CloudDesk

Detailed AWS deployment guide and required repository changes

Document date: January 25, 2026

This document describes the concrete changes you must implement in the CloudDesk repository to support a complete AWS serverless deployment, then provides an end to end deployment procedure for a production like review. It assumes you already created IAM identities and can authenticate the AWS CLI locally. If your IAM identity does not have permissions to deploy CloudFormation based stacks, this guide includes a permissions checklist.

Scope of this guide:

- Backend infrastructure using AWS SAM and AWS CloudFormation
- API Gateway HTTP API secured with a JWT authorizer backed by Amazon Cognito
- Lambda handlers for the four API endpoints documented in the CloudDesk README
- DynamoDB single table data model that supports both user and agent queries
- Frontend integration steps for authentication and API calls
- Deployment steps for backend, then frontend, including verification and troubleshooting
- Operational review checklist, including logging, cost controls, and clean up

Contents

1. Current repository status and target architecture

The CloudDesk README describes a serverless backend deployed via AWS SAM, with Lambda function handlers in src/handlers and a SAM template named template.yaml. However, the current repository root file list shows a Vite plus React frontend layout (public, src, vite.config.ts) and does not visibly include template.yaml at the repository root. This mismatch means sam build and sam deploy will not work until you add the missing infrastructure files and the backend code paths expected by SAM.

Target architecture (as described by the README):

```
Browser UI
  -> Amazon Cognito (authentication)
  -> Amazon API Gateway HTTP API (JWT authorizer)
  -> AWS Lambda (Node.js 18+ handlers)
  -> Amazon DynamoDB (ticket storage and indexes)
  -> Amazon CloudWatch (logs and metrics)
```

API endpoints to implement:

Method	Endpoint	Role	Purpose
POST	/tickets	User	Create a new ticket
GET	/tickets	User	List my tickets
GET	/agent/tickets?status=OPEN	Agent	List tickets by status
PATCH	/agent/tickets/{ticketId}	Agent	Update ticket status

Deployment strategy in this guide:

- Deploy backend first using AWS SAM so you can capture Outputs such as API base URL and Cognito identifiers
- Configure frontend environment variables with those Outputs
- Deploy frontend using Amplify Hosting (recommended) or S3 plus CloudFront

2. Required repository changes (folders and files to add)

To make the repository deployable with SAM, add a backend folder that contains template.yaml and Lambda source code. Separating frontend and backend reduces build confusion and makes CI pipelines simpler. If you prefer to keep the frontend at the repository root, you can still add backend as a sibling folder. The key requirement is that SAM must have a template file and handler entry points that match what the template references.

Recommended repository layout:

```
CloudDesk/
  frontend/                   # Vite + React app (existing code moved here)
    public/
    src/
    index.html
    package.json
    package-lock.json
    vite.config.ts
    tsconfig.json
  backend/
    template.yaml             # NEW: SAM template
    package.json              # NEW: backend dependencies
    package-lock.json         # NEW: backend lockfile
    src/
      handlers/               # NEW: Lambda handlers
        createTicket.ts
        listMyTickets.ts
        listTicketsByStatus.ts
        updateTicketStatus.ts
      lib/
        auth.ts
        ddb.ts
        validate.ts
        response.ts
    events/                   # optional: sample events for local tests
  README.md
```

Folder by folder change list:

- Create backend/template.yaml with all infrastructure resources: HttpApi, Lambda functions, DynamoDB table, Cognito resources, and Outputs
- Create backend/package.json to manage Lambda dependencies independently from frontend dependencies
- Create backend/src/handlers with four handlers mapped to the documented endpoints
- Create backend/src/lib with shared utilities: DynamoDB client, auth claim parsing, request validation, and consistent HTTP responses
- Optional: create backend/events with example event payloads for sam local invoke
- Move current frontend code into frontend/ and update build instructions accordingly (recommended). If you do not move files, keep existing frontend files at root and only add backend/

Git ignore updates:

Update .gitignore to exclude backend build artifacts. For SAM, common artifact folders include .aws-sam and any dist output if you compile TypeScript.

```
# Backend artifacts
backend/.aws-sam/
backend/node_modules/
backend/dist/
frontend/node_modules/
frontend/dist/
```

3. Backend infrastructure specification (SAM template)

The SAM template is the source of truth for deployment. It defines all AWS resources and their wiring. The minimum viable template for CloudDesk includes: a DynamoDB table, a Cognito user pool and app client, an HTTP API with a JWT authorizer, and four Lambda functions with route integrations.

Why HTTP API with a JWT authorizer

API Gateway HTTP APIs support a native JWT authorizer that validates tokens issued by an external identity provider. When using Amazon Cognito, you configure the authorizer issuer URL for the user pool and the audience set to the Cognito app client id. The client calls the API with an access token in the Authorization header.

Key template components:

- Parameters: StageName, CorsAllowedOrigin, LogLevel
- Globals for Lambda timeout and environment variables
- Cognito: UserPool, UserPoolClient (no client secret for SPA), optional UserPoolDomain for hosted UI testing
- HttpApi: CORS settings and Auth JWT authorizer configuration
- DynamoDB: TicketsTable plus GSI for status based queries
- Functions: four AWS::Serverless::Function resources with HttpApi events
- Outputs: ApiBaseUrl, UserPoolId, UserPoolClientId, Region

CORS configuration requirements:

- AllowOrigins must include your frontend domain (Amplify domain or CloudFront domain) and localhost during development
- AllowHeaders must include Authorization and Content-Type
- AllowMethods must include GET, POST, PATCH, OPTIONS
- If you later use cookies, you must handle credentials explicitly. For token based Authorization headers, credentials are not required

Logging and tracing recommendations:

- Enable structured logs in Lambda and include requestId and userSub for debugging
- Consider enabling AWS X-Ray tracing for Lambda and API Gateway once the basic flow works
- Avoid logging raw tokens or sensitive data

4. Authentication and authorization design (Cognito and roles)

CloudDesk requires two roles: User and Agent. Authentication proves identity, and authorization enforces what that identity can do. For this project, Amazon Cognito is the most direct solution because API Gateway HTTP API can validate Cognito issued JWTs natively using a JWT authorizer.

