MCP Registry Architecture

DNS-Based Discovery for AI Agents

Mario Thomas
mario@mariothomas.com
Version 1.5
19 March 2026

Abstract

This paper proposes a lightweight, deployable architecture for solving the agent discovery problem in Model Context Protocol (MCP) ecosystems. The core proposal is a DNS-based convention — an _mcp TXT record — that points any compliant AI agent to an organisation's MCP registry. DNS-based discovery is not new: MX, SRV, _dmarc, and WebFinger all use the same pattern. The novelty here is the specific convention for MCP, the design decision to make the registry itself an MCP server (requiring no special client code), and a fully serverless reference implementation: Amazon CloudFront for global distribution, Lambda@Edge for request parsing, RS256 JWT authentication, and JSON-RPC routing, and Amazon DynamoDB Global Tables for dynamic registry data. No novel infrastructure is required. The proposal composes existing, operationally proven technologies into a governance-first discovery layer that any organisation can deploy in a day and run for under $5 per month at typical registry volumes.

Document History

Version	Date	Status	Summary of Change
1.0	25 February 2026	Published	Initial publication.
1.1	27 February 2026	Published	Added Section 2.3 — What This Proposal Does Not Solve — clarifying that the _mcp DNS record addresses discovery only, and that authentication, authorisation, and tool capability enumeration are explicitly out of scope for the DNS layer.
1.2	28 February 2026	Published	Extended Section 7.4 to address registry-level content filtering as a mitigation for prompt injection attacks, drawing on the DNS reputation services analogy. Added Section 10.6 — Agent Peer Discovery: A Natural Extension — sketching the `/.well-known/mcp` peer discovery model, referencing WebRTC and data mesh parallels, and positioning direct agent capability advertisement as a complementary layer to the registry architecture. Acknowledgements section added.
1.3	2 March 2026	Published	Extended Section 8.1 to document path-based and subdomain-based registry URL patterns as equally compliant implementation approaches, with trade-offs for each. Updated SPEC.md accordingly.
1.4	2 March 2026	Published	Updated Section 8.1 to reflect that the reference implementation at mcp.mariothomas.com uses path-based routing. Updated Section 8.4 and 8.5 to use correct filename index.js and handler index.handler. Rewrote Section 12 in present tense to reflect live deployment, confirmed GitHub repository URL, and noted path-based routing pattern used in the reference implementation.
1.4.1	3 March 2026	Published	Added copyright notice. No content changes.
1.5	19 March 2026	Published	Expanded Section 2.3 to include the explicit 255-byte DNS TXT record constraint as the hard technical rationale for the pointer-not-container design decision. Added version field to Section 5.4 DynamoDB schema for semantic versioning of registry entries, with accompanying field description distinguishing it from protocol_version. Added Section 5.7 — The Read Path: Audit Trail — making explicit that CloudFront and Lambda@Edge logs provide a queryable record of every agent access. Added Section 7.5 — Context Window Constraints in

			Large Registries — noting this as a known limitation and recommending aggressive use of capability_filter, concise server entries, and pagination at scale. Acknowledgements updated to reflect feedback from Cole Johnston.

1. The Problem This Paper Solves

AI agents are only as useful as the tools they can access. Model Context Protocol (MCP) defines how agents connect to those tools — databases, APIs, file systems, business applications — using a standardised interface. But it leaves one question entirely unanswered: how does an agent discover which MCP servers exist in the first place?

Today, the answer is manual configuration. A developer decides which MCP servers an agent has access to, hard-codes those addresses at build time, and hopes the list stays accurate. This approach works for a single developer with a handful of tools. It does not work for an enterprise deploying dozens of agents across hundreds of systems. It creates no governance trail, no central catalogue, and no mechanism for agents to discover capabilities they were not explicitly told about at the moment of their creation.

The problem has a name in distributed systems: the n × m integration problem. If you have n agents and m MCP servers, without a discovery layer you potentially need n × m bespoke configuration decisions. With a discovery layer, you need n + m: each agent knows where to find the registry, and each MCP server registers itself once. The difference between n × m and n + m is the difference between a fragile, manually maintained mesh and a governed, self-describing ecosystem.

This paper proposes a specific, implementable solution to that problem using infrastructure that already exists and is already trusted.

2. Background: What is MCP?

Model Context Protocol is an open standard created by Anthropic that defines how AI models connect to external tools and data sources. Before MCP, every integration between an AI agent and an external system required custom code — a bespoke connector written specifically for that combination. A company wanting its AI agents to access Salesforce, SAP, Google Drive, and an internal database needed four separate, custom-built integrations. When those systems changed their APIs, all four connectors needed updating.

MCP solves this through standardisation. An MCP server is a lightweight service that exposes a set of capabilities — tools — in a consistent, machine-readable way. Any MCP-compatible AI client can connect to any MCP server and immediately discover what that server can do, then invoke those capabilities using a standard protocol. The analogy is USB: before USB, every peripheral required its own connector. USB did not replace the underlying electronics; it standardised the interface so any device could talk to any host. MCP does the same for AI agents and external systems.

2.1 How MCP Works

When an agent connects to an MCP server, it sends a tools/list request. The server responds with a structured manifest — a JSON description of every tool it exposes, including each tool's name, a plain-language description, and the parameters it accepts. The agent's underlying model reads this manifest and can then reason about which tool to call and with what inputs, without any hard-coded logic.

A tool invocation follows the JSON-RPC 2.0 protocol. The agent sends a tools/call request with the tool name and parameters. The MCP server executes the underlying operation — querying a database, calling an API, reading a file — and returns a structured result. The agent incorporates that result into its reasoning and either calls another tool or formulates its response to the user.

Key Concept: The Agentic Loop
An AI agent operating over MCP servers executes a deterministic loop:

1. Receive user query
2. Consult available tools (tools/list)
3. Select and invoke appropriate tool (tools/call)
4. Evaluate result — is the task complete?
5. If not, return to step 3 with new context
6. Return final response to user

MCP provides the interface layer for steps 2, 3, and 5. The agent's sophistication lies in the quality of its reasoning at step 4, not in the underlying protocol.

2.2 What MCP Does Not Solve

MCP defines the communication protocol between an agent and a server. It does not define how an agent discovers which servers exist. Once connected, an agent can query a server's capabilities dynamically. But finding the server to connect to in the first place is left entirely to the implementer. This is the gap this paper addresses.

2.3 What This Proposal Does Not Solve

This proposal solves discovery. It does not solve authentication, authorisation, or capability enumeration — and it is important to be precise about why.

The _mcp DNS record is a naming convention. It tells a compliant agent where to find an organisation's MCP registry. It does not tell the agent whether it is permitted to connect, and it does not authenticate either party. The auth= field in the record is a pointer to a token endpoint — it is a signpost, not a gate. Authentication and authorisation remain the responsibility of the registry and of each individual MCP server. This paper describes one approach to registry-level authentication using RS256 JWT validation at the Lambda@Edge layer; that is an implementation choice, not a property of the DNS record itself.

The registry lists servers, not the capabilities beneath them. An agent that discovers the registry and retrieves the server manifest knows which MCP servers exist and where to connect. It does not yet know what any of those servers can do. Tool enumeration — the actual capabilities an agent can invoke — requires a subsequent tools/list call to each individual server. Discovery and capability enumeration are sequential steps, not a single operation. The DNS record initiates the first step only.

