaces with it through a standard API. Switching fabrics means switching positioning backends, transparently, with seamless handoff.

This abstraction would need to define:

- A standard query interface (position, orientation, confidence, timestamp)
- A backend registration mechanism (fabric declares its positioning capabilities)
- Handoff protocol between backends during environment transitions
- Quality-of-service metadata (accuracy tier, update rate, latency)
- Privacy controls (what positioning data the fabric receives about the user)

No such standard exists today. This is proposed as a new standard to be developed for the spatial internet.

17.9 Current VPS Landscape

Provider	Accuracy	Approach	Architectural Alignment
Niantic Spatial	Centimeter	1M+ pre-mapped global locations, deep learning	Largest coverage, but centralized database model
OVER VPS	Centimeter	TEE patent for encrypted on-device processing	Privacy-focused, aligns with on-device preference
MultiSet AI	Sub-5cm	Scan-agnostic (LiDAR, point cloud, photogrammetry)	Fabric-level ownership friendly (any scan format)
Google ARCore Geospatial	Centimeter (outdoor)	Google Maps street-level coverage	Proprietary, Google-controlled. Contradicts open architecture
EEVE VPS	~27cm	Fully autonomous, works without pre-mapped terrain	Fallback capability where no scan data exists

For the open metaverse, privacy-focused and scan-agnostic approaches are the most architecturally aligned. They avoid lock-in to a single vendor's scan format or cloud infrastructure.

18. Identity

Identity is the thread that follows the user through the metaverse. As the browser connects and disconnects from thousands of spatial fabrics and services automatically, each one needs to know who the user is — without the user doing anything. A warehouse worker walking an aisle, a

surgeon entering an operating room, a shopper browsing a store: the services they encounter must recognize them instantly, apply the right permissions, and respect the right privacy boundaries. Identity is what makes the transition from one service to the next seamless rather than interrupted by login screens, and what ensures that no service learns more about the user than it needs to.

18.1 The Problem

The 10-step rule applies to identity with the same force it applies to service installation. When people move through the metaverse, no one is going to want to create a new user account every ten steps for each new service they encounter. The MBE connects simultaneously to dozens of services. A user walking down a warehouse aisle passively encounters new services every few steps. Each service needs to know who the user is, and the user cannot be involved in that process each time.

18.2 Where Identity Cannot Live

Service-level identity can be eliminated. This is the current web model. Every website has its own account, its own signup flow, its own terms of service. The 10-step rule makes this impossible in the metaverse. A user encountering dozens of new services just by walking cannot create accounts for each one. Service-level identity is the pre-OAuth web, except instead of deliberately navigating to a website, the user passively encounters services every ten steps. The UX is impossible at this scale.

Per-fabric identity can also be eliminated. In this model, each fabric independently manages identity for the users within it, issuing its own credentials and maintaining its own account systems. This fails for the same reasons as service-level identity, raised one level in the hierarchy:

- **Centralized control.** Whoever operates the fabric controls identity for everyone in it. This is the AOL problem: centralized authority that contradicts the open standards vision.
- **Privacy violation.** All identity tokens flow from the fabric, giving it visibility into every service the user interacts with.
- **Not all fabrics are equipped.** A global fabric can handle identity management, but a small fabric (a local shop, a home) is not equipped to be an identity authority.
- **Interoperability.** Two users on different fabrics may not have compatible identities.

18.3 Viable Approaches

With service-level and per-fabric identity eliminated, three viable approaches remain. All three assume a protocol-based identity standard — the question is where that identity is hosted and managed.

Option A: Universal-fabric-owned identity. The user creates an account on one or more universal primary fabrics (the top-level fabric they enter the metaverse through). The universal primary fabric issues identity credentials that child fabrics and services accept. This is the most familiar

model — it mirrors how a platform like Meta or Apple manages identity across its ecosystem. It can be made to work, but it has several significant flaws:

- It forces the browser to enter the metaverse through a select few number of universal primary fabrics.
- The universal primary fabric operator controls the user's identity. If they revoke it, the user loses access to everything.
- The user's activity across all child fabrics and services is visible to the universal primary fabric operator.
- Identity is not portable. Moving between different universal primary fabrics means creating new identities and losing accumulated reputation, relationships, and history.
- Competition among universal primary fabrics becomes competition for identity lock-in — the same dynamic that has produced walled gardens on the web.

This option is viable for closed or enterprise deployments where a single operator controls the environment. It is a poor fit for the open metaverse.

Option B: Universal-fabric-managed identity. Identity is protocol-based and decentralized (the user owns their credentials), but a universal primary fabric acts as the wallet and management layer — storing credentials, presenting them to services, and handling the protocol on the user's behalf. This avoids the ownership problems of Option A: the user's identity is portable because it is defined by the protocol, not by the fabric. But it introduces its own issues:

- Identity management is only available when entering the metaverse through a universal primary fabric.
- The browser must delegate identity operations to whatever fabric the user happens to be in, creating inconsistency across environments.
- Every universal primary fabric must implement the full identity management stack, duplicating effort that the browser could handle once.
- The architecture already requires the browser to be the persistent, always-present layer across all fabrics. Adding identity to any fabric layer adds complexity with no corresponding gain.

This option is technically viable, and some deployments may prefer it. But fabrics should not be in the identity business — identity belongs with a dedicated service that exists for that purpose alone.

Option C: Identity service provider (recommended). Identity is an open protocol standard, managed by a dedicated identity service provider chosen by the user. The user creates their identity with a provider — the same way they create an email account with Gmail, Outlook, or ProtonMail — and the provider stores, manages, and secures the user's credentials, root identity, personas, and verifiable credentials. The browser acts as a conduit: it authenticates with the user's identity service, caches credentials locally for performance and resilience, and presents them to fabrics and services automatically on the user's behalf.

The identity service provider is not the identity. The identity exists according to the protocol standard. The provider is the custodian — the user can transfer their identity to a different provider at any time, just as they can transfer a domain name between registrars. Multiple providers will compete: large cloud companies (Google, Apple, Microsoft, Amazon) will likely offer identity services as part of their platforms, while smaller providers will serve users who prefer not to entrust personal identity to a technology giant. The competitive market and the portability guarantee prevent any single provider from becoming a gatekeeper.

This is the strongest option for the open metaverse:

- **Always available.** Identity service providers are cloud infrastructure companies that maintain the same availability standards as email, banking, and authentication services (99.99%+ uptime). Identity does not go offline when a device sleeps, a browser closes, or a network connection drops.
- **Multi-device.** The user signs into their identity service from any browser on any device — phone, AR glasses, desktop, a friend's headset. No credential export or import. No migration step. The identity is already there because it lives with the provider, not on any single device.
- **Device loss is not catastrophic.** Losing a phone or breaking a headset does not mean losing an identity. The user signs in from a new device and their full identity — root credentials, personas, verifiable credentials — is restored from the provider.
- **Fabric-independent.** The identity service is entirely separate from any spatial fabric. Identity follows the user across every fabric boundary without any fabric operator mediating, observing, or participating in the identity transaction.
- **Consistent.** Every service encounter is handled the same way: the browser retrieves the appropriate credentials from the identity service and presents them automatically. No variation based on which fabric, which device, or which browser.
- **Portable across providers.** Switching identity providers means transferring credentials from one provider to another — not starting over. The protocol defines the identity, not the provider that hosts it. This is the same portability model that prevents email lock-in: the protocol is the standard, the provider is a choice.
- **Accessible to everyone.** Creating an identity is as simple as creating an account with a provider. No cryptographic key management, no local wallet configuration, no backup procedures that require technical knowledge. The provider handles the complexity. A non-technical user can browse the metaverse without ever knowing what a DID or a private key is.

The mechanics:

- **Identity creation.** Done once. The user creates an identity with an identity service provider of their choice. The provider generates and stores the cryptographic material

(key pairs, DIDs) on the user's behalf. To the user, this feels like creating any other account.

- **Storage.** The identity service provider is the authoritative store for the user's credentials, root identity, personas, and verifiable credentials. The browser caches what it needs locally for the current session — enough to present credentials even during brief network interruptions — but the provider is the source of truth.
- **Service encounter.** As a service comes into range, the browser automatically presents the user's credential through the identity protocol. No user interaction needed. The service verifies the credential cryptographically. The identity service provider is not contacted for each encounter — the browser's cached credentials are sufficient.
- **Terms of service.** Services publish their terms in a standardized format. The browser handles presentation: auto-accept if terms match user-defined policy, prompt for review, or reject and skip the service. Agreement is recorded as a signed acknowledgment.

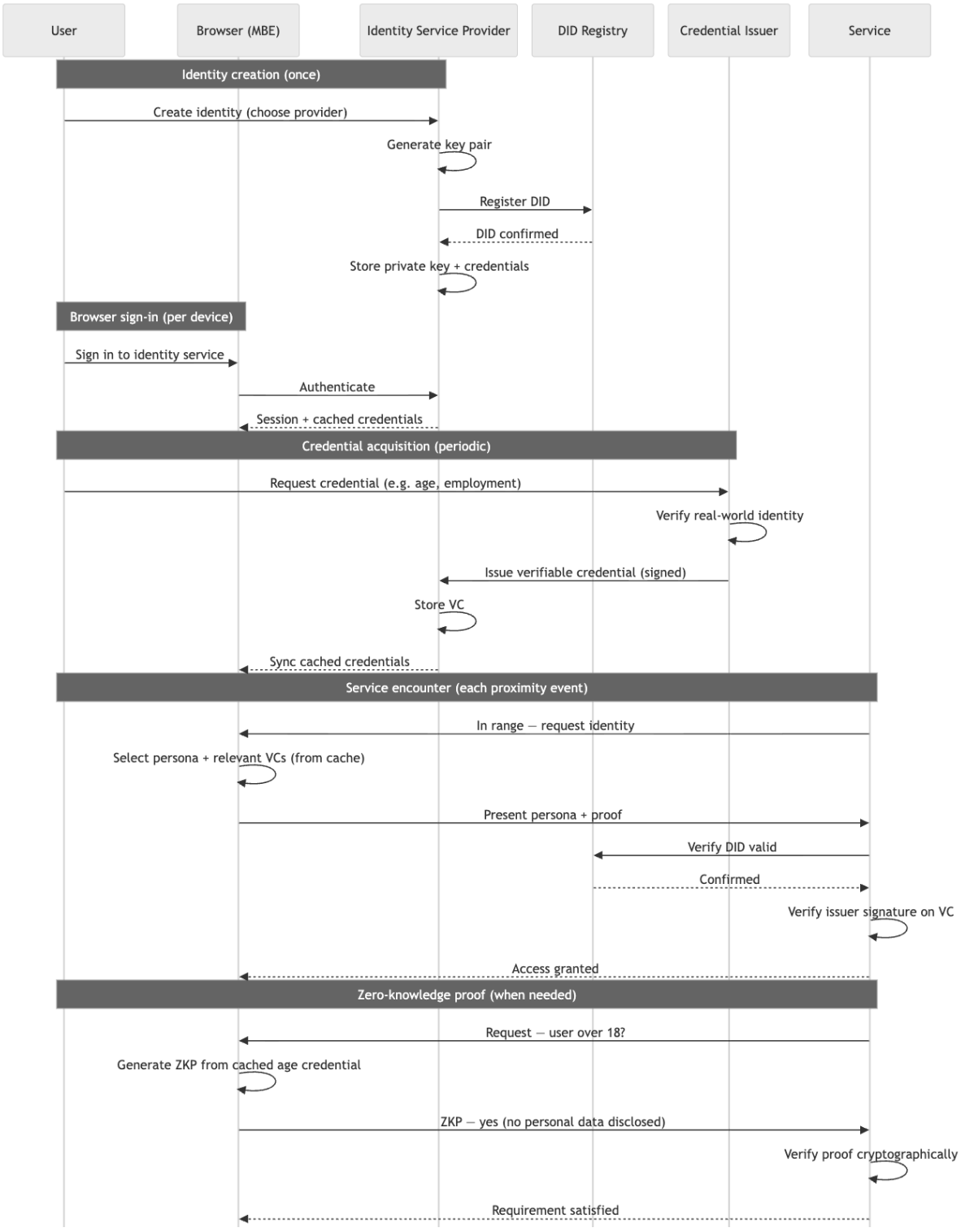


Figure 18.3: *Identity flow. The identity service provider stores credentials and the DID registry provides decentralized verification. The browser acts as a conduit — caching credentials locally and presenting them to services on the user's behalf. Credential issuers verify real-world identity; services never see more than the browser chooses to disclose.*

18.4 Root Identity vs. Personas

There must be a clear separation between a user's root identity and their public-facing personas.