Option 1 (recommended): one user pool plus groups

Create a single Cognito user pool and use groups to represent roles. Create a group named Agents and optionally a group named Users. An authenticated user is considered a normal user by default. Agent only endpoints verify that the JWT claim cognito:groups includes Agents.

- Pros: simpler frontend configuration, one login flow, one JWT authorizer
- Pros: easier user management and fewer moving parts
- Cons: agent and user accounts share the same pool, so you must enforce group based access in Lambda

Option 2 (matches README wording): separate user pool and agent pool

Create two Cognito user pools: one for normal users and one for agents. Then create two JWT authorizers (or two audiences) and attach them to different routes. This reduces group logic in Lambda, but increases infrastructure and frontend complexity.

- Pros: clear separation of user types at the identity provider level
- Cons: frontend must support two pools or two sign in flows
- Cons: API must route to different authorizers, which complicates the SAM template

Recommendation for review deployment:

Use Option 1 unless you have a strong reason to isolate pools. It aligns with typical serverless proof of concept patterns and reduces failure points during deployment.

Authorization enforcement rules (required)

- All endpoints require authentication via the HTTP API JWT authorizer
- User endpoints enforce ownership: users can only read tickets that match their Cognito subject id
- Agent endpoints enforce role: request must include group Agents in cognito:groups
- Agent status updates should enforce allowed transitions: OPEN -> IN_PROGRESS -> RESOLVED

Token placement and expected headers:

The frontend must send a JWT access token in the Authorization header using the Bearer scheme. Example: Authorization: Bearer . API Gateway validates the token before invoking Lambda when a JWT authorizer is attached.

5. DynamoDB data model (keys, GSIs, access patterns)

DynamoDB design must match the access patterns. CloudDesk has two primary queries: list tickets for a specific user, and list tickets by status for agents. A single table design with a small number of item types supports both queries efficiently.

Access patterns to support:

- Create ticket (write)
- List my tickets (Query by user id)
- List tickets by status (Query by status index)
- Update ticket status (Update by ticket id)

Single table key schema:

Table name: TicketsTable

Attribute	Type	Description
PK	String	Partition key
SK	String	Sort key
GSI1PK	String	Partition key for status index
GSI1SK	String	Sort key for status index

Item types (two item approach):

- Ticket metadata item (authoritative record)
- User lookup item (supports list my tickets quickly)

Ticket metadata item example:

```
PK      = TICKET#<ticketId>
SK      = META
ownerSub= <cognitoSub>
status  = OPEN | IN_PROGRESS | RESOLVED
createdAt = <ISO timestamp>
title   = <short summary>
description = <full text>
GSI1PK  = STATUS#<status>
GSI1SK  = CREATED#<createdAt>#TICKET#<ticketId>
```

User lookup item example:

```
PK      = USER#<ownerSub>
SK      = TICKET#<createdAt>#<ticketId>
ticketId= <ticketId>
status  = OPEN | IN_PROGRESS | RESOLVED
createdAt = <ISO timestamp>
title   = <short summary>
```

Implementation notes:

- Use a UUID for ticketId to avoid collisions
- Use an ISO 8601 timestamp for createdAt so lexical order matches chronological order
- When updating status, update both the ticket metadata item and the user lookup item so the user view stays consistent
- Use conditional updates to prevent invalid transitions, for example only allow RESOLVED when current status is IN_PROGRESS
- Keep ticket descriptions within DynamoDB item size limits (400 KB). For large attachments, use S3, but that is out of scope

6. Lambda handlers (contracts, validation, logging)

Each endpoint maps to a Lambda handler. Handlers must validate input, enforce authorization rules, perform DynamoDB operations, and return consistent JSON responses.

Common handler conventions (recommended):

- Use a shared response helper that sets statusCode, headers, and JSON body consistently
- Return errors as JSON objects with fields: error, message, requestId
- Log structured JSON lines that include requestId, route, userSub, and statusCode
- Never log the Authorization header value or JWT token

HTTP API v2 event fields you will use:

```
event.requestContext.http.method
event.requestContext.http.path
event.requestContext.requestId
event.pathParameters
event.queryStringParameters
event.body
event.headers
event.requestContext.authorizer.jwt.claims   # present when JWT authorizer is configured
```

Shared auth parsing helper (backend/src/lib/auth.ts):

```
- Read claims from event.requestContext.authorizer.jwt.claims
- Extract userSub from claims.sub
- Extract groups from claims["cognito:groups"] (may be a string or list depending on runtime)
- Provide helpers:
  - requireAuthenticated(event) -> { userSub, groups }
  - requireAgent(event) -> { userSub, groups } with group Agents required
```

Endpoint A: POST /tickets (User)

- Auth: requireAuthenticated
- Input: JSON body with title and description
- Validation: title required, length limit, description required, length limit
- Write: Put ticket metadata item and user lookup item using a TransactWrite for atomicity
- Output: 201 with ticketId, status, createdAt

