React API Integration Options

You are a **senior cloud/solutions architect with deep hands-on experience in:**

- AWS networking (VPC, ALB, Route 53, ECS on Fargate, Cloud Map)
- Web application architecture (React SPA frontends, Python backends)
- Enterprise identity and access management (Azure Entra ID / Azure AD, OAuth2/OIDC for SPAs and APIs)
- Browser security (CORS, cookies, same-origin policy, secure token handling)
- Multi-environment configuration management (dev/uat/prod) and CI/CD for containerized apps

I will describe my architecture and constraints. Then I want you to propose **all reasonable options for how my **React UI** should call my **Python backend API**, and finally give a **clear recommendation**, with **granular implementation details**.**

1. Current and Planned Architecture (Context)

1.1 Core Stack

- **Cloud provider: AWS**
- **Container orchestration: ECS **on Fargate****
- **Cluster topology:**
** - A **single ECS cluster****
** - Two separate ECS services:**
**   - `ECS-UI` — hosts the **React** SPA**
**   - `ECS-API` — hosts the **Python** backend (e.g., FastAPI/Flask/Django-style REST API)**
- **Load balancing:**
** - One **Application Load Balancer (ALB)** shared between UI and API services**
** - Path-based routing on the same ALB:**
**   - `//ui` (or root `/` if you recommend) → **UI target group → ECS-UI****
**   - `/api/*` → **API target group → ECS-API****
- **ALB exposure:**
** - **Internal-only ALB** (not internet-facing)**

  - Route 53 **private hosted zone** with a **CNAME** (or A/alias) record pointing to the ALB
- **DNS / domains**:
  - Will use **custom domains** via Route 53 Hosted Zones
  - ALB will be accessed through these custom domain names, not via the raw ALB DNS
- **No API Gateway or CloudFront**:
  - Explicit constraint: **Do NOT introduce**:
    - Amazon API Gateway
    - Amazon CloudFront
  - The design should be based on **ALB + ECS + Route 53 + ECS Service Discovery/Cloud Map (if relevant)**

1.2 Environments

- Environments: **dev**, **uat**, **prod**
- Expect **separate ECS services and ALBs per environment**, with their own Route 53 records (you can suggest detailed naming conventions, e.g. `app-dev.internal.example.com`, etc.)
- Requirement: **No hardcoding of URLs** inside the React app code for each environment; configuration must be systematically managed.

1.3 Identity & Security

- **Identity Provider (IdP)**: **Microsoft Azure Entra ID** (Azure AD)
- **Frontend**: React SPA (browser-based client)
- **Backend**: Python API service
- Requirements:
  - Use **Azure Entra** for user authentication (OIDC/OAuth2) for the React SPA
  - Use access tokens (e.g., JWT) to authenticate calls from the SPA to the backend
  - Cover:
    - **CORS vs same-origin approach**
    - **Cookie usage vs Authorization headers**
    - **Cookie security flags** (`Secure`, `HttpOnly`, `SameSite`)
    - Token storage best practices for SPAs (avoid XSS, CSRF, etc.)

1.4 Service Discovery and Internal Communication

- ECS launch type: **Fargate**
- We are using / planning to use **ECS Service Discovery / AWS Cloud Map** for **internal service-to-service calls**.
- I want a **dedicated section** that compares and contrasts:
  - Using **ALB** vs using **Cloud Map** for internal service

discovery and communication between services
- How each affects:
- Latency
- Reliability
- Operability
- Security
- Flexibility
- And **explicitly call out why **browsers (React UI)** cannot directly use Cloud Map, and what that implies for architecture choices.**

2. Key Question

I need to decide **how the React UI should call the backend Python API in this environment, considering:**

- Low latency
- Clean separation of concerns
- Secure integration with **Azure Entra authentication**
- Ease of deployment and environment promotion (dev → uat → prod**) with:**
- Minimal or **no code changes between environments**
- **No hardcoded URLs in the React application**
- Simple and maintainable DevOps story (CI/CD pipelines, configuration, secrets management)
- Using **ALB and ECS only (no API Gateway / CloudFront), but you MAY leverage:**
- Route 53 (public and/or private)
- Internal ALB
- ECS Service Discovery / Cloud Map for internal microservice-to-microservice communication (server-side)

3. What I Want From You (Output Requirements)

Organize your answer into **clear sections with headings. Use diagrams (ASCII if helpful), tables, and code/config examples where it clarifies.**

3.1 Explicitly List and Describe UI → API Call Patterns

Describe at least the following **main patterns and any other relevant ones:**

Pattern A — Same-Origin Path-Based API (Recommended candidate if appropriate)
- React UI and API share **the same domain and origin**, using different paths:
  - e.g. `https://app-dev.internal.example.com/` → UI
  - `https://app-dev.internal.example.com/api/...` → API
- UI calls the API using **relative URLs** such as `/api/...`.
- ALB does path-based routing:
  - `/` or `/ui` → ECS-UI target group
  - `/api/*` → ECS-API target group

For **Pattern A**, provide:

1. **DNS and ALB configuration details**:
   - How to configure Route 53 zones and records (e.g., private hosted zones, `A`/alias vs `CNAME`)
   - How to configure ALB listeners and path-based rules for UI and API
   - How to attach ECS services (target groups, health checks, ports)

2. **React implementation details**:
   - How to structure API calls (using relative paths like `/api/...` instead of absolute URLs)
   - How this simplifies configuration (no need to know host/port at runtime)
   - How to handle different base paths if UI is served from `/ui` vs `/`

3. **Configuration management across dev/uat/prod**:
   - If using same-origin relative URLs, what config is still needed in React?
   - How to manage environment-specific non-URL configuration (`REACT_APP_*` env vars, or runtime config file)
   - Recommended approaches:
     - Build-time configuration via `.env` files and CI/CD injecting env vars
     - **Runtime configuration** pattern: a `/config/config.json` hosted by the UI container and fetched on app startup (to avoid rebuilds just for URL changes)

4. **Authentication & security**:
   - How to integrate **Azure Entra ID** with this pattern:
     - Using an SPA library like MSAL.js (or describe at conceptual level)
     - OIDC code flow with PKCE from the React app
     - Obtaining an **access token** for the API (configured as an app

registration / scope in Entra)
  - How to send the access token to the API:
    - In `Authorization: Bearer ` headers from the browser to `/api/...`
  - Same-origin implications:
    - Typically **no CORS** required if truly same-origin
    - If any CORS is needed (e.g., subtle differences in domains/ports), spell out the exact `Access-Control-*` headers and where to configure them.

5. **Pros and cons** of Pattern A:
  - Pros:
    - Simpler DNS and networking
    - Typically **no CORS configuration** needed
    - Easier SPAs (relative URLs only)
    - Good for cookie-based auth if ever desired
    - Clean path-based routing with ALB
  - Cons:
    - Tighter coupling of UI and API deployment behind same ALB and domain
    - Constraints on future scaling where you might want separate API domains
  - Comment on **latency**: ALB-level overhead, connection reuse, internal VPC-level latency.

Pattern B — Separate Subdomain for API (But Still Same ALB or Another ALB)
- UI served at one host, API at another host, for example:
  - `https://app-dev.internal.example.com` → UI
  - `https://api-dev.internal.example.com` → API
  - Both could still terminate on the **same internal ALB** or on **separate internal ALBs**.
- React calls API through **absolute URLs** (e.g. `https://api-dev.internal.example.com/...`).

For **Pattern B**, cover:

1. **DNS and routing options**:
  - Using multiple Route 53 records mapping to same ALB (via host-based routing)
  - Or separate ALBs for UI and API
  - Internal-only aspects (private hosted zones, network connectivity requirements for clients)

2. **React config**:
  - How to avoid hardcoding:
    - Build-time environment variables

(`REACT_APP_API_BASE_URL`)
- Runtime config JSON
- Show a concrete example of config structure and how React reads it (e.g. `window.__APP_CONFIG__` or `config.json`).

3. **CORS & security:**
- If UI and API are on **different origins (even if both internal), explain:**
- When CORS is required
- Exact CORS policies to configure on the Python backend:
- Allowed origins (per env)
- Allowed methods, headers
- Credentials handling
- Differences if using:
- **Bearer tokens in Authorization header, vs**
- **Cookies for session/auth (and how `SameSite`, `Secure`, `HttpOnly` interact with cross-site requests)**
- How to correctly integrate Entra-issued tokens:
- Configure the API as a resource/app registration
- Validate JWT tokens in Python (e.g., via middleware)
- Best practices around scopes/audiences

4. **Pros and cons relative to Pattern A:**
- More flexibility in independent scaling and DNS
- Requires CORS and more complex configuration
- More moving parts for environment promotion (`dev/uat/prod` DNS and config)
- Discuss any latency implications (DNS, TLS, connection reuse).