Root identity is the cryptographic anchor. It is tied to a real human through proof of personhood, stored by the user's identity service provider, cached locally by the browser, and never revealed to any fabric or service. It is the one thing that proves the user is a unique, authenticated human being.

Personas are the public-facing representations presented to fabrics and services: username, avatar, preferences, history. A user can have multiple personas: professional in a corporate fabric, anonymous in a social space, pseudonymous in a game. Services see the persona, never the root identity.

The persona does not prove who the user is. It proves that the user is: a valid, unique, authenticated human entity, without revealing anything about that entity's real-world identity.

One human, one root identity is the target. Root identity creation requires proof of personhood anchored to real-world verification (government-issued credentials) that is extremely difficult to duplicate.

Multiple personas, unlinkable by default. Service A sees Persona X. Service B sees Persona Y. Neither service can determine they belong to the same root identity unless the user chooses to reveal the connection.

18.5 Selective Disclosure and Zero-Knowledge Proofs

A service that requires age verification receives a zero-knowledge proof that the user is of legal age, without learning the user's name, birthdate, or country. A payment service receives payment authorization without seeing browsing history. Each service gets only the minimum information required for its function.

Worked example: online gambling. A user visits a legal gambling service. The service needs to verify legal eligibility, facilitate gameplay, and report winnings to the appropriate tax authority, all without learning anything personal about the user.

- **Qualification.** The browser generates a zero-knowledge proof: "This persona is of legal age and resides in a jurisdiction where this activity is legal." The service receives a cryptographic "yes" and nothing more.
- **Gameplay.** The user plays under their persona. The service knows only that the persona is backed by a verified, eligible human.
- **Tax reporting.** The user wins. The service reports the taxable event against the persona. The tax authority, which issued the original credential, matches the persona to the taxpayer through the credential chain.

18.6 Safety and Accountability

The identity architecture addresses a serious failure of the current web: the inability to hold bad actors accountable while preserving privacy for everyone else.

Prevention through architecture:

1. **Privacy, not anonymity.** Every persona is backed by a verified human. A bad actor knows there is a cryptographic chain connecting their persona to a real person, even though no service can see it. The deterrent effect is significant.
2. **Permanent consequences.** A persona caught in illegal activity is banned. The root identity is flagged. The person cannot create a new persona to evade the ban because proof of personhood is anchored to real-world credentials that cannot be duplicated.
3. **Age verification that works.** Zero-knowledge proofs backed by government credentials mean age-restricted services can cryptographically verify legal age. This is a verifiable proof, not a checkbox.

Enforcement through selective de-anonymization:

4. **Legal authority follows the chain.** Under normal operation, the persona is completely private. With proper legal authority (court order, warrant), the cryptographic chain can be followed: the service reports the persona to law enforcement, law enforcement obtains a court order, and the court compels the credential issuer to reveal the person behind it. The service never learns who the person is. Only law enforcement with legal authority does.
5. **Multi-party requirement prevents abuse.** No single entity can unilaterally de-anonymize a user. It requires cooperation between the reporting service, the court system, and the credential issuer.

The design principle is privacy by default, accountability by legal process, the same model as telecommunications.

18.7 Protocol Options

Two protocol families satisfy the requirements. They are not mutually exclusive and may be combined.

Self-Sovereign / Decentralized:

Standard	Status	Role
W3C DIDs v1.1	Candidate Recommendation	Root identity: globally unique identifier controlled by the user

Standard	Status	Role
W3C VCs v2.0	Active working group	Persona claims: cryptographically signed credentials held by the identity service provider

DID methods include both blockchain and non-blockchain options (did:peer for direct peer-to-peer, did:web for web infrastructure, blockchain-based for maximum decentralization).

Standard	Status	Role
OpenID Connect	Mature, deployed at scale	Federated authentication through identity providers
OpenID Federation 1.0	Finalized February 2026	Trust networks between independent identity providers
FedCM	W3C standard	Privacy-preserving federated auth without third-party cookies

Potential hybrid: DIDs + VCs for the identity and credential layer (self-sovereign, portable), with OpenID Federation as the trust and verification protocol between services.

19. Avatars

Every user in the metaverse needs a visual representation — an avatar, an extension of the persona described in Section 18 — and the scale of that problem is unlike anything in current 3D applications. In an open plaza, hundreds or even thousands of avatars may be visible simultaneously. Each avatar must work across every fabric the user enters, without the user switching representations or creating new ones.

Avatar data must travel over the network, which means megabyte-sized avatar models are impractical at scale. The avatar must animate in real time from minimal tracking input (head and hands), support devices ranging from high-end VR to phones, accommodate non-humanoid forms, integrate with the identity architecture so that humans, AI agents, and NPCs are distinguishable, and respect content rating policies that vary by fabric. No existing standard solves this full problem. The architecture must define a model that balances fidelity, data size, portability, and scale.

Note that the Khronos glTF working group is actively developing avatar-related extensions. As that work matures, it may influence or supersede parts of the avatar architecture described below. The OMBI will track these developments and incorporate them as they become viable.

19.1 Responsibility Model

Browser responsibilities:

- Independent retrieval of users' avatar data from identity service providers when they come into proximity
- Rendering all avatars in the SOM via ANARI
- Inverse kinematics
- LOD selection based on device capability and distance
- Enforcing content rating policies
- Caching retrieved avatars

Fabric responsibilities:

- Presence management: who is here, where are they, broadcasting position updates between connected browsers
- Avatar policies: content rating requirements
- Providing NPC and AI agent avatars as part of the fabric's own content

The handoff. The fabric tells the browser "User X is at position Y." The browser independently retrieves User X's avatar from the persona layer, computes IK from the streamed input positions, and renders the avatar in the SOM. The fabric never touches the actual avatar data.

19.2 Entity Types

Every avatar carries a required entity type, tied to the identity architecture:

Entity Type	Meaning	Identity Backing
Human	Real person	Persona backed by verified root identity
AI Agent	Autonomous AI entity	Service-issued credential (not human-verified)
NPC	Fabric-controlled character	No identity; part of the fabric's content

The protocol must make it impossible for an AI agent or NPC to claim human entity type without a valid human-backed root identity. Every user gets an honest answer to "am I talking to a real person?"

19.3 Parametric Strategies and the Kilobyte Goal

Megabyte-sized avatars in public spaces are not practical. Avatar data needs to be reduced toward the low-kilobyte range for open-world scenarios with hundreds or thousands of visible users. This drives the entire avatar architecture toward parametric and shared-asset approaches.

- **Parametric definition.** A few hundred bytes of parameters (body shape, proportions, skin tone, hair style, accessory slots) evaluated against a shared reference skeleton and

material library built into every conformant browser. The browser generates the mesh locally. Data over the wire: hundreds of bytes.

- **Common skeletons and textures.** The standard defines reference skeletons and common base texture libraries, built into the browser or downloaded once and cached permanently. A specific avatar transmits only: which reference skeleton, parameter deltas, texture modifications, and custom attachment references. The shared foundation eliminates most per-avatar data.
- **Minimal-input animation.** Per-frame animation data is not streamed as bone transforms. Instead, minimal input is streamed: hand positions, head position, and possibly hip/foot hints from OpenXR (approximately 50-100 bytes per frame). Each receiving browser computes the full skeleton pose locally via inverse kinematics.
- **Progressive enhancement.** Start with the parametric definition (instant, kilobytes). If a custom high-fidelity override exists and bandwidth permits, stream it progressively.

19.4 High-Fidelity Scenarios

Not all scenarios require kilobyte-range avatars. In a boardroom meeting with six participants on high-bandwidth connections, avatars can be extremely high fidelity, even lifelike. The architecture must support the full spectrum from parametric silhouettes in a crowd of thousands to photorealistic representations in intimate settings.

19.5 Non-Humanoid Avatars

The avatar standard cannot assume a humanoid skeleton. Users may appear as robots, animals, abstract shapes, or anything else. The standard must support arbitrary meshes with arbitrary skeletal rigs.

The implications go beyond geometry. A humanoid avatar driven by a human user has a fixed, sparse set of animation inputs: head position, two hand positions, and whatever additional channels the XR hardware provides (fingers, face, eye gaze). The per-frame data packet for a human-driven avatar is predictable and small. But a software-driven entity faces no such constraint. An AI agent animating a cat has independent control over the head, facial expressions, four legs, a tail, ear orientation, and spine curvature — far more output channels than any human tracking system produces. An AI-driven dragon adds wing articulation, jaw movement, and independent claw control. A swarm of AI fireflies might each have only a position and a luminance value but number in the hundreds.

The animation input model must therefore be variable, not fixed. Per-frame animation data is not a fixed-format packet sized for two hands and a head — it is a variable-length stream of named channel values whose count and composition depend on the entity's skeletal rig and its animation source. A human user with basic tracking sends three channels per frame. An AI pet sends a dozen. An AI centaur with an articulated bow arm sends more. The standard defines the channel vocabulary and encoding; the entity defines which channels it uses. The browser's IK system

(Section 19.8) and network transport must all accommodate this variability without special-casing any particular skeleton or input source.

When a human user drives a non-humanoid avatar, the sparse human tracking data must be retargeted to the avatar's skeleton. Human arm movement maps to wing movement on a dragon. Human walking maps to quadruped locomotion on a horse. The avatar creator defines the retargeting map — a mapping from human input channels to the avatar's rig — and the standard needs a slot for that definition so the browser can apply IK correctly. The standard should therefore support two animation modes: retargeted animation (human input mapped to a non-humanoid rig) and direct animation (software-generated poses for arbitrary skeletal configurations, requiring no retargeting).

19.6 Level of Detail

Target devices range from high-end PC VR and desktops to standalone headsets, phones, and lightweight AR glasses. A single triangle budget cannot serve all devices. But device capability is only half the equation — the number of avatars currently on screen is equally important. The browser must balance fidelity against the total rendering budget available at any given moment.