Endpoint B: GET /tickets (User)

- Auth: requireAuthenticated
- Query: DynamoDB Query PK = USER#, sort by SK descending if desired
- Output: 200 with array of tickets

Endpoint C: GET /agent/tickets?status=OPEN (Agent)

- Auth: requireAgent (must be in group Agents)
- Input: query parameter status, must be one of OPEN, IN_PROGRESS, RESOLVED
- Query: DynamoDB Query on GSI1 where GSI1PK = STATUS#

- Output: 200 with array of tickets, include ownerSub for auditing

Endpoint D: PATCH /agent/tickets/{ticketId} (Agent)

- Auth: requireAgent
- Input: path parameter ticketId and JSON body with newStatus
- Validation: enforce allowed transitions, for example OPEN -> IN_PROGRESS -> RESOLVED
- Update: Update ticket metadata status and GSI attributes, then update the user lookup item status
- Output: 200 with updated ticket fields

Error status codes (standardize across handlers):

HTTP code	When to use	Example message
400	Invalid request body or parameters	Invalid status value
401	Missing or invalid token (mostly handled by API Gateway)	Unauthorized
403	Authenticated but not allowed (role or ownership)	Agent role required
404	Ticket not found	Ticket not found
409	Conflict, such as invalid status transition	Invalid status transition
500	Unexpected errors	Internal server error

Build system for TypeScript (esbuild via SAM):

- Each function in template.yaml should declare Metadata: BuildMethod: esbuild
- Set entry points to backend/src/handlers/.ts and handler export name handler
- Set platform: node, target: es2020, and minify: false for debuggability
- Ensure package.json includes dependencies: @aws-sdk/client-dynamodb, @aws-sdk/lib-dynamodb, uuid (or use crypto.randomUUID in Node 18)

7. Local development and testing

Before deploying, you should validate handler logic locally. SAM can invoke functions locally with a simulated API Gateway event. JWT authorizers are not typically validated locally, so your local event fixtures should include a requestContext.authorizer.jwt.claims object.

Local prerequisites:

- Node.js 18+ installed
- AWS CLI configured with credentials
- AWS SAM CLI installed
- Docker installed (SAM local uses Docker for Lambda runtime emulation)

Backend install and unit test baseline:

```
cd backend
npm install
# Optional: add a simple test runner later (vitest or jest). For now, manual invoke is enough.
```

Build and invoke a handler locally:

```
sam build
sam local invoke CreateTicketFunction -e events/createTicket.json
```

Example local event fixture (events/createTicket.json):

```
{
  "version": "2.0",
  "routeKey": "POST /tickets",
  "rawPath": "/tickets",
  "headers": { "content-type": "application/json" },
  "requestContext": {
    "requestId": "local-test-1",
    "http": { "method": "POST", "path": "/tickets" },
    "authorizer": {
      "jwt": {
        "claims": {
          "sub": "00000000-0000-0000-0000-000000000000",
          "cognito:groups": "Users"
        }
      }
    }
  },
  "body": "{\"title\":\"Printer not working\",\"description\":\"The printer in room 401 is offline.\
  "isBase64Encoded": false
}
```

Notes about DynamoDB in local tests:

- Simplest: run against real DynamoDB in your AWS account. This requires setting AWS credentials in your environment.
- More isolated: run DynamoDB Local and point the SDK endpoint to it. This adds setup complexity and is optional for a proof of concept.
- If you run against real AWS, use a dedicated dev stack name so you can delete it cleanly.

8. Backend deployment with AWS SAM

This section is the exact deployment runbook for the backend. It uses sam deploy --guided for first deployment and then standard sam deploy for updates.

Step 8.1: Verify AWS CLI identity and region

```
aws sts get-caller-identity
aws configure list
aws configure get region
```

Confirm the account id is correct and set a default region (for example us-east-1) unless your organization requires a specific region.

Step 8.2: Build the SAM application

```
cd backend
sam build
```

If you are using esbuild builds, sam build should bundle TypeScript to JavaScript automatically. If build fails, confirm that all handler entry points exist and that package.json is valid.

Step 8.3: First deploy with guided configuration

```
sam deploy --guided
```

Recommended guided prompt answers:

Prompt	Recommended value
Stack Name	clouddesk-dev (or clouddesk-prod)
AWS Region	Your preferred region
Confirm changes before deploy	Yes for first run
Allow SAM CLI IAM role creation	Yes
Disable rollback	No
Save arguments to samconfig.toml	Yes
Parameter: StageName	dev (or prod)
Parameter: CorsAllowedOrigin	http://localhost:5173 for dev, then your Amplify domain for prod
Parameter: LogLevel	INFO

Step 8.4: Capture Outputs for frontend configuration

After deployment completes, SAM prints CloudFormation Outputs. Record these values because the frontend needs them:

- ApiBaseUrl (for example https://abc123.execute-api.region.amazonaws.com)
- UserPoolId
- UserPoolClientId
- Region

You can reprint outputs any time using the CloudFormation CLI:

```
aws cloudformation describe-stacks --stack-name clouddesk-dev --query "Stacks[0].Outputs"
```

Step 8.5: Create Cognito groups and users (post deploy)

If you use a single user pool with groups, create a group named Agents, then create test users and add the agent user to that group. You can do this in the AWS Console or with CLI commands. For review deployments, Console is fine. For repeatability, prefer CLI.

```
# Example CLI flow (replace placeholders)
aws cognito-idp create-group --user-pool-id <UserPoolId> --group-name Agents

# Create user and set a temporary password
aws cognito-idp admin-create-user --user-pool-id <UserPoolId> --username agent1 --temporary-password

# Add the user to Agents group
aws cognito-idp admin-add-user-to-group --user-pool-id <UserPoolId> --username agent1 --group-name A
```

Step 8.6: Smoke test backend with curl

Once you have a valid access token from Cognito, you can test endpoints directly. Obtain a token using the frontend or hosted UI, then run:

```
curl -X GET "<ApiBaseUrl>/tickets"   -H "Authorization: Bearer <accessToken>"
```

9. Frontend changes and deployment

The frontend must (1) authenticate users via Cognito, (2) obtain an access token, and (3) call the backend API with Authorization: Bearer . Because the repository currently contains a Vite plus React app, this section assumes you will keep that stack.

