Source Code .Docx File Type

Table of Contents

React Frontend:

1.App.js

```
import './App.css';
import Navbar from './Navbar';
import Uploadpage from './Uploadpage';
import Homepage from './Homepage';
import UserGuidepage from './UserGuidepage';
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";


function App() {
  return (
    <Router>
      <div className="App">
        <Navbar />
        <div className="content">
          <Routes>
            <Route path="/" element={<Homepage />} />
            <Route path="/upload" element={<Uploadpage />} />
            <Route path="/userguide" element={<UserGuidepage />} />
          </Routes>
        </div>
      </div>
    </Router>
  );
}

export default App;
```

2.App.css

```
.App {
  text-align: center;
  background-color: #f2ffe3;
  min-height: 100vh;
}

.App-logo {
  height: 40vmin;
  pointer-events: none;
}
```

```
@media (prefers-reduced-motion: no-preference) {
  .App-logo {
    animation: App-logo-spin infinite 20s linear;
  }
}

.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

3.Navbar.js

```
import { Link } from "react-router-dom";
import "./Navbar.css";


const Navbar = () => {
    return (
        <nav className="navbar" style={{ backgroundColor: '#333333', padding: '1rem',
color: 'white', boxShadow: "0 4px 8px rgba(0,0,0,0.3)"}}>
            <h1>Medical Bill Adjudication System</h1>
            <div className="links">
                <Link to="/" style={{ color: 'white', marginLeft: '1rem',
textDecoration: "none"}}>Home</Link>
```

```
                <Link to="/upload" style={{ color: 'white', marginLeft: '1rem',
textDecoration: "none"}}>Upload Bills</Link>
                <Link to="/userguide" style={{ color: 'white', marginLeft: '1rem',
textDecoration: "none"}}>User Guide</Link>
            </div>
        </nav>
    );
}

export default Navbar;
```

4.Navbar.css

```
.links a:hover{
    background-color: white;
    color: black !important;
    border-radius: 6px;
    padding: 0.35rem 0.7rem;
    transition: all 0.3s ease;
}
```

5.Homepage.js

```


const Homepage = () => {
    return(
        <div className="homepage" style={{ textAlign: "left", padding: "16px" }}>
            <h2>Home Page</h2>
            <h3 style={{borderBottom: "1px solid black", width: "100%", paddingBottom:
"8px"}}>Automated Medical Bill Adjudication System</h3>
            <p style={{ maxWidth: "700px" }}>
                Welcome and Thank you for using our service: Automated Medical Bill
Adjudication System - presented by Team 9.

                <br /> <br /> <br /> <br />
                The aim of this system is to decrease medical bill adjudication time
by leveraging Artificial Intelligence so both insurance companies and patients can
benefit from faster results.

                <br /><br />

                How to use:
                This website has 3 pages - you are currently at the Home page.
                You can navigate between the pages: 'Home', 'Upload Bills', and 'User
Guide' from the header of the pages.
                <br /><br />
                Look at the User Guide page to know the step by step process on how to
use our system.
                <br /><br />
                Once you know how to use this, navigate to the Upload Bills page to
upload your medical bill.
                <br /><br />
                Happy Adjudicating!




            </p>
        </div>
    );


}




export default Homepage;
```

6.Uploadpage.js

```
import { useState, useRef } from "react";
import { v4 as uuidv4 } from "uuid";
import "./Uploadpage.css";

const Uploadpage = () => {
  const [fileName, setFileName] = useState(null);
  const [resultsJSON, setResultsJSON] = useState(null);
  const fileInputRef = useRef();
  const [isProcessing, setIsProcessing] = useState(false);

  // API URL for uploading the PDF to Lambda 0
  const API_URL =
    "https://l83auaa4j8.execute-api.us-east-2.amazonaws.com/upload";

  // API URL to fetch the results json from Lambda 4
  const API_GET_URL =
    "https://l83auaa4j8.execute-api.us-east-2.amazonaws.com/testjson";

  // Opens the file selector dialog
  const handleButtonClick = () => {
    fileInputRef.current.click();
  };

  const handleReset = () => {
    setFileName(null);
    setResultsJSON(null);
    setIsProcessing(false);

    if (fileInputRef.current) {
      fileInputRef.current.value = "";
    }
  };

  // For fetching the results json
  const fetchResultsWithRetry = async (
    jobIdToCheck,
    retries = 6,
    delay = 30000
  ) => {
    for (let i = 0; i < retries; i++) {
      try {
        const response = await fetch(`${API_GET_URL}?jobId=${jobIdToCheck}`, {
          method: "GET",
          headers: { "Content-Type": "application/json" },
        });

        if (response.ok) {
          const data = await response.json();
          setResultsJSON(data);
          setIsProcessing(false);
          console.log("Fetched JSON:", data);
          return;
        } else if (response.status === 404) {
          console.log("Made An Attempt: JSON Not ready yet");
        } else {
          console.error("Error fetching JSON: ", response.statusText);
          break;
        }
      } catch (err) {
        console.error("Fetch error:", err);
      }

      await new Promise((res) => setTimeout(res, delay));
    }
    alert("Failed to fetch the results after multiple attempts");
  };

  // Handles the PDF upload
  const handleFileChange = (event) => {
    const file = event.target.files[0];
    if (file) {
      if (file.type !== "application/pdf") {
        alert(
          "Invalid file type. Please upload your medical bill in PDF format."
        );
        event.target.value = "";
        return;
      }

      setFileName(file.name);
      const newJobId = uuidv4();
      console.log("selected file:", file);
      const reader = new FileReader();
      reader.readAsDataURL(file);
      reader.onload = async () => {
        const base64Data = reader.result.split(",")[1];
        try {
          const response = await fetch(API_URL, {
            method: "POST",
            headers: {
              "Content-Type": "application/json",
            },
            body: JSON.stringify({
              job_id: newJobId,
              file_name: file.name,
              file_content: base64Data,
            }),
          });

          const data = await response.json();
          console.log("server response:", data);

          if (!response.ok) {
            alert(
              data.error || data.message || "Upload Failed. Lambda had an error"
            );
            return;
          }

          alert(data.message || "Upload Complete!");
          setIsProcessing(true);

          fetchResultsWithRetry(newJobId);
        } catch (err) {
          console.error("Upload error:", err);
          alert("Failed to upload file");
        }
      };
    }
  };