LOD Tier	Triangle Budget	Typical Conditions
LOD 0	50,000-100,000+	High-end device, few avatars visible, close range
LOD 1	10,000-20,000	Mid-range device or moderate crowd, mid-distance
LOD 2	2,000-5,000	Lower-end device or large crowd, far distance
LOD 3	500-1,000	Any device at extreme distance, or massive crowds

The LOD tier assigned to each avatar is a runtime decision driven by the combination of device capability, avatar count, and distance. A standalone headset rendering five avatars in a conference room can afford LOD 1 for each. The same headset in a public plaza with two hundred visible avatars must drop most of them to LOD 2 or LOD 3, reserving higher fidelity for the nearest few. The browser manages this dynamically: as avatars enter and leave the scene, the LOD distribution shifts to make the best use of the available budget.

Parametric avatars make this particularly efficient: the generator produces whatever triangle budget the browser requests, with no separate LOD pipeline or storage multiplication. The browser simply asks for a different budget per avatar per frame based on current conditions.

Scene-level LOD (buildings, terrain, equipment, infrastructure) is a separate problem with different solutions. See Section 25.3.

19.7 Custom Shaders

The goal is for glTF PBR materials and their extensions (Section 23.3) to cover the vast majority of avatar surfaces without custom shaders. For effects that PBR cannot express (hair rendering,

cloth simulation, animated tattoos, holographic skin, particle trails), avatar creators can include custom SPIR-V shaders shipped alongside the avatar definition.

19.8 Inverse Kinematics

OpenXR provides partial body tracking: head position, hand positions, and potentially hip and foot positions depending on hardware. Newer devices and extensions add further input channels: individual finger positions (hand tracking), facial expression blendshapes (face tracking via camera or sensor array), and eye gaze direction. IK computes the full body pose from this sparse input, while facial expressions and finger poses are applied directly to the avatar's blendshapes and hand rig. The richer the tracking input, the more expressive and lifelike the avatar — but the architecture must degrade gracefully when input channels are unavailable. A device with only head and hand tracking still produces a usable avatar; one with full face and finger tracking produces a dramatically more expressive one.

This is a per-frame GPU compute workload that scales with the number of visible avatars. With 200 visible avatars in an open plaza, IK, finger posing, and facial expression mapping are 200 parallel computations per frame.

19.9 Content Rating

The avatar standard includes a content rating field tied to the identity architecture's age verification:

Rating	Meaning
G	General audiences, suitable for all fabrics
M	Mature, requires age-verified persona
A	Adult, requires age-verified persona and fabric permits adult content

Fabrics set their policy. The browser enforces it. If the user's current avatar exceeds the fabric's rating, the browser substitutes a default avatar or prompts the user to switch.

19.10 Existing Standards

Standard	What It Is	Relevance
VRM 1.0	Open standard built on glTF for VR/metaverse avatars	Strongest candidate as a base container format
ARF (ISO/IEC 23090-39)	Open ISO standard for avatar storage, carriage, and animation	Worth adopting for transport and animation streaming
glTF	Base 3D interchange format	Foundation for skinned meshes, skeletal animation, morph targets

Standard	What It Is	Relevance
SMPL/SMPL-X	Parametric body model (10-300 parameters → complete body)	Gold standard for parametric body generation; body only

No production-ready, open-standard parametric avatar system provides a full solution (body + face + hair + clothing + accessories, parameterized, kilobyte-range, standardized LOD). SMPL solves the body but stops there. VRM solves the container format but not parametric generation. ARF solves interchange but not generation. A realistic path combines SMPL-class parametric body generation with VRM as the output container, ARF for interchange, and a new parametric schema for face, hair, clothing, and accessories.

Avatars sit at the intersection of more disciplines than any other component in this architecture: 3D modeling, animation, identity, accessibility, real-time rendering, network transport, and human perception. No single working group owns this problem. The OMBI specifically invites collaboration from the avatar standards community (VRM Consortium, ISO SC 29/WG 3), the parametric body modeling community (Max Planck / SMPL), the animation and IK community, and accessibility advocates — to close the gap between what exists today and what the metaverse browser requires.

20. Spatial Audio

Sound is probably the most underestimated dimension of presence. Research in VR perception consistently shows that accurate spatial audio contributes more to the feeling of "being there" than increases in visual fidelity — a finding that surprises people until they experience it. A photorealistic room with flat stereo audio feels like watching a screen. A modest-fidelity room where footsteps land behind you, voices come from the right direction, and ambient hum shifts as you turn your head feels like a place.

Sound tells the brain “where it is”. When a colleague's voice comes from the correct position in a meeting room, the brain stops processing the experience as a simulation and starts processing it as a conversation. When environmental audio (traffic outside, ventilation overhead, a machine running three aisles over) is spatially coherent, the user's situational awareness operates the same way it does in the physical world. Getting audio right is not a polish pass. It is foundational to whether the metaverse feels real enough to use for work, communication, and daily life.

Note that the Khronos glTF working group is actively developing spatial audio extensions. As that work matures, it may influence or supersede parts of the audio architecture described below. The OMBI will track these developments and incorporate them as they become viable.

20.1 Voice

Voice is mixed server-side. The fabric's audio server receives each user's mono voice stream, performs all spatial mixing (HRTF, distance attenuation, occlusion, environmental effects), and returns a single stereo stream to each client, already spatialized for that client's position and head orientation. This model is driven by scalability (the server sends each client one stream

regardless of how many people are present), privacy (no client receives any other user's raw audio — a muted or blocked user's voice never reaches the device), and security (A/V zone enforcement happens server-side, where it cannot be bypassed).

From the browser's perspective, voice is simple:

- Capture and encode microphone input using the negotiated codec
- Send the mono stream to the fabric's audio server
- Receive the mixed stereo stream back
- Decode and output to the user's audio speaker device

The browser does not perform voice mixing, HRTF processing, or spatial voice computation. All of that is the server's concern.

20.2 Non-Voice: Client-Side Spatial Mixing

Non-voice audio (music, ambient sounds, machine noise, notification chimes, alerts) is mixed client-side. These sounds originate from objects in the SOM. A radio on a shelf plays music. A machine on the factory floor produces a whine. A safety alert emits a tone. The browser receives these audio sources as data attached to SOM objects and services and spatializes them locally based on the user's head position.

This is a GPU compute workload: for each sound source, compute 3D spatialization (direction, distance attenuation, occlusion against scene geometry, environmental reverb) relative to the listener, then mix all sources into the final output alongside the server-provided voice stream.

20.3 The Browser's Audio Stack

Audio Type	Source	Mixing Location	Browser Role
Voice	User microphone	Server-side (fabric)	Capture, encode, send; receive, decode, output
Non-voice	SOM objects and services	Client-side	Spatialize, mix, output

The two streams are combined at the final output stage: the server-mixed voice stream plus the client-mixed non-voice stream together form the user's audio experience.

20.4 A/V Zones

Spatial fabrics support designated audio/video zones where audio behavior differs from open space. These zones are defined by the fabric and represented in the SOM as spatial regions with associated audio policies:

- An auditorium where speakers on stage are heard uniformly and the audience is muted.

- A private conversation group where voices inside a spatial boundary do not reach clients outside it.
- A quiet zone where non-voice audio from outside is attenuated or muted.
- A conference room where only participants inside the room hear each other, with no audio leakage to the hallway outside.

A/V zones are fundamental to any social or enterprise space: conference rooms, classrooms, therapy sessions, legal consultations, private meetings on a factory floor. The browser must read zone definitions from the SOM and adjust its local mixing behavior (for non-voice) and its microphone routing (for voice) accordingly.

20.5 Codec

The architecture requires a codec optimized for speed over compression. In a server-side mixing model, the server must decode, spatially mix, and re-encode thousands of streams per audio frame. Consumer codecs like Opus optimize for compression at significant CPU cost per stream, which limits how many streams a server can process simultaneously. The metaverse's scale requirements invert this trade-off: compute time is the bottleneck, not bandwidth.

The browser supports a negotiable set of codecs. The fabric's audio server broadcasts which codecs it accepts; the browser selects one it supports. The recommended baseline codec prioritizes:

Parameter	Baseline Codec
Sample rate	24,000 Hz
Sample depth	16-bit
Channels	Mono (up), Stereo (down)
Frame rate	64 frames/second
Compression	~2:1 (lossy, imperceptible quality loss)
Encode/decode cost	Near zero per stream

Codec comparison:

Codec	Compression	Encode Cost	Delay	Scalability
Baseline (speed-optimized)	~2:1	Near zero	<1 frame	Thousands of streams per server
Opus	20-30:1	~1-3% CPU core	5-65ms	~50-100 streams/core

Codec	Compression	Encode Cost	Delay	Scalability
G.711	2:1	Negligible	<1ms	Fast but 8kHz narrowband only. Telephone quality
AAC-LC	10-15:1	Moderate	200-300ms	Latency disqualifies real-time use

The negotiable codec model allows fabrics to support higher-fidelity codecs when the use case and infrastructure warrant it, while ensuring every browser and every fabric can communicate at the baseline.

21. Inter-Service Communication

Services run in separate WASM sandboxes with isolated memory. In some scenarios, services may need to communicate with each other: a safety service might need to signal a navigation service to reroute around a hazard zone, or a training service might need to coordinate with an equipment monitoring service.

How services in separate memory spaces communicate is a deferred topic. A solution is being developed separately and will be integrated into this document when ready.

22. Filtering and User Consent

Proximity-based discovery (Section 5) means the browser automatically connects to spatial fabrics and their services as they come into range. Without a filtering layer, this model has the potential to overwhelm users with unsolicited content — the spatial equivalent of pop-up ads and notification spam. The filtering and consent system is the mechanism that gives users (and organizations) control over what actually reaches the scene. These controls apply to both fabrics and services alike. A user may want to suppress an individual service within a fabric, or dismiss an entire fabric — its geometry, its services, everything — from the scene entirely.

This is an area where the architectural requirements are clear but the detailed design requires significant community input. The mechanisms described below represent the categories of control that the system should consider supporting.

22.1 Per-Item Dismissal

The user hides a specific fabric or service instance (a particular restock alert, a specific promotional display, an avatar) and it remains suppressed until explicitly re-enabled. This is the most granular level of control.

22.2 Provider-Level Blocking

The user blocks all fabrics and services from a specific provider, across all locations. This is analogous to blocking a domain in a web browser's content blocker. If a user blocks a provider, none of that provider's fabrics or services activate, regardless of proximity.

22.3 Category-Based Filtering

Fabrics and services self-declare a category (advertising, social, entertainment, safety, operations, etc.), and users can suppress entire categories. A user who suppresses "advertising" would see neither ad-category services nor ad-category overlay fabrics. This requires a standardized category taxonomy and an enforcement mechanism — likely tied to the identity and credential system — to prevent fabrics or services from miscategorizing themselves to bypass filters.

22.4 Fabric-Level Muting

The user mutes an entire fabric. At the lightest level, the fabric's geometry remains visible but none of its services activate — the equivalent of muting a tab in a web browser. At the heaviest level, the fabric is removed from the scene entirely: its geometry, its services, and its contribution to the SOM are all suppressed. The browser may offer both options.

22.5 Proximity Threshold Control

Users can adjust the activation radius — how close a fabric or service must be before the browser connects. A tighter radius means fewer fabrics and services competing for attention. A wider radius means richer ambient information. The default should be sensible; the override should be accessible.