Step 9.1: Add environment variables

In Vite, environment variables exposed to the browser must start with VITE_. Create frontend/.env.local for local development and set the values from SAM Outputs.

```
# frontend/.env.local
VITE_API_BASE_URL=https://<ApiId>.execute-api.<region>.amazonaws.com
VITE_AWS_REGION=<region>
VITE_COGNITO_USER_POOL_ID=<UserPoolId>
VITE_COGNITO_USER_POOL_CLIENT_ID=<UserPoolClientId>
```

Step 9.2: Add Amplify Auth dependencies (recommended)

Amplify provides a straightforward way to sign in users and retrieve tokens without implementing OAuth flows manually.

```
cd frontend
npm install aws-amplify @aws-amplify/ui-react
```

Step 9.3: Configure Amplify (frontend/src/amplify.ts)

```
import { Amplify } from "aws-amplify";

export function configureAmplify() {
  Amplify.configure({
    Auth: {
      Cognito: {
        userPoolId: import.meta.env.VITE_COGNITO_USER_POOL_ID,
        userPoolClientId: import.meta.env.VITE_COGNITO_USER_POOL_CLIENT_ID
      }
    }
  });
}
```

Step 9.4: Initialize Amplify in main entry file

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import { configureAmplify } from "./amplify";

configureAmplify();

ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Step 9.5: Add a minimal sign in and role gating UI

For a demo review, the easiest UI is a sign in screen, then two tabs: User view and Agent view. Agent view should only render if the authenticated user belongs to Agents group. The group claim is available after token decode or from Cognito user attributes depending on the library. A simple approach is to call the /agent endpoints and handle 403 with a friendly message.

Step 9.6: API client wrapper (frontend/src/api.ts)

```
import { fetchAuthSession } from "aws-amplify/auth";

const API_BASE = import.meta.env.VITE_API_BASE_URL;

async function getAccessToken(): Promise<string> {
  const session = await fetchAuthSession();
  const token = session.tokens?.accessToken?.toString();
  if (!token) {
    throw new Error("No access token available");
  }
  return token;
}

export async function apiFetch(path: string, options: RequestInit = {}) {
  const token = await getAccessToken();
  const headers = new Headers(options.headers || {});
  headers.set("Authorization", `Bearer ${token}`);
  headers.set("Content-Type", "application/json");
  const res = await fetch(`${API_BASE}${path}`, { ...options, headers });
  const text = await res.text();
  const body = text ? JSON.parse(text) : null;
  if (!res.ok) {
    const msg = body?.message || body?.error || `HTTP ${res.status}`;
    throw new Error(msg);
  }
  return body;
}
```

Step 9.7: Deploy frontend with Amplify Hosting (recommended)

Amplify Hosting is the lowest friction option. It provides a managed build pipeline, static hosting, TLS, and SPA routing rewrites.

- Open AWS Amplify Console and create a new app
- Connect the GitHub repository and select the branch
- If the repo is a monorepo, set app root to frontend
- Set build settings: install command npm ci, build command npm run build, artifacts directory dist
- Set environment variables in Amplify (same keys as .env.local)
- Add rewrite rule: 404 to /index.html with 200 status (for SPA routes)
- Deploy and copy the Amplify domain URL

Step 9.8: Update backend CORS after frontend URL is known

Once you know the Amplify URL, update the CorsAllowedOrigin parameter in SAM and redeploy. This prevents browsers from blocking API calls due to CORS.

```
cd backend
```

```
sam deploy
```

Alternative frontend hosting: S3 plus CloudFront

If you prefer S3 plus CloudFront, you must configure SPA routing so that client side routes return index.html. This is typically done by configuring custom error responses for 403 and 404 to respond with /index.html and a 200 status.

- Create an S3 bucket for static hosting (private bucket recommended)
- Create a CloudFront distribution with the S3 bucket as origin and Origin Access Control
- Upload frontend dist/ content to the bucket
- Configure custom error responses for 403 and 404 to /index.html with response code 200
- Use the CloudFront domain as the frontend base URL, then update backend CORS

10. Verification, troubleshooting, and operational checklist

Use this section as a deployment review checklist. It is organized in the same order reviewers typically validate a serverless application: identity, API authorization, data access, and UI integration.

