Detailed AWS Architecture Design

You are an AWS Solutions Architect with 20+ years of software development experience. Design a detailed end-to-end solution for the following scenario, including multiple options, trade-offs, and a recommended best option:

1. Context and Current Setup
Cloud provider: AWS
Compute platform: ECS on AWS Fargate (no EC2)
Network:
ECS Services are private, running in private subnets inside a VPC.
Access is through an Application Load Balancer (ALB) in public subnets.
The ALB has an AWS-provided DNS name.
There is also a custom domain configured in Route 53:
Hosted Zone in Route 53 with a CNAME (or ALIAS) pointing to the ALB DNS.
ACM certificate is used for HTTPS termination on the ALB.
Applications:
Frontend: React single-page application (SPA), containerized, deployed on its own ECS Fargate Service.
Backend API: Python FastAPI service, containerized, deployed on another ECS Fargate Service.
Cluster: Both ECS services run in the same ECS cluster.
Authentication & Authorization:
Uses Microsoft Entra ID (Azure AD) for both authentication and authorization.
Single-tenant Entra ID setup (only one Azure AD tenant).
Routing / Domains:
The React UI and FastAPI backend are both reachable via the ALB.
You can assume something like:
https://app.example.com → React SPA (ECS Service 1)
https://api.example.com or https://app.example.com/api → FastAPI backend (ECS Service 2)
The exact path vs. subdomain split can be part of your design options.
2. Requirements
2.1 Authentication / Access Requirements
UI (React SPA):
End-users access the UI URL directly in the browser via the ALB and custom domain.
When they access the UI, they must be prompted to sign in via Microsoft Entra ID if they are not already authenticated.

Once signed in, the SPA should have the necessary tokens to:
Display user info (e.g., name, email).
Call the backend FastAPI service securely on behalf of the user.
Service/API (FastAPI backend):
Must be accessible via:
Browser (e.g., direct GET request to some endpoint).
API clients such as Postman.
The React UI when user is already authenticated (SPA → API).
All these access paths should enforce Microsoft Entra ID authentication:
If calling from SPA, the SPA should attach a valid access token in the Authorization: Bearer header.
If calling from Postman or browser directly, the caller must obtain a token from Entra ID and pass it to the API as a Bearer token.
The API should validate tokens and enforce authorization based on claims (e.g., scopes, roles, groups).
Centralized Authorization in Entra ID:
Both authentication and authorization decisions (at least coarse-grained) should be based on Entra ID:
Users sign in using Entra ID.
User roles, groups, or app-specific permissions are defined and managed in Entra ID.
The FastAPI backend should consume these claims (e.g., roles, groups, scp, app_roles) to make authorization decisions.
Constraints & Prohibitions:
Do NOT use AWS API Gateway.
Do NOT use AWS Cognito.
Keep latency low and connections secure (TLS everywhere, minimal extra hops).
2.2 Non-Functional Requirements
Security:
All traffic over HTTPS only.
Private ECS services: only the ALB should have public access, services stay in private subnets.
Tokens should be short-lived; refresh handled appropriately.
Avoid storing tokens insecurely (e.g., no access tokens in localStorage if that can be avoided; consider secure cookies vs. memory storage tradeoffs).
Performance / Latency:
Avoid unnecessary proxy layers that would add significant latency.
Prefer regional alignment (Entra ID endpoints vs AWS region/ location where reasonable).
Consider ALB OIDC offload tradeoffs vs. app-level token validation.
Developer Experience:
API must be easily testable via Postman:
Document how developers obtain a token from Entra ID (client

credentials, OAuth 2.0 authorization code, device code, etc.).
Show how they configure Postman to call the API with Bearer tokens.
Frontend developers should have a clear React + MSAL (or similar) integration model.
3. Solution Scope
Provide a detailed architecture and solution design, including:

High-level Architecture Diagram Description
Describe (in text) how you would draw an architecture diagram with:
VPC, subnets (public for ALB, private for ECS/Fargate services).
ALB listeners & target groups.
Two ECS Fargate services (React SPA, FastAPI API).
Route 53 hosted zone and DNS routing (e.g., app.example.com, api.example.com).
Microsoft Entra ID (Authorization Server / IdP).
Token flows (SPA → Entra, SPA → API, Postman → Entra → API, browser direct → Entra → API).
Multiple Design Options for Authentication & Integration Present several options (at least 4–5 distinct options) for how to implement authentication and UI→service integration, explicitly considering:
Option A – App-Level Auth with SPA & API Both Integrated Directly with Entra ID
Frontend (React SPA):
Use OAuth 2.0 Authorization Code Flow with PKCE via MSAL.js or equivalent library.
SPA registers as a public client / SPA application in Entra ID.
SPA obtains:
ID token for user identity.
Access token(s) for calling the FastAPI backend (registered as a separate Web/API app in Entra).
Token storage:
Discuss the pros/cons of storing tokens in memory, sessionStorage, or localStorage.
Recommend secure patterns (e.g., in-memory with silent token renew if using MSAL, or using hidden iframe if allowed, or rotating refresh tokens).
Backend (FastAPI):
Register a separate Entra ID application representing the API (with its own client ID, app ID URI, scopes / app roles).
SPA requests specific scopes (e.g., api:///user.read) for calling the API.
FastAPI validates the access token on each request:
Use Entra's OpenID Connect discovery document to fetch JWKS, issuer, etc.

Validate signature, issuer, audience, and scopes.
Extract user identity and authorization claims (roles, groups, custom claims).
Postman / Browser Direct:
Users/developers obtain an access token from Entra ID (via OAuth 2.0 Authorization Code, device code, or client credentials for service-to-service) and pass it as Authorization: Bearer to the API.
Authorization:
Use app roles or Entra AD groups.
Map them to claims in tokens and enforce in FastAPI via decorators/middleware.
Option B – ALB-Level OIDC Authentication (ALB as OIDC Client) + Token Forwarding to Backend
Configure ALB authenticate-oidc action against Microsoft Entra ID.
ALB enforces login before requests reach SPA or API:
Discuss:
One ALB with different listeners/rules:
https://app.example.com → React SPA target group.
https://api.example.com or /api/* → FastAPI target group.
ALB side OIDC:
ALB redirects unauthenticated users to Entra ID.
Entra ID returns an authorization code; ALB exchanges it and obtains tokens.
ALB attaches ID token or user info headers to the backend.
Show challenges and/or trade-offs:
ALB's OIDC integration is generally more suitable for web backends rather than pure SPAs.
Handling API access via Postman: how to bypass ALB auth or utilize x-amzn-oidc-data headers, or alternatively require explicit Bearer tokens anyway.
Token injection vs. direct token validation by backend.
Evaluate whether ALB OIDC is a good fit here, given the requirement that:
The API must be callable by SPA, browser, and Postman with Entra-based tokens.
The SPA is a true frontend application (not server-side rendered).
Option C – Backend-For-Frontend (BFF) Pattern
SPA is served by a minimal backend (Node.js or Python) that:
Handles authentication server-side using Entra ID (Auth Code Flow).
Stores session cookies; backend calls API using on-behalf-of (OBO) flow or direct API tokens.
Explore pros/cons, but clearly note:
This may complicate the current architecture (React SPA running in ECS as static assets).
Good for security (no tokens in browser), but may diverge from the