This layering is intentional. DNS is the right primitive for the discovery problem precisely because it is lightweight, globally distributed, and operationally well-understood. Extending it to carry authentication state or capability manifests would undermine those properties. Each layer should do one thing well.

There is also a hard technical constraint that reinforces this design decision. DNS TXT records are limited to 255 bytes per string, with a practical record size limit of approximately 512 bytes before EDNS0 is required. That is sufficient to carry a registry URL, a version string, an auth endpoint pointer, and a small number of flags. It is not sufficient to carry a governed catalogue of MCP servers with capabilities, authentication details, data residency constraints, ownership metadata, and deprecation state. The DNS record is a pointer by necessity as much as by design. Implementations that

attempt to encode a full server catalogue directly into the TXT record will encounter this constraint at modest scale.

3. The Discovery Problem in Detail

Discovery is a solved problem in other areas of internet infrastructure. Email clients find mail servers via MX records in DNS. Web browsers find servers via A and AAAA records. Service endpoints are advertised via SRV records. In each case, the pattern is the same: a well-known naming convention points any compliant client to the appropriate service, without requiring prior configuration or bilateral agreement between the two parties.

The MCP ecosystem currently lacks an equivalent. Emerging proposals such as NANDA (Networked Agents and Decentralised Architecture) from MIT recognise this gap and propose new federated infrastructure to address it. The ambition is sound, but the approach introduces significant complexity: new protocols, new trust hierarchies, new infrastructure components that organisations must operate and trust.

This paper takes a different position. The discovery problem requires a naming convention, not new infrastructure. That convention can be applied to infrastructure that already exists, is already globally distributed, and is already the subject of mature operational governance: the Domain Name System.

To be precise about what is and is not novel here: DNS-based service discovery is a well-established pattern. This paper does not claim to have invented it. The specific contributions are: (1) a standardised TXT record format for advertising an organisation-scoped MCP registry, distinct from the global public registry at registry.modelcontextprotocol.io; (2) the architectural decision to make the registry itself a fully compliant MCP server, so that agents require no special discovery client — only the same tools/list and tools/call calls they use for everything else; (3) edge-enforced public/private access control with no origin compute in the read path; and (4) a detailed, costed, deployable implementation blueprint using managed cloud infrastructure. The claim is not that no one has thought about DNS and MCP. The claim is that no prior work has combined these elements into a complete, enterprise-ready, governance-first pattern — one that enables organisations to publicly externalise their own MCP servers to the world via a simple DNS record while also supporting fully private registries.

4. The Proposed Solution: DNS as MCP Discovery Layer

4.1 The _mcp DNS Record

The proposal is straightforward: organisations publish a TXT record at a well-known subdomain of their domain that points any compliant agent to their MCP registry. The convention follows established precedent — _dmarc for email authentication policy, _well-known for OpenID Connect discovery, _acme-challenge for certificate issuance.

The record takes the form:

```
_mcp.example.com.   IN   TXT   "v=mcp1;
registry=https://mcp.example.com/registry;
public=true;
auth=https://auth.example.com/token;
version=2026-02"
```

The fields are as follows:

Field	Description
v=mcp1	Protocol version indicator. Allows future versions without breaking existing clients.
registry	The URL of the organisation's MCP registry — the directory of available MCP servers.
public	Boolean. Indicates whether unauthenticated agents may query the public registry. If false, all access requires authentication.
auth	The token endpoint for obtaining access credentials. Follows OAuth 2.0 conventions. Present only when private servers exist.
version	The MCP protocol version supported by this registry, in YYYY-MM format.

An agent that knows an organisation's domain can discover its MCP ecosystem with a single DNS query. No prior bilateral agreement, no manual configuration, no out-of-band communication required. The agent queries _mcp.example.com, receives the registry URL, connects to the registry, and calls tools/list to discover available servers. The entire discovery flow follows from a single DNS lookup.

4.2 Public and Private Servers

The public field in the DNS record signals whether the registry exposes any capabilities to unauthenticated agents. An organisation might expose certain MCP servers publicly — a product catalogue, a customer service capability, a public data feed — while keeping internal servers private. The registry enforces this boundary: unauthenticated requests

receive the public manifest only. Authenticated requests receive the full manifest, including private server details.

This maps cleanly onto real organisational needs. A company's CRM data is not public. Its product catalogue might be. The architecture allows both to coexist in the same registry with different access controls, without requiring separate infrastructure for each.

Where DNSSEC is deployed and validated, it can provide strong authenticity guarantees on the DNS record itself — an agent can verify that the _mcp record it received was signed by the authoritative nameserver for the domain, protecting against DNS spoofing attacks that might redirect agents to malicious registries. DNSSEC adoption is not universal, however, and implementers should not assume clients will validate signatures. Organisations publishing _mcp records should sign their zones; agents consuming them should validate where possible and treat unsigned records with appropriate caution.

5. The Registry Architecture

The MCP registry — the service the DNS record points to — is implemented as a fully serverless stack on AWS. No servers are provisioned, patched, or maintained. The architecture composes three AWS services: Amazon CloudFront for global distribution, Lambda@Edge for request processing, and Amazon DynamoDB Global Tables for registry data. Amazon S3 provides optional storage for extended capability manifests.

An important technical constraint shapes this choice: CloudFront Functions — the lighter-weight alternative to Lambda@Edge — cannot access the HTTP request body. Since the MCP protocol uses JSON-RPC 2.0, where the method name and parameters are carried in the POST body rather than the URL, body access is non-negotiable for a fully compliant MCP server. Lambda@Edge provides full Node.js runtime, access to the complete request body, and standard cryptographic libraries for RS256 JWT validation. These capabilities come at the cost of marginally higher latency — Lambda@Edge executes at regional edge locations rather than every CloudFront edge node — but the difference is typically 1–5ms and immaterial for registry queries.

The design principle is that every read operation — which is the overwhelming majority of registry traffic — is served from the edge without any compute infrastructure in the hot path. Write operations, which are infrequent and deliberate, go through a controlled path that provides the governance audit trail the architecture requires.

5.1 Architecture Overview

The Request Flow — Step by Step

Step 1: An AI agent queries DNS for _mcp.example.com
→ Receives: registry URL, public flag, auth endpoint

Step 2: Agent connects to https://mcp.example.com/registry
→ CloudFront receives the request at nearest edge location

Step 3: Lambda@Edge function executes at nearest regional edge
→ Parses JSON-RPC method and parameters from request body
→ Validates JWT using RS256 (full Node.js crypto available)
→ Queries DynamoDB Global Table for registry entries
→ Filters by public/private based on auth status
→ Assembles and returns JSON-RPC response

Step 4: Agent receives server manifest
→ Connects to appropriate MCP servers
→ Calls tools/list on each to discover capabilities

No servers to provision. No instances to manage. No infrastructure to patch.

5.2 CloudFront: Global Edge Distribution

Amazon CloudFront is a content delivery network with edge locations in over 90 cities across 47 countries. Every request to the registry is served from the edge location geographically nearest to the requesting agent — typically within single-digit milliseconds of network latency, regardless of where the registry is nominally hosted.

For a registry that may be queried by AI agents running anywhere in the world — including agents embedded in third-party systems that have discovered the _mcp record — this global distribution is not an optimisation. It is a baseline requirement. A registry served from a single regional endpoint would introduce meaningful latency for geographically distributed agents and create a single point of failure.

CloudFront also handles TLS termination, HTTP/2, and DDoS protection through AWS Shield Standard at no additional cost. The registry inherits enterprise-grade security posture from the first line of infrastructure.

