Agentic RTB Framework

A specification for using agent-driven containers in OpenRTB and Digital Advertising

Version 1.0

Released November 12, 2025

Please email support@iabtechlab.com for public comments and questions. This document is available online at https://iabtechlab.com/standards/artf/

About this document

The Agentic RTB Framework specification defines a foundation for implementing agent services which operate within a host platform and that the orchestrating platform can call directly to accomplish a shared goal. The model leverages containers which are deployed into the infrastructure of a host to enable delegation of critical aspects of bidstream processing to service agents in a consistent manner, with minimal cost, latency and operational impacts. The framework enables this by establishing standard requirements for container runtime behavior and by defining an API which enables reliable, protected and private bidstream mutation.

With this approach, service providers package their offering once and deploy it to any standard-compliant platform, which means they are able to focus on their unique value proposition while offloading operational concerns and scaling to the host platforms. It also creates new possibilities for innovation because host platforms maintain control of data and SLAs and therefore can provide greater access to data and more interaction opportunities to service agents without concerns about leakage, misappropriation or latency.

The Agentic RTB Framework provides significant value to host platforms which are able to “drop-in” new capabilities with minimal integration overhead and compose bid processing pipelines configured specifically for their target use cases. The standard also enables platforms to adapt quickly to changing market demands by simply adding, updating and removing components as needed, all while maintaining control of operational costs and requirements and without incurring significant integration overhead and cost.

While this approach is agentic in nature, the primary focus here is on systematic agentic integration (service to service integration), but autonomic agentic functionality (model to service) is also envisioned as part of this specification as the integrating technology matures.

There are many use cases; for example identity resolution provided by an agent, deal or segmentation activation, fraud detection pre-impression etc. This list of use cases is not fully enumerated here, because part of the goal of this specification is to allow new use cases to be integrated into the bid stream via the agentic framework.

The specification aims to provide a general framework and best practices for deploying and operating these agents. It is not limited to a predefined set of use cases. Each use case is meant to be supported via an “intent” using the specification.

Additionally, the specification describes a standard interface with which containers can be managed using AI agents. This enables sub millisecond real-time bidding operations driven by agentic systems.

This document is primarily for technical audiences, in particular engineers and product managers wishing to implement products and features which can be solved with hosted containers and AI agents. The key takeaways for readers are:

- Understanding why to use containers and AI agents for certain use cases
- Learning what a standard container deployment involves including digital formats for describing the container, requirements, and functions
- Learning how to declare container capabilities and service definitions
- Understanding example use cases and workflows
- Recommendations of best practices for facilitating adoption across the industry

This document is developed by the IAB Tech Lab Container Project Task Force which is a subgroup of the Programmatic Supply Chain Working Group.

License

Significant Contributors

Joshua Prismon, *Index Exchange*; Joe Wilson, *Chalice*; Arpad Miklos, *The Trade Desk*; Brian May, *Individual*; Roni Gordon, *Index Exchange*; Ran Li, *Index Exchange*; Ben White, *OpenX*

IAB Tech Lab Leads

Miguel Morales, Director Addressability & Privacy Enhancing Technologies (PETs)
Shailley Singh, EVP Product & COO

About IAB Tech Lab

The IAB Technology Laboratory is a nonprofit research and development consortium charged with producing and helping companies implement global industry technical standards and solutions. The goal of the Tech Lab is to reduce friction associated with the digital advertising and marketing supply chain while contributing to the safe growth of an industry.

The IAB Tech Lab spearheads the development of technical standards, creates and maintains a code library to assist in rapid, cost-effective implementation of IAB standards, and establishes a test platform for companies to evaluate the compatibility of their technology solutions with IAB standards, which for 18 years have been the foundation for interoperability and profitable growth

in the digital advertising supply chain. Further details about the IAB Technology Lab can be found at https://iabtechlab.com.

