ChatGPT

Swift-Based DNS Solution Proposal for FountainAI

Version: 1.0
Date: August 5, 2025
Author: Contexter
Scope: Replace the CoreDNS dependency in FountainAI's hybrid DNS architecture with a self-hosted DNS server written entirely in Swift 6. The public domain `fountain.coach` remains on a managed provider; only the delegated sub-zone used for service discovery is served by the new Swift implementation.

---

1. Executive Summary

FountainAI currently uses a hybrid DNS setup: the public zone `fountain.coach` resides on a hosted provider (Hetzner or Route 53), and an internal sub-zone (e.g. `internal.fountain.coach`) is delegated to a **self-hosted CoreDNS** instance. While CoreDNS is lightweight and production-ready, it is written in Go. To simplify the stack and align the entire reasoning engine around **Swift 6**, this proposal replaces CoreDNS with a custom DNS server built on **SwiftNIO**.

SwiftNIO is "a cross-platform asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients" [1]. By leveraging SwiftNIO's UDP/TCP primitives, FountainAI can implement an authoritative DNS server that serves zone files, reloads changes on the fly and exposes a matching OpenAPI control plane—all in the same language used by its micro-services. The goals are to:

- Maintain the **hybrid architecture**: keep `fountain.coach` on a globally distributed provider while delegating `internal.fountain.coach` to a local server. This ensures high availability for users and complete control over internal names.
- Achieve **100 % service continuity** for internal name resolution with no external dependencies [2].
- Unify the codebase under **Swift 6**, enabling shared libraries and cohesive tooling.
- Provide **GitOps-friendly** zone storage, programmable control via **OpenAPI**, and support for DNS-01 ACME challenges.

---

2. Architecture Overview

2.1 Components

Component	Description
Public DNS Provider	Managed DNS service (Hetzner, Route 53, etc.) authoritative for `fountain.coach`. It hosts records for the web front and delegates the internal sub-zone via NS records.
Swift DNS Server ("FountainDNS")	A custom authoritative DNS server implemented in Swift 6 using SwiftNIO. It listens on UDP/TCP port 53 (or a custom port), serves zone files for `internal.fountain.coach`, and reloads changes dynamically.
Zone Store	YAML or JSON files representing DNS zone data. These files are stored locally and optionally version-controlled. Each file defines the records for a sub-zone.
DNS API Layer	An HTTP server implemented exclusively with SwiftNIO. It exposes OpenAPI-defined operations to list zones, add/update/delete records and trigger reloads. It writes zone files and signals the DNS server to reload.
FountainAI Clientgen	Swift 6 client generator that creates type-safe bindings for the DNS API so that micro-services and the orchestrator can manage DNS records programmatically.
Gateway / Reverse Proxy	Terminates TLS for the public front (`www.fountain.coach`), resolves internal names via the Swift DNS server and routes traffic accordingly.
ACME Client	Performs DNS-01 challenges for certificate issuance. It uses the DNS API to create `_acme-challenge` TXT records and remove them after validation [3].

2.2 Flow Diagram

```
Public Internet
     ↓
[Public DNS Provider] —— answers queries for fountain.coach and delegates
internal sub-zone
     ↓
[Gateway / Reverse Proxy] —— resolves internal services via FountainDNS
     ↓
[FountainDNS] ←— watches → [Zone files on disk]
     ↑             ↑
     |             |
[DNS API Layer] —— manages zone files (OpenAPI)
     ↑
[FountainAI Clientgen / ACME Client]
```

3. Functional Goals

The Swift-based solution must achieve parity with the previous CoreDNS-based design. The key capabilities are:

Feature	Specification
Zone Delegation	Configure the public provider to delegate `internal.fountain.coach` via NS records pointing to the Swift DNS server(s).
Zone Management	Create, list and delete zones under the delegated domain through the HTTP API.
Record Management	Add, update and delete A/AAAA/CNAME/MX/TXT/SRV/CAA records. TXT support is essential for DNS-01 ACME challenges.
Reload Trigger	Reload zone data without restarting the server. The DNS server must watch for file changes or respond to explicit reload requests.
Git Integration	Store zone files in a Git repository for declarative management and change history.
OpenAPI Spec	Provide a full OpenAPI 3.1 definition for the HTTP API so that clients can be generated automatically [4].
Internal DNSSEC (optional)	Implement DNSSEC signing of internal zones if necessary.