  // Filters the json (removes characters like '{}', '[]', '""', and commas)
  const renderJSON = (data, indent = 0) => {
    return Object.entries(data).map(([key, value]) => {
      const padding = " ".repeat(indent * 2);
      const displayKey = key
        .replaceAll("_", " ")
        .split(" ")
        .map((word) => word.charAt(0).toUpperCase() + word.slice(1))
        .join(" ");

      if (key === "coverage_analysis" && Array.isArray(value)) {
        return (
          <div key={key} style={{ marginTop: "20px", padding: "16px" }}>
            <h3 style={{ textAlign: "center" }}>Coverage Analysis</h3>

            <table className="coverage-table">
              <thead>
                <tr>
                  <th>Procedure</th>
                  <th>CPT Code</th>
                  <th>Billed</th>
                  <th>Covered</th>
                  <th>Coverage Type</th>
                  <th>Deductible Applies</th>
                  <th>Deductible Amount</th>
                  <th>Coinsurance Rate</th>
                  <th>Patient Responsibility</th>
                  <th>Insurance Pays</th>
                  <th>Explanation</th>
                </tr>
              </thead>
              <tbody>
                {value.map((proc, i) => (
                  <tr key={i}>
                    <td>{proc.procedure}</td>
                    <td>{proc.cpt_code}</td>
                    <td>{proc.billed_amount}</td>
                    <td>{proc.covered ? "Yes" : "No"}</td>
                    <td>{proc.coverage_type}</td>
                    <td>{proc.deductible_applies ? "Yes" : "No"}</td>
                    <td>{proc.deductible_amount}</td>
                    <td>{proc.coinsurance_rate}</td>
                    <td>{proc.patient_responsibility}</td>
                    <td>{proc.insurance_pays}</td>
                    <td>{proc.explanation}</td>
                  </tr>
                ))}
              </tbody>
            </table>
          </div>
        );
      }

      if (typeof value === "object" && value !== null) {
        return (
          <div key={key} style={{ marginLeft: indent * 10 }}>
            <p>
              {padding}
              <strong>{displayKey}:</strong>
            </p>
            {Array.isArray(value)
              ? value.map((item, i) => (
                  <div key={i} style={{ marginLeft: (indent + 1) * 10 }}>
                    {typeof item === "object"
                      ? renderJSON(item, indent + 2)
                      : item.toString()}
                  </div>
                ))
              : renderJSON(value, indent + 1)}
          </div>
        );
      } else {
        return (
          <p key={key} style={{ marginLeft: indent * 10 }}>
            {padding}
            <strong>{displayKey}:</strong>{" "}
            {value !== null && value !== undefined ? value.toString() : "null"}
          </p>
        );
      }
    });
  };

  return (
    <div className="uploadpage">
```

```
      <h2 style={{ textAlign: "left", padding: "16px" }}>Upload Page</h2>
      <p>
        <strong>
          When uploading the medical bills for adjudication make sure:
        </strong>
      </p>
      <p>
        <strong>File Format:</strong> PDF Only
      </p>
      <p>
        <strong>Maximum Pages:</strong> 3 Pages Per Bill
      </p>
      <button className="outline-btn" onClick={handleButtonClick}>
        Upload Bill
      </button>

      <input
        type="file"
        ref={fileInputRef}
        style={{ display: "none" }}
        onChange={handleFileChange}
        accept="application/pdf"
      />

      {fileName && <p>Selected file: {fileName}</p>}

      {isProcessing && !resultsJSON && (
        <p style={{ fontWeight: "bold", padding: "16px" }}>Processing...</p>
      )}

      {fileName && resultsJSON && (
        <div
          style={{
            display: "flex",
            flexDirection: "column",
            gap: "20px",
            padding: "16px",
          }}
        >
          {/* Results JSON (top)*/}
          <div
            className="fade-block"
            style={{
              flex: 1,
              overflowY: "auto",
              border: "1px solid #ccc",
              textAlign: "left",
              padding: "16px",
            }}
          >
            <h3 style={{ textAlign: "center" }}>Adjudication Results</h3>
            {renderJSON(resultsJSON)}
          </div>

          {/* PDF Viewer (below) */}
          <div style={{ border: "1px solid #ccc", height: "800px" }}>
            <iframe
              src={URL.createObjectURL(fileInputRef.current.files[0])}
              style={{ width: "100%", height: "100%" }}
              title="Uploaded PDF"
            ></iframe>
          </div>

          {/* Submit a New Bill Button */}
          <button
            className="outline-btn"
            onClick={handleReset}
            style={{ alignSelf: "center" }}
          >
            Upload a New Bill
          </button>
        </div>
      )}
    </div>
  );
};

export default Uploadpage;
```

7.Uploadpage.css

```
.fade-block {
    opacity: 0;
    animation: fadeInBlock 1.2s forwards;
}

@keyframes fadeInBlock {
    from { opacity: 0; }
    to { opacity: 1;}
}

.outline-btn {
    padding: 12px 24px;
```

```css
    background: transparent;
    border: 2px solid #4a4a4a;
    border-radius: 10px;
    font-size: 16px;
    font-weight: 600;
    cursor: pointer;
    transition: 0.25s ease;
    color: #4a4a4a;
}

.outline-btn:hover {
    background: #4a4a4a;
    color: white;
    transform: translateY(-2px);
    box-shadow: 0 4px 12px rgba(0,0,0,0.15);
}

.outline-btn:active {
    transform: translateY(0);
    box-shadow: none;
}

.coverage-table {
    width: 100%;
    border-collapse: collapse;
    font-size: 14px;
    border-radius: 8px;
    overflow: hidden;
    box-shadow: 0 2px 6px rgba(0,0,0,0.1);
}