Disclaimer

THE STANDARDS, THE SPECIFICATIONS, THE MEASUREMENT GUIDELINES, AND ANY OTHER MATERIALS OR SERVICES PROVIDED TO OR USED BY YOU HEREUNDER (THE "PRODUCTS AND SERVICES") ARE PROVIDED "AS IS" AND "AS AVAILABLE," AND IAB TECHNOLOGY LABORATORY, INC. ("TECH LAB") MAKES NO WARRANTY WITH RESPECT TO THE SAME AND HEREBY DISCLAIMS ANY AND ALL EXPRESS, IMPLIED, OR STATUTORY WARRANTIES, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AVAILABILITY, ERROR-FREE OR UNINTERRUPTED OPERATION, AND ANY WARRANTIES ARISING FROM A COURSE OF DEALING, COURSE OF PERFORMANCE, OR USAGE OF TRADE. TO THE EXTENT THAT TECH LAB MAY NOT AS A MATTER OF APPLICABLE LAW DISCLAIM ANY IMPLIED WARRANTY, THE SCOPE AND DURATION OF SUCH WARRANTY WILL BE THE MINIMUM PERMITTED UNDER SUCH LAW. THE PRODUCTS AND SERVICES DO NOT CONSTITUTE BUSINESS OR LEGAL ADVICE. TECH LAB DOES NOT WARRANT THAT THE PRODUCTS AND SERVICES PROVIDED TO OR USED BY YOU HEREUNDER SHALL CAUSE YOU AND/OR YOUR PRODUCTS OR SERVICES TO BE IN COMPLIANCE WITH ANY APPLICABLE LAWS, REGULATIONS, OR SELF-REGULATORY FRAMEWORKS, AND YOU ARE SOLELY RESPONSIBLE FOR COMPLIANCE WITH THE SAME, INCLUDING, BUT NOT LIMITED TO, DATA PROTECTION LAWS, SUCH AS THE PERSONAL INFORMATION PROTECTION AND ELECTRONIC DOCUMENTS ACT (CANADA), THE DATA PROTECTION DIRECTIVE (EU), THE E-PRIVACY DIRECTIVE (EU), THE GENERAL DATA PROTECTION REGULATION (EU), AND THE E-PRIVACY REGULATION (EU) AS AND WHEN THEY BECOME EFFECTIVE.

Glossary

Term	Description
Agent	A software service encapsulated in a container that performs a specific, autonomous function (e.g., fraud detection, deal curation) within the bidstream. Agents are designed to operate with explicit intents and within orchestrator-defined constraints.
Agent Manifest	A JSON structure embedded as metadata in a container image that defines an agent's capabilities, resource requirements, intents, and dependencies. Used by orchestrators to deploy and manage containers safely.
Agent Orchestrator	The control layer within a host platform that manages the deployment, execution, and coordination of agent containers participating in the bidstream.
Agent to Agent (A2A)	Direct communication or coordination between two autonomous agents, enabling them to exchange data or decisions without requiring mediation by a central orchestrator.
Agentic System	A distributed architecture where autonomous agents make decisions independently toward shared goals, such as optimizing auction outcomes, while maintaining defined constraints and interoperability.
Autonomous Agentic	Refers to systems or agents that make independent, AI-driven decisions and actions without direct orchestration, operating based on learned models or contextual goals.
Bidstream	The flow of bid requests and responses exchanged between programmatic advertising entities (SSPs, DSPs, exchanges). In ARTF, agents can modify bidstream data under orchestrator supervision.
Bidstream Mutation	A proposed change (addition, deletion, or update) to data within the bidstream, expressed via an OpenRTB Patch. Each mutation is atomic and tied to a specific intent.
Container	A standardized, lightweight, and portable execution environment that packages an agent's code, dependencies, and configuration into a single image. Containers conform to OCI runtime standards (e.g., Docker,

Term	Description
	Kubernetes).
Intent	A declarative statement describing why an agent proposes a mutation (e.g., “adjustBid,” “activateSegments,” “expireDeals”). Intents guide orchestrators’ decision-making about whether to accept or reject mutations.
Manifest (Container Manifest)	The metadata file (usually container.json) defining container resources, intents, dependencies, and health checks, enabling orchestrators to deploy the container consistently.
MCP (Model Context Protocol)	A protocol for structured model-to-agent communication using JSON-RPC. Complements gRPC and supports autonomic (AI-driven) agentic interactions, allowing models to orchestrate services directly.
Mutation	A single, atomic change proposed to a bid request or response. Includes fields such as intent, op (add/replace/remove), and path (semantic reference within OpenRTB payload).
Orchestrator (Orchestrating Entity)	The host platform (e.g., SSP, DSP, exchange) that manages the execution of agent containers, controls access, validates mutations, and decides whether to apply proposed changes.
Patch	A structured set of mutations proposed by an agent to modify an OpenRTB request or response. Each patch is atomic, traceable, and subject to orchestrator approval.
Sidecar	A secondary container that runs alongside a primary application container, providing auxiliary functions (e.g., monitoring, telemetry, or mediation).
Structural Agentic	Agentic systems where interactions are between structured services (service-to-service orchestration), rather than direct AI model control.
Telemetry	Monitoring data (metrics, traces, logs) emitted by containers to track performance, security, and decision impact across orchestrated systems.
User Cohorts /	Groupings of users with shared characteristics or behaviors that agents can

Term	Description
Audience Segments	activate or modify as part of audience segmentation use cases.