10.1 End to end demo script

- Sign in as a normal user
- Create a ticket (title and description) and confirm a 201 response
- List my tickets and confirm the new ticket appears with status OPEN
- Sign in as an agent user (member of Agents group)
- List tickets by status OPEN and confirm the user ticket appears
- Update the ticket status to IN_PROGRESS, then to RESOLVED
- Sign back in as the normal user and confirm the ticket status is RESOLVED

10.2 Common deployment failures and fixes

- sam deploy fails with missing capabilities: ensure CAPABILITY_IAM is allowed in the deploy prompt
- JWT authorizer rejects token with invalid audience: ensure authorizer audience is set to the Cognito app client id and that the frontend uses that same app client
- 403 from agent endpoints: verify the agent user is in Agents group and that handler group parsing reads cognito:groups
- CORS error in browser: update CorsAllowedOrigin to the exact frontend origin (scheme and domain) and redeploy backend
- DynamoDB AccessDeniedException: check Lambda execution role permissions include dynamodb:PutItem, UpdateItem, Query, GetItem on the table and its indexes
- Unexpected 500 errors: inspect Lambda logs in CloudWatch and ensure you return JSON responses even on exceptions

10.3 Where to look for logs

- Lambda logs: CloudWatch Logs log group per function
- API Gateway execution logs: enable access logging on the HTTP API stage if needed
- CloudFormation events: stack Events tab for resource create failures
- Amplify build logs: Amplify Console build history

10.4 Security baseline checklist

Control	Expected state
No public database access	DynamoDB is private by default and only accessible via IAM
Least privilege execution role	Lambda roles include only required DynamoDB actions
Token validation at edge	HTTP API JWT authorizer attached to all routes
Role enforcement	Agent routes check Agents group and return 403 when missing
Input validation	Handlers validate all request bodies and query params

Secret handling	No secrets committed; frontend uses public Cognito identifiers only

10.5 Cost controls and clean up

- DynamoDB on demand is simple for a proof of concept, but review costs if high traffic is expected
- Amplify Hosting has ongoing costs; delete the app after demos if not needed
- CloudWatch Logs can grow; set log retention to a reasonable value (for example 7 or 14 days) for review stacks
- To remove everything, delete the CloudFormation stack and the Amplify app

Clean up commands:

```
# Delete backend stack
aws cloudformation delete-stack --stack-name clouddesk-dev

# Confirm deletion progress
aws cloudformation describe-stacks --stack-name clouddesk-dev

# Amplify clean up is easiest in the console:
# Amplify -> App settings -> Delete app
```

Appendix A. Sample SAM template skeleton

This is a working starting point you can paste into backend/template.yaml and then refine. It implements one user pool plus an Agents group, one HTTP API with a default JWT authorizer, one DynamoDB table with one GSI, and four functions. You must adjust handler paths to match your repo and ensure your functions export a handler function.

```
AWSTemplateFormatVersion: "2010-09-09"
Transform: AWS::Serverless-2016-10-31
Description: CloudDesk serverless backend

Parameters:
  StageName:
    Type: String
    Default: dev
  CorsAllowedOrigin:
    Type: String
    Default: http://localhost:5173
  LogLevel:
    Type: String
    Default: INFO

Globals:
  Function:
    Runtime: nodejs18.x
    Timeout: 10
    MemorySize: 256
    Environment:
      Variables:
        LOG_LEVEL: !Ref LogLevel
        STAGE_NAME: !Ref StageName

Resources:
  UserPool:
    Type: AWS::Cognito::UserPool
    Properties:
      UserPoolName: !Sub "clouddesk-${StageName}-userpool"
      AutoVerifiedAttributes:
        - email
      UsernameAttributes:
        - email

  UserPoolClient:
    Type: AWS::Cognito::UserPoolClient
    Properties:
      UserPoolId: !Ref UserPool
      ClientName: !Sub "clouddesk-${StageName}-client"
      GenerateSecret: false
      ExplicitAuthFlows:
        - ALLOW_USER_SRP_AUTH
        - ALLOW_REFRESH_TOKEN_AUTH
        - ALLOW_USER_PASSWORD_AUTH

  AgentsGroup:
    Type: AWS::Cognito::UserPoolGroup
    Properties:
      GroupName: Agents
      UserPoolId: !Ref UserPool
      Description: Agent role group

  TicketsTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: !Sub "clouddesk-${StageName}-tickets"
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: PK
          AttributeType: S
        - AttributeName: SK
          AttributeType: S
        - AttributeName: GSI1PK
          AttributeType: S
        - AttributeName: GSI1SK
          AttributeType: S
      KeySchema:
        - AttributeName: PK
          KeyType: HASH
        - AttributeName: SK
          KeyType: RANGE
      GlobalSecondaryIndexes:
        - IndexName: GSI1
          KeySchema:
            - AttributeName: GSI1PK
              KeyType: HASH
            - AttributeName: GSI1SK
              KeyType: RANGE
          Projection:
            ProjectionType: ALL

  HttpApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      StageName: !Ref StageName
      CorsConfiguration:
        AllowOrigins:
          - !Ref CorsAllowedOrigin
        AllowHeaders:
          - Authorization
          - Content-Type
        AllowMethods:
          - GET
          - POST
          - PATCH
          - OPTIONS
      Auth:
        Authorizers:
          CognitoJwt:
            JwtConfiguration:
              issuer: !Sub "https://cognito-idp.${AWS::Region}.amazonaws.com/${UserPool}"
              audience:
                - !Ref UserPoolClient
            IdentitySource: "$request.header.Authorization"
        DefaultAuthorizer: CognitoJwt

  CreateTicketFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Handler: dist/handlers/createTicket.handler
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:PutItem
                - dynamodb:UpdateItem
                - dynamodb:TransactWriteItems
                - dynamodb:GetItem
                - dynamodb:Query
              Resource:
                - !GetAtt TicketsTable.Arn
                - !Sub "${TicketsTable.Arn}/index/*"
      Environment:
        Variables:
          TABLE_NAME: !Ref TicketsTable
      Events:
        CreateTicketRoute:
          Type: HttpApi
          Properties:
            ApiId: !Ref HttpApi
            Path: /tickets
            Method: POST
    Metadata:
      BuildMethod: esbuild
      BuildProperties:
        EntryPoints:
          - src/handlers/createTicket.ts
        Minify: false
        Target: es2020
        Sourcemap: true
        Platform: node

  ListMyTicketsFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Handler: dist/handlers/listMyTickets.handler
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:Query
              Resource:
                - !GetAtt TicketsTable.Arn
                - !Sub "${TicketsTable.Arn}/index/*"
      Environment:
        Variables:
          TABLE_NAME: !Ref TicketsTable
      Events:
        ListMyTicketsRoute:
          Type: HttpApi
          Properties:
            ApiId: !Ref HttpApi
            Path: /tickets
            Method: GET
    Metadata:
      BuildMethod: esbuild
      BuildProperties:
        EntryPoints:
          - src/handlers/listMyTickets.ts
        Minify: false
        Target: es2020
        Sourcemap: true
        Platform: node

  ListTicketsByStatusFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Handler: dist/handlers/listTicketsByStatus.handler
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:Query
              Resource:
                - !GetAtt TicketsTable.Arn
                - !Sub "${TicketsTable.Arn}/index/*"
      Environment:
        Variables:
          TABLE_NAME: !Ref TicketsTable
      Events:
        ListByStatusRoute:
          Type: HttpApi
          Properties:
            ApiId: !Ref HttpApi
            Path: /agent/tickets
            Method: GET
    Metadata:
      BuildMethod: esbuild
      BuildProperties:
        EntryPoints:
          - src/handlers/listTicketsByStatus.ts
        Minify: false
        Target: es2020
        Sourcemap: true
        Platform: node

  UpdateTicketStatusFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Handler: dist/handlers/updateTicketStatus.handler
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:UpdateItem
                - dynamodb:GetItem
                - dynamodb:Query
              Resource:
                - !GetAtt TicketsTable.Arn
                - !Sub "${TicketsTable.Arn}/index/*"
      Environment:
        Variables:
          TABLE_NAME: !Ref TicketsTable
      Events:
        UpdateStatusRoute:
          Type: HttpApi
          Properties:
            ApiId: !Ref HttpApi
            Path: /agent/tickets/{ticketId}
            Method: PATCH
    Metadata:
      BuildMethod: esbuild
      BuildProperties:
        EntryPoints:
          - src/handlers/updateTicketStatus.ts
        Minify: false
        Target: es2020
        Sourcemap: true
        Platform: node

Outputs:
  ApiBaseUrl:
    Value: !Sub "https://${HttpApi}.execute-api.${AWS::Region}.amazonaws.com/${StageName}"
  UserPoolId:
    Value: !Ref UserPool
  UserPoolClientId:
    Value: !Ref UserPoolClient
  Region:
    Value: !Ref AWS::Region
```