.coverage-table th {
    padding: 10px;
    border-bottom: 1px solid #ddd;
    text-align: left;
    background: #f0f0f0;
    font-weight: bold;
}

.coverage-table td {
    padding: 10px;
    border-bottom: 1px solid #eee;
}

.coverage-table tr:nth-child(even) {
    background: #fafafa;
}
```

```
.coverage-table tr:nth-child(odd) {
    background: #fff;
}
```

8.Userguidepage.js

```
import step1 from "./step1.png";
import step2 from "./step2.png";
import step3 from "./step3.png";
import step4 from "./step4.png";

const UserGuidepage = () => {
    return(
        <div className="userguidepage" style={{textAlign: "left", padding: "16px"}}>
            <h2>User Guide Page</h2>
            <h3 style={{padding: "16px"}}>Follow the instructions below:</h3>
            <p style={{padding: "16px"}}><strong>Step 1:</strong> When you want to
process a medical bill click on Upload Bills.</p>
            <img src={step1} alt="Step 1" style={{width: "300px", margin: "16px",
borderRadius: "8px", boxShadow: "0px 4px 12px rgba(0, 0, 0, 0.15)"}} />
            <p style={{padding: "16px"}}><strong>Step 2:</strong> Click on the 'Upload
Bill' button and select your file.</p>
            <img src={step2} alt="Step 2" style={{width: "300px", margin: "16px",
borderRadius: "8px", boxShadow: "0px 4px 12px rgba(0, 0, 0, 0.15)"}} />
            <p style={{padding: "16px"}}><strong>Step 3:</strong> Wait until your
medical bill has been processed.</p>
            <img src={step3} alt="Step 3" style={{width: "300px", margin: "16px",
borderRadius: "8px", boxShadow: "0px 4px 12px rgba(0, 0, 0, 0.15)"}} />
            <p style={{padding: "16px"}}><strong>Step 4:</strong> When the medical
bill has been processed you'll see the results being displayed.</p>
            <img src={step4} alt="Step 4" style={{width: "300px", margin: "16px",
borderRadius: "8px", boxShadow: "0px 4px 12px rgba(0, 0, 0, 0.15)"}} />
            <h4>Information about the results:</h4>
            <p>Covered: This means the medical bill is covered by insurance policy and
the amount covered will be given in the table</p>
            <p>Not Covered: This means that the medical bill is not covered by the
insurance policy</p>
            <p>NOTE: If the bill has the wrong procedural and diagnosis codes the
system will not adjudicate the bill and it will return an invalid medical bill
message</p>
            <p>If that happens you have to resubmit the correct medical bill</p>


        </div>
    );
}


export default UserGuidepage;
```

AWS Backend:

Lambda 0

```
import json
import boto3
import base64
import os
import io

s3 = boto3.client('s3')
BUCKET_NAME = "farhantest01"

def lambda_handler(event, context):
    try:
        body = json.loads(event['body'])
        file_name = body['file_name']
        file_content = body['file_content'] # base64 encoded
        job_id = body.get('job_id')

        if not file_name or not file_content or not job_id:
            return {
                "statusCode": 400,
                "headers": {"Content-Type": "application/json"},
                "body": json.dumps({"error": "Missing job_id, file_name or
file_content"})
            }

        # if statement to check if its not a PDF document it returns a message
        if not file_name.lower().endswith(".pdf"):
            return {
                "statusCode":  400,
                "headers": {"Content-Type": "application/json"},
                "body": json.dumps({"error": "Invalid file type. Only PDF files are
allowed."})
            }

        # decode file content
        file_bytes = base64.b64decode(file_content)

        num_pages = file_bytes.count(b'/Type /Page')

        # if statement to check if the number of pages exceeds 3
        if num_pages > 3:
            return {
                "statusCode":  400,
                "headers": {"Content-Type": "application/json"},
                "body": json.dumps({"error": "Too many pages. Maximum 3 pages
allowed."})
            }

        # Upload to S3
        metadata = {
            "job_id": job_id
        }

        s3_key = file_name
        s3.put_object(Bucket=BUCKET_NAME, Key=s3_key, Body=file_bytes,
ContentType="application/pdf",Metadata=metadata)

        return {
            "statusCode": 200,
            "headers": {"Content-Type": "application/json"},
            "body": json.dumps({"message": f"File {file_name} uploaded successfully.
Job ID: {job_id}"})
        }

    except Exception as e:
            return {
                "statusCode": 500,
                "body": json.dumps({"error": str(e)})
            }
```

Lambda 1

```
import os, json, base64, logging, shutil, re, uuid, datetime, time, random
from typing import Any, Dict
from urllib.parse import unquote_plus
from decimal import Decimal
from string import Template  # kept for future use if needed

import boto3
from botocore.exceptions import (
    ClientError,
    EndpointConnectionError,
    ReadTimeoutError,
    ConnectTimeoutError,
    ConnectionClosedError,
)

# ======================================================================
# Logging Setup
# ======================================================================
logger = logging.getLogger()
logger.setLevel(logging.INFO)
_MAX_LOG_CHUNK = 60000


def _log_big(label: str, text: str | None):
    """
    Log large text in manageable chunks so CloudWatch doesn't truncate it.
    Used for logging long model responses or large JSON blobs.
    """
    if not text:
        logger.info(f"{label}: <empty>")
        return
    logger.info(f"{label} (len={len(text)}):")
    for i in range(0, len(text), _MAX_LOG_CHUNK):
        logger.info(text[i:i + _MAX_LOG_CHUNK])


# ======================================================================
# AWS Clients & Tables
# ======================================================================
AWS_REGION = os.environ.get("AWS_REGION", "us-east-2")

# Core AWS clients
s3 = boto3.client("s3")
bedrock_rt = boto3.client("bedrock-runtime", region_name=AWS_REGION)
ddb = boto3.resource("dynamodb", region_name=AWS_REGION)
lambda_client = boto3.client("lambda", region_name=AWS_REGION)

# Target Lambda (for chaining to the next step in the adjudication pipeline)
TARGET_LAMBDA_NAME = 'farhan_lambda_2'  # os.environ.get("Lambda_2")