22.6 Density Management

The browser imposes a limit on the number of simultaneously active fabrics and services to prevent visual and cognitive overload. When the limit is reached, the browser prioritizes by proximity, user history, and declared urgency. Lower-priority fabrics and services queue until capacity is available.

22.7 Focus Modes

A "do not disturb" model that suppresses all non-essential fabrics and services. Only safety-critical services (which the architecture may designate as non-dismissable), user-pinned fabrics, and user-pinned services remain active. Focus modes may be user-initiated or triggered by context (e.g., entering a meeting space).

22.8 Enterprise Policy Overrides

In enterprise and industrial settings, organizational policy must be able to override individual user preferences in both directions:

- Mandatory services cannot be disabled by the user. A safety alert system on a factory floor, a compliance overlay in a regulated facility — these must remain active regardless of user preference.
- Prohibited fabrics and services are blocked at the organizational level. Advertising, social media overlays, or unapproved third-party fabrics and services may be suppressed across all devices on the premises.

The policy enforcement mechanism must integrate with the identity architecture and the enterprise deployment model. The specific format (policy files, MDM integration, fabric-level declarations) is an open design question.

22.9 The Web Parallel

The web faced this same problem and solved it imperfectly. Pop-up blockers, ad blockers, notification permissions, cookie consent dialogs — each was a reactive patch to an intrusion problem that the original architecture did not anticipate. The metaverse browser has the opportunity to design the filtering layer from the start as a first-class architectural component rather than an afterthought. The community's input on getting this right is essential.

23. Content Provider Ecosystem

The web browser did not succeed because of rendering engines and JavaScript VMs. It succeeded because an ecosystem formed around it. Text editors, image tools, and later sophisticated IDEs and build systems let millions of people create content that the browser could consume. The browser's job was to run that content reliably on any device. Everything else — authoring, compilation, hosting, distribution — happened above the browser, in an ecosystem the browser deliberately did not control.

The metaverse browser requires the same dynamic. The MBE is the rendering engine. Above it sits an ecosystem of authoring tools, compilers, content formats, and hosting infrastructure that content providers use to create spatial experiences. Just as the web browser never dictated whether a developer used Notepad or VS Code, PHP or Node.js, Apache or Nginx, the MBE never dictates how content is created — only what format it arrives in. And just as the web ecosystem matured from hand-written HTML to sophisticated build pipelines (TypeScript compiled to JavaScript, Sass compiled to CSS, JSX compiled to DOM calls), the metaverse ecosystem follows the same compiled-output pattern: author in high-level tools, compile to portable formats, ship the output.

The parallels are direct:

Web Ecosystem	Metaverse Ecosystem
Text editors, IDEs (VS Code, WebStorm)	3D authoring tools (Blender, Maya, Houdini, Revit)
TypeScript → JavaScript compiler	Slang → SPIR-V compiler, C++/Rust → WASM compiler
HTML/CSS/JS (content formats)	glTF, KTX, SPIR-V, .wasm (content formats)
Web servers (Apache, Nginx)	Metaverse servers (Map, other services)
CDNs (Cloudflare, Akamai)	Spatial content delivery (streaming geometry, textures)
npm, package registries	RMAP packages (service driver distribution)
Hosting providers (AWS, Vercel)	Fabric operators, service hosting infrastructure

The critical lesson from the web: the browser team does not need to build the ecosystem. The browser needs to define clear, stable, open formats at the boundary — and the ecosystem builds itself. HTML, CSS, and JavaScript were simple enough that anyone could start, and powerful enough that professionals could build at scale. glTF, SPIR-V, and WASM follow the same principle.

23.1 What Sits Above the MBE

Content providers use authoring tools to create assets that the MBE consumes at runtime. The MBE never sees these tools; it sees only their output.

Tool / Standard	What It Produces	Output Format Consumed by MBE
Slang	Custom shaders	SPIR-V (portable shader bytecode)
Blender, Maya, Houdini, Revit, Cinema 4D	3D models, animations, scenes	glTF
Substance 3D, Quixel Mixer	Materials, textures	KTX textures, PBR materials in glTF
C++, Rust, AssemblyScript	Service logic	.wasm (compiled WASM modules)
Spatial audio tools	3D audio	Spatial audio formats

The compilation boundary between authoring and distribution is the same boundary described in Section 8.5. Content providers author in whatever tools they prefer. What crosses the wire is compiled output.

23.2 Content Formats Consumed by the MBE

The MBE consumes compiled content formats, not authoring formats:

Format	Role	Analogy
glTF	3D asset interchange (geometry, materials, animation)	JPEG/PNG for the web
KTX	GPU-optimized texture format	Pairs with glTF for efficient GPU upload
SPIR-V	Portable shader bytecode	JavaScript for the web (compiled, not authored)
.wasm	Compiled service logic	JavaScript for the web (compiled, not authored)

These formats are handled the same way a web browser handles JPEG, CSS, and JavaScript: the browser knows how to consume them, but the browser does not care how they were created.

23.3 glTF — The MBE's 3D Content Format

GL Transmission Format (glTF) is a Khronos Group standard for 3D asset interchange. It is to the metaverse browser what JPEG is to the web browser — the format that arrives over the network and that the browser knows how to render.

Everything the MBE displays (buildings, furniture, avatars, vehicles, terrain, signage) arrives as glTF. A single glTF file (or its binary variant, GLB) can contain:

- **Geometry.** Triangle meshes, indexed or non-indexed, with vertex positions, normals, tangents, and texture coordinates.
- **Materials.** PBR (Physically Based Rendering) parameter sets that describe surface appearance as data, not shader code. See Section 23.4.
- **Textures.** Image data referenced by materials. When paired with KTX (GPU-optimized texture container), textures upload to the GPU without CPU-side decompression.
- **Skeletal animation.** Bone hierarchies, joint weights, and keyframed transforms for character and object animation.
- **Morph targets.** Blendshape deformations for facial expressions, procedural variation, and animation blending.
- **Scene hierarchy.** A node tree of transforms, enabling complex multi-part objects (a car with doors, wheels, seats, each independently transformable).

What glTF does not carry. glTF is a content delivery format, not a scene description system. It has no concept of streaming, change notification, multi-user editing, or deferred loading. Those capabilities belong to the map service layer (USD, SQL databases) and the SOM. glTF carries the payload; the SOM carries the live state.

Note that the Khronos glTF working group is actively developing scene-level extensions that may narrow this gap. As that work matures, it may influence how the MBE ingests and manages spatial content. The OMBI will track these developments and incorporate them as they become viable.

The PBR material model. glTF's native material system uses Physically Based Rendering (PBR) — a standardised set of surface parameters that produce realistic results under any lighting condition. The core metallic-roughness workflow defines five primary channels:

Parameter	What it controls	Example
Base color	Surface color (or albedo texture)	The red of a fire truck, the grain of oak
Metallic	Whether the surface behaves as a metal (0.0) or dielectric (1.0)	Chrome vs. plastic
Roughness	Surface micro-smoothness (0.0 mirror, 1.0 matte)	Polished glass vs. rough concrete
Normal map	Per-pixel surface detail without additional geometry	Brick texture, fabric weave
Emissive	Self-illumination independent of scene lighting	Neon signs, glowing controls

These parameters are data (values and texture references) not executable code. The rendering engine's built-in PBR shaders know how to interpret them. No compilation step is needed.

Extensions expanding PBR. The Khronos glTF extension pipeline progressively broadens what PBR parameters can express, reducing the need for custom shaders:

Extension	What it adds
KHR_materials_clearcoat	Clear lacquer or varnish layer (car paint, coated wood)
KHR_materials_sheen	Soft fabric-like reflectance (velvet, satin)
KHR_materials_transmission	Light passing through a surface (thin glass, colored liquid)
KHR_materials_volume	Refraction and absorption through solid volumes (thick glass, gemstones)
KHR_materials_ior	Index of refraction control
KHR_materials_iridescence	Thin-film interference (soap bubbles, oil slicks, beetle shells)
KHR_materials_anisotropy	Directional roughness (brushed metal, hair, carbon fibre)
KHR_materials_diffuse_transmission	Light scattering through translucent surfaces (curtains, paper, leaves)

Extension	What it adds
KHR_materials_variants	Multiple material configurations in a single asset (product colour options)

This list continues to grow. Each extension closes a gap that would otherwise require a custom SPIR-V shader, keeping the majority of materials on the simple, data-driven path.

Why glTF is the right choice. glTF aligns with the MBE at every layer. ANARI's material model is being formalised around glTF PBR parameters (expected Q3 2026). Halogen+Filament — the rendering engine currently in use by Sneeze — has strong glTF material support. OGC 3D Tiles uses glTF as its tile payload format for city-scale geospatial datasets. VRM 1.0, the leading avatar interchange standard, is built on glTF. The format is already the gravitational centre of real-time 3D interchange.

The conversion step. Content does not always originate as glTF. Authoring tools (Blender, Maya, Houdini, Revit) export to glTF. Map services that store data in USD convert to glTF before serving through RMAP — the browser never sees USD. BIM data stored in IFC converts to glTF through established pipelines (IfcOpenShell, xBIM). In every case, glTF is the delivery format at the browser boundary, regardless of what sits behind it.

23.4 Materials, Shaders, and GPU Programs

SPIR-V (Standard Portable Intermediate Representation) is a Khronos Group binary intermediate language for shaders and compute kernels. It is the portable bytecode that enables offload onto GPU hardware.

Every visual surface in the metaverse travels a pipeline from the content provider's authoring tool to the user's GPU. For the vast majority of surfaces, that pipeline is direct: the content provider sets glTF PBR parameters (base color, metallic, roughness, normal, emissive), and the rendering engine's built-in shaders consume them directly as data. No compilation, no bytecode, no custom code. For everything else (custom visual effects, procedural textures, novel lighting models, and GPU compute) the pipeline passes through SPIR-V:

Stage	Tool / Standard	Responsibility
Author	Slang (shader authoring language)	Content provider writes in Slang
Compile	Slang compiler → SPIR-V	Content provider compiles to portable bytecode
Distribute	Shipped as part of the service's content	SPIR-V bytecode crosses the network
Validate	SPIR-V validation (in the browser)	Browser rejects malformed or unsafe bytecode

Stage	Tool / Standard	Responsibility
Ingest	Rendering engine (ANARI backend)	Engine accepts SPIR-V and executes on local GPU

Relationship to material authoring tools. SPIR-V is the bottom of a two-stage pipeline. Content providers author shaders and materials using high-level tools — for example, Slang for custom shaders, MaterialX for material node graphs, or any tool that emits SPIR-V. Those high-level representations compile down to SPIR-V bytecode, which then follows the pipeline above. SPIR-V does not care how the bytecode was produced, whether hand-authored in Slang, generated from a MaterialX graph, or emitted by a generative AI service at runtime. It is the universal interchange point. The browser never sees the authoring tool; it sees only SPIR-V.

Validation and safety. SPIR-V shaders are untrusted third-party code destined for the GPU. The browser cannot execute them blindly. SPIR-V has a well-defined validation layer that the browser applies before handing bytecode to the rendering engine. Malformed bytecode, spec-violating constructs, and out-of-bounds resource access patterns are rejected before they ever reach the GPU. This is the GPU-side counterpart to WASM's CPU-side sandboxing — WASM ensures untrusted service logic cannot escape its memory sandbox on the CPU, and SPIR-V validation ensures untrusted shader code cannot reach the GPU without passing structural and safety checks. Full GPU-side sandboxing at the hardware level remains an open industry problem, but validation at the bytecode layer is an achievable and necessary first line of defense.