Note: the handler strings above assume you compile to dist/. If you prefer to deploy plain JavaScript without dist, adjust the Handler fields and disable esbuild or set Output. Also note: if you set StageName to $default, the base URL output should not include the /$default suffix.

Appendix B. Sample handler skeletons

These skeletons are intentionally verbose and include validation and consistent error handling. They are designed to be copy pasted into backend/src/handlers and compiled by esbuild.

backend/src/lib/response.ts

```
export function jsonResponse(statusCode: number, body: any) {
  return {
    statusCode,
    headers: {
      "content-type": "application/json"
    },
    body: JSON.stringify(body)
  };
}

export function errorResponse(statusCode: number, message: string, requestId: string) {
  return jsonResponse(statusCode, {
    error: "RequestError",
    message,
    requestId
  });
}
```

backend/src/lib/auth.ts

```
type Claims = Record<string, any>;

export function getClaims(event: any): Claims {
  return event?.requestContext?.authorizer?.jwt?.claims || {};
}

export function getUserSub(event: any): string | null {
  const claims = getClaims(event);
  return claims.sub || null;
}

export function getGroups(event: any): string[] {
  const claims = getClaims(event);
  const raw = claims["cognito:groups"];
  if (!raw) return [];
  if (Array.isArray(raw)) return raw;
  if (typeof raw === "string") return raw.split(",").map((s) => s.trim()).filter(Boolean);
  return [];
}

export function requireAuthenticated(event: any) {
  const userSub = getUserSub(event);
  if (!userSub) {
    throw new Error("Missing user identity");
  }
  return { userSub, groups: getGroups(event) };
}

export function requireAgent(event: any) {
  const auth = requireAuthenticated(event);
  const isAgent = auth.groups.includes("Agents");
  if (!isAgent) {
    const err = new Error("Agent role required");
    (err as any).statusCode = 403;
    throw err;
  }
  return auth;
}
```

backend/src/lib/ddb.ts

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
export const ddb = DynamoDBDocumentClient.from(client, {
  marshallOptions: { removeUndefinedValues: true }
});
```

backend/src/handlers/createTicket.ts