Table of Contents

Introduction

What is an agentic system?

The digital advertising ecosystem is built on a distributed system that allows for multi-party participation across a variety of business activities. This distributed system has common goals, but individual systems that are called by other partners to accomplish these goals. Each of these distributed systems also work to accomplish their own goals to accomplish their own mandates and make their own decisions to accomplish both the global goal (ie, a specific auction) as well as their own goals and mandates.

A **structurally agentic system** is a distributed architecture where autonomous components operate with explicit mandates, decision-making authority, and structured delegation patterns to accomplish domain-specific goals within defined constraints. Supply-Side Platforms (SSPs) and Demand-Side Platforms (DSPs) function as independent agents that orchestrate real-time auctions by delegating specialized tasks to other agents through standardized protocols such as OpenRTB. Each agent maintains internal state, optimizes toward distinct objectives (e.g., maximizing publisher yield vs. advertiser ROI), and makes autonomous decisions within strict latency budgets while the system's overall behavior emerges from the interaction of these specialized components rather than centralized control. This architectural pattern enables composability, fault isolation, independent optimization, and the ability to integrate new capabilities without modifying core auction logic, making it fundamentally different from monolithic systems where a single component possesses global knowledge and decision-making authority.

This standard introduces the ability to add new agents into existing orchestrations by introducing two new constructs - containers and OpenRTB patch. The focus of this document is to enable Agentic extensibility for the RTB process, but both the container and the OpenRTB patch are a generic agentic approach, and can be used for non real-time bidding use cases as well.

Requirements

For agentic extensibility to be adopted widely in the OpenRTB ecosystem, implementers need to adhere to a basic set of principles. These principles are:

1) **Agents must be able to participate in the core bidstream.** The bidstream may have many different points of integration with containers. This standard is focused on entities that are transacting via the bidstream in real-time, and is not a general-purpose standard for containerized services at the edge.
2) **Agents must accomplish a specific goal.** Each agent must declare specific intents for itself, and any changes that it wants to make to auctions moving through the bidstream. The orchestrating entity may then choose to accept the change or not to facilitate multi-party participation.

3) **Agents must be composable and deployable in standard enterprise infrastructure.** Agents must be structured as Containers and must adhere to the standard OCI runtime and image specifications (i.e., Docker Containers) so they are manageable via distributed cluster orchestration systems such as Kubernetes, Docker Compose / Swarm or cloud based systems like Amazon's Elastic Containers service.
4) **Agents must be performant and efficient**. Agents must communicate via a high performance protocol such as gRPC or in some cases MCP. Agents should be written in an efficient language and leverage an efficient ecosystem (for example, Rust, Go, or Java) or a highly optimized interpreted language. Agents should not consume resources they do not need. Orchestrating entities must provide guidance on expected response times to container providers.
5) **Agents must adhere to the policy of least-privilege and least-data. Agents will not have unfettered access to the outside world**, and must leverage appropriate services provided by the orchestrating entity. Orchestrating entities should implement appropriate measures to ensure containers adhere to this requirement.

What is a Container?

A container is a standard technology, originally popularized by Docker that provides an image-based, versioned, lightweight, and portable execution environment that encapsulates an application and all its dependencies—code, runtime, system tools, libraries, and settings—into a single package. This approach ensures that the containerized application behaves consistently across different computing environments, whether running locally, in a private data center, or in the cloud.

Containers provide a form of operating system-level virtualization where each container runs in isolation with its own runtime image, while sharing the host OS kernel. They are built using Open Container Initiative (OCI) standards, which define image formats and runtime specifications, making containers interoperable across various orchestration platforms such as Kubernetes, Docker Compose, or cloud-based services like Amazon Elastic Container Service (ECS).