Native SPIR-V ingestion. SPIR-V is consumed natively by Vulkan today and DirectX 12 starting with the upcoming Shader Model 7. On Apple platforms, Vulkan-on-Metal translation layers (KosmicKrisp, MoltenVK) handle SPIR-V ingestion transparently — the layer translates SPIR-V to Metal Shading Language internally, without the browser's involvement. Native SPIR-V ingestion is the norm; where platform-specific translation is needed, it is an implementation detail of the rendering engine or driver, not a browser architectural concern.

glTF PBR. The vast majority of surfaces in the metaverse (metals, plastics, glass, fabric, skin, wood, stone, painted surfaces, coated materials) skip the SPIR-V pipeline entirely. They are described by glTF PBR parameters — data values on SOM objects (base color, metallic factor, roughness, normal map, emissive) plus the extensions listed in Section 23.3. The rendering engine's built-in PBR shaders consume these parameters directly. A content provider authors a material in any tool (MaterialX, Substance 3D, a game engine's material editor), and the export pipeline maps it to glTF PBR parameters. What arrives at the browser is render-ready data.

Custom SPIR-V shaders. SPIR-V is the escape hatch for content that needs visual effects that PBR parameters cannot express (procedural textures, animated materials, holographic surfaces, stylised non-photorealistic rendering, particle-driven appearances, novel lighting models). For these, the content provider authors a custom shader in a high-level language (Slang, or any language that targets SPIR-V), compiles to SPIR-V bytecode at build time, and ships it alongside the glTF asset.

Dynamic materials at runtime. A service that needs to change an object's appearance at runtime (a product configurator cycling through colour options, a building façade responding to time of day, a vehicle changing livery) simply writes new PBR parameter values to the SOM through host functions. Change the base colour from blue to red. Increase roughness. Swap the normal map texture. These are data writes, not code execution. No compilation, no server round-trip, no SPIR-V. ANARI reads the updated parameters next frame and renders the new appearance. This is the same mechanism services use to move objects (write a new position) or animate them (write new bone transforms). A service creating an object entirely from scratch (a chess piece, a dashboard widget, a procedural tree) creates the mesh geometry, creates a material, and sets PBR parameters through host functions. SPIR-V only enters if the service needs a visual effect that PBR cannot describe.

Dynamic shaders. The overwhelming majority of spatial content will use pre-authored materials, not runtime-generated ones. A service that wants to generate a novel visual effect at runtime (a generative AI creating materials from text prompts, a simulation producing appearance data on the fly) beyond what glTF PBR can handle will need to generate SPIR-V on the fly. But this is difficult because the browser carries no shader compiler. This is an acceptable limitation of the current architecture. One possible workaround is a server round-trip: the service sends a request to its server-side backend, the server generates and compiles the shader to SPIR-V, and returns the bytecode to the client for validation and execution through the standard SPIR-V pipeline.

Internal GPU compute. The browser itself makes internal use of SPIR-V compute pipelines for workloads that are highly parallel and must complete within the frame budget: spatial audio mixing, proximity queries, inverse kinematics, and physics simulation. Rather than prescribing a specific compute programming model at the architectural level, the browser dispatches these workloads as SPIR-V compute shaders through the platform's GPU API (Vulkan, Metal, or DX12), each of which natively supports compute alongside graphics. This is an internal implementation detail — services do not interact with these compute pipelines.

Service-submitted GPU compute. Whether services should be able to submit their own GPU compute programs (for tasks such as client-side physics, local data processing, or ML inference) is a known gap in the current architecture. ANARI does not support compute dispatch — it is a scene description API for rendering. A mechanism for service-submitted GPU compute would require a separate abstraction. Until this gap is resolved, services that require heavy parallel computation should perform that work server-side and stream results to the browser through RMAP, or perform it within their WASM sandbox on the CPU.

23.5 USD on the Map Service Side

Universal Scene Description (USD) is not part of the MBE's software architecture. It is one of several potential storage mediums that a map service may use internally. The map service reads USD, converts geometry and materials to map nodes that reference glTF files, and serves the result to the browser through RMAP. USD never crosses the wire to the browser.

The MBE's internal scene graph (the SOM) is its own data structure, not a USD Stage. The browser receives objects from map services via RMAP and places them into the SOM regardless of how the map service stores them. The transmission protocol between the map service and the browser is abstracted by RMAP; the browser does not know or care whether the map service uses USD, SQL, or any other storage medium.

Why USD matters on the map service side. USD is a scene description system: a live, composable, streamable, multi-user scene graph with an active composition engine. Its core properties (hierarchical namespace, non-destructive layering, deferred loading, change notification, concurrent multi-user editing, extreme scale, references over copies) align closely with what a map service needs to store and manage spatial data.

The barrier to entry. When a map service uses USD as its storage medium, any application that exports USD can provide the content for a spatial fabric:

- An architect designing a building in Autodesk Revit exports USD. A map service reads it directly and serves it as a spatial fabric.
- A game designer building a world in Unreal Engine exports USD. A map service reads it directly and serves it as a spatial fabric.
- An artist modeling a scene in Blender exports USD. A map service reads it directly and serves it as a spatial fabric.
- An entire city planned in NVIDIA Omniverse is already USD. A map service reads it directly and serves it as a spatial fabric.

No special tools, no custom formats, no learning a new pipeline. This is the same dynamic that made the web explode: any text editor could create HTML. USD means any 3D tool can produce the spatial content that a map service can read to turn into a browsable fabric.

USD and SQL map databases. For implementers using relational databases to store map service data, the conceptual overlap with USD is near-complete:

Property	SQL Map Database	USD
Hierarchical object system	Root → Category → Type → Physical objects	Hierarchical Prim namespace
Streaming / deferred loading	Load on demand via navigation	Payloads (structure visible, detail deferred)
Change tracking / events	Event system with ordered indices	Change notification
Typed object system	Distinct object types with defined columns	Schemas with defined attributes

Property	SQL Map Database	USD
Coordinate transforms	Geographic, Cylindrical, Cartesian	Transform properties on Prims
Concurrent multi-user editing	SQL transactions, multiple clients	Non-destructive layering
Extreme scale	Billions of objects	Designed for Pixar-level complexity
References, not copies	Objects reference shared assets	Instancing by reference
Pointers to external formats	Metadata and references to assets	References to glTF, KTX, etc.

A relational map database is essentially a USD-shaped system built in SQL. The scene description and storage layer (what the world looks like, how it is structured, how changes propagate) is what USD was built to do. In both cases, the map service reads from the storage medium and serves map nodes that reference glTF files to the browser through RMAP.

24. Existing 3D Applications and the Metaverse

The most common question about the open metaverse browser is some variation of, "I have a Unity or Unreal app. How do I bring it into the metaverse?" This question comes from game developers, enterprise simulation teams, architectural visualization studios, and training content creators — anyone who has invested years building 3D applications on existing engines. The question is reasonable, and the answer requires an honest assessment about what transfers, what must be rebuilt, and why.

24.1 History: How We Got Here

Unity and Unreal Engine emerged as general-purpose 3D application platforms. They provide complete, vertically integrated runtime environments: their own rendering pipelines, their own physics engines, their own audio systems, their own scripting runtimes, their own scene graphs, their own asset pipelines, and their own platform abstraction layers. An application built in Unity or Unreal is compiled into a monolithic executable that includes the engine itself. The app and the engine are inseparable.

This model made sense for standalone applications (games, simulations, training programs) where one application owns the entire screen and all of the hardware resources. It is the same model that native desktop and mobile apps follow: the developer ships everything, the app runs in isolation, and the platform provides only basic services (windowing, input, file access).

The web disrupted this model for documents and 2D applications. The browser became the runtime, and content was expressed in standard formats (HTML, CSS, JavaScript) that any

browser could execute. Developers stopped shipping their own rendering engines. They shipped content, and the browser handled the rest.

The metaverse browser applies the same disruption to 3D spatial content.

24.2 The Fundamental Incompatibility

A Unity or Unreal application is a self-contained runtime. The MBE is also a runtime. You cannot nest one runtime inside another any more than you can run a native iOS app inside a web browser. The conflict is structural:

Property	Game Engine App	MBE Service
Rendering	Engine's own pipeline (Unity URP/HDRP, Unreal Nanite/Lumen)	ANARI + rendering engine (backend-agnostic)
Scene graph	Engine's internal scene graph	SOM (shared, multi-source)
Physics	Engine's physics (PhysX, Havok, Chaos)	Server-side services (client-side GPU compute is an open gap — Section 23.4)
Scripting	C# (Unity), C++/Blueprints (Unreal)	WASM (compiled from any language)
Audio	Engine's audio system (FMOD, Wwise integration)	MBE spatial audio
Networking	Custom or engine-specific (Netcode, Replication)	RMAP
Platform abstraction	Engine handles per-platform builds	MBE + OpenXR + ANARI
Isolation	App owns entire screen and all resources	Service runs sandboxed alongside dozens of others

The last row is the critical one. A game engine app assumes it is the only thing running. A metaverse service is one of dozens sharing the SOM, the GPU, and the user's attention. These are incompatible operating models.

24.3 What Can Be Preserved

Despite the runtime incompatibility, a substantial portion of an existing 3D application's investment transfers directly:

- **3D assets (geometry, textures, materials, animations).** Both Unity and Unreal export to glTF, which is the MBE's content format. Models, skeletal animations, PBR materials, and

textures move across with standard export tools. This is the largest category of reusable work, often representing the majority of a project's budget.

- **Scene structure via USD.** Both engines increasingly support USD export. A scene exported as USD can serve as the source data for a map service, which reads the USD, converts to map nodes that reference glTF files, and serves the content through RMAP. The geometry, hierarchy, transforms, and spatial relationships are preserved. For architectural visualization, urban planning, and facility modeling, this path is particularly powerful. An entire building modeled in Revit, exported through USD, can be served as a spatial fabric with minimal additional work beyond standing up the map service.
- **Service logic (with refactoring).** Unreal's C++ compiles to WASM. Unity's C# compiles to WASM (via NativeAOT or Blazor). The application's business logic (inventory management, training sequences, data visualization, interaction rules) can be recompiled as WASM service modules. However, any code that calls engine-specific APIs (rendering commands, physics queries, engine UI) must be refactored to use the MBE's host function API instead. The logic is preservable; the engine coupling is not.