current Fargate deployment model.
Option D – Hybrid: SPA Uses Auth Code + PKCE; FastAPI Validates Tokens; Postman Uses Same API Registration
Similar to Option A, but explicitly show:
How to configure Entra ID with:
One SPA client app registration.
One API (Web) app registration.
How to share these between:
SPA → API calls.
Postman → API calls.
Browser direct → API calls (if desired).
This might end up being one of the recommended options, so describe it in greater detail.
Option E – httpd (Apache) Web Server Acting as Auth Proxy in the Same ECS Services (for UI and API Separately)
Describe an option where you introduce an Apache httpd reverse proxy in front of both the UI and the API, each in their own ECS Fargate Service:
Overall Idea:
For the UI ECS Service:
Run a container (or multi-container task) where Apache httpd:
Terminates incoming HTTP from the ALB (inside VPC).
Performs OIDC authentication with Microsoft Entra ID using mod_auth_openidc (or similar).
Serves static React build assets (or proxies to a Node server serving the SPA).
Once the user is authenticated, Apache sets secure cookies / headers and forwards authenticated requests to the SPA content.
For the API ECS Service:
Another httpd proxy container in front of FastAPI:
Handles OIDC auth against Entra ID.
Validates the ID/access tokens.
Forwards authenticated requests (with user claims injected as headers) to the FastAPI app behind it on localhost or another container in the same task.
Entra Integration:
Detailed configuration for mod_auth_openidc:
OIDCProviderMetadataURL pointing to Entra's discovery endpoint.
OIDCClientID, OIDCClientSecret (for confidential client flows, especially for the API).
OIDCRedirectURI on the respective domains (app.example.com for UI; api.example.com for API).
Requested scopes (e.g., openid profile email api:///user.read).
How cookies are stored (mod_auth_openidc session cookies) and how access tokens are managed by Apache.
Flow for UI:

Browser hits https://app.example.com.
ALB routes to UI target group → httpd container.
mod_auth_openidc detects no session → redirects to Entra login.
After successful login, Entra redirects back to OIDCRedirectURI, Apache completes the OIDC flow, sets session, and then:
Serves SPA static files.
Optionally injects user info into headers or a bootstrap JSON endpoint for SPA to consume.
Discuss whether the SPA itself needs direct access tokens (for calling API) or whether:
The API is called via the same httpd proxy, and httpd handles token exchange / on-behalf-of, or
SPA still uses MSAL and obtains its own token for API, while httpd only protects UI access.
Flow for API:
Client (SPA / Postman / browser) hits https://api.example.com.
ALB routes to API target group → httpd container.
There are two main patterns:
Browser & SPA via session:
mod_auth_openidc authenticates and keeps a session cookie.
For SPA: calls to /api/* might use the same cookie, and httpd injects tokens/claims as headers to FastAPI.
Bearer token mode:
mod_auth_openidc configured in a "token verification" mode:
Expects Authorization: Bearer header from SPA/Postman.
Validates the JWT (signature, issuer, audience, scopes).
On success, forwards request to FastAPI with claims in headers; on failure, returns 401.
Discuss which of these is better, given the requirement that Postman and direct browser calls must be supported.
FastAPI Behavior:
With httpd handling OIDC/token validation, FastAPI can:
Either trust headers from Apache for identity and authorization.
Or still validate Bearer tokens directly (defense in depth).
Discuss security implications of trusting proxy headers (e.g., use of a private network, mTLS or security groups to ensure only httpd can access FastAPI).
Pros and Cons:
Pros:
Centralizes OIDC logic in Apache; simpler app code in FastAPI/ React.
Standard, battle-tested mod_auth_openidc behavior.
Potential to hide tokens from the SPA in some designs (if using pure server-side session for API).
Cons:
Additional complexity (Apache config, session management).

Another component in the path, some added latency.
Need to ensure pattern still supports Postman and direct access appropriately.
Clearly compare this Apache proxy option against the other options and indicate where it shines and where it is not ideal.
For each option (A–E):
Describe:
Token flow (step-by-step).
Where validation occurs (ALB vs. Apache vs. app).
How UI → API integration works.
How Postman / direct calls work.
Provide pros, cons, complexity, security, and latency analysis.
Best-Practice Recommendation Among the presented options, choose and clearly mark one best option for this scenario, considering:
Strong adherence to the requirement to use Entra ID only (no API Gateway, no Cognito).
Clear and consistent Entra ID usage for both SPA and API.
Secure handling of tokens.
Simplicity and maintainability.
Support for UI, browser, Postman, and (if relevant) httpd proxy flows.
Good developer experience.
Likely, the best option will be some variant of:
React SPA using Auth Code + PKCE via MSAL.
FastAPI validating access tokens from Entra ID directly.
Separate Entra App Registrations for SPA and API, with defined scopes/app roles.
Postman configured to acquire tokens for the same API app registration.
Justify this choice deeply and explicitly, and also explicitly mention why you would or would not choose the httpd proxy option vs ALB OIDC or pure app-level auth.
4. Detailed Entra ID Configuration
Provide a step-by-step, detailed configuration guide for Microsoft Entra ID (single-tenant) that supports the recommended architecture:

App Registrations
SPA App (e.g., “MyApp SPA”):
Type: public client SPA.
Redirect URIs:
https://app.example.com
https://app.example.com/auth/callback (or similar, depending on routing).
Implicit grant vs Auth Code with PKCE:

Use Authorization Code with PKCE, not implicit flow.
Grant it permission to call the API application.
Describe required manifest or configuration changes (e.g., spa redirectUris, allowPublicClient, etc.).
API App (e.g., “MyApp API”):
Type: Web/API (confidential client).
Expose an API:
Define Application ID URI (e.g., api://).
Define scopes: e.g., user.read, user.write, admin.
Optionally define app roles for authorization: Reader, Writer, Admin.
Describe how to configure “Authorized client applications” so that the SPA can request tokens for the scopes.
Optionally, if using httpd mod_auth_openidc with confidential client flows:
Describe whether separate app registrations are needed for httpd acting as a confidential client (especially for API proxy patterns).
Permissions, Scopes, and Consent
Configure the SPA app to request the API app’s scopes:
e.g., api:///user.read.
Configure admin consent as needed for organizational rollout.
Discuss delegated permissions vs. application permissions and clarify which are used in this scenario (primarily delegated, since the user signs in).
User Roles / Groups & Authorization
Describe how to:
Define app roles within the API App registration (e.g., role IDs, display names, allowed member types = Users/Groups).
Assign users or groups to app roles in Entra ID enterprise application blade.
Explain how roles appear in the token (e.g., roles claim vs. groups claim).
Discuss token size concerns with many groups; suggest enabling group overage or using app roles for more predictable claim sizes.
Give guidelines on whether to use:
App roles for API-level authorization.
Entra ID groups for coarse RBAC.
Or custom claims (via optional claims or custom extension attributes) when necessary.
FastAPI Integration Provide detailed guidance (pseudo-code or code-level concepts) for how to integrate FastAPI with Entra ID both with and without a proxy:
Without Apache proxy:
Use Python libraries like:
python-jose, PyJWT, msal, or others for token validation.
Configuration in FastAPI:
Entra tenant ID.

Authority / issuer URL, e.g.:
https://login.microsoftonline.com//v2.0
Audience (API Application ID URI or Client ID).
Required scopes (e.g., api:///user.read).
Implement a reusable dependency or middleware that:
Extracts Authorization header.
Validates JWT against Entra's JWKs.
Checks iss, aud, exp, nbf, etc.
Checks that the user has required scope(s)/role(s).
Injects the user principal (with claims) into the FastAPI path operations.
With Apache proxy:
Describe how FastAPI would:
Trust user identity and roles passed via headers set by mod_auth_openidc (e.g., X-User, X-Roles, X-Email).
Optionally still re-validate Bearer tokens for critical endpoints (defense in depth).
Discuss how to implement FastAPI dependencies that read identity from headers instead of from JWT directly.
Provide example patterns like:
A get_current_user dependency.
A decorator or dependency like @requires_role("Admin") or @requires_scope("user.read").
React SPA Integration Describe how to integrate the React SPA with Entra ID:
Use MSAL.js (or a similar Entra ID SPA library).
Configure MSAL with:
Client ID of the SPA app registration.
Authority: https://login.microsoftonline.com/.
Redirect URIs matching Entra registration.
postLogoutRedirectUri.
On initial load:
If not authenticated, redirect to login or show "Sign In".
After login, store the MSAL account and tokens.
For calling the API:
Use acquireTokenSilent (with fallback to acquireTokenRedirect/ acquireTokenPopup) to get an access token for the API scopes.
Attach the access token in the Authorization: Bearer header in fetch/Axios calls.
If using the Apache proxy pattern for UI:
Explain how the SPA would behave if Apache already enforces login.
Clarify whether SPA still needs to obtain tokens for the API or can rely on session cookies / same-origin requests against an httpd-fronted API.
State management:

Where to store the MSAL instance and user account (React context or a global store).
Token storage & security:
Discuss risk of XSS and how that affects token storage choice.
Suggest using MSAL's recommended protected iframe/hidden refresh mechanisms rather than manually persisting tokens.
5. ALB, ECS, and Networking Details
Provide a precise design on the AWS side:

ALB Configuration
Listeners:
HTTPS : 443 with ACM certificate for *.example.com or specific hostnames like app.example.com.
Rules:
Host-based routing:
Host: app.example.com → Target Group: React UI ECS service (or httpd+SPA).
Host: api.example.com → Target Group: FastAPI ECS service (or httpd+FastAPI).
Or path-based if using a single domain:
/ → UI Target Group.
/api/* → API Target Group.
Target groups:
Type: IP or instance as per Fargate best practices.
Health checks configured on appropriate endpoints (e.g., /healthz on FastAPI or Apache).
ECS Fargate Services
Network mode: awsvpc.
Tasks run in private subnets only.
ALB is associated with the ECS services via service -> targetGroupArn.
Security groups:
ALB SG:
Inbound: HTTPS (443) from internet.
Outbound: to ECS tasks' security group.
ECS Tasks SG:
Inbound: from ALB SG on the container port (e.g., 80/8080).
Outbound: to the internet via NAT Gateway (for talking to Entra ID, fetching JWKS, etc.).
If using multi-container tasks for httpd + FastAPI or httpd + SPA:
Describe the task definition with multiple containers on the same ENI, how they communicate (localhost/ports), and which container is the one registered in the target group (httpd).
DNS / Route 53 / Certificates
Route 53 Hosted Zone: example.com.
Records:

app.example.com → ALIAS to ALB DNS.
api.example.com → ALIAS to ALB DNS (if using host-based routing).
ACM:
Request a certificate for app.example.com and api.example.com (or *.example.com) in the same region as the ALB.
Ensure all traffic is HTTPS with HTTP→HTTPS redirection (optional but recommended).
6. API Access from Postman and Browser
Explain in detail how:

Postman Access:
Developers configure an OAuth 2.0 client in Postman:
Authorization URL: https://login.microsoftonline.com//oauth2/v2.0/authorize
Token URL: https://login.microsoftonline.com//oauth2/v2.0/token
Client ID: SPA client or a dedicated "Postman client" app (explain which is better and why).
Scopes: api:///user.read (and others as needed).
Obtain an access token and call:
https://api.example.com/endpoint with Authorization: Bearer .
Discuss any differences if API is fronted by httpd:
How mod_auth_openidc can validate Bearer tokens.
Any headers or configuration changes needed.
Describe typical errors (401, 403) and how to debug (invalid audience, missing scope, wrong tenant, etc.).
Direct Browser Access:
For endpoints intended for human users, describe:
Option of redirecting unauthenticated users to SPA (preferred UX).
Or presenting raw JSON for authenticated tokens (if user already logs in via SPA & sends Bearer tokens).
Clarify how this fits with the chosen architecture and with/without the Apache proxy.
7. Security, Latency, and Operational Considerations
Security:
Key management: tokens are signed by Entra; no custom key mgmt is needed, but FastAPI or httpd must regularly fetch JWKS and cache them.
TLS at ALB; traffic in VPC can be HTTP or mTLS (discuss pros/cons).
CSRF, XSS considerations:
For SPA, emphasize protecting against XSS so tokens aren't exfiltrated.
For Apache session cookies, discuss SameSite, HttpOnly, Secure flags.
Suggest hardening (Content Security Policy, secure React coding