Pattern C — Internal Service Discovery (Cloud Map) and Where It Fits

- Explain in **detail:**
- How **ECS Service Discovery / Cloud Map works:**
- Service registry
- DNS names like `api.service.local` or `api.dev.svc.cluster.local` (use accurate AWS examples)
- How ECS tasks register/deregister
- Where Cloud Map is appropriate:
- **Internal, server-side communication (e.g., one ECS service calling another within the VPC)**
- Why **browsers cannot directly use Cloud Map:**
- Private VPC-only DNS names
- Not resolvable from client networks
- Security and routing implications

Then:

1. Compare **using ALB vs using Cloud Map for **internal service-to-service communication**, with a **dedicated subsection** and a **comparison table**:**
- **Dimensions:**
- Latency
- Health checking
- Traffic control / routing rules
- Observability (access logs, metrics)
- Operational complexity
- Resilience and fault handling
- Security controls (security groups, auth at app level)
- Explain typical patterns:
- API-to-API calls going via internal ALB vs via Cloud Map name directly (and how each is configured).

2. Provide a **clear statement on:**
- Cloud Map is **NOT for direct browser traffic**
- UI → API should be via:
- ALB + DNS (Route 53), i.e., HTTPS endpoints from the browser
- Cloud Map is useful behind the scenes **only between backend services.**

3. If relevant, describe a **hybrid pattern:**
- Public/internal ALB for browser traffic into a “gateway”/edge service
- Internal microservices behind Cloud Map that the gateway calls.
- But do **NOT introduce API Gateway/CloudFront (respect the constraint); stay within ALB + ECS.**

3.2 Detailed Implementation Guidance

For each pattern, provide **step-by-step, concrete implementation details, including:**

1. **ALB and ECS configuration:**
- Listener configuration (HTTP/HTTPS, ports)
- Path-based / host-based routing rules
- Target group settings (health check paths, thresholds)
- Example AWS CLI / Terraform / CloudFormation snippets are welcome but not required; pseudo-config is fine.

2. **Route 53 setup:**

- Example zone and record configuration for `dev`, `uat`, `prod`
- Specifically for **internal ALBs and **private hosted zones****
- Suggestions on naming conventions (e.g. `app-dev.internal.example.com`, `api-dev.internal.example.com`).

3. **React app configuration management:**
- Pattern 1: Build-time `.env` with `REACT_APP_*` variables:
- How to structure per-environment env files
- How CI/CD pipelines inject these values
- Pattern 2: Runtime `config.json`:
- Example `config.json` structure holding `apiBaseUrl` and other settings
- Example React code that fetches `config.json` before rendering the app
- How to mount/inject the config file at container startup without rebuilding the image
- Explain **trade-offs between build-time vs runtime configuration for environment promotion and speed of change.**

4. **Python backend configuration:**
- How to:
- Configure the base path (`/api`)
- Handle CORS in frameworks like FastAPI, Flask, Django REST (conceptual, with example snippets)
- Validate Azure Entra JWT tokens:
- JWKS endpoint configuration
- Audience/scope validation
- Token lifetime and refresh considerations

5. **Authentication with Azure Entra ID:**
- SPA flow:
- User → React UI → Entra login → redirect back with code → token exchange → store tokens in memory or secure storage
- Best practice token storage for SPAs:
- Pros/cons of localStorage/sessionStorage vs in-memory storage with refresh tokens in secure cookies
- Backend API:
- Configure Entra app registration for the API (expose scopes)
- Validate incoming tokens on each request
- How this impacts CORS, same-origin, and cookie settings if any cookies are used.

3.3 Non-Functional Considerations

Explicitly analyze for each pattern:

- **Latency and performance:**
** - Extra ALB hops?**
** - DNS lookup overhead (and mitigation via caching)**
** - Keep-alive and connection reuse behavior**
- **Scalability:**
** - How easy it is to independently scale UI vs API**
** - ECS service autoscaling and impact on routing**
- **Reliability & fault isolation:**
** - What happens if API is down but UI is up, and vice versa**
- **Security:**
** - TLS termination points**
** - Network segmentation (security groups, subnets)**
** - Principle of least privilege (ALB → ECS security groups, etc.)**
- **Operational simplicity:**
** - How easy is it to reason about and debug?**
** - Logs (ALB access logs vs app logs)**
** - Health checks and rollout/rollback strategies (e.g., blue/green deployments with ECS)**

3.4 Final Recommendation

After walking through all the patterns:

1. Provide a **clear, opinionated recommendation for **my specific scenario**, taking into account:**
**   - Internal-only ALB**
**   - Route 53 private hosted zones**
**   - Fargate-based ECS**
**   - dev/uat/prod environments**
**   - Use of Azure Entra ID**
**   - Strong preference for:**
**   - Low latency**
**   - Minimal hardcoding**
**   - Quick integration and deployments**
**   - Simple, maintainable DevOps**

2. Summarize in a **short comparison table:**
**   - Rows: Pattern A, Pattern B, any others you propose**
**   - Columns: Latency, Complexity, Security, CORS overhead, Config management complexity, Recommended? (Yes/No)**

3. Provide a **practical migration or implementation plan:**

- If I'm starting from scratch, which pattern should I implement first?
- If I later want to evolve (e.g., from same-origin to separate subdomain), what is a smooth transition path?

4. Style & Depth

- Assume the reader is **technical and familiar with AWS, but wants **architect-level clarity** and **implementation-ready detail**.**
- Use:
- Clear headings and subheadings
- Occasional small diagrams or request/flow sequences
- Short, focused code examples where they aid understanding
- Avoid generic textbook-level explanations; **focus all your reasoning on the concrete scenario and constraints described above.**

Claude Sonnet 4.5

Comprehensive Architecture Guide: React SPA + Python API on AWS ECS with Azure Entra ID

Executive Summary

You're building a multi-tier application with a React frontend and Python backend, both containerized on ECS Fargate, fronted by an internal ALB, authenticated via Azure Entra ID, across dev/uat/prod environments. The core architectural decision is **how the browser-based React app calls the Python API**.

My recommendation: **Pattern A (Same-Origin Path-Based)** for your scenario, with runtime configuration for the React app and Cloud Map reserved strictly for future internal service-to-service communication.

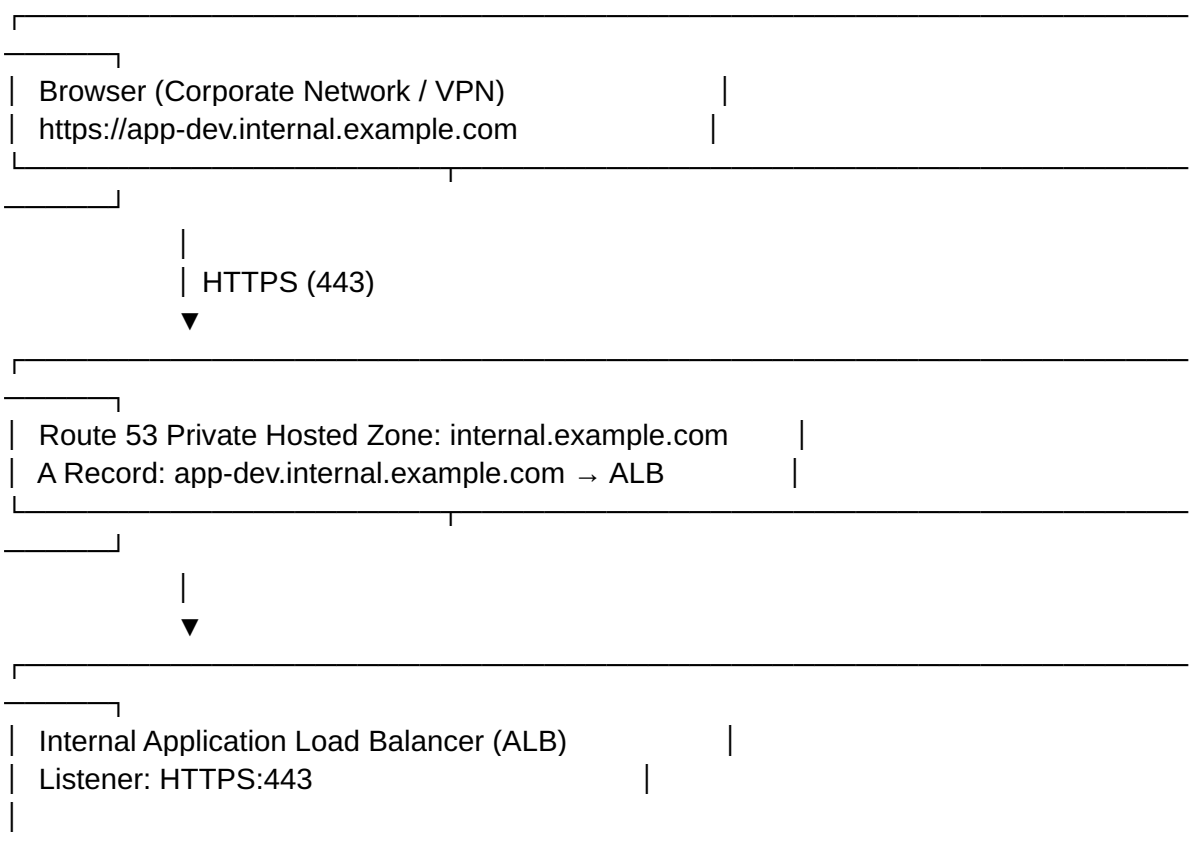
This document provides exhaustive implementation details for all viable patterns, a dedicated Cloud Map analysis, and concrete guidance for Azure Entra integration.