# DynamoDB table names (overridable via environment variables for different
environments)
PATIENT_TABLE = os.environ.get("PATIENT_TABLE", "Patient_info")                # PK:
patient_id
PROVIDER_TABLE = os.environ.get("PROVIDER_TABLE", "Provider_info")             # PK:
provider_id
```

```
BILL_TABLE = os.environ.get("BILL_TABLE", "temporary_patientMedicalBillInfo") # PK:
table_id
BILL_ITEMS_TBL = os.environ.get("BILL_ITEMS_TABLE", "temporary_med_bill2")    # PK:
code, SK: table_id
ICD10_TABLE = os.environ.get("ICD10_TABLE", "temporary_med_icd")              # PK:
code, SK: table_id

# DynamoDB table resources
patient_tbl = ddb.Table(PATIENT_TABLE)
provider_tbl = ddb.Table(PROVIDER_TABLE)
bill_tbl = ddb.Table(BILL_TABLE)
bill_items_tbl = ddb.Table(BILL_ITEMS_TBL)
icd10_tbl = ddb.Table(ICD10_TABLE)

# =====================================================================
# Behavior Flags / Tunables
# =====================================================================
# These tunables allow you to adjust retry behavior and token limits via env vars.
RETRY_SLEEP_BASE = float(os.environ.get("RETRY_SLEEP_BASE", "0.8"))      # base delay
for backoff
RETRY_JITTER_MAX = float(os.environ.get("RETRY_JITTER_MAX", "0.6"))      # random
jitter added to backoff
MAX_RETRIES = int(os.environ.get("MAX_RETRIES", "8"))                    # max invoke
retries
EXTRA_COOLDOWN_ON_FAIL_SECS = float(os.environ.get("EXTRA_COOLDOWN_ON_FAIL_SECS",
"3.0"))
DOC_MAX_TOKENS = int(os.environ.get("DOC_MAX_TOKENS", "4500"))           # token
budget for full document extraction


# =====================================================================
# LLM Prompt (single full-document extraction)
# =====================================================================
# This prompt defines the strict JSON schema and behavior the LLM must follow.
FULL_DOC_INSTRUCTIONS = """
You are extracting structured data from a medical receipt. READ THE ENTIRE PDF FILE IN
ITS ENTIRETY.

Return ONLY a JSON object matching EXACTLY this schema:

{
  "patient_info": {
    "firstname": string|null,
    "lastname": string|null,
    "age": number|null,
    "phone": string|null,
    "address": string|null,
    "city": string|null,
    "state": string|null,
    "zipcode": string|null
  },
  "hospital_info": {
    "name": string|null,
    "phone": string|null,
    "address": string|null,
    "city": string|null,
    "state": string|null,
    "zipcode": string|null
  },
  "medical_bill_info": {
    "items": [
      {"code": string|null, "description": string|null, "bill": number|null}
    ],
    "subtotal": number|null,
    "discount": number|null,
    "tax_rate_percent": number|null,
    "total_tax": number|null,
    "balance_due": number|null
  },
  "icd_10_codes": [
    {"code": string|null, "description": string|null}
  ]
}

Rules:
- Output MUST be valid JSON only. No backticks, no markdown, no extra commentary.
- Amounts (bill, subtotal, discount, taxes, balance_due) MUST be numbers (no $ or
commas).
- If a field is missing on the document, set it to null.
- If a ZIP code appears inside an address, also copy it to zipcode.
- Do not invent data. Be conservative.
- Make sure to get all the medical data.
- "items" typically contain billing procedure codes (e.g., CPT/HCPCS) with amounts.
- "icd_10_codes" should contain any diagnosis codes labeled as ICD-10, DX, or similar,
and their human-readable descriptions.
- the "icd_10 codes" are not guaranteed to exist in the document. If you do not find
it, enter Null as the value
"""


# ======================================================================
# Helper Functions: Bedrock formatting and parsing
# ======================================================================
def _build_pdf_content_block(b64_data: str) -> Dict[str, Any]:
    """
    Build a Bedrock 'document' content block for a PDF.
    We assume all inputs are PDFs, so no format detection is needed.
    """
    return {
        "type": "document",
        "source": {
            "type": "base64",
            "media_type": "application/pdf",
            "data": b64_data,
        },
    }


def _first_text_block(body_json: Dict[str, Any]) -> str:
    """
    Extract concatenated text from the model's response body.
    Bedrock returns a list of 'content' blocks; we merge all text-type blocks.
    """
    content = body_json.get("content", [])
    return "\n".join([c.get("text", "") for c in content if c.get("type") ==
"text"]).strip()


# Regex to pull the outermost JSON object from the model text
_json_outer_re = re.compile(r"\{.*\}", re.DOTALL)


def _extract_json_from_text(text: str) -> Dict[str, Any]:
    """
    Extract the first JSON object found in the text and parse it.
    Raises ValueError if no JSON is found or parsing fails.
    This is defensive in case the model adds stray text around the JSON.
    """
    if not text:
        raise ValueError("Model returned empty text.")
    m = _json_outer_re.search(text)
    if not m:
        raise ValueError("Model did not return JSON.")
    return json.loads(m.group(0))


# Regex to detect general numeric substrings (for money, percentages, etc.)
_money_re = re.compile(r"[-+]?\d*\.?\d+")


def _to_number(x):
    """
    Convert various numeric-like values to float (or None).
    Strips currency symbols and commas from strings.
    Intended to normalize LLM outputs for numeric fields.
    """
    if x is None:
        return None
    if isinstance(x, (int, float)):
        return float(x)
    if isinstance(x, str):
        m = _money_re.search(x.replace(",", ""))
        return float(m.group(0)) if m else None
    return None