---

4. Justification for a Swift Implementation

The existing fallback solution uses CoreDNS because it is lightweight and production-ready [5]. However, adopting a pure Swift implementation offers several benefits:

1. **Unified technology stack.** FountainAI's micro-services and orchestrator are written in Swift 6. Implementing DNS in Swift eliminates the need to maintain a Go binary and simplifies cross-compilation and deployment pipelines.
2. **Leverage SwiftNIO for high-performance networking.** SwiftNIO is explicitly designed for high-performance protocol servers; it is a "cross-platform asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients" [1]. This makes it suitable for implementing a protocol like DNS, which uses both UDP and TCP and benefits from non-blocking I/O.
3. **Extensibility through Swift packages.** Additional features such as DNSSEC, metrics or DNS-over-TLS can be added as Swift modules, aligning with FountainAI's modular architecture.
4. **Better integration with Swift concurrency.** The server can use Swift 6's `async/await` and structured concurrency along with SwiftNIO's event loops to manage requests efficiently.
5. **Reduced external dependencies.** Self-hosting eliminates rate limits and zone restrictions imposed by hosted providers [6] [7] and ensures offline operation [2].

---

5. Technical Design

5.1 Overview

The custom DNS server (hereafter **FountainDNS**) will consist of two primary subsystems:

1. **DNS Query Engine (UDP/TCP)** – built on SwiftNIO. It listens on UDP and TCP ports (default 53) for DNS queries, parses DNS messages according to RFC 1035, and responds with records from zone files. Running on top of SwiftNIO ensures high throughput and non-blocking I/O [1].

2. **HTTP Control Plane** – implemented using SwiftNIO's HTTP modules. This component provides the OpenAPI-defined endpoints for managing zones and records and signalling the DNS engine to reload.

The DNS engine and control plane can run in the same process or separate processes communicating via IPC or file system watchers. For simplicity, the MVP runs them in a single process with a shared `ZoneManager` responsible for reading/writing zone files and caching records in memory.

5.2 DNS Query Engine

1. **Network bootstrap:** Use `DatagramBootstrap` for UDP and `ServerBootstrap` for TCP to listen on port 53 or a configurable port. Each incoming datagram or stream is handled by a `ChannelHandler` that decodes DNS queries.
2. **Message parsing:** Implement a DNS message parser conforming to RFC 1035. The handler should read the header, questions and EDNS0 options (if present), then assemble a response with the appropriate answer, authority and additional sections based on zone data.
3. **Record lookup:** The engine consults the in-memory cache maintained by `ZoneManager`. If the query matches a record (e.g. A, AAAA, CNAME, MX, TXT, SRV), it returns the record with the configured TTL. Unmatched queries return NXDOMAIN or are forwarded to an upstream resolver if you choose to support recursion (not required for an authoritative server).
4. **Reload mechanism:** When zone files change or when the control plane triggers a reload, the engine reloads the in-memory cache. Swift's concurrency primitives (e.g. `Actor`) can ensure thread-safe updates.

5.3 Zone Storage

- **File format:** Store zone files as YAML (e.g. `internal.fountain.coach.yaml`) or as RFC 1035 zone files. YAML is easy to serialize/deserialize in Swift using the `Yams` package and aligns with the existing GitOps workflow.
- **Structure:** Each file contains zone metadata and a list of records, including fields like `name`, `type`, `value`, `ttl`, `priority`, `weight` and `port` (for SRV). The control plane persists changes by rewriting these files.
- **Version control:** Committing the zone files to Git provides an audit trail and allows roll-backs.

5.4 HTTP Control Plane

- **Framework:** Use SwiftNIO's `HTTPServerPipelineHandler` and related HTTP modules to implement an HTTP API. Each endpoint defined in the OpenAPI spec maps to a handler function.
- **Operations:** Endpoints include listing zones, creating zones, listing records, upserting records, deleting records and triggering a reload. Request bodies are validated against JSON Schema generated from the OpenAPI definition.