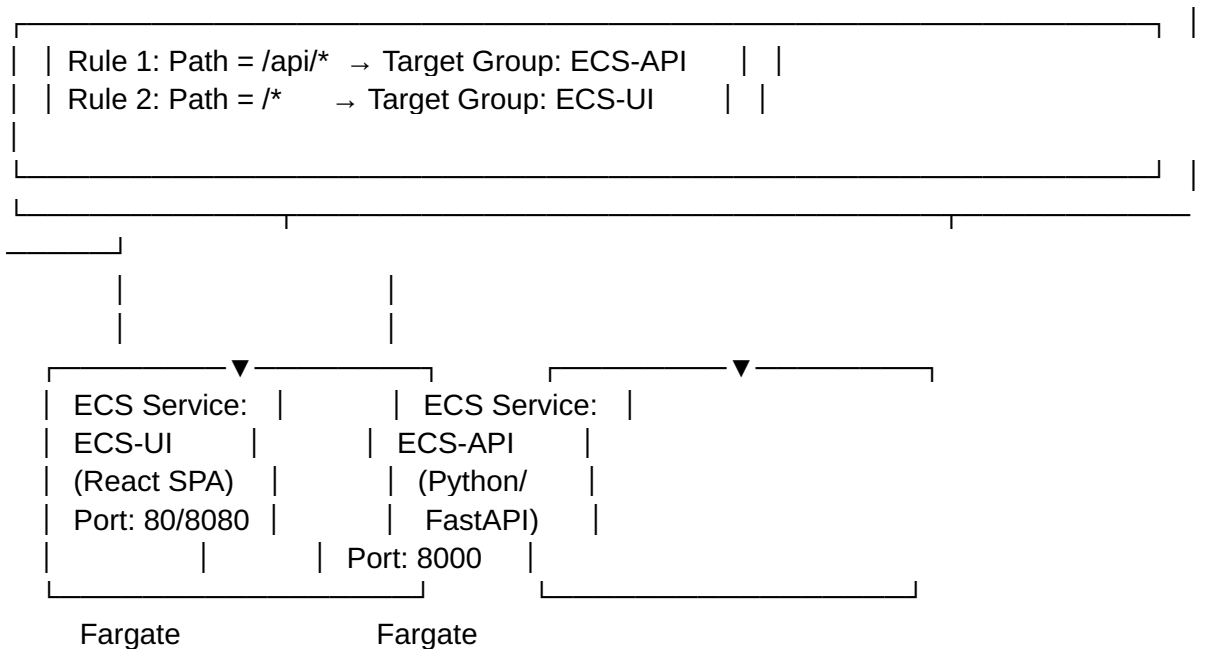
Table of Contents

---

1. Pattern A: Same-Origin Path-Based API RECOMMENDED

1.1 Architecture Overview





Key principle: The browser sees a **single origin** (https://app-dev.internal.example.com). All API calls use **relative paths** (/api/users, /api/orders). The ALB routes based on path prefix.

1.2 DNS and ALB Configuration

1.2.1 Route 53 Setup

Private Hosted Zone (one per environment or shared):

Zone: internal.example.com (Private, associated with VPC)

Records:

- app-dev.internal.example.com → A (Alias) → ALB-dev
- app-uat.internal.example.com → A (Alias) → ALB-uat
- app-prod.internal.example.com → A (Alias) → ALB-prod

Why A (Alias) instead of CNAME?

- Alias records are free, have better performance, and can point to AWS resources (ALB) at the zone apex if needed.
- Use Alias for ALBs; it's the AWS best practice.

Naming convention:

- app-{env}.internal.example.com for the unified UI+API endpoint
- Alternatively: {app-name}-{env}.internal.example.com if you have multiple apps

1.2.2 ALB Configuration

Listener Configuration:

```
Listener: HTTPS:443
  Certificate: ACM certificate for *.internal.example.com (or specific cert per env)
  Default Action: Forward to ECS-UI target group (fallback)

  Rules (evaluated in order):
    Priority 1:
      Condition: Path is /api/*
      Action: Forward to Target Group: TG-API-dev

    Priority 2 (default):
      Condition: Path is /*
      Action: Forward to Target Group: TG-UI-dev
```

Target Group: TG-UI-dev

```
Target Type: IP (for Fargate)
Protocol: HTTP
Port: 8080 (or whatever your React container serves on)
VPC: your-vpc
Health Check:
  Path: /health or /index.html
  Interval: 30s
  Timeout: 5s
  Healthy threshold: 2
  Unhealthy threshold: 2
  Matcher: 200
Deregistration delay: 30s
```

Target Group: TG-API-dev

```
Target Type: IP
Protocol: HTTP
Port: 8000 (FastAPI default)
VPC: your-vpc
Health Check:
  Path: /api/health
```

Interval: 30s
Timeout: 5s
Healthy threshold: 2
Unhealthy threshold: 2
Matcher: 200
Deregistration delay: 30s

ECS Service Integration:

- Each ECS service (ECS-UI, ECS-API) is configured with a loadBalancers block pointing to its respective target group.
- ECS automatically registers/deregisters Fargate task IPs as they scale.

Security Groups:

ALB Security Group (SG-ALB-dev):
Inbound:

- HTTPS (443) from Corporate CIDR or VPN CIDR

Outbound:

- HTTP (8080) to SG-ECS-UI
- HTTP (8000) to SG-ECS-API

ECS-UI Security Group (SG-ECS-UI):
Inbound:

- HTTP (8080) from SG-ALB-dev

Outbound:

- HTTPS (443) to 0.0.0.0/0 (for pulling images, etc.)

ECS-API Security Group (SG-ECS-API):
Inbound:

- HTTP (8000) from SG-ALB-dev

Outbound:

- HTTPS (443) to 0.0.0.0/0
- Database ports to RDS/Aurora SG if applicable

1.3 React Implementation Details

1.3.1 API Call Structure

No hardcoded URLs. Use relative paths:

```
// src/api/client.js
const API_BASE = '/api';  // Relative path

export async function fetchUsers(token) {
  const response = await fetch(`${API_BASE}/users`, {
    method: 'GET',
    headers: {
      'Authorization': `Bearer ${token}`,
      'Content-Type': 'application/json'
    }
  });
  return response.json();
}
```

Why this works:

- Browser is at https://app-dev.internal.example.com/
- Fetch to /api/users resolves to https://app-dev.internal.example.com/api/users
- Same origin → no CORS preflight needed
- ALB routes /api/* to the API target group

1.3.2 Handling Different Base Paths

If you serve UI from /ui instead of /:

ALB Rule:

```
Priority 1: Path = /api/*  → TG-API
Priority 2: Path = /ui/*   → TG-UI
Priority 3: Path = /       → Redirect to /ui (or serve a landing page)
```

React Router Configuration:

```
// src/App.js
import { BrowserRouter } from 'react-router-dom';

function App() {
  return (
```

```
    <BrowserRouter basename="/ui">
      {/* Routes */}
    </BrowserRouter>
  );
}
```

Nginx/Static Server in UI Container:

```
# nginx.conf
server {
  listen 8080;
  root /usr/share/nginx/html;

  location /ui {
    alias /usr/share/nginx/html;
    try_files $uri $uri/ /ui/index.html;
  }
}
```

Recommendation: Serve UI from / (root) for simplicity. Reserve /api for the backend. This avoids basename complexity in React Router.

1.4 Configuration Management Across Environments

1.4.1 The Problem

You need to avoid:

- Rebuilding the React Docker image for each environment
- Hardcoding environment-specific values in code

1.4.2 Solution 1: Build-Time Environment Variables (Simple, Less Flexible)

Structure:

```
my-react-app/
├── .env.dev
├── .env.uat
├── .env.prod
├── src/
└── Dockerfile
```

.env.dev:

```
REACT_APP_ENV=dev
REACT_APP_ENTRA_CLIENT_ID=abc123-dev
REACT_APP_ENTRA_TENANT_ID=your-tenant-id
REACT_APP_ENTRA_REDIRECT_URI=https://app-dev.internal.example.com

# No API URL needed if using relative paths!
```

.env.uat:

```
REACT_APP_ENV=uat
REACT_APP_ENTRA_CLIENT_ID=def456-uat
REACT_APP_ENTRA_TENANT_ID=your-tenant-id
REACT_APP_ENTRA_REDIRECT_URI=https://app-uat.internal.example.com
```

CI/CD Pipeline (e.g., GitHub Actions, GitLab CI):

```
# .github/workflows/deploy-ui.yml

- name: Build React App for Dev
  run: |
    cp .env.dev .env
    npm run build
    docker build -t my-react-app:dev .

- name: Push to ECR
  run: |
    docker tag my-react-app:dev 123456789.dkr.ecr.us-east-1.amazonaws.com/my-react-app:dev
    docker push ...
```

Pros:

- Simple, standard React pattern
- No runtime config fetching

Cons:

- Separate Docker image per environment (my-react-app:dev, my-react-app:uat, etc.)
- Cannot promote the same image artifact across environments
- Slower feedback loop (rebuild required for config changes)

1.4.3 Solution 2: Runtime Configuration (RECOMMENDED for Multi-Env)

Concept: Bake a **generic** React build into the Docker image. At container startup, inject a config.json file that the React app fetches before rendering.