In the context of OpenRTB and programmatic advertising, containers act as execution units for business logic that must interact with the bidstream in real-time. These execution units can host logic for bid modification, audience activation, deal curation, and other bidstream processes, while offering flexibility, modularity and simplified deployment for participants without requiring the container or the orchestrator to provide bespoke core platform infrastructure to each partner.

Why Containers?

Containers enable a portable (capable of running in many environments), composable (can be combined with other containers or systems to provide value), dynamically scalable (capacity can be ramped up and down as demand does), and protected mode (manage ingress and egress, protect IP and sensitive data allowing execution of decisioning logic across diverse

programmatic environments). Container's standardized packaging allows partners to deploy once and operate across multiple orchestrating entities and cloud infrastructures without retooling, while composability ensures that logic from buyers, sellers, identity providers, and measurement vendors can be integrated into a cohesive, real-time auctioning workflow. Critically, containers encapsulate business logic in a self-contained image isolating proprietary IP while still allowing participation in bidstreams without requiring independent scaling.

Integration into the Container Ecosystem

To run a container from a partner company in an orchestrator's infrastructure, both parties must address specific requirements and restrictions:

- The agent provider must ensure their application runs properly as a non-root user
- The orchestrator will never run the container as a root user
- The container must implement a Kubernetes compatible health and readiness probes (HTTP endpoints, TCP checks, or exec commands)
- Containers must follow the principle of least privilege and remove all unnecessary capabilities.
- The container must be built to handle graceful shutdowns, respect security contexts, and must not require privileged access or host network/PID namespaces.
- The container must run without external network access: all network ingress and egress is prohibited with the exception of service communications with the orchestrating entity. Agent providers and orchestrating entities may negotiate specific network access policies to facilitate efficient operation of containers in the orchestrator's environment.

On the orchestrating side, the orchestrator needs to implement appropriate RBAC policies that grant containers only the minimum required resource permissions. Network policies must be configured to control both ingress and egress traffic and explicitly define which services containers can communicate with and which ports are accessible. Depending on the required security level of network traffic control, the orchestrator may want to allow agent providers alternatives such as volume mappings with external data synchronization facilities to enable container data import and export. For secrets management, the orchestrator must securely inject any required credentials, API keys, or certificates using Kubernetes Secrets or similar methods, such as mounted secrets filesystem or secure environment variables) and will ensure they're properly scoped and rotated. Additionally, the orchestrator is expected to implement resource quotas and limits, use Pod Security Policies or Pod Security Standards or similar methods to enforce security constraints, and support monitoring and logging to track the container's behavior within their infrastructure.

Why OpenRTB Patch?

OpenRTB itself is a multi-party structurally agentic system. Complex interactions between many different parties are built into the OpenRTB and AdCOM domain models which describe the parameters of the real time bidding process. Introducing new agents into the existing system