5.3 Lambda@Edge: Request Processing

Lambda@Edge functions execute Node.js in AWS's regional edge locations — over 100 locations globally. Unlike CloudFront Functions, Lambda@Edge has full access to the HTTP request body, supports the complete Node.js runtime including cryptographic libraries, and can make network calls to AWS services such as DynamoDB. These capabilities are essential for a fully MCP-compliant registry that must parse JSON-RPC 2.0 request bodies and perform RS256 JWT validation.

The Lambda@Edge function in this architecture performs three operations on every inbound request:

1. Request body parsing. The function reads the raw POST body and parses the JSON-RPC 2.0 envelope to extract the method name and parameters. This is the operation that makes Lambda@Edge a requirement: CloudFront Functions do not expose the request body and cannot perform this step.
2. JWT authentication. If an Authorization: Bearer header is present, the function validates the token using RS256 asymmetric verification via the Node.js crypto module. The public key is embedded in the function package. Invalid tokens are rejected with a 401 response before any registry data is accessed. Valid tokens are decoded to determine the requester's access level.
3. Registry query and response assembly. Based on the JSON-RPC method, the function queries DynamoDB for the relevant registry entries, applies the public/private filter based on authentication status, wraps the results in a valid JSON-RPC response envelope, and returns it directly. The CloudFront origin is never consulted for standard registry operations.

The function package is approximately 120 lines of JavaScript including dependencies. Lambda@Edge functions have a 5-second execution timeout and a 1MB package limit — both comfortably within the requirements for registry operations.

5.4 DynamoDB Global Tables: Dynamic Registry Data

Amazon DynamoDB Global Tables provide multi-region, fully managed NoSQL storage accessible from Lambda@Edge functions. Each registry entry is a DynamoDB item, queryable by primary key or via a prefix-based scan on the server identifier attribute. DynamoDB was chosen over CloudFront KeyValueStore (KVS) because KVS is accessible only from CloudFront Functions — the lighter-weight execution environment that cannot read request bodies. Lambda@Edge requires a data store it can reach via the AWS SDK, and DynamoDB Global Tables provide the globally distributed, low-latency reads that a registry serving agents worldwide requires.

Each MCP server in the registry is stored as a DynamoDB item. The key structure follows a hierarchical naming convention that allows efficient prefix-based retrieval:

```
Key:    server:crm-readonly
Value:  {
            "name": "CRM Read-Only Access",
            "url": "https://crm-mcp.internal.example.com",
            "public": false,
            "protocol_version": "2024-11",
            "capabilities": ["accounts", "opportunities", "contacts"],
            "data_residency": "eu-west-1",
            "auth_required": "bearer",
            "deprecated": false,
            "added": "2026-01-15",
            "owner": "sales-ops@example.com",
            "version”: “1.0.0”
        }
```

The fields are self-describing but the distinction between `protocol_version` and `version` is worth making explicit. `protocol_version` identifies the MCP protocol version the server implements — this is an interoperability constraint for connecting agents. `version` carries the semantic version of the registry entry itself, allowing agents and registry consumers to detect when a server entry has been materially updated without comparing full item contents. Agents that cache registry responses should treat a changed `version` value as a signal to refresh their local copy.

DynamoDB item sizes are limited to 400KB, which is ample for registry metadata. DynamoDB Global Tables replicate data across chosen AWS regions within seconds, ensuring that Lambda@Edge functions in any region read from a local replica rather

than crossing the Atlantic or Pacific on every request. This replication is automatic and requires no operational management.

DynamoDB makes the registry genuinely dynamic. New MCP servers can be added without any redeployment of the Lambda@Edge function or any code change. The registry is a data store with a consistent query interface, not a static file that requires manual updates and redeployment.

5.5 S3: Extended Manifests

For MCP servers with extensive capability manifests — detailed tool descriptions, parameter schemas, example invocations — the DynamoDB item may contain a reference to an S3 object rather than the full manifest inline. The Lambda@Edge function returns the S3 URL, and the requesting agent fetches the full manifest directly. S3 is also used as the origin for any static content the registry serves, such as documentation or schema files.

This two-tier approach keeps the hot path — the registry query — within CloudFront and Lambda@Edge, while S3 handles bulk content delivery for agents that need comprehensive capability information before connecting to a server.

5.6 The Write Path: Governed Registry Updates

Read operations are fully serverless. Write operations — adding, updating, or deprecating MCP server entries — follow a controlled path that enforces governance. The recommended pattern uses a Git repository as the source of truth for the registry, with a GitHub Actions workflow pushing changes to DynamoDB:

1. An engineer or system raises a pull request to the registry repository adding or modifying a server definition file (JSON or YAML).
2. The pull request is reviewed and approved by the designated registry owner — a governance role, typically in the AI Centre of Excellence or infrastructure team.
3. On merge, a GitHub Actions workflow triggers. It calls the DynamoDB API using the AWS SDK to write the new or updated item to the Global Table.
4. The change is immediately available globally. No cache invalidation, no redeployment, no restart required.

Every change to the registry is therefore a named commit by an identifiable author, approved by a designated reviewer, with a complete audit trail in version control. This is governance that emerges from the architecture rather than being bolted on as an afterthought. It also means the registry is fully recoverable: if a bad entry is published, the previous state is one revert away.

Section 5.7 — The Read Path: Audit Trail

The read path has a complementary audit trail that is equally useful and requires no additional instrumentation. CloudFront access logs record every request made to the registry, including the requesting IP address, timestamp, HTTP method, path, response code, and bytes transferred. Lambda@Edge logs, available in CloudWatch Logs in each edge region, record function execution details including request headers — which carry the bearer token presence, though not their contents — and response codes. Together these logs provide a queryable record of every agent that accessed the registry, what it requested, and when. Organisations with compliance or security requirements should enable both log sources from the outset; they are off by default in CloudFront but trivial to enable, and the cost at typical registry query volumes is negligible.

6. The Registry Protocol

The registry exposes a subset of the MCP protocol. It is itself an MCP server — specifically, one whose tools describe other MCP servers. This design choice means agents interact with the registry using exactly the same protocol they use for all other MCP servers: no special handling, no separate discovery SDK, no additional library dependencies.

6.1 tools/list Response

A tools/list request to the registry returns the four tools the registry exposes:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "tools": [
            {
                "name": "discover_servers",
                "description": "Returns all MCP servers available
                                to this client, filtered by
                                authentication status.",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "capability_filter": {
                            "type": "string",
                            "description": "Optional. Filter
                                            servers by capability
                                            tag."
                        }
                    }
                }
            },
            {
                "name": "get_server_details",
                "description": "Returns full connection details
                                for a specific server.",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "server_id": {
                            "type": "string",
                            "description": "Server identifier."
                        }
                    },
                    "required": [
                        "server_id"
                    ]
                }
            },
            {
                "name": "search_servers",
```

```
                "description": "Searches the registry by
                                capability, data type, or
                                keyword.",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "query": {
                            "type": "string"
                        }
                    },
                    "required": [
                        "query"
                    ]
                }
            },
            {
                "name": "check_server_health",
                "description": "Returns last-known health status
                                of a server.",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "server_id": {
                            "type": "string"
                        }
                    },
                    "required": [
                        "server_id"
                    ]
                }
            }
        ]
    }
}
```

6.2 discover_servers Response

A tools/call request invoking discover_servers returns the full server manifest for all servers the requesting agent is authorised to see. The registry returns structured data as a JSON string within the MCP text content envelope — the standard content type for MCP tool results. This keeps the registry compatible with all MCP-compliant clients without requiring any special response handling:

```
{
    "jsonrpc": "2.0",
    "id": 2,
    "result": {
        "content": [
            {
                "type": "text",
                "text": "{
```

```
                        'servers': [
                                {
                                        'id': 'crm-readonly',
                                        'name': 'CRM Read-Only Access',
                                        'url': 'https://crm-
mcp.internal.example.com',
                                        'public': false,
                                        'capabilities': ['accounts',
'opportunities', 'contacts'],
                                        'data_residency': 'eu-west-1',
                                        'deprecated': false
                                },
                                {
                                        'id': 'product-catalogue',
                                        'name': 'Product Catalogue',
                                        'url': 'https://catalogue-
mcp.example.com',
                                        'public': true,
                                        'capabilities': ['products', 'pricing',
'availability'],
                                        'data_residency': 'global',
                                        'deprecated': false
                                }
                        ]
                        }"
            }
        ]
    }
}
```

7. Security Considerations

Security considerations are not optional in an IETF context, and they are not an afterthought in this proposal. The architecture makes specific, deliberate security choices at each layer.

7.1 DNS Record Integrity

The _mcp DNS record should be protected by DNSSEC where the zone is signed and the consuming agent validates signatures. Without DNSSEC validation, an attacker capable of poisoning the DNS cache of a target agent could redirect it to a malicious registry. DNSSEC can provide cryptographic proof that a DNS record was published by the authoritative nameserver for the domain — but this protection is only effective when both sides of the exchange honour it. Organisations publishing _mcp records should sign their zones as a baseline practice; this is increasingly straightforward with modern DNS providers and registrars.

Organisations implementing this proposal should also consider TTL carefully. A short TTL allows rapid updates but increases DNS query volume and the exposure window during which a poisoned record might be served from cache. A TTL of 300 to 900 seconds (5 to 15 minutes) is recommended as a balance between agility and resilience.

7.2 Registry Authentication

The registry is a high-value target. An attacker who can modify registry entries — replacing legitimate MCP server URLs with malicious endpoints — can redirect all agent traffic connecting through that registry. The write path governance described in Section 5.6 is therefore not merely a convenience; it is a security control. Every registry change must be attributed, reviewed, and logged.

JWT validation at the Lambda@Edge layer uses RS256 asymmetric cryptography via the Node.js crypto module, which is fully available in the Lambda@Edge runtime. The private key never leaves the token issuer. The public key is embedded in the Lambda@Edge function package and used only for verification. This is straightforward and well-tested in production Lambda environments. Key rotation requires redeployment of the function package, which is a deliberate and logged operation.

The MCP specification has evolved to include OAuth 2.1 client registration patterns as the preferred authentication mechanism for MCP servers. Implementers should align token issuance with the MCP OAuth 2.1 profile: dynamic client registration, PKCE for public clients, and token binding where available. The Lambda@Edge authentication layer described here is intentionally agnostic to the specific OAuth flow used upstream — it validates the resulting JWT regardless of how it was issued. This means the registry

is forward-compatible with whatever authentication profile the MCP specification standardises, provided the token format remains JWT.

7.3 The Registry as Single Point of Failure

A centralised registry creates a single point of failure: if the registry is unavailable, agents cannot discover new servers. In practice, CloudFront's global distribution and 99.99% SLA make this an unlikely operational concern. But it remains a valid architectural consideration, particularly for agents that must be resilient to infrastructure failures.

The recommended mitigation is for agents to cache the registry manifest locally with a configurable TTL. An agent that has previously retrieved a server manifest can continue operating against known servers even if the registry is temporarily unreachable. This is the same pattern used by DNS resolvers caching previously resolved records.

7.4 Prompt Injection via Registry

A novel attack surface this proposal introduces is prompt injection through registry metadata. If an attacker can insert entries into the registry — or compromise a legitimate server's DynamoDB item — they could include text in the capability descriptions or server names designed to manipulate an agent's reasoning. For example, a malicious server description might include instructions that, when read by the agent as part of its tools/list context, cause the agent to take unintended actions.

Mitigations include: strict input validation on all DynamoDB writes, limiting the length and character set of free-text fields, and treating all registry content as untrusted input in the agent's reasoning pipeline. Agents should not execute instructions embedded in tool descriptions; they should use those descriptions only to reason about which tool to invoke.

As the MCP ecosystem matures, registry-level content filtering services will likely emerge — analogous to DNS reputation services such as Cisco Umbrella or Cloudflare Gateway, which maintain continuously updated lists of known malicious domains and block responses before they reach the client. The equivalent for MCP registries would be a content validation layer that inspects capability descriptions, server names, and free-text fields against known malicious patterns before entries are written to the registry and before they are returned to agents. Organisations operating registries at scale should design their write path to accommodate such a filtering layer — a validation step between the pull request approval and the DynamoDB write that can be updated independently of the registry infrastructure itself. This is not a requirement for initial deployment, but it is the right seam to build into the architecture from the start. Agents consuming registry content should treat all free-text fields as untrusted input regardless

of the registry's apparent reputation, applying the same scepticism a well-configured DNS resolver applies to every response it receives.

7.5 Context Window Constraints in Large Registries

As the number of MCP servers in a registry grows, the volume of data returned by `discover_servers` may become a practical constraint for agents with limited context windows. An agent that retrieves a full registry manifest containing dozens or hundreds of server entries, each with capability lists, authentication details, and metadata, may exhaust a significant portion of its available context before any tool invocation has taken place.

This is a known limitation of the current design. The primary mitigation is the `capability_filter` parameter on the `discover_servers` tool, which allows agents to request only servers matching a specific capability tag rather than retrieving the full manifest. Agents operating in large registry environments should use this parameter aggressively. Registry operators should also consider keeping individual server entries concise — verbose capability descriptions compound the problem. Agents that cache registry responses locally should store only the entries they actively use rather than the full manifest.

A secondary mitigation is pagination. The current reference implementation does not implement pagination on `discover_servers`, but registry operators deploying at scale should consider adding a `limit` and `offset` parameter to the tool schema to allow agents to retrieve the registry in pages rather than in a single response.

8. Implementation Guide

This section provides a practical walkthrough for implementing the complete registry architecture. The estimated time to a working deployment for an engineer familiar with AWS is four to six hours.

8.1 Vendor-Neutral Description

The registry architecture described in this paper is not inherently AWS-specific. Any implementation that satisfies the following functional requirements is compliant with the proposal:

- A globally distributed HTTP endpoint that accepts POST requests with JSON-RPC 2.0 bodies at a URL published in the _mcp DNS record.
- An edge-layer function — executing in under 50ms — that validates bearer tokens, reads server entries from a low-latency data store, and assembles JSON-RPC responses without consulting a compute backend for standard read operations.
- A key-value data store readable by the edge function with sub-millisecond latency, supporting prefix-based key enumeration.
- A controlled write path that updates the key-value store, with change attribution and audit logging.
- A private object store for binary assets (documents, manifests) accessible via time-limited signed URLs.

In pseudocode, the edge function logic is:

```
function handle(request):
body = parse_json_rpc(request.body)
is_auth = validate_jwt(request.authorization_header, PUBLIC_KEY)

if body.method == 'tools/list':
return tools_manifest(is_auth)

if body.method == 'tools/call':
tool = body.params.name
args = body.params.arguments

if tool == 'discover_servers':
entries = store.list(prefix='server:')
servers = filter(entries, public=True OR is_auth)
return mcp_response(servers)

if tool == 'get_server_details':
entry = store.get('server:' + args.server_id)
if entry.public OR is_auth:
return mcp_response(entry)
return error(403, 'Authorisation required')
```



```
return error(404, 'Method not found')
```

In the pseudocode above, store denotes any low-latency key-value or document store reachable from the edge compute runtime — DynamoDB Global Tables in the AWS reference implementation, Cloudflare KV in a Workers-based implementation, or any equivalent. The abstraction is intentional: the logic is identical regardless of the underlying store.

Registry URL Patterns

The registry= field in the _mcp DNS record points to the registry root. The convention does not mandate how MCP servers are addressed relative to that root. Two patterns are in common use, each with distinct trade-offs.

Path-based routing serves the registry and all MCP servers from a single domain, differentiating them by URL path:

```
registry=https://mcp.example.com/registry
```

Individual servers are addressed as:

```
https://mcp.example.com/articles
https://mcp.example.com/locations
https://mcp.example.com/documents
```

Path-based routing requires a single CloudFront distribution, a single certificate, and simpler DNS configuration. The Lambda@Edge function inspects the request path to route each request to the correct handler. The trade-off is that all servers share the same deployment lifecycle — updating one server's logic requires redeploying the shared function. This pattern is well suited to small teams operating all servers centrally.

Subdomain-based routing gives each MCP server its own subdomain under the registry domain:

```
registry=https://mcp.example.com
```

Individual servers are addressed as:

```
https://articles.mcp.example.com
https://locations.mcp.example.com
https://documents.mcp.example.com
```


Subdomain-based routing requires a wildcard certificate for *.mcp.example.com — note that a single-level wildcard such as *.example.com does not cover two levels deep — and either a separate CloudFront distribution per server or a single wildcard distribution with subdomain-aware routing in the Lambda@Edge function. The trade-off is additional infrastructure complexity. The benefit is genuine independence: each server can be deployed, updated, and owned by a different team without touching shared infrastructure. This pattern is better suited to larger organisations where different teams own different MCP servers.

Both patterns are fully compliant with the _mcp DNS convention. The reference implementation at mcp.mariothomas.com uses path-based routing. Implementers should choose based on their organisational structure, team ownership model, and operational preferences rather than any constraint imposed by this convention.

The following subsections describe the reference implementation of this pattern using Amazon Web Services. Equivalent implementations are possible using Cloudflare Workers + KV (Cloudflare Workers can read request bodies and support JWT validation natively), Azure Front Door + Azure Cosmos DB, or any CDN provider that supports edge computing with request body access and an associated data store. The AWS implementation is provided as a complete, tested reference — not as a requirement.

8.2 Prerequisites

- An AWS account with appropriate IAM permissions
- A registered domain with DNS management access
- AWS CLI configured with credentials
- Node.js 18 or higher (for the Lambda@Edge function and local testing)
- Git and GitHub (or equivalent) for the write-path governance workflow

8.3 Step 1 — Create the DynamoDB Global Table

```
# Create the registry table with global replication
aws dynamodb create-table \
--table-name mcp-registry \
--attribute-definitions AttributeName=server_id,AttributeType=S \
--key-schema AttributeName=server_id,KeyType=HASH \
--billing-mode PAY_PER_REQUEST \
--region us-east-1

# Enable Global Tables replication to additional regions as needed
aws dynamodb create-global-table \
--global-table-name mcp-registry \
--replication-group RegionName=us-east-1 RegionName=eu-west-1 RegionName=ap-
southeast-1

# Note the table ARN — you will need it for the Lambda@Edge execution role
```


8.4 Step 2 — Write the Lambda@Edge Function

Create a file named index.js. Lambda@Edge runs in the us-east-1 region and is deployed via CloudFront behaviours. The function uses the AWS SDK for DynamoDB access and the Node.js crypto module for JWT validation:

```
const { DynamoDBClient, ScanCommand } = require('@aws-sdk/client-dynamodb');
const { unmarshall } = require('@aws-sdk/util-dynamodb');
const crypto = require('crypto');

const TABLE_NAME = 'mcp-registry';
const dynamo = new DynamoDBClient({ region: 'us-east-1' });

// Embed your RS256 public key here
const PUBLIC_KEY = `-----BEGIN PUBLIC KEY-----
YOUR_PUBLIC_KEY_HERE
-----END PUBLIC KEY-----`;

exports.handler = async (event) => {
const request = event.Records[0].cf.request;

// Parse JSON-RPC body (requires Lambda@Edge — unavailable in CF Functions)
let body;
try {
const raw = Buffer.from(request.body.data, 'base64').toString('utf8');
body = JSON.parse(raw);
} catch (e) {
return errorResponse(-32700, 'Parse error');
}

// RS256 JWT validation via Node.js crypto (unavailable in CF Functions)
let isAuthenticated = false;
const authHeader = (request.headers['authorization'] || [{}])[0]?.value;
if (authHeader?.startsWith('Bearer ')) {
try {
const token = authHeader.substring(7);
isAuthenticated = verifyJWT(token, PUBLIC_KEY);
} catch (e) {
return { status: '401', body: 'Unauthorised' };
}
}

// Route on JSON-RPC method
if (body.method === 'tools/list') {
return mcpResponse(buildToolsList());
}

if (body.method === 'tools/call') {
const tool = body.params?.name;
const args = body.params?.arguments || {};
if (tool === 'discover_servers') {
const servers = await queryRegistry(isAuthenticated, args.capability_filter);
return mcpResponse({ servers });
}
```

```
if (tool === 'get_server_details' && args.server_id) {
const item = await getServer(args.server_id, isAuthenticated);
return item ? mcpResponse(item) : errorResponse(-32602, 'Not found or
unauthorised');
}
}
return errorResponse(-32601, 'Method not found');
};

async function queryRegistry(isAuth, capFilter) {
const result = await dynamo.send(new ScanCommand({ TableName: TABLE_NAME }));
return result.Items
.map(unmarshall)
.filter(s => s.public || isAuth)
.filter(s => !capFilter || (s.capabilities || []).includes(capFilter));
}

function verifyJWT(token, publicKey) {
const [h, p, sig] = token.split('.');
const verify = crypto.createVerify('RSA-SHA256');
verify.update(h + '.' + p);
if (!verify.verify(publicKey, sig, 'base64url')) throw new Error('Invalid');
return JSON.parse(Buffer.from(p, 'base64url').toString());
}

function mcpResponse(data) {
return { status: '200',
headers: { 'content-type': [{ value: 'application/json' }] },
body: JSON.stringify({ jsonrpc: '2.0', id: 1,
result: { content: [{ type: 'text', text: JSON.stringify(data) }] }
}) };
}
function errorResponse(code, msg) {
return { status: '400',
body: JSON.stringify({ jsonrpc: '2.0', error: { code, message: msg } }) };
}
```

Note on Production Completeness

The Lambda@Edge function above is illustrative. The DynamoDB scan with filtering, error handling, and helper functions are shown but abbreviated for clarity. A full, production-ready implementation (~120 lines, complete with DynamoDB pagination, RS256 JWT validation, capability filtering, and structured error responses) will be published in the companion GitHub repository, linked from mariothomas.com at publication.

Production note: ScanCommand performs a full table scan and is appropriate for small registries (under a few hundred servers). For larger deployments, a partition-key design with a GSI on the public attribute is recommended to avoid scan costs and improve query efficiency. The reference implementation in the GitHub repository demonstrates both patterns.

8.5 Step 3 — Deploy the Lambda@Edge Function