Structure:

```
my-react-app/
├── public/
│   └── config.json.template
├── src/
│   ├── config.js
│   └── App.js
├── Dockerfile
└── entrypoint.sh
```

public/config.json.template:

```
{
  "env": "${ENV}",
  "entraClientId": "${ENTRA_CLIENT_ID}",
  "entraTenantId": "${ENTRA_TENANT_ID}",
  "entraRedirectUri": "${ENTRA_REDIRECT_URI}",
  "apiBaseUrl": "/api"
}
```

Dockerfile:

```
FROM node:18 AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
```

```
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
COPY --from=build /app/public/config.json.template /usr/share/nginx/html/
config.json.template
COPY entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

EXPOSE 8080
ENTRYPOINT ["/entrypoint.sh"]
CMD ["nginx", "-g", "daemon off;"]
```

entrypoint.sh:

```
#!/bin/sh
set -e

# Substitute environment variables into config.json
envsubst < /usr/share/nginx/html/config.json.template > /usr/share/nginx/html/config.json

# Start nginx
exec "$@"
```

ECS Task Definition (dev environment):

```
{
  "family": "ecs-ui-dev",
  "containerDefinitions": [
    {
      "name": "ui",
      "image": "123456789.dkr.ecr.us-east-1.amazonaws.com/my-react-app:latest",
      "environment": [
        { "name": "ENV", "value": "dev" },
        { "name": "ENTRA_CLIENT_ID", "value": "abc123-dev" },
        { "name": "ENTRA_TENANT_ID", "value": "your-tenant-id" },
        { "name": "ENTRA_REDIRECT_URI", "value": "https://app-dev.internal.example.com" }
      ],
      "portMappings": [{ "containerPort": 8080 }]
    }
  ]
}
```

React Code to Load Config:

src/config.js:

```
let config = null;

export async function loadConfig() {
  if (config) return config;

  const response = await fetch('/config.json');
  config = await response.json();
  return config;
}

export function getConfig() {
  if (!config) {
    throw new Error('Config not loaded. Call loadConfig() first.');
  }
  return config;
}
```

src/index.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { loadConfig } from './config';

const root = ReactDOM.createRoot(document.getElementById('root'));

loadConfig()
  .then(() => {
    root.render(<App />);
  })
  .catch(err => {
    console.error('Failed to load config:', err);
    root.render(<div>Configuration error. Please contact support.</div>);
  });
```

src/App.js:

```
import { getConfig } from './config';

function App() {
  const config = getConfig();

  // Use config.entraClientId, config.apiBaseUrl, etc.
  return <div>App for {config.env}</div>;
}
```

Benefits:

- **Single Docker image** promoted across dev → uat → prod
- Configuration injected at deployment time via ECS task definition environment variables
- Fast config changes (update task definition, redeploy—no rebuild)
- Aligns with immutable infrastructure principles

Trade-offs:

- Slightly more complex setup (entrypoint script, config loading in React)
- Config is publicly readable (but that's acceptable for client-side apps; secrets should never be in the frontend)

1.5 Authentication & Security

1.5.1 Azure Entra ID Integration (MSAL.js)

App Registrations in Azure Entra:

1. **UI App Registration** (SPA):
   - Name: MyApp-UI-Dev
   - Platform: Single-page application
   - Redirect URIs: https://app-dev.internal.example.com
   - Implicit grant: **Disabled** (use Auth Code Flow with PKCE)
   - API permissions: Delegated permissions to the API app registration (see below)
2. **API App Registration**:
   - Name: MyApp-API-Dev
   - Expose an API:
     - Application ID URI: api://myapp-api-dev
     - Scopes: api://myapp-api-dev/access_as_user
   - App roles (optional, for RBAC)

React MSAL Configuration:

```
// src/authConfig.js
import { getConfig } from './config';

export function getMsalConfig() {
  const config = getConfig();
```

```
  return {
    auth: {
      clientId: config.entraClientId,
      authority: `https://login.microsoftonline.com/${config.entraTenantId}`,
      redirectUri: config.entraRedirectUri,
    },
    cache: {
      cacheLocation: 'sessionStorage', // or 'localStorage'
      storeAuthStateInCookie: false,
    }
  };
}

export const loginRequest = {
  scopes: ['api://myapp-api-dev/access_as_user']
};
```

src/index.js (with MSAL):

```
import { PublicClientApplication } from '@azure/msal-browser';
import { MsalProvider } from '@azure/msal-react';
import { loadConfig } from './config';
import { getMsalConfig } from './authConfig';

const root = ReactDOM.createRoot(document.getElementById('root'));

loadConfig()
  .then(() => {
    const msalInstance = new PublicClientApplication(getMsalConfig());
    return msalInstance.initialize().then(() => msalInstance);
  })
  .then(msalInstance => {
    root.render(
      <MsalProvider instance={msalInstance}>
        <App />
      </MsalProvider>
    );
  })
  .catch(err => {
    console.error('Initialization error:', err);
  });
```

Protected API Call:

```
// src/api/client.js
import { useMsal } from '@azure/msal-react';
```

```
import { loginRequest } from '../authConfig';

export function useApiClient() {
  const { instance, accounts } = useMsal();

  async function callApi(endpoint, options = {}) {
    // Acquire token silently
    const request = {
      ...loginRequest,
      account: accounts[0]
    };

    const response = await instance.acquireTokenSilent(request);
    const token = response.accessToken;

    return fetch(`/api${endpoint}`, {
      ...options,
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json',
        ...options.headers
      }
    });
  }

  return { callApi };
}
```

1.5.2 Token Storage Best Practices

Recommended approach for SPAs:

- **Access tokens**: Store in **memory** (JavaScript variable, React state/context)
    - Pros: Not vulnerable to XSS-based token theft from localStorage
    - Cons: Lost on page refresh
- **Refresh tokens**: MSAL.js handles this internally using sessionStorage or localStorage (configurable)
    - MSAL automatically refreshes access tokens using the refresh token

Do NOT:

- Store access tokens in localStorage or sessionStorage if you can avoid it (XSS risk)
- Use cookies for access tokens in SPAs (complicates CORS, and cookies are better for server-rendered apps)

For this pattern (same-origin):

- Since UI and API are same-origin, you *could* use HttpOnly cookies for session management, but:
    - Azure Entra's SPA flow is designed for bearer tokens
    - Mixing cookie-based auth with Entra tokens is non-standard
- **Stick with bearer tokens in Authorization headers**

1.5.3 CORS Configuration

Do you need CORS?

No, if:

- UI and API are truly same-origin (same scheme, domain, port as seen by the browser)
- In Pattern A, the browser sees https://app-dev.internal.example.com for both UI and API

Edge case: If your UI container serves on port 8080 and API on 8000, but the ALB terminates HTTPS and forwards to both, the browser only sees port 443 (HTTPS). So still same-origin.

If you do need CORS (e.g., during local development where UI is localhost:3000 and API is localhost:8000):

FastAPI Example:

```
# main.py
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

# For local dev
origins = [
    "http://localhost:3000",
    "https://app-dev.internal.example.com",
    "https://app-uat.internal.example.com",
    "https://app-prod.internal.example.com",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
```

```
# If using cookies
    allow_methods=["*"],
    allow_headers=["*"],
)
```

For production (Pattern A): You can **remove or disable CORS middleware** entirely, since same-origin requests don't trigger CORS.

1.5.4 Python Backend Token Validation

FastAPI Example with Azure Entra JWT Validation:

```
# auth.py
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import jwt, JWTError
import requests

security = HTTPBearer()

TENANT_ID = "your-tenant-id"
CLIENT_ID = "api-client-id"

# API app registration client ID
JWKS_URI = f"https://login.microsoftonline.com/{TENANT_ID}/discovery/v2.0/keys"

# Cache JWKS keys
_jwks_cache = None

def get_jwks():
    global _jwks_cache
    if not _jwks_cache:
        _jwks_cache = requests.get(JWKS_URI).json()
    return _jwks_cache

def verify_token(credentials: HTTPAuthorizationCredentials = Depends(security)):
    token = credentials.credentials

    try:
        # Decode header to get kid
        unverified_header = jwt.get_unverified_header(token)
        kid = unverified_header['kid']

        # Find the right key
        jwks = get_jwks()
```

```
        key = next((k for k in jwks['keys'] if k['kid'] == kid), None)
        if not key:
            raise HTTPException(status_code=401, detail="Invalid token key")

        # Verify and decode
        payload = jwt.decode(
            token,
            key,
            algorithms=['RS256'],
            audience=f"api://{CLIENT_ID}",

# Must match your API's App ID URI
            issuer=f"https://login.microsoftonline.com/{TENANT_ID}/v2.0"
        )

        return payload

# Contains user info, roles, etc.

    except JWTError as e:
        raise HTTPException(status_code=401, detail=f"Token validation failed: {str(e)}")

# Usage in endpoints
@app.get("/api/users")
def get_users(token_payload: dict = Depends(verify_token)):
    user_id = token_payload.get('oid')