```
import { TransactWriteCommand } from "@aws-sdk/lib-dynamodb";
import { randomUUID } from "crypto";
import { ddb } from "../lib/ddb";
import { requireAuthenticated } from "../lib/auth";
import { jsonResponse, errorResponse } from "../lib/response";

function nowIso() {
  return new Date().toISOString();
}

function validateBody(body: any) {
  const title = body?.title;
  const description = body?.description;

  if (!title || typeof title !== "string" || title.trim().length < 3) {
    const err = new Error("Title is required and must be at least 3 characters");
    (err as any).statusCode = 400;
    throw err;
  }

  if (!description || typeof description !== "string" || description.trim().length < 5) {
    const err = new Error("Description is required and must be at least 5 characters");
    (err as any).statusCode = 400;
    throw err;
  }

  return { title: title.trim(), description: description.trim() };
}

export const handler = async (event: any) => {
  const requestId = event?.requestContext?.requestId || "unknown";
  try {
    const { userSub } = requireAuthenticated(event);

    const body = event?.body ? JSON.parse(event.body) : {};
    const { title, description } = validateBody(body);

    const ticketId = randomUUID();
    const createdAt = nowIso();
    const status = "OPEN";
    const tableName = process.env.TABLE_NAME;
    if (!tableName) {
      throw new Error("Missing TABLE_NAME");
    }

    const ticketMeta = {
      PK: "TICKET#" + ticketId,
      SK: "META",
      ticketId,
      ownerSub: userSub,
      status,
      createdAt,
      title,
      description,
      GSI1PK: "STATUS#" + status,
      GSI1SK: "CREATED#" + createdAt + "#TICKET#" + ticketId
    };

    const userLookup = {
      PK: "USER#" + userSub,
      SK: "TICKET#" + createdAt + "#" + ticketId,
      ticketId,
      status,
      createdAt,
      title
    };

    await ddb.send(new TransactWriteCommand({
      TransactItems: [
        { Put: { TableName: tableName, Item: ticketMeta } },
        { Put: { TableName: tableName, Item: userLookup } }
      ]
    }));

    return jsonResponse(201, { ticketId, status, createdAt });
  } catch (e: any) {
    const statusCode = e?.statusCode || 500;
    const message = statusCode === 500 ? "Internal server error" : (e?.message || "Error");
    console.log(JSON.stringify({ requestId, statusCode, message }));
    return errorResponse(statusCode, message, requestId);
  }
};
```

backend/src/handlers/listMyTickets.ts

```
import { QueryCommand } from "@aws-sdk/lib-dynamodb";
import { ddb } from "../lib/ddb";
import { requireAuthenticated } from "../lib/auth";
import { jsonResponse, errorResponse } from "../lib/response";

export const handler = async (event: any) => {
  const requestId = event?.requestContext?.requestId || "unknown";
  try {
    const { userSub } = requireAuthenticated(event);
    const tableName = process.env.TABLE_NAME;
    if (!tableName) throw new Error("Missing TABLE_NAME");

    const pk = "USER#" + userSub;

    const out = await ddb.send(new QueryCommand({
      TableName: tableName,
      KeyConditionExpression: "PK = :pk",
      ExpressionAttributeValues: { ":pk": pk },
      ScanIndexForward: false
    }));

    return jsonResponse(200, { tickets: out.Items || [] });
  } catch (e: any) {
    const statusCode = e?.statusCode || 500;
    const message = statusCode === 500 ? "Internal server error" : (e?.message || "Error");
    console.log(JSON.stringify({ requestId, statusCode, message }));
    return errorResponse(statusCode, message, requestId);
  }
};
```

backend/src/handlers/listTicketsByStatus.ts

```
import { QueryCommand } from "@aws-sdk/lib-dynamodb";
import { ddb } from "../lib/ddb";
import { requireAgent } from "../lib/auth";
import { jsonResponse, errorResponse } from "../lib/response";

const ALLOWED = new Set(["OPEN", "IN_PROGRESS", "RESOLVED"]);

export const handler = async (event: any) => {
  const requestId = event?.requestContext?.requestId || "unknown";
  try {
    requireAgent(event);

    const status = event?.queryStringParameters?.status || "OPEN";
    if (!ALLOWED.has(status)) {
      const err = new Error("Invalid status value");
      (err as any).statusCode = 400;
      throw err;
    }

    const tableName = process.env.TABLE_NAME;
    if (!tableName) throw new Error("Missing TABLE_NAME");

    const out = await ddb.send(new QueryCommand({
      TableName: tableName,
      IndexName: "GSI1",
      KeyConditionExpression: "GSI1PK = :pk",
      ExpressionAttributeValues: { ":pk": "STATUS#" + status },
      ScanIndexForward: false
    }));

    return jsonResponse(200, { tickets: out.Items || [] });
  } catch (e: any) {
    const statusCode = e?.statusCode || 500;
    const message = statusCode === 500 ? "Internal server error" : (e?.message || "Error");
    console.log(JSON.stringify({ requestId, statusCode, message }));
    return errorResponse(statusCode, message, requestId);
  }
};
```

backend/src/handlers/updateTicketStatus.ts (transition enforcement)

```
import { GetCommand, UpdateCommand } from "@aws-sdk/lib-dynamodb";
import { ddb } from "../lib/ddb";
import { requireAgent } from "../lib/auth";
import { jsonResponse, errorResponse } from "../lib/response";

const ALLOWED = new Set(["OPEN", "IN_PROGRESS", "RESOLVED"]);
const NEXT: Record<string, string[]> = {
  OPEN: ["IN_PROGRESS"],
  IN_PROGRESS: ["RESOLVED"],
  RESOLVED: []
};

export const handler = async (event: any) => {
  const requestId = event?.requestContext?.requestId || "unknown";
  try {
    requireAgent(event);

    const ticketId = event?.pathParameters?.ticketId;
    if (!ticketId) {
      const err = new Error("Missing ticketId");
      (err as any).statusCode = 400;
      throw err;
    }

    const body = event?.body ? JSON.parse(event.body) : {};
    const newStatus = body?.newStatus;
    if (!newStatus || typeof newStatus !== "string" || !ALLOWED.has(newStatus)) {
      const err = new Error("Invalid newStatus");
      (err as any).statusCode = 400;
      throw err;
    }