```
# Install dependencies
npm install @aws-sdk/client-dynamodb @aws-sdk/util-dynamodb
zip -r registry-function.zip index.js node_modules/

# Create the Lambda function in us-east-1 (required for Lambda@Edge)
aws lambda create-function \
--function-name mcp-registry-edge \
--runtime nodejs20.x \
--role arn:aws:iam::ACCOUNT:role/mcp-registry-edge-role \
--handler index.handler \
--zip-file fileb://registry-function.zip \
--region us-east-1

# Publish a version (Lambda@Edge requires a versioned ARN)
aws lambda publish-version \
--function-name mcp-registry-edge \
--region us-east-1

# The IAM role needs: lambda.amazonaws.com + edgelambda.amazonaws.com principals
# and DynamoDB read permissions on the mcp-registry table
```

8.6 Step 4 — Create the CloudFront Distribution

Create a CloudFront distribution with your S3 bucket as origin. In the default cache behaviour, attach the Lambda@Edge function to the viewer request event using the versioned ARN from Step 3. Set the Lambda include body option to true — this is what allows the function to access the POST body. Configure your custom domain (mcp.example.com) with an ACM certificate in us-east-1. This can be done via the AWS Console or CloudFormation.

8.7 Step 5 — Publish the DNS Record

```
_mcp.example.com.   300   IN   TXT   "v=mcp1;
registry=https://mcp.example.com/registry;
public=true;
auth=https://auth.example.com/token;
version=2026-02"
```

8.8 Step 6 — Add Servers to the Registry

```
# Add a server entry to DynamoDB
aws dynamodb put-item \
--table-name mcp-registry \
--item '{
"server_id":      { "S": "crm-readonly" },
"name":           { "S": "CRM Read-Only" },
"url":            { "S": "https://crm-mcp.internal.example.com" },
```



```
"public":         { "BOOL": false },
"capabilities":   { "SS": ["accounts", "opportunities"] },
"data_residency": { "S": "eu-west-1" },
"deprecated":     { "BOOL": false }
}' \
--region us-east-1
```

9. Cost Analysis

One of the strongest arguments for this architecture is its cost profile. At any realistic traffic volume for an MCP registry, the monthly cost is negligible.

Component	Estimated Cost
Lambda@Edge	$0.60 per million invocations + $0.00005001/GB-second compute. At 100,000 queries/month with 128MB/50ms average: ~$0.09
DynamoDB Global Tables	On-demand reads: $0.283 per million read units. At 100,000 queries: ~$0.03. Global replication adds ~$0.08/GB storage.
CloudFront Data Transfer	First 1TB/month: $0.0085/GB. A registry response is ~5KB. 100,000 queries ≈ 500MB: ~$0.004
S3 (extended manifests)	Negligible. Storage costs pennies per GB; retrieval is infrequent.
Route 53 (DNS)	$0.50 per hosted zone per month. DNS queries: $0.40 per million.
Total (100K queries/month)	Under $2/month at typical registry volumes
Total (1M queries/month)	Under $5/month

These figures assume AWS standard pricing as of early 2026 and exclude free tier benefits, which reduce costs further. The figures are indicative rather than precise: actual cost depends on average Lambda@Edge execution duration, DynamoDB read unit consumption, and data transfer. In all realistic scenarios, monthly cost is well under $5. An organisation already using CloudFront for other purposes may have no incremental cost at all for the registry infrastructure.

The cost profile makes this architecture accessible to organisations of any size. A startup and a FTSE 100 company face the same infrastructure bill for their MCP registry. The governance maturity of how they manage the write path will differ; the hosting cost will not.

10. Relationship to Existing Work

10.1 The Official MCP Registry (registry.modelcontextprotocol.io)

Anthropic launched the official central MCP Registry in September 2025 at registry.modelcontextprotocol.io. It is the authoritative, globally searchable catalogue for publicly accessible MCP servers — the equivalent of a public telephone directory that any agent or developer can query by name or keyword.

This proposal solves a complementary and equally important problem: organisation-scoped discovery. The _mcp DNS record and self-hosted registry is analogous to a company's own corporate directory. Critically, an organisation can use it in two ways. With public=true, any agent that knows the organisation's domain can instantly discover and connect to the organisation's publicly exposed MCP servers — no central registry submission required. This is ideal for product catalogues, customer-facing tools, open data feeds, or any capability the organisation wants the world to find automatically. With public=false or selective authentication, the same mechanism securely surfaces internal-only servers to the organisation's own agents.

The two approaches coexist naturally and are explicitly encouraged by the official registry's own documentation, which recommends self-hosted registries for private servers. An organisation can publish its public MCP servers to both the central registry for maximum global visibility and its own _mcp registry for instant domain-based discovery. A well-designed agent will consult both: the _mcp record first when it already knows the domain, and the central registry for broader search. This is not competition — it is two pieces of the same ecosystem fitting together precisely as intended.

10.2 DNS Verification SEP (GitHub SEP #1959, December 2025)

A related proposal in the MCP community (SEP #1959, December 2025) proposes using _mcp._tcp SRV and TXT records at the individual server level for server identity verification and capability advertisement. The architectural distinction is precise: SEP #1959 is a per-server primitive — it answers the question 'is this specific server what it claims to be?' This paper is a bootstrap-to-registry pattern — it answers the question 'given a domain, what MCP ecosystem does this organisation expose?' The two layers are designed to stack. An agent can use the _mcp TXT record to discover an organisation's registry, use the registry to find a list of MCP servers, and then use SEP #1959-style per-server DNS records to verify the authenticity of each server before connecting. Neither proposal makes the other redundant; they operate at different levels of the discovery hierarchy.

10.3 NANDA

NANDA (Networked Agents and Decentralised Architecture), developed at MIT, addresses the same discovery problem through a federated directory protocol. Where NANDA introduces new trust hierarchies and new infrastructure components, this proposal composes existing infrastructure in a new configuration. The two are not mutually exclusive: a NANDA-style federation could be built on top of this discovery layer, with _mcp records providing the bootstrap hook.

10.4 Agent-to-Agent Protocol (A2A)

Google's Agent-to-Agent protocol addresses inter-agent communication — how one agent delegates to another. This paper addresses intra-session discovery — how an agent finds available tools. The two protocols operate at different layers and are complementary. An agent might use DNS-based MCP discovery to find a specialised agent, then use A2A to delegate a subtask to that agent.

10.5 WebFinger

WebFinger (RFC 7033) provides a similar mechanism for discovering information about entities identified by URI, using a well-known URL pattern. The _mcp DNS record proposal follows the same conceptual model — a well-known location returns structured metadata about a service — but operates at the DNS layer rather than the HTTP layer. DNS-layer discovery is more appropriate for agent-to-agent contexts where the discovery client may not have prior knowledge of the target's HTTP infrastructure.

10.6 Agent Peer Discovery: A Natural Extension

The registry model described in this paper is centralised by design. An organisation operates a registry; agents query it to discover available servers. This is the right model for enterprise MCP deployments where governance, access control, and auditability are primary concerns. But it is not the only possible model, and the DNS convention described here points naturally toward a complementary peer discovery layer that does not require a registry intermediary at all.

The extension is straightforward in concept. Rather than — or in addition to — publishing a registry at the URL advertised in the _mcp DNS record, an agent or service could publish its own capability manifest directly at a well-known HTTP path: /.well-known/mcp. Any other agent that knows the domain can query this path and discover what that agent can do, without consulting a central registry. The _mcp DNS record bootstraps the initial discovery; the /.well-known/mcp path serves the capability manifest directly. No intermediary required.

This maps to a peer-to-peer discovery model that several developments in the agent ecosystem are converging toward. Google's Agent-to-Agent protocol already anticipates agents calling each other directly rather than routing all interactions through MCP registries. The original vision of WebRTC — browsers connecting peer-to-peer for audio and video without a centralised media server — provides a useful historical parallel: the peer-to-peer capability was architecturally sound even if signalling infrastructure emerged in practice to handle NAT traversal and session coordination. Similarly, a peer discovery layer for agents does not eliminate the value of registries; it adds a complementary primitive that agents can use when they already know a domain and want to discover capabilities directly.

The data mesh parallel is equally instructive. In a data mesh architecture, data products are owned and published by the teams that produce them rather than registered in a central catalogue maintained by a platform team. Discoverability is a property of the data product itself, not a function of a central registry. The equivalent for agents would be agents owning and publishing their own capability manifests — making discoverability a property of the agent rather than a function of the registry it happens to be listed in. This shifts the governance model from registry-centric to agent-centric, with the _mcp DNS convention providing the common naming layer that makes peer discovery work across organisational boundaries.

The extension also generalises beyond agents. The same /.well-known/mcp path could advertise data products, knowledge bases, or any structured capability that another agent might need to discover — not just MCP servers in the current sense. As the agentic web matures, the boundary between an agent, a data product, and a knowledge source will blur; a discovery convention that treats all of these as first-class primitives is more durable than one scoped to the current MCP server taxonomy.

This peer discovery extension is explicitly out of scope for this paper, which addresses the registry bootstrap problem. It is identified here as a natural and important next layer — one that composes cleanly with the DNS convention and registry architecture described in the preceding sections rather than competing with them. The three layers stack: the _mcp DNS record bootstraps discovery of the registry, the registry provides governed, access-controlled enumeration of servers, and direct /.well-known/mcp paths enable peer-to-peer capability advertisement for agents that choose to publish themselves. Each layer is independently useful; together they form a complete discovery architecture for the agentic web.

11. IANA Considerations

This proposal does not define a new DNS record type. It uses the existing TXT record type with a well-known prefix convention — _mcp — following the pattern established by _dmarc, _domainkey, and similar service-specific DNS conventions. The _mcp label is an underscore-prefixed name as defined in RFC 8552, which scopes the interpretation of TXT records to the specific service identified by the underscore label. Implementers should note that RFC 8552 requires underscore labels to be registered or documented to avoid collisions. This proposal constitutes the documentation of _mcp for the purpose of the MCP registry bootstrap convention; formal IANA registration is identified as a future action below.

This proposal requests no new IANA registry entries at this time. If the convention achieves broad adoption and an IETF Working Group elects to standardise it, the following IANA considerations would apply:

- Registration of the _mcp DNS name prefix in an appropriate IANA registry of underscore names, following the process established by RFC 8552.
- Registration of the v=mcp1 version string in a new IANA registry for MCP DNS record versions.
- Definition of the authoritative IANA registry for MCP registry tool names (discover_servers, get_server_details, search_servers, check_server_health).

12. Live Reference Implementation: mcp.mariothomas.com

This paper is accompanied by a live reference implementation at mcp.mariothomas.com — a fully functional MCP registry and three MCP servers deployed on the architecture described in this paper and discoverable via a live _mcp DNS record. The implementation validates the architecture end-to-end and gives developers a working example they can query, fork, and test immediately.

The reference implementation uses path-based routing. All servers are served from a single CloudFront distribution with a single Lambda@Edge function routing on URL path.

Reference Implementation Endpoints (live at publication)
DNS record: _mcp.mariothomas.com
Registry: https://mcp.mariothomas.com/registry (public)
Articles MCP: https://mcp.mariothomas.com/articles (public)
Locations MCP: https://mcp.mariothomas.com/locations (public)
Documents MCP: https://mcp.mariothomas.com/documents (private — bearer token required)

The articles server exposes a queryable catalogue of published articles from mariothomas.com, filterable by tag and date range. The locations server exposes a log of cities visited with arrival and departure dates. The documents server exposes private advisory materials and is accessible only with a valid RS256 JWT bearer token, demonstrating the public/private access model described in Section 6.

Full source code is available at github.com/mariothomas/mcp-dns-registry. The implementation serves three purposes: to validate that the architecture works as specified; to provide a concrete reference that implementers can inspect and learn from; and to demonstrate the public/private access model with real data rather than hypothetical examples.

12.1 The DNS Record

The live DNS record is published at _mcp.mariothomas.com and takes the following form:

```
_mcp.mariothomas.com.   300   IN   TXT   "v=mcp1;
registry=https://mcp.mariothomas.com/registry;
public=true;
auth=https://auth.mariothomas.com/token;
version=2026-02"
```


Any agent that knows the domain mariothomas.com can discover the full MCP ecosystem from this single record. The TTL of 300 seconds balances agility — allowing rapid updates during the early life of the implementation — with resilience against the record being unavailable.

12.2 Registry DynamoDB Structure

The registry DynamoDB table contains three items corresponding to the three MCP servers in this implementation:

DynamoDB server_id	Server
server:articles	Published articles MCP server — public
server:locations	Cities and dates MCP server — public
Server:documents	Private documents MCP server — authentication required

An unauthenticated agent querying the registry receives the two public server entries. An agent presenting a valid bearer token receives all three. The Lambda@Edge function enforces this boundary by reading the request body, validating the JWT, and querying DynamoDB before any response is assembled.

12.3 Server 1: Published Articles (Public)

The articles MCP server exposes a queryable catalogue of published articles from mariothomas.com. The article catalogue is held in DynamoDB, the Lambda@Edge function assembles responses by scanning items with a server_type attribute of article, and no origin is consulted. A new article is added by writing a new DynamoDB item — no code changes, no redeployment. At demo scale, a full scan is appropriate; production deployments should index by server_type to avoid scan costs.

Tool	Description
list_articles	Returns all published articles with title, publication date, URL, tags, and estimated read time. Optional filters: tag, after (ISO date), before (ISO date).
get_article_summary	Returns the summary and key themes for a specific article identified by slug.

DynamoDB item structure for article entries:

```
Key:    article:agentic-ai-board-guide
Value:  {
"title": "Agentic AI: A Board Director's Guide",
"published": "2025-09-03",
"url": "https://mariothomas.com/blog/agentic-ai-board-guide",
```



```
"tags": ["agentic-ai", "board-governance", "ai-strategy"],
"read_time_minutes": 12,
"summary": "Explains agentic AI through the lens of the do-while loop..."
}
```

12.4 Server 2: Locations and Dates (Public)

The locations MCP server exposes a log of cities visited with corresponding dates. It demonstrates that the architecture is not limited to document or business data — any structured information can be surfaced to agents via an MCP server. The dataset is deliberately simple, which makes it an ideal entry point for developers examining the implementation for the first time.

Tool	Description
list_locations	Returns all locations with city, country, arrival date, and departure date. Supports filtering by country or date range.
get_location_detail	Returns full detail for a specific city visit including associated context such as conferences or engagements.

DynamoDB item structure for location entries:

```
Key:    location:2026-02-london
Value:  {
"city": "London",
"country": "United Kingdom",
"arrival": "2026-02-01",
"departure": "2026-02-28",
"context": "IoD board advisory; LSE guest lecture",
"public": true
}
```

The key naming convention — location:{YYYY-MM}-{city-slug} — enables efficient date-range filtering. The Lambda@Edge function scans DynamoDB items with a server_type attribute of location and applies date comparisons in memory. As with all scan-based queries in this reference implementation, production deployments should replace full scans with indexed queries.

12.5 Server 3: Private Documents (Private)

The documents MCP server is the most architecturally significant of the three. It demonstrates that the same serverless pattern used for public data can serve private documents with full access controls. The documents served are illustrative examples

clearly labelled as fictional — their purpose is to demonstrate the retrieval architecture, not to expose real sensitive information.

Tool	Description
list_documents	Returns a catalogue of available documents with title, type, year, and file size. Requires valid bearer token.
get_document_metadata	Returns full metadata for a specific document. Requires valid bearer token.

The retrieve_document tool is the critical case. The Lambda@Edge function generates a CloudFront signed URL for the corresponding S3 object using an embedded CloudFront private key. The signed URL is scoped to the specific object and expires after 15 minutes. The agent receives the URL and fetches the PDF directly from CloudFront — binary content never transits the MCP layer.

DynamoDB item structure for document entries:

```
Key:    doc:2024-tax-return
Value:  {
"title": "Tax Return — Demonstration Example 2024",
"type": "tax_return",
"year": 2024,
"file_size_kb": 245,
"s3_key": "documents/demo-tax-return-2024.pdf",
"public": false,
"description": "Fictional demonstration document. Not real tax data."
}
```

Security for the private server rests on three controls in sequence: RS256 JWT validation at the Lambda@Edge layer rejects unauthenticated requests before any DynamoDB query is made; the S3 bucket has no public access policy and is inaccessible without a signed URL; and the signed URL expires after 15 minutes, limiting the window of exposure if a URL were intercepted.

12.6 End-to-End Discovery Flow

The following sequence illustrates a complete authenticated flow — from domain name to PDF download — with no prior knowledge of the implementation:

1. Agent queries DNS: _mcp.mariothomas.com. Receives registry URL and auth endpoint.
2. Agent requests bearer token from auth.mariothomas.com/token using its credentials.

3. Agent calls tools/list on the registry with bearer token. Function validates JWT, queries DynamoDB for all three server items, returns full manifest including private documents server.
4. Agent connects to mcp.mariothomas.com/documents, calls tools/list. Receives catalogue of three available tools.
5. Agent calls list_documents. Function validates token, queries DynamoDB for all document items, returns catalogue.
6. Agent calls retrieve_document for a specific document. Function validates token, retrieves S3 key from DynamoDB item, generates 15-minute signed URL, returns it. Agent fetches PDF directly from CloudFront.

What the Live Implementation Demonstrates

- Public/private access control enforced at the edge — not at origin
- Dynamic registry via DynamoDB — no redeployment needed to add or modify servers
- Binary document retrieval via signed URLs — content never transits the MCP layer
- Complete discovery from domain name alone — no prior configuration required
- All three MCP servers share the same CloudFront distribution and Lambda@Edge function codebase
- Total infrastructure cost for the entire implementation: approximately $1/month
- Servers to provision, patch, or maintain: zero

13. Conclusion

The agent discovery problem is real, and it currently lacks a widely adopted, deployable enterprise pattern. As AI deployments scale from individual projects to organisation-wide infrastructure, the cost of manual configuration compounds. An agent that cannot discover what it can do is an agent that must be hand-configured by a human — which reintroduces exactly the scaling constraints that agentic AI is supposed to eliminate.

This paper has proposed a solution that requires no new infrastructure, no new protocols, and no new trust relationships. It requires one DNS record, one CloudFront distribution, one Lambda@Edge function of approximately 120 lines, and one DynamoDB Global Table. Any organisation with an AWS account and a registered domain can implement it in a day. Any organisation that implements it gains an auditable, governed, globally distributed catalogue of their AI agent capabilities that any compliant agent can discover from a domain name alone.

The architecture is deliberately conservative. It uses boring, proven technology in a new configuration rather than proposing exotic new primitives. In the context of enterprise AI governance — where boards and regulators are paying close attention to what AI systems can do and how that is controlled — boring technology that works is preferable to elegant technology that requires explanation.

The single most important architectural decision in the paper deserves emphasis. By making the registry itself a fully compliant MCP server — one whose tools are discover_servers, get_server_details, and search_servers — this proposal requires no new client behaviour whatsoever. An agent discovers a new organisation's entire capability ecosystem using exactly the same tools/list and tools/call calls it already makes against every other MCP server. There is no discovery SDK to install, no new protocol to implement, no special handling required. The composition is the contribution.

The proposal is available for implementation, critique, and extension. A reference implementation is available at mcp.mariothomas.com, including a live _mcp DNS record and a working registry serving public MCP servers. Feedback and collaboration are welcomed.

14. References

[1] Anthropic. (2024). Model Context Protocol Specification. https://spec.modelcontextprotocol.io

[2] MIT CSAIL. (2025). NANDA: Networked Agents and Decentralised Architecture. Project NANDA.

[3] Mockapetris, P. (1987). Domain Names — Implementation and Specification. RFC 1035. IETF.

[4] Gulbrandsen, A., Vixie, P., & Esibov, L. (2000). A DNS RR for Specifying the Location of Services. RFC 2782. IETF.

[5] Crocker, D. (2011). Scoped Interpretation of DNS Resource Records for Defined Underscored Node Names. RFC 8552. IETF.

[6] Jones, M., & Sakimura, N. (2013). WebFinger. RFC 7033. IETF.

[7] Amazon Web Services. (2025). Lambda@Edge Developer Guide. AWS Documentation.

[8] Amazon Web Services. (2025). Amazon DynamoDB Global Tables Developer Guide. AWS Documentation.

[8a] Amazon Web Services. (2025). CloudFront Signed URLs Developer Guide. AWS Documentation.

[9] Amazon Web Services. (2025). Amazon Bedrock AgentCore Documentation. AWS Documentation.

[10] Thomas, M. (2025). Agentic AI: A Board Director's Guide. mariothomas.com.

[11] Google. (2025). Agent-to-Agent Protocol Specification. Google DeepMind.

Acknowledgements

The author thanks Jonathan Jenkyn for peer review that clarified the scope of the DNS convention with respect to authentication and authorisation, and Jonathan Taws for substantive feedback on registry security controls and the agent peer discovery extension. Cole Johnston provided valuable feedback on tool versioning, context window constraints in large registries, and the read-path audit trail, each of which has been addressed in this version

.

About the Author

Mario Thomas is Head of Applied AI & Emerging Technology Strategy at Amazon Web Services (AWS) and a Chartered Director and Fellow of the Institute of Directors. He operates at the intersection of AI governance, board advisory, and strategic transformation, helping boards move from tactical AI project approvals to comprehensive AI governance.

Mario guest lectures at the London School of Economics on their Data Science and AI for Executives programme and maintains an independent thought leadership practice at mariothomas.com, publishing the Board in the Machine newsletter. He serves as an AWS press spokesperson on AI and emerging technology.

The views expressed in this paper are the author's own and do not represent the position of Amazon Web Services or any other organisation.

Contact: mario@mariothomas.com
Web: mariothomas.com