24.4 What Cannot Be Directly Preserved

Components tightly coupled to the engine's runtime do not survive the transition. These are the areas where existing work must be re-authored or fundamentally rethought for the MBE's architecture:

- **Engine-specific rendering.** Unity's Universal Render Pipeline, Unreal's Nanite and Lumen, custom post-processing effects, engine-specific shader graphs — none of these transfer. The MBE renders through ANARI and its rendering engine, and materials are expressed as glTF PBR parameters or, for custom effects, SPIR-V shaders. Visual effects must be re-authored for the MBE's rendering model. In practice, PBR materials transfer well through glTF; custom shader effects require re-implementation.
- **Engine-specific physics and simulation.** PhysX configurations, Chaos destruction setups, cloth simulation parameters — these are tied to their respective engines. Physics in the MBE runs primarily as server-side services. Client-side GPU compute for physics is a known architectural gap (Section 23.4). Simple physics (collision, rigid body) can be re-implemented server-side or within WASM; complex engine-specific simulations require rethinking.
- **Engine-specific networking.** Unity's Netcode for GameObjects and Unreal's Replication system are replaced entirely by RMAP. The networking model is fundamentally different: RMAP provides model-based access to server-maintained objects, not engine-level state replication. Backend services must be redesigned around RMAP's model access pattern.
- **The "app owns the screen" assumption.** Any design that assumes the application controls the entire viewport, all input, and all audio must be reconceived. In the metaverse, the content is a service contributing to a shared scene, not an application monopolizing a screen.

24.5 The USD Path: Easiest On-Ramp

For content that is primarily spatial (architectural visualization, urban models, facility layouts, product showrooms, training environments) the USD export path offers the most direct route into the metaverse:

- Author the environment in any tool that supports USD (Revit, Blender, Maya, Houdini, Omniverse, Unreal, Unity).
- Export the scene as USD.
- Host it through a map service that reads the USD data, converts it to map nodes that reference glTF files, and serves them through RMAP. The map service and its content form the foundation of a spatial fabric.
- Add services as needed (interactive elements, data overlays, IoT feeds) as independent WASM modules communicating through RMAP.

This path preserves the spatial investment almost entirely. The environment looks the same. What changes is how interactivity is delivered. Instead of monolithic engine logic, behavior comes from composable services.

24.6 Enterprise First, Gaming Later

The near-term adoption path for existing 3D content is likely enterprise-led, not gaming-led. Enterprise 3D applications (facility management, digital twins, architectural walkthroughs, training simulations, product configurators) are typically asset-heavy and logic-light. Their value is in the spatial data (the building model, the factory layout, the equipment geometry). This content transfers cleanly through the USD and glTF paths described above.

Gaming applications, by contrast, tend to be logic-heavy and deeply coupled to their engine's runtime capabilities. A complex game with custom physics, AI systems, networked multiplayer, and engine-specific visual effects requires substantially more work to decompose into the MBE's service model. This is not a permanent barrier. As the MBE ecosystem matures, tooling will emerge to automate parts of the migration, and new games will be built natively for the metaverse browser from the start. But the honest near-term picture is that enterprise content migrates more easily than gaming content.

24.7 The Web Parallel

The web faced this exact transition. When web browsers became the dominant platform, developers did not "import" their Flash applications, Java applets, or native desktop apps into the browser. They rebuilt them using web technologies. The assets (images, text, data) were reusable, but the runtime was different. Flash developers learned JavaScript. Native UI developers learned CSS. The transition was real work, but the result was content that ran everywhere, on any device, with no installation.

The same transition is ahead for 3D content: the assets are transferred; the logic is recompiled; the engine-as-runtime model gives way to the browser-as-runtime model. Developers who build for the open metaverse will reach every device and every user, just as web developers reach every browser. The investment required is real, but the alternative — a fragmented landscape of incompatible proprietary engines, each requiring its own app store and its own installation — is a world the metaverse browser will complement.

24.8 Future Possibilities

Several developments could ease the transition further:

- **Automated migration tooling.** Community or vendor-built tools that analyze a Unity or Unreal project and generate MBE-compatible output: glTF assets, WASM service stubs, RMAP backend scaffolding. This does not eliminate refactoring, but it accelerates it significantly.
- **Engine-as-service adapters.** A Unity or Unreal application could potentially run server-side as a service backend, with a thin RMAP adapter translating its output into model objects that the MBE renders. The engine runs on the server; the browser renders the result. This sacrifices client-side performance for compatibility and is most viable for enterprise applications where server infrastructure is available.
- **Native MBE development in existing tools.** As the MBE's content formats and service APIs stabilize, authoring tools (including Unity and Unreal) could add native MBE export targets producing glTF, SPIR-V, and WASM directly, without the intermediate step of building a standalone engine app. This is the long-term convergence point — the same tools, targeting a different runtime.
- **Progressive complexity.** Early metaverse services will be simpler than today's AAA games. As the ecosystem matures (better tooling, more capable WASM runtimes, richer host function surfaces, community-shared libraries) the complexity ceiling rises. What is difficult today becomes routine tomorrow, following the same trajectory that took web development from hand-written HTML to React and Three.js.
- **AI-assisted migration.** The most transformative possibility. An AI system that understands both the source engine's architecture and the MBE's service model could analyze an existing Unity or Unreal project (scene hierarchy, scripting logic, material setup, physics configuration, and networking model) and generate a working MBE migration (refactored WASM service modules, RMAP backend scaffolding, glTF PBR materials, and a restructured scene exported through USD or glTF). The developer reviews, adjusts, and ships rather than rebuilding from scratch. This is not speculative — AI code migration across languages and frameworks is already demonstrating viability, and the structured nature of game engine projects (typed APIs, well-defined asset pipelines, documented patterns) makes them particularly amenable to automated analysis. AI does not eliminate the need to understand the MBE's architecture, but it could compress what is currently months of migration work into days.

The transition from proprietary engines to an open browser runtime is not instantaneous. But the direction is the same one the web established — open standards, universal reach, and content that outlives any single platform.

25. Further Considerations

The following topics are important concerns for a shipping metaverse browser that do not yet have definitive architectural solutions. They are documented here to ensure they are not overlooked and to invite community input.

25.1 Permissions Model for Sensor Data

The web gates camera, microphone, and location behind permission prompts. The MBE must gate access to significantly more sensitive data:

Data Type	Sensitivity	Risk
Eye tracking	Extreme	Gaze patterns reveal medical conditions, cognitive load, attention, and purchasing intent
Hand/body tracking	High	Full skeletal tracking reveals physical characteristics, disabilities, emotional state
Room geometry	High	The physical shape and contents of the user's real-world space
Camera passthrough	Extreme	Raw camera feed of the user's surroundings
GPS/VPS location	High	Precise physical location

Services should never receive raw sensor data by default. The browser must mediate: services request capabilities, the user grants or denies, and the browser provides only the minimum necessary data (a service might need "user is looking at object X" without receiving the raw gaze vector).

Eye tracking deserves particular attention. Gaze behavior is individually distinctive (like a fingerprint), and users are often unaware of what gaze patterns reveal.

25.2 Content Trust and Code Signing

When the MBE downloads and executes a WASM module from a service, it must verify the module's integrity and provenance:

- **Code signing.** Services sign their WASM modules; the browser verifies signatures before execution.

- **Content hashing.** The service registry publishes expected hashes; the browser verifies downloads match.
- **Chain of trust.** A service's signing key links to its identity credential, establishing a verifiable chain from code to publisher.
- **Revocation.** If a service is found malicious, its signing key is revoked and all browsers refuse its modules.

25.3 Level of Detail and Streaming

Every enterprise use case the architecture describes (factory floors, warehouses, airports, hospital campuses, city streets) contains far more geometry than any device can render simultaneously at full fidelity. The browser needs hierarchical level of detail (LOD) — coarse representations at distance, progressively finer detail near the viewer, with smooth transitions between levels. It also needs streaming — content that arrives incrementally as the user moves, with spatial indexing so the browser knows what to request without downloading everything first.

For the rendering engine to handle LOD selection (choosing which level to display), the content itself must be structured to support multiple levels. The architecture currently specifies glTF as the delivery format but does not specify how glTF content carries LOD hierarchies or how the browser requests progressive detail from a map service.

A promising development is underway within the Khronos 3D Formats working group. OGC 3D Tiles, the leading standard for streaming massive 3D datasets, was built on top of glTF and added spatial indexing, hierarchical LOD, and view-dependent refinement capabilities that glTF alone does not provide. The 3D Tiles technology was originally designed for planet-to-city-scale geospatial data but proved applicable at any scale. The Cesium team has offered to merge this LOD and streaming technology back into the glTF standard. Work is actively underway on several glTF extensions:

- EXT_node_lod defines LOD at the node level with explicit screen-coverage thresholds.
- The glTF External Reference Format enables lazy loading where only immediately necessary scene parts load based on spatial proximity.
- KHR_meshopt_compression (merged January 2026) provides high-speed geometry compression at approximately 1 GB/sec decompression.

A new glTF version incorporating these capabilities is expected in 2026. If this convergence delivers on its trajectory, the MBE's content format will natively carry LOD hierarchies, and the browser's glTF loader will gain view-dependent refinement without requiring a separate streaming protocol. Map services would serve glTF files (with LOD built in) through RMAP, and the browser would handle progressive refinement internally.

Open questions remain. Whether glTF will absorb 3D Tiles' spatial indexing capabilities (knowing what content exists where across a large area, before downloading any of it) or whether that remains a map-service-side concern is not yet clear. The OMBI is engaging with the Khronos 3D

Formats working group and the 3D Tiles community to track this convergence and ensure the MBE's content pipeline aligns with the emerging standard.

25.4 Cookies and Persistent Storage

The web browser's cookie model is widely regarded as a failed design: opaque key-value pairs with inconsistent scoping, third-party tracking abuse, and a consent regime that annoys users without protecting them.

The MBE needs a persistent storage model designed from scratch:

- Per-persona storage for preferences, bookmarks, and avatar customizations that travel with the user across devices.
- Per-service storage for a service's local state for this user (game progress, settings, shopping cart).
- Scoped and sandboxed so a service can only access its own storage, never another's.
- No third-party tracking equivalent. No mechanism for services to correlate storage across fabrics.
- Structured data. Typed, queryable data rather than string-only key-value pairs.

25.5 Caching

3D asset caching differs fundamentally from HTTP resource caching:

- **Spatial locality.** Cache assets near where the user is or is likely to go. Pre-fetch geometry for the next block ahead.
- **Size variance.** Assets range from kilobytes (parametric avatars) to megabytes (building geometry, textures). Eviction policies must account for this.
- **Freshness models.** A building's geometry may not change for months; a live event changes every second.
- **Shared assets.** Common skeletons, base textures, and standard materials cached once and shared across services and fabrics.
- **WASM modules.** Compiled code cached across sessions; recompilation on every visit is wasteful.

25.6 Navigation Model

The web has URLs, bookmarks, back/forward, and tabs. The metaverse needs equivalents:

- **Spatial addressing.** A universal scheme for identifying a location across fabrics. What is the "URL" of a place?
- **Bookmarks.** Save and return to specific locations.
- **Sharing.** Send someone a link to "where I am right now."

- **History.** Places visited, with spatial context.
- **VR teleportation.** The navigation model for VR users who are not physically moving through the world.

25.7 Accessibility

How users with disabilities interact with a 3D spatial environment is an unsolved problem industry-wide:

- **Visual impairment.** Screen readers for 3D space, audio descriptions of environments, haptic navigation aids.
- **Hearing impairment.** Visual indicators for spatial audio cues (directional subtitles, visual alerts for sounds).
- **Motor impairment.** Alternative input methods (gaze-based interaction, voice control, switch access) when hand tracking is the primary model.
- **Cognitive accessibility.** Simplified environments, reduced sensory load, clear wayfinding.

The web has WCAG and ARIA. The metaverse has no equivalent standard. This is both a moral obligation and an anticipated legal requirement (ADA, EU European Accessibility Act). The MBE should be designed with accessibility hooks from the beginning rather than retrofitted.