practices).
Logging and monitoring:
Log authentication failures with enough detail (but not tokens).
Use CloudWatch Logs & Metrics; optionally integrate with SIEM.
Latency:
Entra endpoints are internet-accessible; FastAPI/Apache and SPA will talk to them via NAT.
Explain impact and how token caching (and JWKs caching) reduce latency.
Compare latency impacts of:
Direct app-level token validation.
ALB OIDC.
Apache proxy with mod_auth_openidc.
Operations:
Token lifetime and refresh:
Describe default lifetime and how refresh happens in SPA via MSAL, and/or in Apache via mod_auth_openidc.
Rotating Entra settings:
Impact of rotating client secrets (for confidential clients like httpd proxy; SPA is a public client and has no secret).
Rolling deployments and blue/green:
Ensure tokens validated by new versions of API/proxy remain valid.
8. Final Output Expectations
Your final answer should:

Provide a structured, step-by-step solution design with headings and subheadings.
Clearly list:
Multiple design options (including the Apache httpd proxy option).
A recommended best option (with rationale).
Include enough concrete detail (URLs, claims, typical configuration values, Apache directives, etc.) to be practically implementable.
Focus heavily on:
AWS ECS Fargate + ALB network and security setup.
Deep integration with Microsoft Entra ID for both authentication and authorization.
UI (React SPA) → API (FastAPI) integration, including token flows.
How an httpd proxy option would work and how it compares.
Ensuring:
Secure connections.
Low latency.
No use of AWS API Gateway or Cognito.

GPT-5.1 Thinking

0. Target End-State (Executive Summary)

You're already in a very good place: ECS Fargate, ALB, Route 53, Entra ID (Azure AD). The cleanest, most secure, and maintainable solution **for this scenario** is:

> **Recommended Option (D / "Hybrid"):**
> - **React SPA** uses **Authorization Code + PKCE** with **MSAL.js** against **Entra ID SPA app registration**.
> - **FastAPI** is a **separate Entra ID Web/API app** and **validates JWT access tokens** from Entra ID on every call.
> - **Postman / browser / other clients** also obtain access tokens for that same API registration and call https://api.example.com with Authorization: Bearer <token>.

This keeps all auth logic at the **application level**, gives you:

- One clear **identity provider (Entra ID)**.
- Clean separation of **SPA vs API** concerns.
- First-class support for **Postman**, **browser** and **service-to-service** calls.
- No need for API Gateway / Cognito / complex ALB OIDC gymnastics.

Below is a full design, including alternative options (ALB OIDC, BFF, Apache proxy) and why they are less ideal.

---

1. High-Level Architecture (Text Diagram)

1.1 Network / AWS Components

Describe the diagram like this:

1. **VPC (10.0.0.0/16)**
    - **Public subnets** (e.g., 10.0.1.0/24, 10.0.2.0/24):
        - **Application Load Balancer (ALB)** in at least 2 AZs.

- **Private subnets** (e.g., 10.0.11.0/24, 10.0.12.0/24):
    - **ECS Fargate Service 1 – UI** (React SPA, likely served via Nginx or similar)
    - **ECS Fargate Service 2 – API** (FastAPI)
- **NAT Gateway(s)** in public subnets so private tasks can reach:
    - Entra ID endpoints (login & JWKS).
    - Container registries, etc.

2. **Security Groups**
    - **ALB SG**:
        - Inbound: TCP 443 from Internet.
        - Outbound: TCP 80/443 to ECS Tasks SG.
    - **ECS Tasks SG**:
        - Inbound: from ALB SG on app ports (e.g., 80/8080).
        - Outbound: 80/443 to Internet via NAT (for Entra ID, JWKS).
3. **ALB**
    - **HTTPS listener 443** with ACM certificate for app.example.com, api.example.com (or *.example.com).
    - **Host-based rules**:
        - Host: app.example.com → Target Group tg-ui → Fargate UI service.
        - Host: api.example.com → Target Group tg-api → Fargate API service.
    - (Alternative: single host app.example.com with path routing / vs /api/*.)
4. **ECS Fargate Services**
    - **UI Service**:
        - Task definition: container ui-container (Nginx serving built React).
        - Registered in tg-ui, port 80.
    - **API Service**:
        - Task definition: container api-container (FastAPI, e.g., Uvicorn on port 8000).
        - Registered in tg-api, port 8000 (or 80).
5. **Route 53**
    - Hosted zone: example.com.
    - app.example.com → ALIAS → ALB DNS.
    - api.example.com → ALIAS → ALB DNS.
6.

Microsoft Entra ID (Azure AD)

- **Tenant**: single-tenant.
- **App Registrations**:
    - MyApp SPA (public client / SPA).
    - MyApp API (Web/API).
    - (Optional) MyApp Dev Client for Postman / tools.

1.2 Token & Call Flows (Recommended Option)

1. **SPA Login & Token Acquisition**
    1. User opens https://app.example.com.
    2. ALB routes to UI service → React SPA bundle.
    3. SPA initializes MSAL; sees no account → redirects user to Entra ID / authorize (Auth Code + PKCE).
    4. User authenticates, Entra ID redirects back to https://app.example.com/auth/callback.
    5. MSAL exchanges auth code for:
        - **ID token** (user identity).
        - **Access token(s)** for MyApp API scopes (api://<api-app-id>/user.read, etc.).
    6. Tokens stored securely (see section 4. React).
2. **SPA → FastAPI call**
    1. SPA calls https://api.example.com/resource with:
        - Header: Authorization: Bearer <access_token_for_MyApp_API>.
    2. ALB routes to API service.
    3. FastAPI validates JWT (signature, issuer, aud, exp, scopes/roles).
    4. If valid & authorized, FastAPI executes logic and returns JSON.
3. **Postman → FastAPI**
    1. Developer configures OAuth2 in Postman (Auth Code or Device Code or a “Dev Client” app).
    2. Postman obtains access token for MyApp API scopes.
    3. Call https://api.example.com/resource with Authorization: Bearer <token>.
    4. FastAPI validates and authorizes exactly as for SPA.
4. **Browser direct → FastAPI**
    - If it presents Authorization: Bearer <token> (e.g., via JS client), FastAPI handles it like above.
    - If no Bearer token → 401/302 (depending on pattern); recommended is to keep API as pure resource server (401 JSON).

---

2. Design Options for Authentication & Integration

Option A – Direct App-Level Auth (SPA + API Each Integrate with Entra)

This is the “basic” version of the recommended design.