# Regex to detect US ZIP codes in arbitrary address strings
_zip_re = re.compile(r"\b(\d{5})(?:-\d{4})?\b")


def _maybe_zip_from_address(addr):
    """
    Try to extract a ZIP code from an address string.
    If not found, returns None.
    """
    if not addr or not isinstance(addr, str):
        return None
    m = _zip_re.search(addr)
    return m.group(0) if m else None


def _to_decimal(n) -> Decimal | None:
    """
    Convert to Decimal where possible for DynamoDB numeric fields.
    DynamoDB requires Decimal for non-integer numeric types.
    """
    if n is None:
        return None
    if isinstance(n, Decimal):
        return n
    if isinstance(n, (int, float)):
        return Decimal(str(n))
    if isinstance(n, str):
        try:
            return Decimal(n)
        except Exception:
            return None
    return None


def _clean_for_ddb(obj):
    """
    Recursively clean Python object for DynamoDB:
    - Remove None and empty strings (to avoid validation issues).
    - Convert int/float to Decimal.
    - Preserve nested structures and strings.
    This is applied to items right before DDB writes.
    """
    if isinstance(obj, dict):
        out = {}
        for k, v in obj.items():
            cleaned = _clean_for_ddb(v)
            if cleaned is None:
                continue
            if isinstance(cleaned, str) and cleaned == "":
                continue
            out[k] = cleaned
        return out

    elif isinstance(obj, list):
        return [
            x
            for x in (_clean_for_ddb(v) for v in obj)
            if not (x is None or x == "")
        ]

    elif isinstance(obj, (int, float)):
        return _to_decimal(obj)

    elif isinstance(obj, (str, Decimal)):
        return obj

    else:
        return obj


def _put_safe(table, item: dict):
    """
    Clean and write an item safely into DynamoDB.
    - Sanitizes the item with _clean_for_ddb
    - Performs put_item
    Returns the cleaned item for logging/debugging.
    """
    item_clean = _clean_for_ddb(item)
    table.put_item(Item=item_clean)
    return item_clean


# ======================================================================
# Bedrock Invocation Helpers (with retry/backoff)
```

```python
# =====================================================================
def _is_retryable_client_error(e: ClientError) -> bool:
    """
    Determine whether a ClientError is retryable (throttling, 5xx, etc.).
    This prevents immediate hard failure on transient Bedrock issues.
    """
    err = e.response.get("Error", {}) if hasattr(e, "response") else {}
    code = (err.get("Code") or "").lower()
    msg = (err.get("Message") or "")
    retryable_codes = {
        "throttlingexception",
        "toomanyrequestsexception",
        "serviceunavailableexception",
        "internalfailure",
        "internalservererror",
        "requesttimeout",
        "limitexceededexception",
    }
    if code in retryable_codes:
        return True
    if "too many requests" in str(msg).lower():
        return True
    status = (
        e.response.get("ResponseMetadata", {}).get("HTTPStatusCode")
        if hasattr(e, "response")
        else None
    )
    if status and (status == 429 or 500 <= status < 600):
        return True
    return False


def _sleep_with_backoff(attempt: int):
    """
    Exponential backoff with random jitter based on the attempt number.
    Backoff = RETRY_SLEEP_BASE * 2^(attempt-1) + random jitter.
    """
    delay = (RETRY_SLEEP_BASE * (2 ** (attempt - 1))) + random.uniform(
        0, RETRY_JITTER_MAX
    )
    time.sleep(delay)


def _bedrock_invoke(model_id: str, payload: dict) -> dict:
    """
    Robust Bedrock invocation with automatic retries and backoff.
    - Handles network issues, throttling, and 5xx responses.
    - Raises RuntimeError after exhausting MAX_RETRIES.
    """
    body_bytes = json.dumps(payload).encode("utf-8")
    last_err = None

    for attempt in range(1, MAX_RETRIES + 1):
        try:
            resp = bedrock_rt.invoke_model(modelId=model_id, body=body_bytes)
            return json.loads(resp["body"].read())
        except (
            EndpointConnectionError,
            ReadTimeoutError,
            ConnectTimeoutError,
            ConnectionClosedError,
        ) as e:
            # Low-level connectivity or timeout issues → retry
            last_err = e
            logger.warning(
                f"Bedrock connection/timeout (attempt {attempt}/{MAX_RETRIES}): {e}"
            )
            _sleep_with_backoff(attempt)
            continue
        except ClientError as e:
            # AWS service-level error; decide if retryable based on code/status
            last_err = e
            if _is_retryable_client_error(e):
                logger.warning(
                    f"Bedrock retryable error (attempt {attempt}/{MAX_RETRIES}): {e}"
                )
                _sleep_with_backoff(attempt)
                continue
            # Non-retryable → propagate immediately
            logger.error(f"Bedrock non-retryable error: {e}", exc_info=True)
            raise
        except Exception as e:
            # Unknown error; treat as transient and retry until limit
            last_err = e
            logger.warning(
                f"Bedrock invoke transient error (attempt {attempt}/{MAX_RETRIES}):
{e}",
                exc_info=True,
            )
            _sleep_with_backoff(attempt)

    # All retries exhausted
    logger.error("Bedrock invoke failed after retries.", exc_info=True)
    raise RuntimeError(f"Bedrock invoke failed after retries: {last_err}")
```

```python
def _invoke_full_extraction(model_id: str, content_block: dict) -> Dict[str, Any]:
    """
    Invoke the model ONCE for the entire PDF to extract structured data.
    The prompt and PDF content_block are sent together as a single message.
    """
    messages = [
        {
            "role": "user",
            "content": [
                {"type": "text", "text": FULL_DOC_INSTRUCTIONS},
                content_block,
            ],
        }
    ]
    payload = {
        "anthropic_version": "bedrock-2023-05-31",
        "messages": messages,
        "max_tokens": DOC_MAX_TOKENS,
        "temperature": 0,
    }

    logger.info("Invoking model for full-document extraction ...")
    body = _bedrock_invoke(model_id, payload)
    raw_text = _first_text_block(body)
    _log_big("[full-doc] Full OCR/Model output", raw_text)
    _log_big("[full-doc] Model raw (first 2k)", raw_text[:2000])
    return _extract_json_from_text(raw_text)