The browser does not have to solve this alone. Because the MBE is built on open standards with a service-based architecture, accessibility solutions can come from anywhere. A third-party service can provide real-time audio descriptions of the environment for visually impaired users. Another can overlay directional subtitles for the hearing impaired. A specialized input service can translate voice commands or switch inputs into spatial interactions for users with motor impairments. The open architecture means that accessibility innovation is not bottlenecked by the browser team — anyone can build and distribute services that make the metaverse more accessible, and those services reach every user on every conformant browser.

25.8 Internationalization

The metaverse is global from day one. Text rendered in 3D space, on UI panels, and within service content must support the full range of human writing systems: Latin, CJK (Chinese, Japanese, Korean), Arabic and Hebrew (right-to-left), Devanagari, Thai, and every other script in common use. This affects:

- The UI toolkit (Section 15)
- Avatar name display
- Service content
- Any text the browser itself renders

Beyond rendering, translation is a significant opportunity. Every service in the metaverse presents text (labels, instructions, menus, descriptions) to users who may speak any language. Several approaches could make the metaverse natively multilingual in a way the web has never achieved:

- **Crowd-sourced translation service.** A dedicated translation service, accessible through RMAP, maintains a database of human-verified translations. AI generates the initial translation for new content, but the authoritative result is reviewed and refined by human translators — native speakers who correct nuance, idiom, and context that AI misses. The database grows over time as more content is translated and verified. Services query the translation model for their text in the user's preferred language. This produces the highest-quality translations and improves continuously as the community contributes.
- **Real-time AI translation.** The browser translates service content on the fly using AI inference, either locally on the device or through a server-side AI service. Content arrives in the author's language and is translated before display. This provides immediate coverage for any language pair without waiting for human review, but with lower accuracy for nuanced or domain-specific content. Useful as a fallback when the crowd-sourced database has no entry for a given string.

25.9 Developer Tools

If the MBE is to attract a developer ecosystem, it needs tooling on par with web browser developer tools:

- **SOM Inspector.** Browse the scene graph hierarchy, inspect node properties, visualize branch ownership (analogous to the DOM inspector).
- **Service Monitor.** View running WASM services, their CPU and memory usage, fuel consumption, and execution state.
- **Network Monitor.** Inspect fabric connections, streaming bandwidth, object update rates, latency.
- **Performance Profiler.** Frame timing breakdown: time in WASM execution, ANARI rendering, internal GPU compute, OpenXR submission.
- **Shader Debugger.** Inspect SPIR-V shaders in use, material parameters, per-object rendering output.
- **SOM Library.** SDK for services to access and interface with the SOM through performant and securable classes.

25.10 Versioning and Backward Compatibility

As the MBE standard evolves (new host functions, expanded SOM capabilities, richer UI toolkit APIs, RMAP protocol revisions) older services and fabrics must continue to work. The browser cannot assume every deployed service targets the latest specification. Services activate automatically through proximity; there is no opportunity to prompt the user to "update" a service

they never installed. A service deployed today must still function when the browser updates next year.

The versioning strategy must address:

- **Host function API versioning.** Services declare which API version they target. The browser supports multiple versions simultaneously or provides compatibility shims.
- **RMAP protocol versioning.** As the service connectivity protocol evolves, older backends must remain reachable.
- **Graceful degradation.** A service targeting a newer API than the browser supports should fail cleanly — not crash, corrupt the SOM, or degrade other services.
- **Deprecation lifecycle.** A defined process for retiring old API versions, with sufficient lead time for service providers to update.

The web's experience here is instructive. Backward compatibility has been both the web's greatest strength (a page from 1998 still renders) and its heaviest burden (every browser carries decades of legacy behavior). The MBE should learn from both sides of that lesson.

Appendices

A. Standards Evaluation

This appendix provides the detailed evaluation of metaverse-related standards that informed the architectural decisions in this document. The assessment was performed in the context of building a cross-platform native metaverse browser that maintains simultaneous connections to multiple spatial fabrics and services, blending output into a single Scene Object Model.

A.1 MBE Core Abstractions

Standard	Role in MBE	Domain
ANARI	Rendering abstraction. Thin, high-level scene-graph API where the application works with abstract objects (meshes, materials, lights, cameras) and ANARI delegates to an interchangeable rendering engine, which translates scene descriptions into GPU draw calls.	Rendering
SPIR-V	GPU shader interchange. Portable bytecode for custom visual effects. Cross-compiled at runtime to the local GPU's native shader language. Also used internally by the browser for GPU compute workloads.	Shaders / Compute
OpenXR	XR device abstraction. Unified API for headset discovery, tracking, input, rendering submission, and composition.	Device I/O
WASM	Content execution runtime. Sandboxed execution of third-party service logic with per-service memory isolation.	Content Execution
RMAP	Model access protocol. Service driver layer that provides standardized access to server-maintained object models, decoupling the browser from underlying service protocols and wire formats.	Networking

A.2 Content-Provider Ecosystem (Above the MBE)

Tool / Standard	What It Produces	Format Consumed by MBE
Slang, GLSL, HLSL	Custom shaders and compute kernels	SPIR-V
Blender, Maya, Houdini, Revit	3D models, animations, scenes	glTF

Tool / Standard	What It Produces	Format Consumed by MBE
Substance 3D, Quixel Mixer	Materials, textures	KTX textures, PBR materials in glTF
C++, Rust, AssemblyScript	Service logic	.wasm modules
MaterialX	Materials, shading networks	glTF PBR parameters, SPIR-V (custom effects)

A.3 Standards Eliminated

Standard	Category	Reason for Elimination
OpenGL	GPU graphics	Frozen since 2017. Apple deprecated it. Superseded by Vulkan on every platform.
OpenGL ES	Mobile GPU graphics	Superseded by Vulkan on modern mobile devices.
OpenGL SC	Safety-critical GPU	The MBE has no safety certification requirements.
Vulkan SC	Safety-critical GPU	Same as OpenGL SC.
WebGL	Browser GPU graphics	The MBE is a native application, not a web browser. GPU rendering through ANARI and native rendering engines.
WebGPU	Browser GPU (modern)	A WebGPU-based engine could serve as an ANARI backend, but WebGPU is not relevant as a browser-facing API.
COLLADA	3D asset interchange	Superseded by glTF and OpenUSD
SPIR	Shader IR (old)	Original IR for OpenCL. Fully superseded by SPIR-V.
OpenVG	2D vector graphics	MBE rendering is 3D scene-graph-based through ANARI. 2D UI handled by OpenXR composition layers.
OpenWF	Window compositing	Dormant/deprecated. Handled by platform compositors and OpenXR composition layers.
NNEF	Neural network format	ML model packaging is a separate concern from the rendering/compute/device stack.
EGL	Rendering context bridge	Fully abstracted by the architecture: OpenXR manages display surfaces in XR mode, rendering engines create their own GPU contexts.

Standard	Category	Reason for Elimination
OpenVX	Computer vision	Fully abstracted by OpenXR's extension system. AR capabilities provided through OpenXR extensions.

A.4 Standards Set Aside

These standards will be incorporated into the MBE but are content-handling concerns, not architectural decisions:

Standard	Category	Note
glTF	3D asset format	Incorporated like JPEG in a web browser
KTX	Texture format	Pairs with glTF for efficient GPU texture upload

B. Standards Landscape

The MBE's architecture draws primarily on Khronos Group standards for rendering, compute, shaders, and device I/O, with W3C and OGC standards for identity and geospatial positioning. These are the core partners, but the architecture depends on standards from several additional consortia — particularly in transport, indoor positioning, identity implementation, and building data. This appendix maps the broader standards ecosystem to the architecture's requirements.

B.1 Standards Organisations Referenced in This Document

Organisation	Standards Referenced	Architecture Sections
Khronos Group	ANARI, OpenXR, glTF, KTX, SPIR-V	Throughout — core runtime and content standards
W3C	DIDs v1.1, VCs v2.0, FedCM	Section 18 (identity)
OGC	GeoPose 1.0/1.x, IndoorGML, CityGML, 3D Tiles	Section 17 (positioning), Section 23.5 (map services), Appendices B.3, B.5
ISO	ARF (ISO/IEC 23090-39), E57 (ASTM E2807)	Section 19.10 (avatars), Section 17.5 (scan data)

B.2 Transport Protocols — IETF

The architecture describes extensive real-time networked communication (RMAP, avatar position streaming, spatial audio, fabric streaming) but deliberately abstracts away the transport layer

through RMAP's Source adapter model (Section 13). The underlying transport protocols are the domain of the Internet Engineering Task Force (IETF):

Standard	Relevance
QUIC (RFC 9000)	UDP-based transport with built-in encryption, multiplexing, and connection migration. QUIC's stream multiplexing maps naturally to the MBE's simultaneous connections to dozens of fabrics and services. A strong candidate for RMAP Source adapter implementations.
WebTransport	Low-latency, bidirectional communication over QUIC with both reliable streams and unreliable datagrams. The unreliable datagram mode suits ephemeral data like avatar positions, where the latest state matters more than guaranteed delivery.
HTTP/3	HTTP over QUIC. Relevant for content delivery (glTF assets, KTX textures, WASM modules) where reliable delivery is required but low latency is desirable.
TLS 1.3 (RFC 8446)	Encryption for all network communication. Implicit in the architecture but worth making explicit as a baseline requirement.

RMAP does not mandate a transport protocol — each service's Source adapter brings whatever wire protocol that service requires (Section 13). IETF protocols are the natural candidates for Source adapter implementations and should be acknowledged as a key standards dependency. The RMAP protocol specification, when formalised, will need to specify recommended transport bindings.

B.3 Positioning — IEEE, Bluetooth SIG, FiRa, OGC

The architecture identifies indoor positioning as critical for enterprise use cases (Section 17.1) and proposes a positioning abstraction layer (Section 17.8), but the discussion of positioning technologies focuses on GPS and camera-based VPS. Radio-based indoor positioning standards (already deployed in enterprise environments) and OGC standards for positioning output formats and indoor spatial data are equally relevant.

Organisation	Standard	Relevance
IEEE	802.11az (WiFi Fine Time Measurement)	Centimeter-level indoor positioning using WiFi round-trip time measurement. Ratified 2022. Available on recent WiFi chipsets. Directly relevant to factory, warehouse, and hospital environments where WiFi infrastructure already exists.
IEEE	802.15.4z (Enhanced UWB ranging)	Ultra-wideband ranging for centimeter to sub-centimeter indoor positioning. The technology behind Apple's U1/U2 chips and Samsung's UWB

Organisation	Standard	Relevance
		implementations. Already in smartphones and enterprise asset tracking.
Bluetooth SIG	Bluetooth 5.1+ Direction Finding	Angle-of-arrival and angle-of-departure for indoor positioning using BLE beacons. Lower precision than UWB (sub-meter vs. sub-centimeter) but vastly larger installed base. Many enterprise indoor positioning systems are BLE-based.
Bluetooth SIG	Bluetooth Channel Sounding (6.0)	Secure distance measurement with sub-centimeter ranging and security against relay attacks.
FiRa Consortium	FiRa UWB specifications	Interoperability standards for UWB ranging and positioning across device manufacturers (Apple, Samsung, NXP). Ensures cross-vendor UWB positioning.
OGC	GeoPose 1.0 / 1.x	Standard encoding for real-world position and orientation (6DOF) relative to a geographic reference frame. Already referenced in Section 17.5. The 1.x work packages are converging with the positioning abstraction layer proposed in Section 17.8: Work Package 5 adds picosecond-resolution temporal coordinates, Work Package 7 adds confidence and uncertainty metadata, and Work Package 8 adds VPS support developed in collaboration with the MSF.
OGC	IndoorGML	Standard representation for indoor spatial environments: connectivity between rooms, corridors, and floors, navigation networks, and indoor-to-outdoor coordinate transitions. The enterprise use cases this architecture targets (factories, warehouses, hospitals, airports) are indoor. IndoorGML provides the spatial structure within which indoor positioning systems (UWB, BLE, WiFi RTT) operate and services are anchored. A natural source format for indoor map services.