2.1 Token Flow

- **SPA**
  - Registered as SPA app.
  - Uses **Auth Code + PKCE** via MSAL.
  - Gets ID token & access token for API.
  - Stores tokens (prefer **in-memory** / MSAL cache, not raw localStorage).
- **API (FastAPI)**
  - Registered as Web/API app.
  - Exposes scopes (user.read, admin) & app roles.
  - Validates JWT access token on each request:
    - Uses Entra’s discovery document + JWKS.
- **Postman / Browser Direct**
  - Developer logs in via OAuth2 flow (Auth Code, Device Code), gets access token targeting the API.
  - Sends Authorization: Bearer <token>.

2.2 Validation & Authorization

- **Validation at API level**:
  - iss = https://login.microsoftonline.com/<tenant-id>/v2.0.
  - aud = api://<app-id-guid> **or** API client ID.
  - scp (delegated scopes) or roles (app roles).
- **Authorization**:
  - Use scopes for function-level permissions; app roles or groups for RBAC.

2.3 Token Storage (SPA)

- **In-memory (recommended)**:
    - Tokens kept in JS memory / MSAL's internal storage.
    - Less persistence, less risk if XSS.
    - Relies on silent renew via iframe or hidden tab & refresh tokens (handled by MSAL 2.x).
- **sessionStorage**:
    - Survives refresh in same tab; lost on new tab / browser close.
    - Still vulnerable to XSS.
- **localStorage**:
    - Persistent; highest XSS risk. Avoid for access tokens.

Recommendation: MSAL defaults + **in-memory** or **sessionStorage**; prioritize XSS protection (CSP, linting, no dangerouslySetInnerHTML).

2.4 Pros / Cons

- **Pros**
    - Simple, standard, well-documented.
    - Works cleanly with SPA, Postman, any HTTP client.
    - No extra infra components.
- **Cons**
    - Tokens in browser; you must take XSS seriously.
    - Each app (SPA, API) must implement Entra integration.

Option B – ALB-Level OIDC Auth (ALB as OIDC Client)

ALB supports authenticate-oidc actions. ALB becomes an OIDC client to Entra and can enforce login before forwarding traffic.

2.5 Architecture & Flow

- ALB HTTPS listener 443.

- Rule: for app.example.com and/or /:
    - authenticate-oidc using Entra (client id/secret).
    - On success, forward to UI/API target group.
- ALB sets OIDC headers:
    - x-amzn-oidc-data (JWT).
    - x-amzn-oidc-identity (sub).
- Backend may trust these headers (after its own JWT verify or trusting ALB).

2.6 Challenges for Your Requirements

- Designed mainly for **web apps in browsers**, not rich SPA + API ecosystem.
- **Postman**:
    - Hitting https://api.example.com triggers ALB OIDC flow, which expects browser redirects & cookies.
    - Hard/impossible to cleanly “just send a Bearer token” and have ALB validate it.
- **Pure SPA**:
    - SPA itself no longer “owns” the auth flow; ALB does.
    - Harder for SPA to acquire tokens to call other APIs, perform on-behalf-of flows, etc.
- **Different access paths**:
    - You want **SPA**, **Postman**, and **direct browser** all using Entra tokens. ALB OIDC doesn’t nicely unify those.

2.7 Pros / Cons

- **Pros**
    - Offloads OIDC login from app code.
    - Can protect legacy apps quickly.
- **Cons**
    - Limited to browser-style flows.
    - Poor fit for API clients like Postman.
    - Harder to implement “API as resource server” with Entra scopes/roles.

Conclusion: Not a good fit for this use case; I would not pick this.

Option C – Backend-For-Frontend (BFF)

Introduce a BFF (Node.js / Python) that:

- Serves the React SPA.
- Handles OIDC Auth Code flow server-side.
- Stores session in an **HTTP-only secure cookie**.
- Calls FastAPI on behalf of the user using OBO (on-behalf-of) flow or direct API tokens.

2.8 Flow

1. Browser → app.example.com → BFF.
2. BFF redirects to Entra for Auth Code flow.
3. After callback, BFF stores session (cookie) + obtains access token for API.
4. BFF:
    - Serves React build (static).
    - Provides /api/* endpoints; SPA never directly calls api.example.com.
    - BFF calls FastAPI with Bearer token (OBO or backend client credentials).

2.9 Pros / Cons

- **Pros**
    - Very strong security: access tokens are never in browser.
    - Can centralize auth, CSRF protection, etc.
- **Cons**
    - **Architecture change**: you now have 3 backends: BFF, UI static, API.
    - BFF becomes critical path; more moving parts.
    - More complex Entra flows (OBO, confidential clients).
    - Doesn't naturally satisfy "API easily testable via Postman" unless you also expose direct token-protected API.

Conclusion: Great for very high-security consumer apps, but overkill and adds complexity given your Fargate-only, SPA+API pattern and Postman requirement.

Option D – Hybrid / Refined App-Level Auth (Recommended)

This is Option A, but:

- Clearly structured with **separate SPA vs API app registrations**.
- Shared API registration for **SPA, Postman, browser, service-to-service**.
- Explicit guidance for roles, scopes, and developer tooling.

This best meets your:

- Single Entra tenant.
- SPA + API separation.
- Postman / direct access.
- Low latency (no extra proxies).

I'll detail this in section 3 (recommended architecture).

Option E – Apache httpd (mod_auth_openidc) as Auth Proxy (UI & API)

Put Apache httpd in front of both UI and API (each its own ECS service or multi-container task):

2.10 UI Service (httpd + SPA)

- ECS Task with:
    - Container 1: httpd with mod_auth_openidc.
    - Container 2: ui (Nginx static SPA or Node dev server).
- ALB → target group → httpd container.
- httpd:
    - Enforces OIDC login with Entra ID.
    - After login, serves static SPA content or proxies to Node.
    - Manages tokens via its own session cookies.

Config sketch:

```
OIDCProviderMetadataURL https://login.microsoftonline.com/<tenant-id>/v2.0/.well-known/openid-configuration
OIDCClientID <spa-client-id-or-proxy-client-id>
OIDCClientSecret <secret-if-confidential>
OIDCRedirectURI https://app.example.com/oidc/callback
OIDCCryptoPassphrase <random-strong-passphrase>
OIDCScope "openid profile email"

<Location />
```

```
  AuthType openid-connect
  Require valid-user
</Location>

ProxyPass / http://ui-container:3000/
ProxyPassReverse / http://ui-container:3000/
```

2.11 API Service (httpd + FastAPI)

Two main patterns:

a) Session-Mode

- httpd enforces login; clients only use cookies.
- Not suitable for Postman (no browser redirect, cookies).

b) Bearer-Token Verification Mode (better fit)

- Clients (SPA, Postman, browser JS) send Authorization: Bearer <token>.
- mod_auth_openidc validates JWT against Entra:
    - OIDCOAuthVerifyJwksUri, OIDCOAuthIntrospectionEndpoint or uses provider metadata.
- On success:
    - httpd forwards to FastAPI and injects claims as headers (e.g., X-User-Email, X-User-Roles).