# ======================================================================
# Lambda Chaining Helper
# ======================================================================
def _trigger_next_lambda(
    bucket: str,
    key: str,
    version_id: str | None,
    table_id: str | None,
    job_id: str | None,
):
    """
    Fire-and-forget invocation of the next Lambda in the pipeline.
    Passes:
      - S3 bucket/key/version (for context)
      - table_id (linking to this bill in DynamoDB)
      - job_id (from S3 object metadata, used to correlate to frontend job)
    """
    if not TARGET_LAMBDA_NAME:
        logger.warning("TARGET_LAMBDA_NAME not set; skipping trigger.")
        return

    payload = {
        "source": "lambda-chain",
        "s3": {
            "bucket_name": bucket,
            "object_key": key,
            "version_id": version_id,
        },
        "table_id": table_id,
        "job_id": job_id,
    }

    logger.info(f"Triggering {TARGET_LAMBDA_NAME} with payload: 
{json.dumps(payload)}")

    # InvocationType="Event" → asynchronous invocation; this Lambda does not wait for 
the next one
    lambda_client.invoke(
        FunctionName=TARGET_LAMBDA_NAME,
        InvocationType="Event",
        Payload=json.dumps(payload).encode("utf-8"),
    )
    logger.info(
        f"Triggered next lambda {TARGET_LAMBDA_NAME} for s3://{bucket}/{key}"
        + (f"?versionId={version_id}" if version_id else "")
    )


# ======================================================================
# Main Lambda Handler
# ======================================================================
def lambda_handler(event, context):
    """
    Entry point for S3 -> Lambda:
    - Triggered by S3 PUT (PDF upload).
    - For each PDF:
      * Read PDF from S3 (and read job_id from S3 object metadata).
      * Call Bedrock ONCE with the entire PDF to extract structured JSON.
      * Write patient, provider, bill, items, and ICD-10 rows to DynamoDB.
      * Use ONE table_id per PDF (whole bill).
      * Trigger the next Lambda with table_id and job_id.
      * Delete the source PDF from S3 after processing (cleanup).
    """
    tmp_root = "/tmp"
    model_id = os.environ["MODEL_ID"]

    results = []
    # Keep last parsed info for final debug prints
    last_patient_info = None
    last_hospital_info = None
    last_medical_bill_info = None

    try:
        # ------------------------------------------------------------------
        # 1. Validate event and get S3 records
        # ------------------------------------------------------------------
        records = event.get("Records", [])
        if not records:
            logger.info("No S3 records in event.")
            return {"results": []}

        # ------------------------------------------------------------------
        # 2. Process each S3 record (each uploaded PDF)
        # ------------------------------------------------------------------
        for idx, rec in enumerate(records, start=1):
            bucket = rec["s3"]["bucket"]["name"]
            key = unquote_plus(rec["s3"]["object"]["key"])
            size = rec["s3"]["object"].get("size")
            version_id = None  # ignore object versions for now

            logger.info(f"[{idx}] Start s3://{bucket}/{key} size={size}")

            parsed_any_page = False
            # Conceptually treating the whole PDF as "page 1"
            page_results = []
            table_id: str | None = None
            job_id: str | None = None

            try:
                # ----------------------------------------------------------
                # 2.1 Fetch PDF from S3 and read job_id from metadata
                # ----------------------------------------------------------
                obj = s3.get_object(Bucket=bucket, Key=key)
                content_type = obj.get("ContentType")
                logger.info(f"[{idx}] S3 ContentType: {content_type}")

                # Metadata is lowercase-keyed in S3; expect 'job_id' set by uploader
                metadata = (obj.get("Metadata") or {})
                job_id = metadata.get("job_id")
                logger.info(f"[{idx}] S3 object metadata: {metadata}
(job_id={job_id})")

                # Read full PDF bytes and encode as base64 for Bedrock document input
                file_bytes = obj["Body"].read()
                b64 = base64.b64encode(file_bytes).decode("utf-8")
                # We assume all inputs are PDFs, so directly build a PDF content
block.
                content_block = _build_pdf_content_block(b64)

                # ---------------------------------------------------------
                # 2.2 Single full-document extraction via Bedrock
                # ---------------------------------------------------------
                created_at = datetime.datetime.utcnow().isoformat(
                    timespec="seconds"
                ) + "Z"

                # Generate identifiers for this bill and associated entities
                patient_id = str(uuid.uuid4())
                provider_id = str(uuid.uuid4())
                table_id = str(uuid.uuid4())

                try:
                    data = _invoke_full_extraction(model_id, content_block)
                except Exception as e:
                    # Any model or parsing failure for this PDF is captured, but we
continue

                    logger.error(
                        f"[{idx}] Error during full-doc extraction: {e}",
                        exc_info=True,
                    )
                    page_results.append(
                        {
                            "page_no": 1,
                            "parsed": False,
                            "error": str(e),
                        }
                    )
                    # Small cooldown to avoid hammering the model if many failures
occur

                    time.sleep(EXTRA_COOLDOWN_ON_FAIL_SECS)
                    results.append(
                        {
                            "bucket": bucket,
                            "key": key,
                            "parsed_any_page": False,
                            "pages": page_results,
                        }
                    )
                    # control flow continues to 'finally' where we still trigger next
lambda and delete

                    continue
```

```python
# ----------------------------------------------------------
# 2.3 Normalize JSON segments from full document
# ----------------------------------------------------------
p = data.get("patient_info", {}) or {}
h = data.get("hospital_info", {}) or {}
m = data.get("medical_bill_info", {}) or {}
icd_10_list = data.get("icd_10_codes", []) or []

# ---------------- Patient ----------------
patient_info = {
    "firstname": p.get("firstname"),
    "lastname": p.get("lastname"),
    "age": _to_number(p.get("age")),
    "phone": p.get("phone"),
    "address": p.get("address"),
    "city": p.get("city"),
    "state": p.get("state"),
    "zipcode": p.get("zipcode"),
}
# If zipcode missing, try to infer it from the address string
if not patient_info["zipcode"]:
    patient_info["zipcode"] = _maybe_zip_from_address(
        patient_info["address"]
    )