    const tableName = process.env.TABLE_NAME;
    if (!tableName) throw new Error("Missing TABLE_NAME");

    const key = { PK: "TICKET#" + ticketId, SK: "META" };

    const current = await ddb.send(new GetCommand({ TableName: tableName, Key: key }));
    if (!current.Item) {
      const err = new Error("Ticket not found");
      (err as any).statusCode = 404;
      throw err;
    }

    const oldStatus = current.Item.status;
    const allowedNext = NEXT[oldStatus] || [];
    if (!allowedNext.includes(newStatus)) {
      const err = new Error("Invalid status transition");
      (err as any).statusCode = 409;
      throw err;
    }

    const createdAt = current.Item.createdAt;

    await ddb.send(new UpdateCommand({
      TableName: tableName,
      Key: key,
      UpdateExpression: "SET #s = :ns, GSI1PK = :gpk, GSI1SK = :gsk",
      ExpressionAttributeNames: { "#s": "status" },
      ExpressionAttributeValues: {
        ":ns": newStatus,
        ":gpk": "STATUS#" + newStatus,
        ":gsk": "CREATED#" + createdAt + "#TICKET#" + ticketId,
        ":expected": oldStatus
      },
      ConditionExpression: "#s = :expected"
    }));

    // Update the user lookup record status (best effort)
    const ownerSub = current.Item.ownerSub;
    if (ownerSub && createdAt) {
      const userKey = { PK: "USER#" + ownerSub, SK: "TICKET#" + createdAt + "#" + ticketId };
      await ddb.send(new UpdateCommand({
        TableName: tableName,
        Key: userKey,
        UpdateExpression: "SET #s = :ns",
        ExpressionAttributeNames: { "#s": "status" },
        ExpressionAttributeValues: { ":ns": newStatus }
      }));
    }

    return jsonResponse(200, { ticketId, status: newStatus });
  } catch (e: any) {
    const statusCode = e?.statusCode || 500;
    const message = statusCode === 500 ? "Internal server error" : (e?.message || "Error");
    console.log(JSON.stringify({ requestId, statusCode, message }));
    return errorResponse(statusCode, message, requestId);
  }
};
```

Appendix C. IAM permissions checklist and least privilege notes

CloudDesk deployment touches multiple AWS services. The easiest path is to deploy using an administrative role, then reduce permissions once the stack is stable. If your IAM identity is already created, compare it against the checklist below.

C.1 Deployer permissions checklist (sam deploy)

- cloudformation: CreateStack, UpdateStack, DeleteStack, DescribeStacks, DescribeStackEvents, GetTemplateSummary
- s3: CreateBucket, PutObject, GetObject, DeleteObject, ListBucket (SAM artifact bucket)
- iam: CreateRole, AttachRolePolicy, PutRolePolicy, PassRole (SAM creates roles for Lambda and API Gateway integration)
- lambda: CreateFunction, UpdateFunctionCode, UpdateFunctionConfiguration, AddPermission
- apigateway: Create and manage HTTP API resources (v2)
- dynamodb: CreateTable, UpdateTable, DescribeTable
- cognito-idp: CreateUserPool, CreateUserPoolClient, CreateGroup, AdminCreateUser (for initial test users)
- logs: CreateLogGroup, PutRetentionPolicy

C.2 Quick deployment policy approach

For a short review deployment, attach AdministratorAccess to the deployer role temporarily. After the stack is stable, remove AdministratorAccess and replace it with a constrained policy.

C.3 Minimal Lambda execution policy example

This policy is attached to the Lambda execution role that SAM creates for each function. It grants only DynamoDB operations and CloudWatch logs. Adjust table ARN and index ARN accordingly.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DynamoDbAccess",
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:TransactWriteItems",
        "dynamodb:GetItem",
        "dynamodb:Query"
      ],
      "Resource": [
        "arn:aws:dynamodb:<region>:<account-id>:table/<table-name>",
        "arn:aws:dynamodb:<region>:<account-id>:table/<table-name>/index/*"
      ]
    },
    {
      "Sid": "Logs",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

C.4 Notes on security reviews

- Avoid embedding secrets in the frontend. Cognito user pool id and client id are public identifiers and are safe to ship to the browser
- Use HTTPS only. Amplify and CloudFront provide TLS automatically
- Restrict CORS origins to your known domains in production
- Consider adding rate limiting at API Gateway or WAF once the core flow is validated

References

Key documentation pages used while preparing this guide:

- CloudDesk repository: https://github.com/mabaan/CloudDesk
- Deploy with AWS SAM: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/using-sam-cli-deploy.html
- SAM sam deploy command reference: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-cli-command-reference-sam-deploy.html
- SAM AWS::Serverless::HttpApi: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-resource-httpapi.html
- SAM HttpApi Auth property: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-property-httpapi-httpapiauth.html
- API Gateway HTTP API JWT authorizer: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-jwt-authorizer.html
- Troubleshoot HTTP API JWT authorizers: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-troubleshooting-jwt.html
- Cognito user pool tokens: https://docs.aws.amazon.com/cognito/latest/developerguide/amazon-cognito-user-pools-using-tokens-with-identity-providers.html
- Amplify redirects and rewrites: https://docs.aws.amazon.com/amplify/latest/userguide/redirects.html
- Amplify rewrite examples for SPAs: https://docs.aws.amazon.com/amplify/latest/userguide/redirect-rewrite-examples.html
- CloudFront custom error responses: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/GeneratingCustomErrorResponses.html