Config sketch:

```
OIDCProviderMetadataURL https://login.microsoftonline.com/<tenant-id>/v2.0/.well-known/openid-configuration
OIDCOAuthClientID <api-app-client-id>
OIDCOAuthClientSecret <api-app-client-secret>
OIDCOAuthIntrospectionEndpoint https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token
OIDCOAuthVerifyClaimsIssuer on

<Location /api/>
  AuthType oauth20
  Require valid-user
  # Or Require claim "roles:Admin"
</Location>
```

ProxyPass /api/ http://fastapi:8000/
ProxyPassReverse /api/ http://fastapi:8000/

FastAPI can then:

- Trust headers (private network, SG restrictions).
- Or revalidate tokens (defense in depth).

2.12 Pros / Cons

- **Pros**
  - Centralizes OIDC & token handling in Apache.
  - Can hide tokens from SPA (if session mode).
  - Very flexible and battle-tested.
- **Cons**
  - Additional hop & complexity (httpd config, tuning).
  - Doesn't fundamentally solve Postman/direct access better than app-level (still need Bearer mode).
  - Now your containers are more complex (multi-container or additional service).

Conclusion: Viable if you want strong separation of auth from app code, or you have many backend languages. In **this** scenario, it's extra complexity without strong benefit over the recommended app-level approach.

---

3. Recommended Architecture in Detail (Option D)

3.1 Entra ID Configuration

3.1.1 API App Registration – MyApp API

1. **Register App**
   - Go to **Entra ID → App registrations → New registration**.

- Name: MyApp API.
- Supported account types: *Accounts in this organizational directory only* (single tenant).
- Redirect URI: not required for pure API (Web API).

2. **Expose an API**
   - Expose an API → Set Application ID URI: api://<myapp-api-app-id-guid> (default suggestion is fine).
   - Add **scopes**:
     - Example:
       - Scope name: user.read
         Full: api://<api-app-id>/user.read
         Who can consent: Admins and users.
         Description: “Read user data”.
       - Scope name: user.write, etc.
3. **App Roles (API-level Authorization)**
   - App roles:
     - Reader: value = Reader, allowed member types = Users/Groups.
     - Writer: value = Writer.
     - Admin: value = Admin.
   - After creation, go to **Enterprise applications → MyApp API**:
     - Assign users and/or groups to roles.
4. **Optional Claims / Groups**
   - Token configuration:
     - Add optional claim groups or roles (roles come automatically when using app roles).
   - If you have many groups, prefer app roles over raw groups to keep token size manageable.

3.1.2 SPA App Registration – MyApp SPA

1. **Register App**
   - Name: MyApp SPA.
   - Type: Accounts in this org directory only.
   - Platform: SPA.
2.

Redirect URIs

- Under Authentication → Add a platform → Single-page application:
  - Add:
    - https://app.example.com
    - https://app.example.com/auth/callback (or similar route used by MSAL).

3. **Authentication Settings**
   - Ensure:
     - Implicit grant **NOT** used (no Access tokens via implicit).
     - **Auth Code flow with PKCE** is used (default for SPA).
   - Enable Allow public client flows for SPA if needed (depends on UX; for SPA type it's implicit).
4. **API Permissions**
   - API permissions:
     - Add a permission → My APIs → choose MyApp API.
     - Select delegated scopes: user.read, user.write, etc.
   - Grant admin consent if required for your org.
5. **Authorized Client Applications**
   - In MyApp API → Expose an API:
     - Under Authorized client applications add the SPA's client ID to allow it to request the scopes.

3.1.3 Optional Postman / Tools App – MyApp Dev Client

You can:

- **Reuse MyApp SPA** for Postman (interactive login).
  Or,
- Create a dedicated **public client**: MyApp Dev Client:
  - Platform: Mobile and desktop applications or Public client (native).
  - Redirect URIs: https://oauth.pstmn.io/v1/callback for Postman.
  - Permissions: same delegated scopes from MyApp API.

Using a separate Dev Client avoids confusing production SPA logs / consent with dev tools.

3.2 AWS Infrastructure Details

3.2.1 VPC, Subnets, NAT

- **VPC**: e.g., 10.0.0.0/16.
- **Public subnets (2+)**:
    - ALB, NAT Gateways.
- **Private subnets (2+)**:
    - ECS tasks for UI & API.

Routing:

- Private subnet route: 0.0.0.0/0 → NAT Gateway.
- Public subnet route: 0.0.0.0/0 → Internet Gateway.

3.2.2 ALB Configuration

- **ACM**:
    - Obtain certificate for app.example.com and api.example.com, or wildcard *.example.com, in the same region as ALB.
- **Listeners**:
    - :443 (HTTPS) with the ACM cert.
    - Optional :80 → redirect to :443.
- **Rules**:
    - If **host-based**:
        - Rule 1:
            - Condition: Host is app.example.com
            - Action: Forward to tg-ui (port 80).
        - Rule 2:
            - Condition: Host is api.example.com
            - Action: Forward to tg-api (port 8000).
    - If **path-based** on one host:
        - Host is app.example.com and Path is /api/* → tg-api.
        - Otherwise → tg-ui.

3.2.3 ECS Fargate Services

-

ECS Cluster: myapp-cluster.

- **Service 1 – UI (React SPA)**
  - Task definition:
    - Network mode: awsvpc.
    - CPU/memory as needed.
    - Container: ui-container:
      - Image: e.g., Nginx with /usr/share/nginx/html = React build.
      - Port: 80.
  - Service:
    - Launch type: Fargate.
    - Subnets: private.
    - Security group: sg-ecs-ui (inbound from ALB SG on 80).
    - Target group: tg-ui.
- **Service 2 – API (FastAPI)**
  - Task definition:
    - Network mode: awsvpc.
    - Container: api-container:
      - Image: Python FastAPI + Uvicorn.
      - Port: 8000 (health endpoint /healthz).
  - Service:
    - Subnets: private.
    - SG: sg-ecs-api (inbound from ALB SG on 8000).
    - Target group: tg-api.
- **Egress**:
  - Both SGs allow outbound to 443 → NAT → Entra endpoints & JWKs.