# Object ID of the user
    # ... business logic
    return {"users": [...]}
```

Key points:

- Validate aud (audience) matches your API's App ID URI
- Validate iss (issuer) matches Azure Entra
- Validate signature using JWKS from Microsoft
- Extract user identity from oid, preferred_username, or roles claims

---

1.6 Pros and Cons of Pattern A

Aspect	Pros	Cons
Simplicity	Single domain, relative URLs, no CORS	Tighter coupling of UI and API behind same ALB
Configuration	Minimal (no API URL needed in React)	Still need Entra client IDs per env
Latency	Single DNS lookup, single TLS handshake	ALB adds ~1-3ms per request (negligible)
Security	Same-origin = simpler security model	All services share same domain (less isolation)
Scalability	Can independently scale ECS services	ALB is a single point; must scale ALB if needed
DevOps	Path-based routing is standard ALB feature	Requires careful ALB rule ordering
Future flexibility	Easy to add more services under / service-name	Harder to split API to separate domain later

Latency deep-dive:

- ALB adds ~1-3ms of latency per request (AWS internal benchmarks)
- Connection reuse: ALB maintains connection pools to backend targets (HTTP/1.1 keep-alive or HTTP/2)
- DNS caching: After first lookup, browser caches DNS for TTL duration (typically 60s)
- TLS session resumption: Reduces handshake overhead on subsequent requests

Verdict: For internal apps with moderate traffic (<10k RPS), ALB latency is negligible. Pattern A is **optimal for your use case**.

---

2. Pattern B: Separate Subdomain for API

2.1 Architecture Overview

Browser
  ├─ https://app-dev.internal.example.com (UI)

└─ https://api-dev.internal.example.com (API)
   │
   ├─ Option 1: Same ALB, host-based routing
   └─ Option 2: Separate ALBs

Key difference: UI and API are on **different origins** (different subdomains). This triggers CORS.

2.2 DNS and Routing Options

Option 2A: Same ALB, Host-Based Routing

Route 53:

app-dev.internal.example.com → A (Alias) → ALB-dev
api-dev.internal.example.com → A (Alias) → ALB-dev (same ALB)

ALB Listener Rules:

Listener: HTTPS:443
  Certificate: *.internal.example.com (wildcard)

  Rules:
    Priority 1:
      Condition: Host is api-dev.internal.example.com
      Action: Forward to TG-API-dev

    Priority 2:
      Condition: Host is app-dev.internal.example.com
      Action: Forward to TG-UI-dev

Pros:

- Single ALB to manage
- Shared infrastructure cost

Cons:

- Both services share ALB capacity (though ALB auto-scales)
- Less isolation

Option 2B: Separate ALBs

Route 53:

app-dev.internal.example.com → A (Alias) → ALB-UI-dev
api-dev.internal.example.com → A (Alias) → ALB-API-dev

Pros:

- Complete isolation of UI and API traffic
- Independent scaling and monitoring
- Easier to apply different WAF rules, rate limits, etc.

Cons:

- Higher cost (2 ALBs per environment)
- More infrastructure to manage

2.3 React Configuration

Runtime config.json:

```
{
  "env": "dev",
  "apiBaseUrl": "https://api-dev.internal.example.com",
  "entraClientId": "abc123-dev",
  "entraTenantId": "your-tenant-id",
  "entraRedirectUri": "https://app-dev.internal.example.com"
}
```

API Client:

```
// src/api/client.js
import { getConfig } from '../config';

const config = getConfig();
const API_BASE = config.apiBaseUrl;  // Absolute URL

export async function fetchUsers(token) {
  const response = await fetch(`${API_BASE}/users`, {
    method: 'GET',
```

```
    headers: {
      'Authorization': `Bearer ${token}`,
      'Content-Type': 'application/json'
    },
    credentials: 'include'  // If using cookies; omit for bearer tokens
  });
  return response.json();
}
```

Key difference: apiBaseUrl is now an **absolute URL** and must be configured per environment.

2.4 CORS & Security

2.4.1 CORS is Required

Since https://app-dev.internal.example.com (UI) is calling https://api-dev.internal.example.com (API), the browser sees this as **cross-origin** and will:

1. Send a **preflight OPTIONS request** for non-simple requests (e.g., with Authorization header)
2. Check Access-Control-Allow-Origin in the response

2.4.2 Python Backend CORS Configuration

FastAPI:

```
from fastapi.middleware.cors import CORSMiddleware
import os

app = FastAPI()

# Load allowed origins from environment variable
ALLOWED_ORIGINS = os.getenv('ALLOWED_ORIGINS', '').split(',')

# Example: ALLOWED_ORIGINS=https://app-dev.internal.example.com,https://app-uat.internal.example.com

app.add_middleware(
```

```
    CORSMiddleware,
    allow_origins=ALLOWED_ORIGINS,
    allow_credentials=False,

# True only if using cookies
    allow_methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"],
    allow_headers=["Authorization", "Content-Type"],
    max_age=3600,

# Cache preflight for 1 hour
)
```

ECS Task Definition (API):

```
{
  "environment": [
    {
      "name": "ALLOWED_ORIGINS",
      "value": "https://app-dev.internal.example.com"
    }
  ]
}
```

Flask:

```
from flask_cors import CORS

app = Flask(__name__)
CORS(app, origins=["https://app-dev.internal.example.com"], supports_credentials=False)
```

2.4.3 Bearer Tokens vs Cookies

Bearer Tokens (Recommended for Pattern B):

- Send token in Authorization: Bearer <token> header
- No need for credentials: 'include' in fetch
- CORS config: allow_credentials=False
- Simpler, stateless

Cookies (If you must):

- API sets cookies with:
  ```python
  response.set_cookie(
  'session_id',
  ```

```
    value='...',
    secure=True,
```

HTTPS only

```
    httponly=True,
```

Not accessible via JS

```
    samesite='None',
```

Required for cross-site cookies

```
    domain='.internal.example.com'
```

Share across subdomains

```
)
```

- React fetch: credentials: 'include'
- CORS config: allow_credentials=True, allow_origins must be **explicit** (not *)

Recommendation: Stick with **bearer tokens** for Azure Entra integration. Cookies add complexity with SameSite=None and require HTTPS everywhere.

2.4.4 Azure Entra Token Validation

Same as Pattern A (see section 1.5.4). The backend validates the JWT regardless of how it's delivered (header or cookie).

2.5 Pros and Cons of Pattern B

Aspect	Pattern A (Same-Origin)	Pattern B (Separate Subdomain)
CORS	Not needed	Required; adds complexity
Configuration	Minimal (relative URLs)	Must configure API URL per env
Flexibility	Coupled	Independent UI and API domains
Latency	1 DNS lookup, 1 TLS handshake	2 DNS lookups, 2 TLS handshakes (if separate ALBs)
Security	Simpler (same-origin)	More isolation, but CORS attack surface
Scalability	Shared ALB	Can use separate ALBs for isolation
DevOps	Simpler	More moving parts (2 DNS records, CORS config)

When to use Pattern B:

- You plan to expose the API to **other clients** (mobile apps, third-party integrations) that aren't on the same domain
- You want **complete isolation** of UI and API infrastructure
- You have **multiple UIs** (e.g., admin portal, customer portal) calling the same API

For your scenario: Pattern A is simpler and sufficient unless you have the above requirements.

---

3. Pattern C: Cloud Map & Internal Service Discovery

3.1 What is AWS Cloud Map?

AWS Cloud Map (part of ECS Service Discovery) is a **service registry** that:

- Automatically registers ECS tasks as they start
- Creates DNS records in a **private namespace** (e.g., service.local)
- Provides **service-to-service discovery** within the VPC

Example:

- Namespace: dev.svc.local (private DNS namespace)

- Service: api
- Full DNS name: api.dev.svc.local
- Resolves to: IP addresses of all healthy ECS tasks running the API service

---

3.2 How Cloud Map Works with ECS

ECS Service Definition:

```
{
  "serviceName": "api-service",
  "taskDefinition": "api-task",
  "desiredCount": 3,
  "launchType": "FARGATE",
  "networkConfiguration": { ... },
  "serviceRegistries": [
    {
      "registryArn": "arn:aws:servicediscovery:us-east-1:123456789:service/srv-abc123",
      "containerName": "api",
      "containerPort": 8000
    }
  ]
}
```

Cloud Map Service:

```
{
  "Name": "api",
  "NamespaceId": "ns-xyz789",  // Points to dev.svc.local namespace
  "DnsConfig": {
    "DnsRecords": [
      { "Type": "A", "TTL": 10 }
    ]
  },
  "HealthCheckCustomConfig": {
    "FailureThreshold": 1
  }
}
```

Result:

- ECS automatically registers each task's IP with Cloud Map
- DNS query for api.dev.svc.local returns all task IPs (A records)
- Client-side load balancing (or use SRV records for port info)

3.3 Why Browsers Cannot Use Cloud Map

Critical limitation: Cloud Map DNS names (e.g., api.dev.svc.local) are **only resolvable within the VPC**.

Why:

1. **Private DNS namespace**: Cloud Map creates Route 53 private hosted zones associated with your VPC
2. **VPC DNS resolver**: Only EC2 instances, ECS tasks, Lambda functions, etc. **inside the VPC** can resolve these names
3. **Browsers are outside the VPC**: Even if users are on a corporate network with VPN access to the VPC, their DNS queries go to corporate DNS servers, not the VPC DNS resolver

Implications:

- **Browsers cannot directly call** https://api.dev.svc.local/users
- You **must** use an ALB (or NLB) with a Route 53 record that browsers can resolve

Workaround (not recommended):

- Set up DNS forwarding from corporate DNS to Route 53 Resolver endpoints in the VPC
- Complex, adds latency, and defeats the purpose of Cloud Map (which is for internal service mesh)

3.4 ALB vs Cloud Map: Detailed Comparison

Dimension	ALB (Application Load Balancer)	Cloud Map (Service Discovery)
Use Case	External or internal **client-to-service** (including browsers)	**Service-to-service** within VPC
DNS Resolution	Public or private Route 53 records; resolvable by browsers	Private VPC-only DNS; **not resolvable by browsers**
Load Balancing	Layer 7 (HTTP/HTTPS); advanced routing (path, host, headers)	DNS-based (returns multiple IPs); client-side load balancing
Health Checks	Active health checks by ALB; automatic target deregistration	ECS task health checks; Cloud Map updates DNS based on task state
Latency	+1-3ms per request (ALB processing)	Direct task-to-task; ~0.5ms lower latency
Connection Pooling	ALB maintains pools to targets; efficient reuse	Client must manage connections
TLS Termination	ALB terminates TLS; can offload from tasks	Tasks must handle TLS (or use service mesh like App Mesh)
Observability	ALB access logs, CloudWatch metrics, request tracing	Requires application-level logging; less built-in observability
Traffic Control	Weighted target groups, blue/green, canary deployments	Requires custom logic or service mesh
Security	Security groups, WAF integration, TLS policies	Security groups only; no WAF
Cost	~$16-25/month per ALB + $0.008/LCU-hour	~$0.50/month per service + $0.0000001/query (negligible)
Operational Complexity	Moderate (ALB config, target groups, listeners)	Low (ECS handles registration automatically)
Failure Handling	ALB retries failed targets, drains connections	Client must implement retries
Best For	Browser → API, External → Internal, API Gateway pattern	Microservice mesh, internal API-to-API calls

---

3.5 When to Use Each

Use ALB When:

- **Browsers or external clients** need to access the service
- You need **advanced routing** (path-based, host-based, header-based)
- You want **centralized observability** (access logs, metrics)
- You need **TLS termination** without modifying application code
- You want **managed health checks and failover**

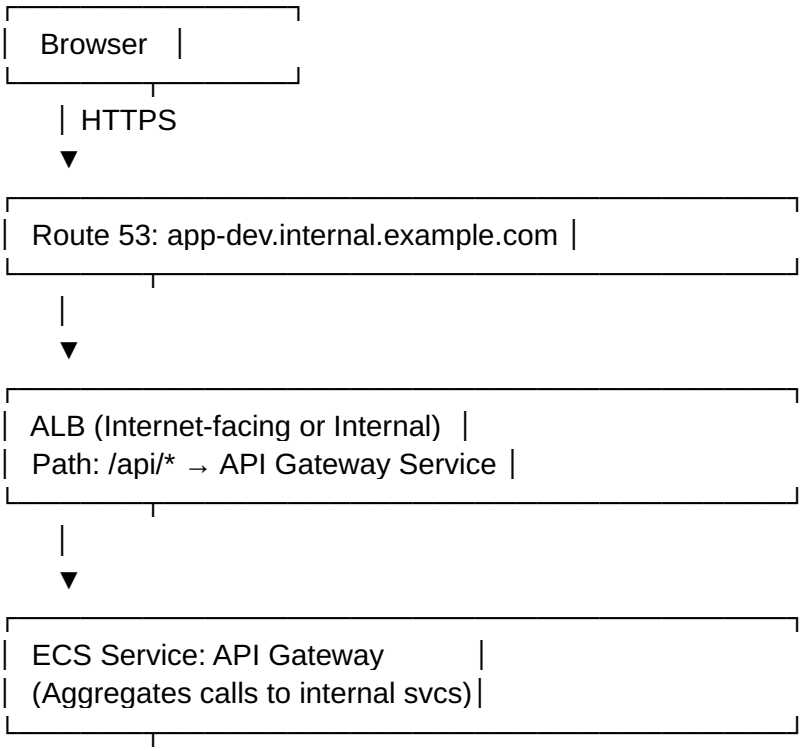
Use Cloud Map When:

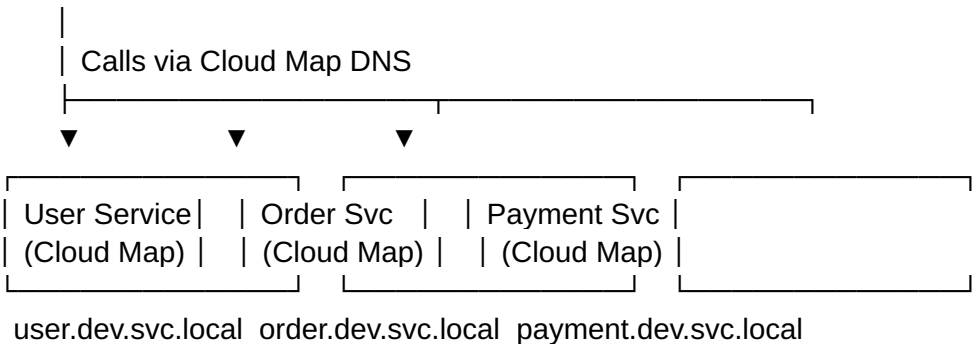
- **Internal service-to-service** communication (e.g., API → Database Proxy, API → Auth Service)
- You want **lowest latency** (direct task-to-task, no ALB hop)
- You're building a **microservices architecture** with many small services
- You're using a **service mesh** (AWS App Mesh integrates with Cloud Map)
- You want **dynamic service discovery** without manual DNS updates

---

3.6 Hybrid Pattern: ALB for Browsers, Cloud Map for Internal Services

Architecture:





How it works:

1. Browser calls https://app-dev.internal.example.com/api/orders → ALB → API Gateway service
2. API Gateway service (Python/FastAPI) calls internal services:
   ```python
   # Inside API Gateway service
   import httpx

async def get_user(user_id: str):
# Resolves via Cloud Map
response = await httpx.get(f"http://user.dev.svc.local:8000/users/{user_id}")
return response.json()
```

1. Cloud Map resolves user.dev.svc.local to task IPs
2. Direct task-to-task communication (no ALB hop)

Benefits:

- **Best of both worlds**: ALB for browser access, Cloud Map for low-latency internal calls
- **Scalability**: Internal services can scale independently without ALB limits
- **Security**: Internal services not exposed via ALB; only accessible within VPC

When to use this:

- You have **multiple backend microservices** that need to call each other
- You want to **minimize latency** for internal calls
- You're building a **service-oriented architecture**

For your current scenario (single React UI + single Python API):

- **Not needed yet**
- Stick with Pattern A (ALB for both UI and API)
- **Introduce Cloud Map later** when you split the API into multiple microservices

---

3.7 Cloud Map Configuration Example

Create Namespace:

```
aws servicediscovery create-private-dns-namespace \
  --name dev.svc.local \
  --vpc vpc-abc123 \
  --region us-east-1
```

Create Service:

```
aws servicediscovery create-service \
  --name api \
  --namespace-id ns-xyz789 \
  --dns-config "DnsRecords=[{Type=A,TTL=10}]" \
  --health-check-custom-config FailureThreshold=1
```

ECS Service with Service Discovery:

```
{
  "cluster": "my-cluster",
  "serviceName": "api-service",
  "taskDefinition": "api-task:1",
  "desiredCount": 3,
  "launchType": "FARGATE",
  "networkConfiguration": {
    "awsvpcConfiguration": {
      "subnets": ["subnet-abc", "subnet-def"],
      "securityGroups": ["sg-api"],
      "assignPublicIp": "DISABLED"
    }
  },
  "serviceRegistries": [
    {
      "registryArn": "arn:aws:servicediscovery:us-east-1:123456789:service/srv-abc123"
    }
  ]
}
```

Testing from another ECS task:

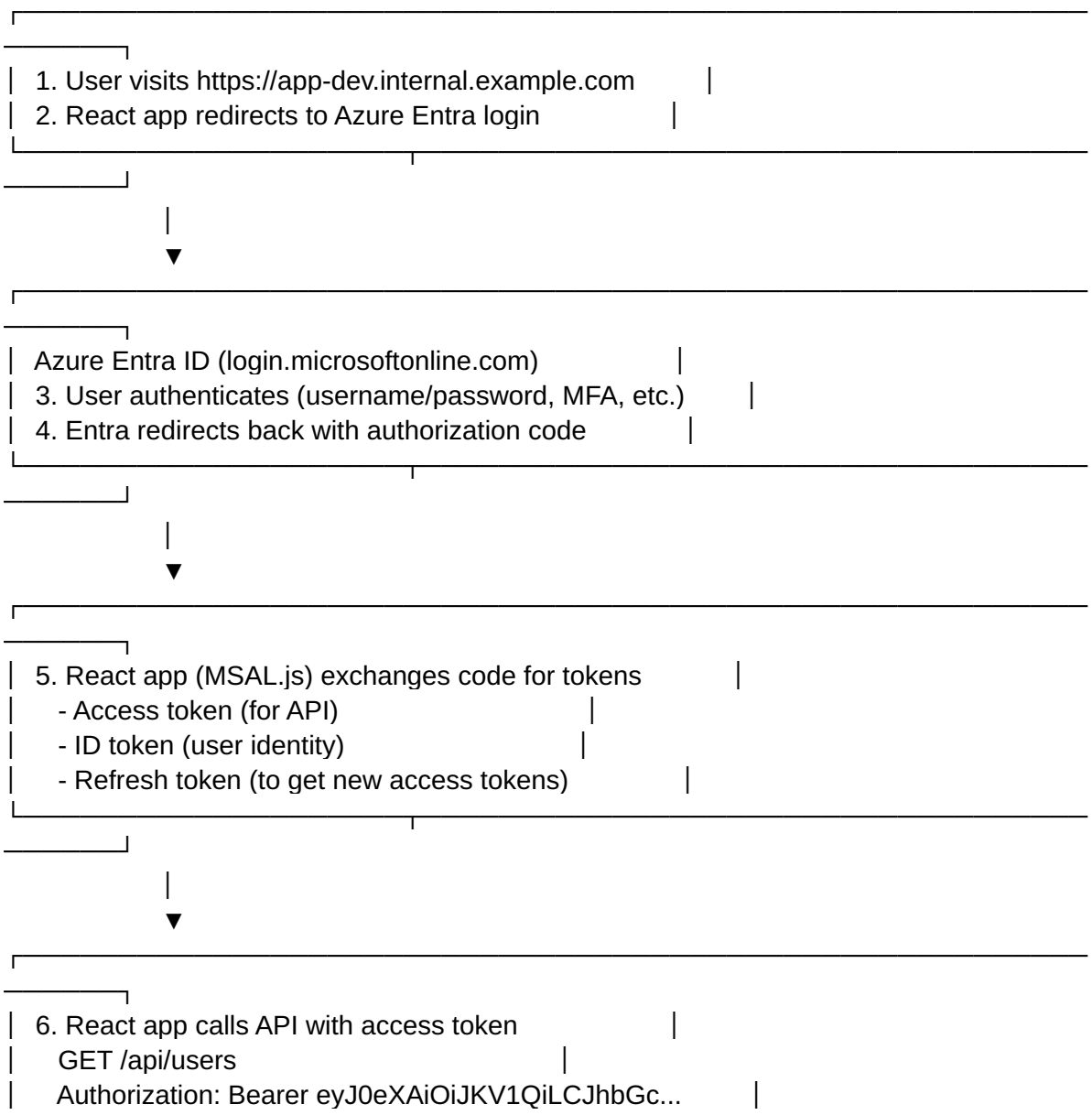
```
# Inside an ECS task in the same VPC
curl http://api.dev.svc.local:8000/health

# DNS resolution
dig api.dev.svc.local
```

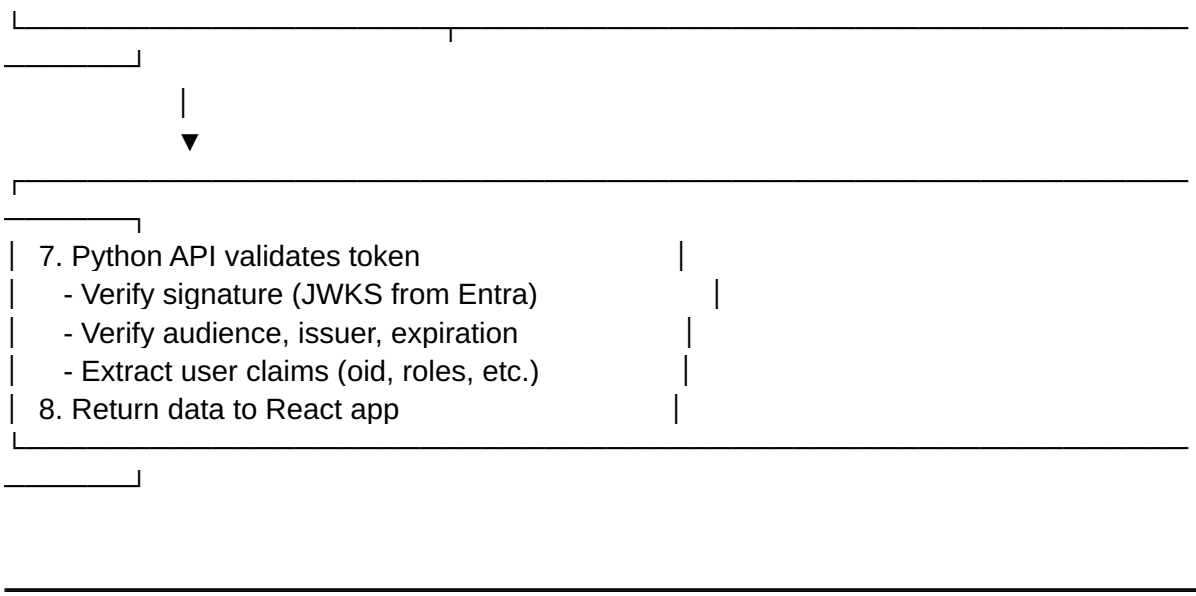
Returns A records for all healthy tasks

---

4. Authentication with Azure Entra ID (Deep Dive)

4.1 Architecture Overview





4.2 Azure Entra App Registrations (Detailed Setup)

4.2.1 API App Registration

Purpose: Represents your Python backend API.

Steps:

1. Azure Portal → Entra ID → App registrations → New registration
2. **Name**: MyApp-API-Dev
3. **Supported account types**: Single tenant (your org only)
4. **Redirect URI**: Leave blank (API doesn't redirect)
5. Click **Register**

Expose an API:

1. Go to **Expose an API**
2. **Application ID URI**: Click "Set" → Use default api://<client-id> or custom like api://myapp-api-dev
3. **Add a scope**:
    - Scope name: access_as_user
    - Who can consent: Admins and users
    - Display name: "Access MyApp API as a user"
    - Description: "Allows the app to access the API on behalf of the signed-in user"
4. **Save**

App Roles (Optional, for RBAC):

1. Go to **App roles**
2. Create roles like Admin, User, ReadOnly
3. Assign users/groups to roles in **Enterprise Applications**

Note the following:

- **Application (client) ID**: e.g., 12345678-1234-1234-1234-123456789abc
- **Application ID URI**: e.g., api://myapp-api-dev

4.2.2 UI App Registration (SPA)

Purpose: Represents your React SPA.

Steps:

1. Azure Portal → Entra ID → App registrations → New registration
2. **Name**: MyApp-UI-Dev
3. **Supported account types**: Single tenant
4. **Redirect URI**:
    - Platform: **Single-page application**
    - URI: https://app-dev.internal.example.com
5. Click **Register**

Authentication Settings:

1. Go to **Authentication**
2. **Redirect URIs**: Ensure https://app-dev.internal.example.com is listed
3. **Implicit grant and hybrid flows**: **Uncheck** both (use Auth Code Flow with PKCE)
4. **Supported account types**: Single tenant
5. **Allow public client flows**: No

API Permissions:

1. Go to **API permissions**
2. **Add a permission** → **My APIs** → Select MyApp-API-Dev
3. **Delegated permissions** → Check access_as_user
4. **Add permissions**
5. (Optional) **Grant admin consent** if required by your org

Note the following:

- **Application (client) ID**: e.g., abcdef12-3456-7890-abcd-ef1234567890

4.3 MSAL.js Configuration (Complete Example)

Install MSAL:

npm install @azure/msal-browser @azure/msal-react

src/authConfig.js:

```
import { getConfig } from './config';

export function getMsalConfig() {
  const config = getConfig();

  return {
    auth: {
      clientId: config.entraClientId,  // UI app registration client ID
      authority: `https://login.microsoftonline.com/${config.entraTenantId}`,
      redirectUri: config.entraRedirectUri,
      postLogoutRedirectUri: config.entraRedirectUri,
      navigateToLoginRequestUrl: true,
    },
    cache: {
      cacheLocation: 'sessionStorage',  // or 'localStorage'
      storeAuthStateInCookie: false,    // Set to true if IE11 support needed
    },
    system: {
      loggerOptions: {
        loggerCallback: (level, message, containsPii) => {
          if (containsPii) return;
          console.log(message);
        },
        logLevel: 'Info',  // 'Error', 'Warning', 'Info', 'Verbose'
      }
    }
  };
}

export const loginRequest = {
  scopes: ['api://myapp-api-dev/access_as_user'],  // API scope
};