- **Authentication:** Protect endpoints using API keys, bearer tokens or mutual TLS to prevent unauthorized changes.

5.5 ACME Integration

Let's Encrypt's DNS-01 challenge requires the ability to add a TXT record to `_acme-challenge.<domain>` [3]. Because the control plane supports adding TXT records, an ACME client (Lego, Certbot or a custom Swift implementation) can request a certificate for both public and internal hostnames by:

1. Calling `upsertRecord` with a TXT record containing the challenge string.
2. Triggering a reload via the API or waiting for the file watcher to update the DNS engine.
3. Completing the ACME challenge and then deleting the TXT record via the API.

This process is identical to the CoreDNS-based solution and ensures the new server remains compatible with existing automation.

5.6 DNSSEC and DoT/DoH (Optional)

FountainDNS can optionally implement DNSSEC signing and support DNS-over-TLS/HTTPS (DoT/DoH). SwiftNIO provides TLS support through `swift-nio-ssl` [8], enabling encrypted DNS on port 853. DNSSEC would require key management and record signing; this could be implemented using a Swift cryptography library in a later iteration.

---

6. OpenAPI Interface

The control plane API remains nearly identical to the hybrid CoreDNS proposal. Below is the **OpenAPI 3.1** specification. Paths are versioned under `/v1/` to allow evolution.

```
openapi: 3.1.0
info:
  title: FountainDNS Control API
  version: 1.0.0
  description: |
    API for managing the internal fountain.coach delegated zone via the
Swift-based DNS server.

Supports listing zones, managing records (including TXT for ACME DNS-01), and
triggering reloads.
servers:
  - url: http://localhost:8000
paths:
  /v1/zones:
    get:
      summary: List all managed zones
      operationId: listZones
      responses:
        '200':
          description: A list of zones
```

```yaml
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Zone'
    post:
      summary: Create a new zone under the delegated domain
      operationId: createZone
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Zone'
      responses:
        '201':
          description: Zone created
  /v1/zones/{zoneName}/records:
    parameters:
      - name: zoneName
        in: path
        required: true
        schema:
          type: string
    get:
      summary: List records in a zone
      operationId: getZoneRecords
      responses:
        '200':
          description: List of DNS records
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/DNSRecord'
    post:
      summary: Add or update a DNS record
      operationId: upsertRecord
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/DNSRecord'
      responses:
        '200':
          description: Record upserted
    delete:
      summary: Remove a DNS record
      operationId: deleteRecord
      parameters:
        - name: recordId
          in: query
          required: true
          schema:
            type: string
      responses:
        '204':
          description: Record deleted
  /v1/zones/{zoneName}/reload:
    post:
      summary: Trigger a reload of the DNS server for this zone
      operationId: reloadZone
      parameters:
        - name: zoneName
          in: path
          required: true
          schema:
            type: string
      responses:
        '202':
          description: Reload triggered
components:
  schemas:
    Zone:
      type: object
      required:
        - name
      properties:
        name:
          type: string
          description: Fully qualified zone name (e.g.
internal.fountain.coach)
        records:
          type: array
          items:
            $ref: '#/components/schemas/DNSRecord'
    DNSRecord:
      type: object
      required:
        - type
        - name
        - value
      properties:
        id:
          type: string
          description: Unique identifier for the record (optional on create)
        type:
          type: string
          enum: [A, AAAA, CNAME, TXT, MX, NS, SRV, CAA]
        name:
          type: string
          description: Relative record name (e.g. service for
service.internal.fountain.coach)
        value:
          type: string
        ttl:
          type: integer
          default: 3600
          description: Time to live in seconds
        priority:
          type: integer
          description: MX/SRV priority where applicable
        weight:
          type: integer
          description: SRV record weight where applicable
        port:
          type: integer
          description: SRV record port where applicable
```

Notes

- The API does not manage records in the public `fountain.coach` zone; those continue to reside on the hosted provider.
- After upserting or deleting records, clients should call `reloadZone` to ensure the DNS engine picks up changes.