3.3 FastAPI Integration with Entra ID

3.3.1 Config Needed

Environment variables for API container:

- TENANT_ID
- API_CLIENT_ID (MyApp API's Application (client) ID).
- API_APP_ID_URI = api://<api-app-id-guid> (used as audience).

- AUTHORITY = https://login.microsoftonline.com/<TENANT_ID>/v2.0
- JWKS_URL = https://login.microsoftonline.com/<TENANT_ID>/discovery/v2.0/keys

3.3.2 Token Validation Dependency

Example using python-jose and requests:

```
# requirements: fastapi, uvicorn, python-jose[cryptography], httpx or requests

from fastapi import FastAPI, Depends, HTTPException, status, Request
from jose import jwt
import httpx
from functools import lru_cache
from typing import List

TENANT_ID = "<tenant-id>"
API_APP_ID_URI = "api://<api-app-id-guid>"

# audience
AUTHORITY = f"https://login.microsoftonline.com/{TENANT_ID}/v2.0"
JWKS_URL = f"{AUTHORITY}/discovery/v2.0/keys"
ISSUER = AUTHORITY

app = FastAPI()

@lru_cache()
def get_jwks():
    resp = httpx.get(JWKS_URL, timeout=5.0)
    resp.raise_for_status()
    return resp.json()["keys"]

def get_kid(header):
    return header.get("kid")

def get_key(kid):
    jwks = get_jwks()
    for key in jwks:
        if key["kid"] == kid:
            return key
    raise HTTPException(status_code=401, detail="Invalid token key id")

async def get_current_user(request: Request, required_scopes: List[str] = None,
required_roles: List[str] = None):
    auth = request.headers.get("Authorization")
    if not auth or not auth.lower().startswith("bearer "):
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
```

```
detail="Missing bearer token")

    token = auth.split(" ", 1)[1]
    try:
        header = jwt.get_unverified_header(token)
        key = get_key(get_kid(header))
        claims = jwt.decode(
            token,
            key,
            algorithms=["RS256"],
            audience=[API_APP_ID_URI, "<api-client-id>"],

# support both forms
            issuer=ISSUER
        )
    except Exception as e:
        raise HTTPException(status_code=401, detail=f"Invalid token: {str(e)}")

    # Scope check (delegated permissions)
    if required_scopes:
        token_scopes = claims.get("scp", "").split()
        if not all(scope in token_scopes for scope in required_scopes):
            raise HTTPException(status_code=403, detail="Insufficient scope")

    # Role check (app roles)
    if required_roles:
        token_roles = claims.get("roles", [])
        if not any(role in token_roles for role in required_roles):
            raise HTTPException(status_code=403, detail="Insufficient role")

    return claims

# or a custom User model

def require_scopes(*scopes):
    async def dep(request: Request):
        return await get_current_user(request, required_scopes=list(scopes))
    return dep

def require_roles(*roles):
    async def dep(request: Request):
        return await get_current_user(request, required_roles=list(roles))
    return dep

@app.get("/healthz")
async def healthz():
    return {"status": "ok"}

@app.get("/me")
async def get_me(user=Depends(get_current_user)):
```

```
    return {
        "sub": user["sub"],
        "name": user.get("name"),
        "preferred_username": user.get("preferred_username"),
        "roles": user.get("roles", []),
        "scp": user.get("scp", "")
    }

@app.get("/admin/data")
async def admin_data(user=Depends(require_roles("Admin"))):
    return {"secret": "only admins see this"}
```

Key points:

- Caches JWKS keys for performance.
- Validates iss, aud, exp, etc.
- Enforces scopes/roles via dependencies.

3.4 React SPA Integration with MSAL.js

3.4.1 MSAL Configuration

Install:

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

MSAL config:

```
// src/authConfig.ts
import { Configuration, LogLevel } from "@azure/msal-browser";

export const msalConfig: Configuration = {
  auth: {
    clientId: "<MyApp SPA client-id>",
    authority: "https://login.microsoftonline.com/<tenant-id>",
    redirectUri: "https://app.example.com/auth/callback",
    postLogoutRedirectUri: "https://app.example.com",
  },
  cache: {
    cacheLocation: "sessionStorage", // or "localStorage" if you accept risk
    storeAuthStateInCookie: false,
```

```
  },
  system: {
    loggerOptions: {
      loggerCallback: (level, message) => { console.log(message); },
      logLevel: LogLevel.Info,
      piiLoggingEnabled: false,
    },
  },
};

export const loginRequest = {
  scopes: ["openid", "profile", "email", "api://<api-app-id-guid>/user.read"],
};
```

Wrap app with MsalProvider:

```
// src/index.tsx
import React from "react";
import ReactDOM from "react-dom/client";
import { PublicClientApplication } from "@azure/msal-browser";
import { MsalProvider } from "@azure/msal-react";
import { msalConfig } from "./authConfig";
import App from "./App";

const msalInstance = new PublicClientApplication(msalConfig);

const root = ReactDOM.createRoot(document.getElementById("root")!);
root.render(
  <MsalProvider instance={msalInstance}>
    <App />
  </MsalProvider>
);
```

3.4.2 Login & Token Acquisition

Example React component:

```
// src/App.tsx
import React, { useEffect, useState } from "react";
import { useMsal, useIsAuthenticated } from "@azure/msal-react";
import axios from "axios";
import { loginRequest } from "./authConfig";

const API_BASE = "https://api.example.com";
```

```
function App() {
  const { instance, accounts } = useMsal();
  const isAuthenticated = useIsAuthenticated();
  const [user, setUser] = useState<any>(null);
  const [apiData, setApiData] = useState<any>(null);

  useEffect(() => {
    if (!isAuthenticated) {
      instance.loginRedirect(loginRequest);
      return;
    }
    const account = accounts[0];
    setUser({
      name: account?.name,
      username: account?.username,
      homeAccountId: account?.homeAccountId,
    });
  }, [isAuthenticated, instance, accounts]);

  const callApi = async () => {
    const account = accounts[0];
    if (!account) return;

    const response = await instance.acquireTokenSilent({
      ...loginRequest,
      account,
    });

    const accessToken = response.accessToken;

    const res = await axios.get(`${API_BASE}/me`, {
      headers: { Authorization: `Bearer ${accessToken}` },
    });
    setApiData(res.data);
  };

  if (!isAuthenticated) {
    return <div>Redirecting to sign-in...</div>;
  }

  return (
    <div>
      <h1>MyApp</h1>
      <p>Signed in as: {user?.name} ({user?.username})</p>
      <button onClick={callApi}>Call API</button>
      {apiData && (
        <pre>{JSON.stringify(apiData, null, 2)}</pre>
      )}
    </div>
```

```
  );
}

export default App;
```

3.4.3 Token Storage & Security

- Use MSAL's in-memory/session storage.
- Add strong **Content Security Policy (CSP)**, no unsafe eval, avoid inline scripts.
- Use HTTPS only; set Secure cookies where relevant.
- Handle 401 from API by calling acquireTokenSilent again and, if needed, loginRedirect.