The positioning abstraction layer proposed in Section 17.8 must accommodate WiFi RTT, UWB, and BLE positioning backends alongside GPS and camera VPS.

B.4 Identity Implementation — DIF, FIDO Alliance

Section 18 describes the identity architecture using DIDs, VCs, and zero-knowledge proofs. The organisations building the implementation layer — the protocols and infrastructure that make DIDs and VCs practically deployable — are additional engagement targets:

Organisation	Standards	Relevance
Decentralized Identity Foundation (DIF)	DIDComm Messaging, Presentation Exchange, Well Known DID Configuration	DIF develops the protocols around W3C DIDs and VCs. DIDComm provides secure, transport-agnostic communication between DID-identified parties — directly relevant to browser–identity service provider communication. Presentation Exchange standardises how services request and receive verifiable credentials, mapping to the service encounter flow in Section 18.3.
FIDO Alliance	FIDO2, WebAuthn, Passkeys	Passwordless authentication using public-key cryptography. The architecture's root identity (Section 18.4) is a cryptographic anchor; FIDO's authentication infrastructure is the practical implementation layer — how a user authenticates to their identity service provider. The passkey model aligns with the architecture's goal of making identity creation as simple as creating any other account.
OpenID Foundation	OpenID Connect, OpenID Federation 1.0	Already referenced in Section 18.7 for federation protocols. Given the architecture's proposed hybrid model (DIDs + VCs for identity/credentials, OpenID Federation for trust between providers), the OpenID Foundation is an identity standards partner alongside W3C.

B.5 Built Environment — buildingSMART, OGC

The architecture's enterprise use cases centre on buildings and cities — factories, warehouses, hospitals, airports, retail stores, and urban-scale outdoor environments. These already have digital representations in Building Information Modelling (BIM) systems and geospatial databases. Several open standards are relevant as source formats for map services.

Organisation	Standards	Relevance
buildingSMART International	IFC (Industry Foundation Classes)	Open standard for BIM data. Contains building geometry, spatial structure, materials, mechanical/electrical systems, and rich semantic metadata. The practical path from physical building to spatial fabric is: BIM/IFC data → conversion to USD or glTF (via IfcOpenShell, xBIM, or Autodesk's export tools) → served through a RMAP map service. buildingSMART's openCDE API provides standards-based access to BIM data in cloud environments. For enterprise spatial fabrics, IFC/BIM is a major source format alongside USD and glTF.

Organisation	Standards	Relevance
OGC	CityGML / CityJSON	Semantic 3D city models: buildings, terrain, vegetation, water bodies, transportation, and city furniture with rich semantic information (building function, address, floor count, roof type). Numerous cities publish CityGML datasets (Berlin, Helsinki, Singapore, New York). Like USD and IFC, CityGML is a source format that converts to glTF for browser consumption. The semantic richness could inform how services attach to locations — not just to coordinates, but to semantically identified structures.
OGC	3D Tiles	Streaming format for massive heterogeneous 3D geospatial datasets (cities, terrain, buildings, point clouds, photogrammetry). Originally developed by Cesium, now an OGC community standard. Provides spatial indexing, hierarchical level-of-detail, and view-dependent refinement. Significant existing content ecosystem (Cesium ion, Google Maps 3D, ESRI ArcGIS, municipal datasets). Uses glTF as its tile payload format, aligning directly with the MBE's content format. A candidate storage and streaming format for outdoor urban map services alongside USD.

B.6 USD Governance — Alliance for OpenUSD (AOUSD)

The Alliance for OpenUSD (AOUSD) governs USD's evolution. Given USD's role in the map service architecture (Section 23.5) and the content pipeline, AOUSD decisions directly affect the MBE ecosystem. Members include Pixar, Apple, Autodesk, Adobe, and NVIDIA. Their technical working groups cover camera, geometry, materials, and the USD core — all relevant to how USD content flows into spatial fabrics.

B.7 Other Organisations

Organisation	Relevance	Priority
3GPP	5G network slicing for low-latency fabric streaming; 5G positioning services (sub-meter outdoor)	Track — relevant as 5G edge computing matures
Bytecode Alliance	Maintains Wasmtime, WASI, and the Component Model — the core WASM runtime and system interface the MBE embeds for content execution (Section 12).	Engage
Open AR Cloud (OARC)	OSCP and SpatialDDS protocols (referenced in Section 17.5). Decentralised	Engage

Organisation	Relevance	Priority
	spatial infrastructure vision aligns closely with the MBE's architectural principles.	
SMPTE	Media transport (ST 2110) for video-rich spatial fabrics; immersive video standards	Track — not a priority for initial architecture
Spatial Web Foundation	HSTP, HSML spatial web protocols	Track — early stage, unclear adoption trajectory

Glossary

Term	Definition
ANARI	Analytic Rendering Interface for Data Visualization. A Khronos Group standard providing a thin, high-level, scene-description API that delegates rendering to an interchangeable rendering engine.
AOUSD	Alliance for OpenUSD. The organisation governing USD's evolution. Members include Pixar, Apple, Autodesk, Adobe, and NVIDIA.
A/V Zone	A designated area within a spatial fabric where audio and video behavior differs from open space (e.g., auditorium mode, cone of silence).
Compute shader	A GPU program for general-purpose parallel data processing, as distinct from graphics shaders that produce visual output. The browser uses compute shaders internally for workloads such as spatial audio mixing, IK, and physics. Service-submitted GPU compute is a known architectural gap (Section 23.4).
Compute dispatch	The act of submitting a SPIR-V compute program and associated data buffers for GPU execution.
DID	Decentralized Identifier. W3C standard for globally unique, self-sovereign identifiers.
DIF	Decentralized Identity Foundation. Develops the protocols and reference implementations around W3C DIDs and VCs (DIDComm, Presentation Exchange).
Fabric	See Spatial fabric.
Fabric-level service	A service operated by the fabric owner as part of the fabric's overall offering, not tied to a specific map location (e.g., presence, commerce).
Frame budget	The maximum time available per frame. At 90fps, the frame budget is 11.1 milliseconds.
FIDO	Fast Identity Online. Alliance developing passwordless authentication standards (FIDO2, WebAuthn, Passkeys) using public-key cryptography.
glTF	GL Transmission Format. Khronos Group standard for 3D asset interchange.
glTF PBR	glTF's native Physically Based Rendering material model. Defines standard surface properties (base color, metallic, roughness, normal,

Term	Definition
	emissive) as data parameters on SOM objects. The vast majority of materials in the metaverse are expressed as glTF PBR parameters, flowing directly through ANARI to the rendering engine without shader compilation.
GPS	Global Positioning System. Provides coarse (meter-level) outdoor positioning.
IFC	Industry Foundation Classes. Open standard for Building Information Modelling (BIM) data, governed by buildingSMART International.
HRTF	Head-Related Transfer Function. Transforms audio to simulate spatial positioning relative to the listener's ears.
IK	Inverse Kinematics. Computing full body pose from sparse tracking input (head, hands).
Khronos Group	An open, non-profit, member-driven consortium of over 150 organizations creating royalty-free interoperability standards for 3D graphics, augmented and virtual reality, parallel computing, and related technologies. Responsible for ANARI, SPIR-V, OpenXR, glTF, KTX, and other standards referenced in this document.
KTX	Khronos Texture format. GPU-optimized texture container.
LOD	Level of Detail. Rendering objects at different fidelity based on distance and device capability.
Map-anchored service	A service attached to a specific location in a fabric's map (e.g., equipment dashboard, transit board).
Map service	The required component of a spatial fabric that provides spatial data (geometry, terrain, coordinate system).
MBE	Metaverse browser engine. The core engine of a metaverse browser, and the subject of this document. Analogous to a web browser's rendering engine.
Metaverse browser	A native, cross-platform application that brings the browsing paradigm to three-dimensional space. Comprises the MBE (the engine) and the application shell (user interface).
Metaverse browser engine	See MBE.

Term	Definition
Metaverse Standards Forum	An open forum for cooperation between standards organizations and companies to foster interoperability standards for the metaverse. The OMBI operates within the Metaverse Standards Forum.
OMBI	Open Metaverse Browser Initiative. The group formed within the Metaverse Standards Forum to oversee the specification of the MBE and the broader open metaverse browser endeavor.
Open metaverse browser	The overarching endeavor to create a metaverse browser built entirely on open standards, implementable by anyone, owned by no one. Not a product name; a description of the goal.
OpenXR	A Khronos Group standard providing a unified API for XR device interaction.
QUIC	IETF transport protocol (RFC 9000). UDP-based with built-in encryption, multiplexing, and connection migration.
Overlay	A spatial fabric that superimposes content across the entire scene rather than anchoring to specific locations.
Persona	A public-facing identity presentation used by the browser when interacting with fabrics and services. One user may have multiple personas.
Rendering engine	The software layer between ANARI and the GPU API (Vulkan, Metal, DX12) that translates high-level scene descriptions into low-level draw calls. Examples: OSPRay, VisRTX, VisGL.
RMAP	Remote Model Access Protocol. The MBE's model access protocol for networked services, analogous to ODBC for databases. Developed by Metaversal Corporation; proposed for adoption as an open standard. Defines the model interface (class definitions, properties, events, actions) while leaving wire format and transport to the service provider.
Root identity	The cryptographic anchor tying a user to a verified real human. Never revealed to fabrics or services.
SOM	Scene Object Model. The MBE's central interface; a multi-source, hierarchical, streaming scene graph with ownership semantics and access-controlled APIs. Every subsystem in the browser interacts with the scene through the SOM.
Sneeze	The OMBI's reference MBE implementation, named as a playful parallel to Blink (Chromium's engine). The architecture described in this document is being built as Sneeze.

Term	Definition
Spatial fabric	A collection of services organized around a shared spatial context. The fundamental unit of the metaverse.
SPIR-V	Standard Portable Intermediate Representation. Khronos Group binary format for portable shader bytecode. Also used internally by the browser for GPU compute workloads.
VC	Verifiable Credential. W3C standard for cryptographically signed claims.
VPS	Visual Positioning System. Provides precise (centimeter-level) positioning using camera-based environment recognition.
UWB	Ultra-Wideband. Radio technology for centimeter to sub-centimeter indoor positioning and ranging.
WASM	WebAssembly. Binary instruction format for sandboxed execution of compiled code.
WASI	WebAssembly System Interface. Standardized way for WASM modules to interact with the host system.
WebTransport	IETF protocol providing low-latency, bidirectional communication over QUIC with both reliable streams and unreliable datagrams.