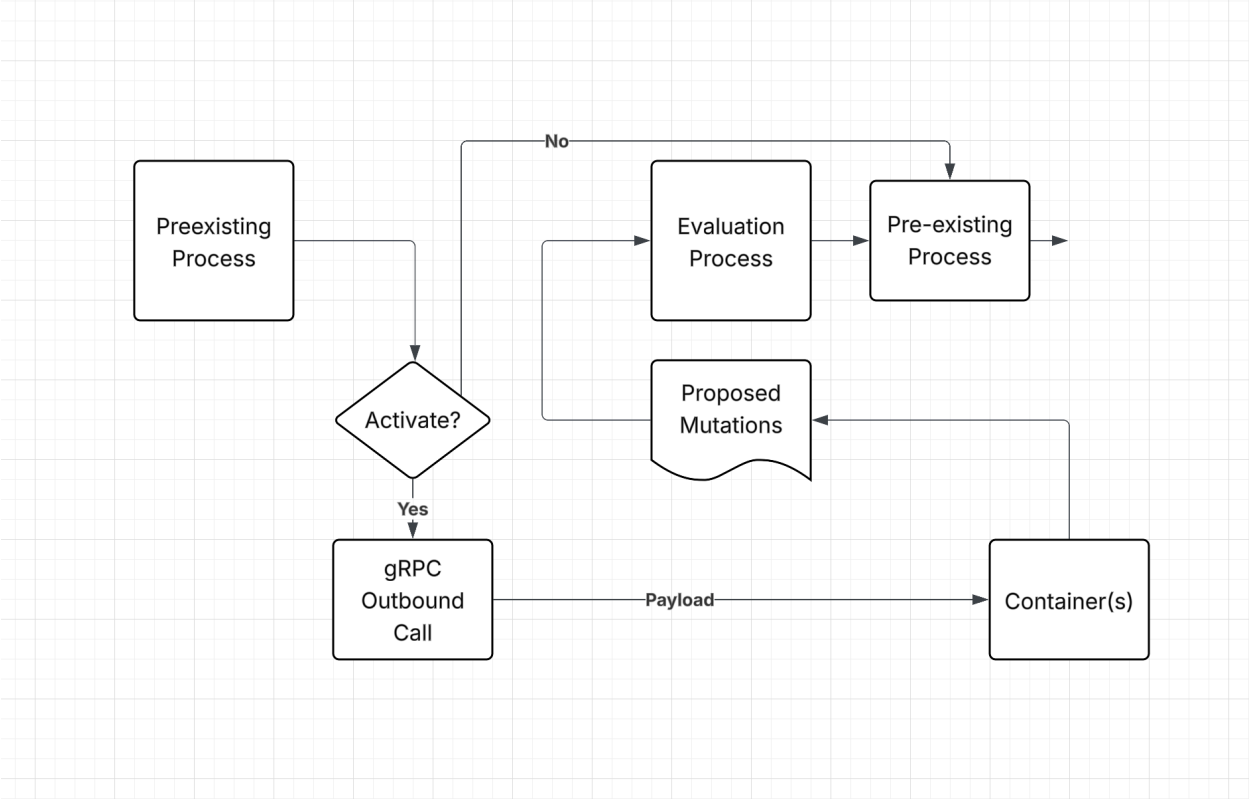
bypassing the full OpenRTB payload to those agents and the agent then returning the full, updated OpenRTB payload has a number of drawbacks:

- Inefficiency of passing the full OpenRTB payload to every entity introduces considerable serialization and deserialization costs.
- Using a sequential approach for agentic calls has unacceptable performance implications
- Running multiple agentic calls with the same payload that all return a modified payload makes it extraordinarily difficult and expensive to understand the changes to the payload that each individual agent makes.
- The full OpenRTB payload exposes all information, including information that may not be germane to the agent's purpose or may contain competitive or restricted information.
- Data may need to be scrubbed for data privacy purposes as well. Reconstructing the payload would be expensive.
- Some information needed by the agent may not be part of the OpenRTB standard.
- It may be very difficult for orchestrating systems to understand *why* some changes are made.

To address these changes, This standard envisions a protocol that meets the following requirements:

1) Each Agent should only get the minimum data allowed by the orchestrator and requested by the agent provider, following the least-data principle. In particular, private and competitive signals must be removed prior to agentic involvement.
2) Agentic Changes to the OpenRTB payload must be isolated to ensure that each change can be evaluated or rejected independently.
3) Each agent should be decoupled from the processing pipeline implementation. An agent implementation should never know internal specifics such as call sites or internal structures of orchestration systems.
4) Agentic Changes to the OpenRTB payload must declare their intentions to ensure that orchestrators understand why proposed changes are made.

To meet these requirements this standard extends the model by introducing a patching mechanism, the *OpenRTB Patch Protocol*, which provides a standardized protocol for requesting changes to an OpenRTB bid request or bid response. When processing bid requests and bid responses, the orchestrator will identify the containers that should be included in its processing and will issue a request to each. Participant responses may request in-flight modifications via patch objects that include desired mutations which the orchestrator may accept or reject at their discretion. The OpenRTB Patch Protocol also introduces a concept of *intents* which are declarative attributes that indicate the purpose of a given mutation which enables orchestrators to better determine when and how containers should be invoked .



Path Clarification

Sometimes agents might return mutations that need to be inserted at a later point, and may specify a particular id (for example, a particular deal-id). In some cases mutations might be inserted in a different location or sequence than the agent anticipates. To maintain this flexibility on an intention by intention basis - mutation paths may use semantic references derived from openRTB concepts. Rather than following explicit paths to the overall RTB structure (which the agent may not know), paths identify business level entities rather than specific JSON locations. This allows the spec to express mutations in terms of auction semantics rather than data layout.

Why gRPC and MCP?