# ---------------- Hospital / Provider ----------------
hospital_info = {
    "name": h.get("name"),
    "phone": h.get("phone"),
    "address": h.get("address"),
    "city": h.get("city"),
    "state": h.get("state"),
    "zipcode": h.get("zipcode"),
}
if not hospital_info["zipcode"]:
    hospital_info["zipcode"] = _maybe_zip_from_address(
        hospital_info["address"]
    )

# ---------------- Bill Items ----------------
# The LLM returns a list of dicts with code/description/bill
items = []
for it in (m.get("items") or []):
    items.append(
        {
            "code": it.get("code"),
            "description": it.get("description"),
            "bill": _to_number(it.get("bill")),
        }
    )

# ---------------- Bill Summary ----------------
medical_bill_info = {
    "subtotal": _to_number(m.get("subtotal")),
    "discount": _to_number(m.get("discount")),
    "tax_rate_percent": _to_number(
        m.get("tax_rate_percent")
    ),
    "total_tax": _to_number(m.get("total_tax")),
    "balance_due": _to_number(m.get("balance_due")),
}

# -------------------------------------------------
# 2.4 Persist to DynamoDB (patient, provider, bill)
# -------------------------------------------------
# Patient row (one per PDF)
patient_item = {
    "patient_id": patient_id,
    "created_at": created_at,
    "table_id": table_id,
    "source_bucket": bucket,
    "source_key": key,
    "page_no": 1,  # full-doc treated as single logical page
    **patient_info,
}
patient_item = _put_safe(patient_tbl, patient_item)

# Provider/hospital row (one per PDF)
provider_item = {
    "provider_id": provider_id,
    "created_at": created_at,
    "table_id": table_id,
    "source_bucket": bucket,
    "source_key": key,
    "page_no": 1,
    **hospital_info,
}
provider_item = _put_safe(provider_tbl, provider_item)

# Bill summary row (one per PDF)
bill_item = {
    "table_id": table_id,
    "created_at": created_at,
    "source_bucket": bucket,
    "source_key": key,
    "patient_id": patient_id,
            "provider_id": provider_id,
            "page_no": 1,
            **medical_bill_info,
        }
        # Ensure numeric fields are Decimal for DynamoDB
        for k in (
            "subtotal",
            "discount",
            "tax_rate_percent",
            "total_tax",
            "balance_due",
        ):
            if bill_item.get(k) is not None:
                bill_item[k] = _to_decimal(bill_item[k])
        bill_item = _put_safe(bill_tbl, bill_item)

        # --------------------------------------------------
        # 2.5 Write bill items to DynamoDB (one row per item)
        # --------------------------------------------------
        # These are the line items (CPT/HCPCS with amounts) linked by
table_id.
        if items:
            with bill_items_tbl.batch_writer() as batch:
                for it in items:
                    bill_dec = _to_decimal(it.get("bill"))
                    code_val = it.get("code")
                    # Guarantee a non-empty PK even if the LLM didn't find a
code
                    if not code_val or code_val == "":
                        code_val = f"NO_CODE_{uuid.uuid4()}"
                    item_row = {
                        "code": code_val,  # PK
                        "table_id": table_id,  # SK (in GSI or composite PK
design)
                        "description": it.get("description"),
                        "bill": bill_dec,
                        "created_at": created_at,
                        "page_no": 1,
                    }
                    batch.put_item(
                        Item=_clean_for_ddb(item_row)
                    )

        # --------------------------------------------------
        # 2.6 Write ICD-10 codes to icd_10_reference_table
        #       PK: code, SK: table_id
        # --------------------------------------------------
        icd10_count = 0
        for icd in icd_10_list:
            code_val = (icd or {}).get("code")
            desc_val = (icd or {}).get("description")
            if not code_val:
                # Skip entries with no code; description alone isn't useful as
a PK
                continue
            icd_item = {
                "code": code_val,       # partition key
                "table_id": table_id,    # sort key
                "description": desc_val,
                "created_at": created_at,
                "source_bucket": bucket,
                "source_key": key,
            }
            _put_safe(icd10_tbl, icd_item)
            icd10_count += 1

        logger.info(
            f"[{idx}] DDB writes OK: "
            f"{PATIENT_TABLE}(patient_id={patient_id}), "
            f"{PROVIDER_TABLE}(provider_id={provider_id}), "
            f"{BILL_TABLE}(table_id={table_id}), "
            f"{BILL_ITEMS_TBL}(items={len(items)}), "
            f"{ICD10_TABLE}(icd10_codes={icd10_count})"
        )

        # Track last successfully parsed info for debug/demo prints
        last_patient_info = patient_info
        last_hospital_info = hospital_info
        last_medical_bill_info = {
            **medical_bill_info,
            "items": items,
        }
        parsed_any_page = True

        # Capture per-PDF summary for the function response
        page_results.append(
            {
                "page_no": 1,
                "parsed": True,
                "table_id": table_id,
                "patient_id": patient_id,
                "provider_id": provider_id,
                "items_count": len(items),
                "icd10_count": icd10_count,
                "patient_info": patient_info,
                "hospital_info": hospital_info,
                "medical_bill_info": medical_bill_info,
            }
        )

    except Exception as e:
        # Catch-all to avoid the whole batch failing due to one PDF
        logger.error(
            f"[{idx}] Error while processing {key}: {e}", exc_info=True
        )
    finally:
        # -----------------------------------------------------------
        # 2.7 Trigger next Lambda (adjudication step)
        # -----------------------------------------------------------
        try:
            if table_id is not None:
                # Debug print for CloudWatch
                print(table_id)
                logger.info(f"[{idx}] Triggering next lambda with
table_id={table_id}, job_id={job_id}")
                _trigger_next_lambda(bucket, key, version_id, table_id,
job_id)
            else:
                logger.warning(
                    f"[{idx}] Skipping next-lambda trigger for
s3://{bucket}/{key} "
                    f"because table_id is None (processing failed early)."
                )
        except Exception as e:
            logger.error(
                f"Failed to trigger next lambda for s3://{bucket}/{key}: {e}",
                exc_info=True,
            )