3.5 Postman & Browser Access

3.5.1 Postman Configuration

1. In Postman, open your API request to https://api.example.com/me.
2. Go to **Authorization** tab:
    - Type: OAuth 2.0.
3. Set:
    - **Auth URL**: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/authorize
    - **Access Token URL**: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token
    - **Client ID**: MyApp Dev Client (or reuse MyApp SPA).
    - **Client Secret**: *empty* if using public client / SPA; if using confidential dev client, put secret.
    - **Scope**: api://<api-app-id-guid>/user.read offline_access openid profile
    - **Grant Type**:
        - Authorization Code (user login via browser), or
        - Device Code (no browser hook).
    - Click **Get New Access Token**, complete login, accept consent.
    - Postman stores the token; ensure Add token to: Header.
    - Call API:
    - Request header: Authorization: Bearer <token>.

- Response: 200 with JSON if token valid and roles/scopes satisfied; else 401/403.

Debug hints:

- 401 “Invalid audience” → check aud in token; must match api://<api-app-id-guid> or API client ID.
- 403 “Insufficient scope” → ensure correct scope(s) checked in FastAPI; confirm scp claim has them.
- Wrong tenant → ensure tenant-id in URLs matches your tenant.

3.5.2 Browser Direct

If a developer uses custom JS or CLI tool that obtains a valid Bearer token, they can:

- Call https://api.example.com/resource with Authorization: Bearer <token>.
- API responds same as for SPA / Postman.

For human-friendly UX, keep backend as pure JSON resource; direct human access should mainly be via SPA.

---

4. httpd Proxy Option vs Recommended Option

Why not httpd (Option E) as primary?

- It shines if:
  - You have many microservices in many languages and want a **single standard OIDC gateway** layer.
  - You explicitly want **no tokens in SPA** (then use session-mode BFFish behavior).
- But for your current setup:
  - You already have a simple topology (SPA + single API, same cluster).
  - You require clean **Postman** and generic client support.
  - Adding httpd adds:
    - Extra container per service.
    - More configs and secrets (client secrets for httpd).
    - Slight latency (~few ms per request).
  - It does not simplify Entra config vs direct app-level JWT validation.

Why not ALB OIDC (Option B)?

- ALB OIDC is awkward for Postman and non-browser clients.
- Doesn’t give SPA a first-class token acquisition experience.
- ALB is not a full OAuth2 server / token validator for generic Bearer tokens.

Why not BFF as default (Option C)?

- Strong security, but:
    - Significantly more moving parts.
    - Harder developer experience (must go via BFF for everything).
    - Not necessary unless you have very sensitive data and want no tokens in JS at all.

Why Option D is best here

- All requirements met directly:
    - SPA uses Entra ID with MSAL.
    - API validates Entra tokens, uses Entra roles/groups.
    - Postman/Browsers reuse the same API registration.
- Minimal infrastructure complexity.
- Secure, low latency, clean DX.

---

5. Security, Latency & Operational Considerations

5.1 Security

- **Transport Security**
    - TLS termination at ALB using ACM.
    - Traffic inside VPC can be HTTP; for very sensitive deployments, consider mTLS or HTTPS between ALB and ECS tasks (more complex).
- **Token Security**
    - JWTs signed by Entra with RSA; no custom key mgmt.
    - FastAPI validates against JWKS and caches keys.
    - Keep access tokens relatively short-lived (Entra defaults are reasonable). Use refresh tokens via MSAL for SPA.
- **XSS & CSRF**
    - SPA:
        - Strict CSP (no unsafe-inline, no unsafe-eval).
        - Avoid dangerous HTML injection.
    - API:
        - Pure JSON; no CSRF since it doesn’t rely on cookies.

    - If you ever use cookies for auth, set them as HttpOnly, Secure, SameSite=Strict/Lax.
- **Least Privilege**
    - Use scopes & app roles to avoid overbroad permissions.
    - Restrict FastAPI endpoints with fine-grained dependencies.

5.2 Latency

- **Token validation cost**
    - First request: JWKS fetch (~10s TTL or more).
    - Subsequent requests: purely local crypto; minimal overhead.
- **Network hops**
    - Client → ALB → ECS task.
      Very straightforward; no extra proxy layers.

Comparisons:

- ALB OIDC → adds Entra roundtrips & ALB auth logic.
- httpd → extra hop + mod_auth_openidc processing.
- BFF → extra network hop + additional processing.

5.3 Operations

- **Token Lifetime & Refresh**
    - SPA: MSAL handles silent refresh; only when tokens expire & silent fails does it redirect user.
    - API: just validates exp and nbf; no state to manage.
- **Secrets**
    - SPA is public client (no secret).
    - API might use client secret only if you add OBO flows; otherwise, for validation no secret needed.
    - Store any secrets (if needed) in **AWS Secrets Manager**; inject via env vars.
- **Rolling Deployments**
    - Since validation uses Entra's public keys, new versions of API still validate existing tokens.
    - No coupling between token issuance and app deployment.
-

Logging & Monitoring

- FastAPI logs:
    - Authentication errors (401) including reason, but **never** log entire tokens.
- ALB access logs to S3 (optional).
- CloudWatch Logs for ECS tasks.
- Optionally, forward logs to SIEM (e.g., via Kinesis / Firehose).

---

6. Summary

- You stay fully on **AWS ECS + ALB** with Entra ID as **sole IdP**.
- **Best option:**
    - **React SPA** uses **MSAL.js** with **Auth Code + PKCE** against a **SPA app** in Entra.
    - **FastAPI** uses **JWT validation** against an **API app** in Entra, enforcing **scopes & app roles**.
    - **Postman & other clients** use the **same API registration** and Bearer tokens.
- Alternative patterns (ALB OIDC, BFF, Apache httpd proxy) are viable in other contexts but add complexity or don't meet your exact needs as cleanly.

If you'd like, I can next:

- Draft Terraform/CloudFormation snippets for the ALB + ECS + SG wiring.
- Or provide full sample FastAPI and React projects wired to Entra ID that you can adapt.