This spec for agents requires that containers support Remote Procedure Calls for communication between orchestrators and agents. GRPC with protobuf is mandated for all interfaces in this document, while MCP may be used when appropriate. While OpenRTB recently added support for GRPC at a native level, most OpenRTB payloads are still REST and JSON. The reasons for supporting GRPC instead of REST are:

1) Performance needs to be as fast as possible for these interfaces. GRPC uses protobuf serialization, which is considerably more space and time efficient than JSON.
2) GRPC is easy to validate and has built-in rejection of invalid payloads.

3) Remote Procedure Calls are a better model for agent-based communication than state transitions.

MCP may also be used (and will likely be required in the future). In particular MCP version 2025-06-18 is the first version that is both performant enough and secure enough to handle significant traffic. This is primarily due to the support of streamable HTTP as well as OAuth authentication. MCP also uses JSON-RPC which makes it a natural successor to REST based interfaces. In addition, while GRPC is ideal for service to service agentic orchestration, MCP can be used not only for service to service orchestration, but also model to agent orchestration, enabling autonomic agentic flows as well.

In both cases, the same set of interfaces are exposed - in GRPC via rpc definitions in protobuf, or via tool definitions in MCP.

Agent Manifest

An agent manifest is a standard field in JSON added to the container image via the image metadata expressed as a label which defines how a container is used and managed by an orchestrator. The manifest is specified as a key-value pair with the key being “agent-manifest” and the value being a JSON structure.

The manifest provides business requirements for the container such as:

- The name of the agent
- The vendor for the agent
- The owner (email address) of the agent

It also includes a number of business metadata fields:

- The intent(s) supported by this agent.

It also includes information about the runtime configuration for the container:

- The minimum CPU and memory resources it requires,
- Any dependencies on other services (by name) that the container needs to communicate with.

Technical Implementation

Infrastructure Considerations

Composable and deployable mean that Agents need to be able to be deployed in a wide variety of infrastructure in a manner that is consistent with the base infrastructure. In practice this means that they must be structured as OCI containers and must be deployable into a distributed container ecosystem (typically Kubernetes)
The following best practices for Docker images must be followed by both the agent provider and orchestrator for any agent acting on bidstream data:

- **Mandatory health checks** should be implemented as both liveness and readiness probes to ensure robust self-healing capabilities, allowing infrastructure to operate autonomously and maintain continuous service availability even during partial failures.
- **Image signing and verification** must be enforced across all deployments to establish a chain of trust from development to production, preventing unauthorized or compromised artifacts from entering the bidstream ecosystem.
- **Images must be built by a managed and auditable CI/CD pipeline.** Pipelines should automate the entire release process with integrated security scanning, compliance checks, and deployment validation to maintain velocity while ensuring quality and consistency.
- **Standard logging and monitoring** must be integrated into all deployments to provide real-time visibility into system health, enabling proactive identification of performance bottlenecks and security anomalies before they impact service delivery. Containers must support Open Telemetry endpoints for metrics and support Open Tracing for distributed trace and span collections.

The following are a few of the best practices for docker images which must be followed for any orchestrator:

- **Network policies and segmentation** must be implemented to control traffic flows between services, between containers and between privacy sensitive services to protect them from unauthorized workloads. Systems should be configured to provide defense-in-depth isolation that prevents lateral movement and limits the blast radius of security breaches within your infrastructure.
- **Resource quotas and limits** must be defined at both namespace and container levels to guarantee predictable performance, prevent resource starvation, and optimize utilization across your infrastructure.

Manifest Requirements

Agents are expected to provide a manifest as described above that expresses their needs:

1) Minimum CPU/Memory requirements

2) The intents they will invoke
3) Other services that they expect to be available (by name)

If the orchestrating entity cannot support the requirements and intents of a container, it should not be started.

Bidstream Integration

Agents are implemented as containers that are essentially black boxes that allow service providers to execute core business logic. Each container “lives” in a detached and isolated compute environment once deployed. For these containers to work, they must integrate into the bid stream using predefined intentions. Because participants in the AdTech ecosystem generally have unique mixes of infrastructure, software, and practices. What extension points are supported and how they integrate with agents will likely be custom for each participant. Since it’s not feasible or desirable to have everyone run the same software, instead, a standard protocol is defined that supports generic interactions with extension points. OpenRTB provides robust support for data communication within the bidstream, with appropriate abstractions for vendor-specific extensions, but lacks standardized support for incorporating changes like those containers may want to propagate back to the bid stream as part of a decisioning cycle. To address that shortfall, this standard introduces “OpenRTB Patch,” a protocol for expressing desired changes back into OpenRTB requests and responses.