export const tokenRequest = {
  scopes: ['api://myapp-api-dev/access_as_user'],
};
```

src/index.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { PublicClientApplication, EventType } from '@azure/msal-browser';
import { MsalProvider } from '@azure/msal-react';
import App from './App';
import { loadConfig } from './config';
import { getMsalConfig } from './authConfig';

const root = ReactDOM.createRoot(document.getElementById('root'));

loadConfig()
  .then(() => {
    const msalInstance = new PublicClientApplication(getMsalConfig());

    // Handle redirect promise
    return msalInstance.initialize().then(() => {
      return msalInstance.handleRedirectPromise().then(() => {
        // Optional: Set active account
        const accounts = msalInstance.getAllAccounts();
        if (accounts.length > 0) {
          msalInstance.setActiveAccount(accounts[0]);
        }

        // Listen for sign-in events
        msalInstance.addEventCallback((event) => {
          if (event.eventType === EventType.LOGIN_SUCCESS && event.payload.account) {
            msalInstance.setActiveAccount(event.payload.account);
          }
        });

        return msalInstance;
      });
    });
  })
  .then(msalInstance => {
    root.render(
      <React.StrictMode>
        <MsalProvider instance={msalInstance}>
          <App />
        </MsalProvider>
      </React.StrictMode>
    );
  })
  .catch(err => {
    console.error('Initialization error:', err);
    root.render(<div>Failed to initialize application. Please refresh.</div>);
  });
```

src/App.js:

```
import React from 'react';
import { AuthenticatedTemplate, UnauthenticatedTemplate, useMsal } from '@azure/msal-
react';
import { loginRequest } from './authConfig';

function App() {
  const { instance } = useMsal();

  const handleLogin = () => {
    instance.loginRedirect(loginRequest).catch(e => {
      console.error('Login error:', e);
    });
  };

  const handleLogout = () => {
    instance.logoutRedirect().catch(e => {
      console.error('Logout error:', e);
    });
  };

  return (
    <div>
      <AuthenticatedTemplate>
        <h1>Welcome to MyApp</h1>
        <button onClick={handleLogout}>Sign Out</button>
        {/* Your app content */}
      </AuthenticatedTemplate>

      <UnauthenticatedTemplate>
        <h1>Please sign in</h1>
        <button onClick={handleLogin}>Sign In</button>
      </UnauthenticatedTemplate>
    </div>
  );
}

export default App;
```

4.4 Making Authenticated API Calls

src/api/client.js:

```
import { useMsal } from '@azure/msal-react';
import { tokenRequest } from '../authConfig';
```

```
import { getConfig } from '../config';

export function useApiClient() {
  const { instance, accounts } = useMsal();
  const config = getConfig();
  const apiBase = config.apiBaseUrl || '/api';  // Relative for Pattern A

  async function callApi(endpoint, options = {}) {
    try {
      // Acquire token silently
      const request = {
        ...tokenRequest,
        account: accounts[0],
      };

      const response = await instance.acquireTokenSilent(request);
      const token = response.accessToken;

      // Make API call
      const apiResponse = await fetch(`${apiBase}${endpoint}`, {
        ...options,
        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': 'application/json',
          ...options.headers,
        },
      });

      if (!apiResponse.ok) {
        throw new Error(`API error: ${apiResponse.status}`);
      }

      return apiResponse.json();

    } catch (error) {
      // If silent token acquisition fails, try interactive
      if (error.name === 'InteractionRequiredAuthError') {
        const response = await instance.acquireTokenPopup(tokenRequest);
        const token = response.accessToken;

        const apiResponse = await fetch(`${apiBase}${endpoint}`, {
          ...options,
          headers: {
            'Authorization': `Bearer ${token}`,
            'Content-Type': 'application/json',
            ...options.headers,
          },
        });

        return apiResponse.json();
```

```
      }

      throw error;
    }
  }

  return { callApi };
}
```

Usage in a component:

```
import React, { useEffect, useState } from 'react';
import { useApiClient } from './api/client';

function UserList() {
  const [users, setUsers] = useState([]);
  const { callApi } = useApiClient();

  useEffect(() => {
    callApi('/users')
      .then(data => setUsers(data.users))
      .catch(err => console.error('Failed to fetch users:', err));
  }, []);

  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}
```

4.5 Python Backend Token Validation (Production-Ready)

Install dependencies:

```
pip install fastapi python-jose[cryptography] requests pydantic
```

auth.py (Complete implementation):

```
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import jwt, JWTError
from typing import Optional, Dict
import requests
import os
from functools import lru_cache

security = HTTPBearer()

# Configuration
TENANT_ID = os.getenv('ENTRA_TENANT_ID')
CLIENT_ID = os.getenv('ENTRA_API_CLIENT_ID')

# API app registration client ID
ISSUER = f"https://login.microsoftonline.com/{TENANT_ID}/v2.0"
JWKS_URI = f"https://login.microsoftonline.com/{TENANT_ID}/discovery/v2.0/keys"
AUDIENCE = f"api://{CLIENT_ID}"

# Or your custom App ID URI

# Cache JWKS keys (refresh every hour)
@lru_cache(maxsize=1)
def get_jwks() -> Dict:
    """Fetch and cache JWKS keys from Azure Entra."""
    response = requests.get(JWKS_URI, timeout=10)
    response.raise_for_status()
    return response.json()

def get_signing_key(token: str) -> Optional[Dict]:
    """Extract the signing key from JWKS based on token's kid."""
    try:
        unverified_header = jwt.get_unverified_header(token)
        kid = unverified_header.get('kid')

        if not kid:
            return None

        jwks = get_jwks()

        for key in jwks.get('keys', []):
            if key.get('kid') == kid:
                return key

        return None
    except Exception:
        return None

def verify_token(credentials: HTTPAuthorizationCredentials = Depends(security)) -> Dict:
    """
```

```
Verify and decode Azure Entra JWT token.
Returns the token payload (claims).
"""
token = credentials.credentials

try:
    # Get signing key
    signing_key = get_signing_key(token)
    if not signing_key:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid token: signing key not found",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # Decode and verify token
    payload = jwt.decode(
        token,
        signing_key,
        algorithms=['RS256'],
        audience=AUDIENCE,
        issuer=ISSUER,
        options={
            'verify_signature': True,
            'verify_aud': True,
            'verify_iss': True,
            'verify_exp': True,
        }
    )

    return payload

except jwt.ExpiredSignatureError:
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Token has expired",
        headers={"WWW-Authenticate": "Bearer"},
    )
except jwt.JWTClaimsError as e:
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail=f"Invalid token claims: {str(e)}",
        headers={"WWW-Authenticate": "Bearer"},
    )
except JWTError as e:
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail=f"Token validation failed: {str(e)}",
        headers={"WWW-Authenticate": "Bearer"},
    )
```

```
    except Exception as e:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Internal error during token validation: {str(e)}",
        )

def require_role(required_role: str):
    """Dependency to check if user has a specific role."""
    def role_checker(token_payload: Dict = Depends(verify_token)) -> Dict:
        roles = token_payload.get('roles', [])
        if required_role not in roles:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail=f"Insufficient permissions. Required role: {required_role}",
            )
        return token_payload
    return role_checker
```

main.py:

```
from fastapi import FastAPI, Depends
from auth import verify_token, require_role
from typing import Dict

app = FastAPI()

@app.get("/api/health")
def health_check():
    """Public health check endpoint."""
    return {"status": "healthy"}

@app.get("/api/users")
def get_users(token_payload: Dict = Depends(verify_token)):
    """Protected endpoint - requires valid token."""
    user_id = token_payload.get('oid')

# Object ID
    user_email = token_payload.get('preferred_username')

    # Your business logic here
    return {
        "users": [
            {"id": 1, "name": "Alice"},
            {"id": 2, "name": "Bob"},
        ],
        "requested_by": user_email,
    }

@app.post("/api/admin/settings")
```

```
def update_settings(
    settings: dict,
    token_payload: Dict = Depends(require_role('Admin'))
):
    """Admin-only endpoint - requires Admin role."""
    return {"message": "Settings updated", "settings": settings}
```

Environment variables (ECS Task Definition):

```
{
  "environment": [
    { "name": "ENTRA_TENANT_ID", "value": "your-tenant-id" },
    { "name": "ENTRA_API_CLIENT_ID", "value":
"12345678-1234-1234-1234-123456789abc" }
  ]
}
```

4.6 Token Refresh and Session Management

How MSAL handles token refresh:

1. Access tokens are short-lived (typically 1 hour)
2. MSAL automatically refreshes access tokens using the refresh token
3. acquireTokenSilent() checks if the cached token is expired and refreshes if needed
4. Refresh tokens are long-lived (typically 90 days) and stored in sessionStorage or localStorage

Best practices:

- **Always use acquireTokenSilent() first** before making API calls
- **Handle InteractionRequiredAuthError**: If silent refresh fails (e.g., refresh token expired), prompt user to re-authenticate
- **Don't store tokens manually**: Let MSAL manage token storage and lifecycle

Session timeout handling:

```
// src/api/client.js
async function callApi(endpoint, options = {}) {
  try {
    const response = await instance.acquireTokenSilent(tokenRequest);
    // ... make API call
  } catch (error) {
    if (error.name === 'InteractionRequiredAuthError') {
```

```
      // Refresh token expired or revoked - user must re-authenticate
      alert('Your session has expired. Please sign in again.');
      instance.loginRedirect(loginRequest);
    }
    throw error;
  }
}
```

---

5. Non-Functional Analysis

5.1 Latency and Performance

Component	Latency Impact	Mitigation
DNS Lookup	10-50ms (	