---

7. Implementation Recommendations

Component	Guidance
DNS Engine	Use SwiftNIO's `DatagramBootstrap` and `ServerBootstrap` to listen on UDP/TCP. Implement a `ChannelHandler` to decode/encode DNS messages. Cache zone data in an `Actor` for thread-safe access.
Zone Manager	Represent zone data in Swift structs. Provide functions to load from disk (YAML/JSON), apply edits and write back. Use `FileWatcher` (e.g. `swift-nio-file-system` or `DispatchSourceFileSystemObject`) to detect external changes and trigger reloads.
HTTP Server	Build the API exclusively with SwiftNIO's HTTP modules. Validate requests against the OpenAPI schema and respond with appropriate status codes. Protect the API using API keys or mTLS.
ACME Client	Continue using an ACME client (e.g. Certbot with a DNS-hook script). Alternatively, implement a lightweight ACME client in Swift using `swift-nio-ssl` for TLS operations.

Component	Guidance
Testing	Use SwiftNIO's `EmbeddedChannel` and `EmbeddedEventLoop` to test the DNS handlers without network I/O. Write integration tests to ensure zone updates reflect in DNS responses.
Performance	Enable caching and efficient data structures for rapid query responses. Use concurrency features to handle multiple requests concurrently.
Metrics and logging	Integrate with a metrics system (e.g. Prometheus) by exposing counters for queries, errors and reloads. Use structured logging via SwiftLog.

8. Deployment

1. **Delegate the internal zone:** On the public DNS provider, create NS records for `internal.fountain.coach` pointing to the IP addresses of the FountainDNS servers. For redundancy, run multiple instances on different hosts and include all of them in the NS set.
2. **Build and run FountainDNS:** Compile the Swift server on your Linux/macOS host. Configure it to listen on port 53 (requires elevated privileges) or on a non-privileged port (e.g. 1053) and update NS records accordingly [9].
3. **Deploy the DNS API:** Run the HTTP control plane on port 8000. Ensure the process has write access to the zone store and can signal the DNS engine. Restrict access via firewall rules and authentication.
4. **Configure resolvers:** On FountainAI hosts, point the resolver (e.g. `/etc/resolv.conf`) at the FountainDNS server for internal domains. External queries will be handled by the system's existing resolver chain.
5. **Certificate issuance:** Set up an ACME client that uses the DNS API to create and remove TXT records for DNS-01 challenges. For public hosts, you may continue using HTTP-01 if preferred.
6. **GitOps pipeline:** Store zone files in a Git repository. A CI/CD job can validate changes, apply them by calling the API and monitor for successful reloads.

9. Future Enhancements

- **DNSSEC:** Implement zone signing using Swift cryptographic libraries and serve DNSSEC records.
- **DoT/DoH:** Add support for DNS-over-TLS and DNS-over-HTTPS using `swift-nio-ssl` [8].
- **Dynamic service registration:** Allow services to register themselves directly with FountainDNS via gRPC or a custom protocol.
- **Clustered mode:** Share zone state across multiple instances using a distributed cache or consensus protocol for high availability.

10. Conclusion

Transitioning from CoreDNS to a **Swift-based DNS server** retains the advantages of the previous fallback solution—cloud independence, offline operation and programmable control [2]—while unifying the FountainAI platform under a single language. SwiftNIO provides the foundation for building high-performance protocol servers [1], enabling FountainAI to handle DNS queries and control operations efficiently. By delegating the internal zone to this custom server and exposing an

OpenAPI-defined API, we achieve a secure, GitOps-friendly and extensible service discovery layer tailored to FountainAI’s needs.

---

[1] [8] GitHub - apple/swift-nio: Event-driven network application framework for high performance protocol servers & clients, non-blocking.
https://github.com/apple/swift-nio

[2] [4] [5] GitHub
https://github.com/Fountain-Coach/codex-deployer/blob/main/Proposals/ CoreDNS Fallback Proposal for FountainAI.txt

[3] Challenge Types - Let's Encrypt
https://letsencrypt.org/docs/challenge-types/

[6] DNS Overview - Hetzner Docs
https://docs.hetzner.com/dns-console/dns/general/dns-overview/

[7] Hetzner DNS Console | DNSControl
https://docs.dnscontrol.org/provider/hetzner

[9] CoreDNS: DNS and Service Discovery
https://coredns.io/manual/configuration/