OpenRTB Patch is designed to adhere to the following principles:

- Containers provide desired mutations to the orchestrator as “patches” which describe a delta from the provided request or response and may include some combination of additions, changes, and deletions to specified portions of the request or response. Whether or not a patch is applied is left to the discretion of the orchestrator, which will decide to accept or reject mutations in accordance with its business requirements. Whether and how orchestrators will communicate choices to accept or reject patches to agent providers is left to the parties.
- Each patch is atomic. A mutation must be accepted in whole or rejected in whole. Multiple patches may be independently accepted or rejected - there are no transactions or any ordering guarantees across mutations.
- Mutations are semantically meaningful: an OpenRTB Patch specifies not only what change is desired, but also why it is requested - this is specified as the intent.
- OpenRTB patch propagates necessary privacy and signal information into containers. It is the container's responsibility to honor all of the specified behavior in the RTB request.
- OpenRTB patch follows the semantics of the OpenRTB standard, including use of “ext” objects for any nonstandard, extended signaling orchestrators that may wish to make available to agent providers.
- OpenRTB patch facilitates auditability within an orchestrator's environment by logging requested and applied patches.

- Orchestrators should provide standard reports and metrics to judge the acceptance rate of mutations and rejection reasons.

Service Naming

As mentioned above, containers can provide specific dependencies in the name of service that they have a dependency on, Since this will often map to proprietary services that orchestrators provide, or names of other containers, the specification is simply that the service dependency be a string to signify the dependency for mapping.

API Design

RPC definitions for both gRPC and for MCP require a bit more structure than a traditional REST/JSON service. For purposes of this document, a few key patterns are defined, but the authoritative definition for this is the gRPC definition found in the IABTechLab's Github Project. These definitions take the form of Protobuf specifications. In particular, the services and messages must be articulated in .proto files. For MCP use cases, the specification is the same as GRPC, with the obvious exception of a tool definition rather than RPC endpoints for each extension point.

For the purpose of the comment period, there is an initial protobuf definition. These are subject to change.

Example Extension Point

```
syntax = "proto2";

package com.iabtechlab.bidstream.mutation.v1;

import "com/iabtechlab/openrtb/v2.6/openrtb.proto";

// service definition
service RTBExtensionPoint {
  // GetMutations returns RTBResponse containing mutations to be
applied at the predetermined auction lifecycle event
  rpc GetMutations (RTBRequest) returns (RTBResponse);
}


message RTBRequest {
  // ENUM as per Programmatic Auction Definition IAB TL doc/spec
```

```
  required Lifecycle lifecycle = 1;

  required string id = 2;
  optional Extensions ext = 3;

  required com.iabtechlab.openrtb.v2.BidRequest bid_request = 4;

  optional com.iabtechlab.openrtb.v2.BidResponse bid_response = 5;


  required int32 tmax = 6;

}
```

This forms the backbone of the extension system. Each individual request exposes a single rpc and a single stream mechanism for a given extension point. Right now the spec does anticipate the need to support multiple called endpoints so that the same container could support multiple different service requests.

AgenticRTB Specific Requirements

Field	Type	Description
id	string	Extension point Request ID
bidRequest.imp	imp message	Impression message (per OpenRTB, with exceptions)
bidRequest.site	site message	Site message (per OpenRTB)
bidRequest.app	app message	App message (per OpenRTB)
bidRequest.device	device message	Device message (per OpenRTB)
bidRequest.user	user message	User message (per OpenRTB)
bidRequest.regs	reg message	Regulation message (per OpenRTB)
bidRequest.source	source message	Source message (per OpenRTB)

bidRequest.tmax	integer	Maximum time in milliseconds the exchange allows for mutations to be received including internet latency to avoid timeout

Extension Response

A mutation represents a change to an existing request —it modifies the system's state by adding, removing, or updating records. Extension Provider responses take the form of these mutations, proposing adjustments to bid requests or responses. An example Extension Response gRPC is below.

```
message RTBResponse { // Or RequestPatch
  string id = 1;
  repeated mutation mutations = 2;
  MetaData metadata = 3;
}

message Mutation {
  string intent = 1;
  string op = 2;
  string path = 3;

  // The structure of value depends on the specified intent.
  // Reserve 100+ for intent-specific payloads
 // The structure of value depends on the specified intent.
  // Reserve 100+ for intent-specific payloads
  oneof value {
    // List of string Identifiers
    IDsPayload ids = 100;

    // Adjust properties of a specific deal
    AdjustDealPayload adjust_deal = 101;

    // Adjust the bid price
    AdjustBidPayload adjust_bid = 102;

    // Metrics or telemetry data
    AddMetricsPayload add_metrics = 103;
```

```
  }


}

message MetaData {
  string api_version = 1;
  string model_version = 2;
}
```

Example - Audience Segments - Activating Cohorts

Below is an example of a response return. Since gRPC is a binary protocol, JSON format is used to express concepts rather than serialization formats.

```
{
    "intent": "activateSegments",
    "op": "add",
    "path": "/user/data/segment",
    "value": {
        "IDsPayload": [“18-35-age-segment","soccer-watchers"]
    }
}
```

Example - Complex Orchestration

Endpoints may return multiple mutations. If so, each mutation is evaluated separately. Note that there is no support for transactions across multiple mutations. Any mutation here may be accepted or rejected independently of other mutations.

```
[{
    "intent": "expireDeals",
    "op": "remove"
    "path": "/imp/1",
    "value": {
        "IDsPayload": [“deal100","deal200"]
    }
},{
    "intent": "activateDeals",
```

```
    "op": "add",
    "path": "/imp/1",
    "value": {
        "IDsPayload": [“deal300","deal201"]
    }
},{
    "intent": "adjustDeals",
    "op": "replace"
    "path": "/imp/2/deals/400",
    "value": {
        “AdjustDealPayload”: {
            "bidfloor": 5.00,
            "wadomain": ["adomain.com"],
        }
     },
},{
    "intent": "adjustDeals",
    "op": "replace",
    "path": "/imp/1/deals/500",
    "value": {
        “AdjustDealPayload”: {
            "bidfloor": 8.00
        }
    },
}]
```

Manifest - Container.json

This is an example of a container JSON. Note that this is associated with the metadata for the docker image for transmission.

```
{
  "name": "openrtb-container-suite",
  "version": "1.0.0",
  "description": "Example container manifest for OpenRTB and Digital Advertising use cases",
  "image": {
    "repository": "registry.example.com/rtb/auction-container",
    "tag": "v1.0.5",
```

```
    "digest": "sha256:abc123def4567890"
  },
  "resources": {
    "cpu": "500m",
    "memory": "256Mi"
  },
  "intents": [
    {
      "name": "bidResponseGeneration",
      "description": "Generate bid responses in real time based on request data"
    },
    {
      "name": "bidValuation",
      "description": "Evaluate incoming bid requests and determine bid amount"
    },
    {
      "name": "bidRequestModification",
      "description": "Propose mutations to a bid request prior to auction execution"
    },
    {
      "name": "auctionOrchestration",
      "description": "Route or prioritize bid requests across multiple buyers"
    },
    {
      "name": "metadataEnhancement",
      "description": "Insert or modify auction metadata such as fraud or viewability signals"
    },
    {
      "name": "dynamicDealCuration",
      "description": "Curate deals in real time, optimize margins or enforce dynamic
inclusion/exclusion lists"
    },
    {
      "name": "audienceSegmentation",
      "description": "Activate or enrich user cohorts and audience segments"
    }
  ],
  "dependencies": {
    "fraudDetectionService": {
      "service": "fraud-svc",
      "ENV_VARIABLE": FRAUD_URL
    },
    "audienceService": {
```

```
      "host": "audience-svc",
      "port": 9010
    }
  },
  "health": {
    "livenessProbe": {
      "httpGet": {
        "path": "/health/live",
        "port": 8080
      },
      "initialDelaySeconds": 10,
      "periodSeconds": 5
    },
    "readinessProbe": {
      "httpGet": {
        "path": "/health/ready",
        "port": 8080
      },
      "initialDelaySeconds": 5,
      "periodSeconds": 5
    }
  },
  "security": {
    "runAsNonRoot": true,
    "dropCapabilities": ["NET_ADMIN", "SYS_PTRACE"],
    "networkPolicies": {
      "ingress": ["fraud-svc", "audience-svc"],
      "egress": ["logging-svc"]
    }
  },
  "maintainers": [
    {
      "name": "IAB Tech Lab Example Team",
      "email": "support@iabtechlab.com"
    }
  ]
}
```