        # -----------------------------------------------------------
        # 2.8 Delete the source file from S3 after processing
        # -----------------------------------------------------------
        try:
            if bucket and key:
                logger.info(f"[{idx}] Deleting source object
s3://{bucket}/{key}")
                s3.delete_object(Bucket=bucket, Key=key)
                logger.info(f"[{idx}] Deleted source object
s3://{bucket}/{key}")
        except Exception as e:
            # Deletion failure is logged but not fatal to the Lambda
            logger.error(
                f"[{idx}] Failed to delete source object s3://{bucket}/{key}:
{e}",
                exc_info=True,
            )

        # Store result summary for this record
        results.append(
            {
                "bucket": bucket,
                "key": key,
                "parsed_any_page": parsed_any_page,
                "pages": page_results,
            }
        )
        logger.info(
            f"[{idx}] Done with full PDF (pages
processed={len(page_results)})."
        )

logger.info(f"Processed {len(records)} record(s).")
return {"results": results}

finally:
# ================================================================
# Final debug/demo prints (non-critical) and /tmp cleanup
# ================================================================
# These prints are for manual testing insight; they do not affect logic.
print("Testing")
try:
    if last_patient_info:
        print(
            last_patient_info.get("firstname"),
            last_patient_info.get("lastname"),
        )
    if last_hospital_info:
        print(last_hospital_info.get("phone"))
    if last_medical_bill_info:
        print(last_medical_bill_info.get("balance_due"))
        for item in last_medical_bill_info.get("items", []):
            code = item.get("code")
            desc = item.get("description")
            bill = item.get("bill")
            print(f"{code} - {desc}: ${bill}")
except Exception as e:
    logger.warning(f"Demo prints failed: {e}")

# Clean up any temporary files/directories in /tmp to keep the environment
clean
try:
    for name in os.listdir(tmp_root):
        path = os.path.join(tmp_root, name)
        try:
            if os.path.isdir(path):
                shutil.rmtree(path, ignore_errors=True)
            else:
                os.remove(path)
        except Exception as e:
            logger.warning(f"Failed to remove {path}: {e}")
    logger.info("Cleaned /tmp.")
except FileNotFoundError:
    # /tmp may not exist or may already be cleaned
    pass
```

Lambda 2

```
import json
import os
import boto3
import logging
from boto3.dynamodb.conditions import Attr
from botocore.exceptions import ClientError
from typing import List, Dict, Optional, Tuple, Any

# ---------------------------------------------------------------------
# AWS & logging setup
# ---------------------------------------------------------------------
region_name = "us-east-2"

logger = logging.getLogger()
logger.setLevel(logging.INFO)
_MAX_LOG_CHUNK = 60000  # reserved for potential large-log helpers

# Core AWS clients/resources
dynamodb = boto3.resource("dynamodb", region_name)
bedrock = boto3.client("bedrock-runtime", region_name)
lambda_client = boto3.client("lambda", region_name=region_name)

# Name/ARN for downstream Lambda in the adjudication pipeline
NEXT_LAMBDA_NAME = os.environ.get("Lambda_3")

# ---------------------------------------------------------------------
# DynamoDB tables
# ---------------------------------------------------------------------
# Line items (CPT etc.) for a given bill, keyed by code + table_id
```

```python
med_bill_table = dynamodb.Table("temporary_med_bill2")

# Temporary ICD codes extracted from the bill, linked via table_id
icd_temp_table = dynamodb.Table("temporary_med_icd")

# Reference CPT code table: authoritative codes + descriptions
reference_table = dynamodb.Table("reference_table")

# Reference ICD table: authoritative ICD codes + descriptions
icd_reference_table = dynamodb.Table("icd_10_reference_table")

# Token budget for Bedrock responses
MAX_TOKENS = 100


# ------------------------------------------------------------------
def trigger_next_lambda(previous_event, validation_result, cpt_codes):
    """
    Fire-and-forget invocation of the next Lambda in the pipeline.

    - Takes the original event (from upstream Lambda),
      the validation_result produced here, and the list of CPT codes.
    - Attaches `validation` and `cpt_codes` to the payload.
    - Invokes NEXT_LAMBDA_NAME asynchronously (InvocationType="Event").
    """
    if not NEXT_LAMBDA_NAME:
        print("TARGET_NEXT_LAMBDA_NAME not set; skipping next-lambda trigger.")
        return

    # Preserve as much of the original event as possible to keep context
    if isinstance(previous_event, dict):
        payload = dict(previous_event)
        payload["validation"] = validation_result
    else:
        payload = {"original_event": previous_event, "validation": validation_result}

    # Always pass CPT list so downstream stages can display them or reuse them
    payload["cpt_codes"] = cpt_codes

    print(f"Triggering {NEXT_LAMBDA_NAME} with payload: {json.dumps(payload)}")

    try:
        lambda_client.invoke(
            FunctionName=NEXT_LAMBDA_NAME,
            InvocationType="Event",  # async, no need to wait for next Lambda
            Payload=json.dumps(payload).encode("utf-8"),
        )
    except Exception as e:
        # Trigger failure does not crash this Lambda; we just log it
        print(f"Error triggering next lambda {NEXT_LAMBDA_NAME}: {e}")


# ---------------------------------------------------------------------
def get_cpt_codes_for_table_id(table_id: str) -> List[Dict[str, str]]:
    """
    Fetch all CPT-like entries from med_bill_table for a given table_id.

    Returns:
        A list of dicts: [{"code": "...", "description": "..."}, ...]
        - Uses a Scan with FilterExpression on table_id.
        - Handles pagination with LastEvaluatedKey.
    """
    try:
        # Initial scan filtered by table_id
        response = med_bill_table.scan(
            FilterExpression=Attr("table_id").eq(table_id)
        )
        items = response.get("Items", [])

        # Paginate through all results if more than 1 page
        while "LastEvaluatedKey" in response:
            response = med_bill_table.scan(
                FilterExpression=Attr("table_id").eq(table_id),
                ExclusiveStartKey=response["LastEvaluatedKey"],
            )
            items.extend(response.get("Items", []))

        codes = []
        for item in items:
            # Only build entries that have a "code" field
            if "code" in item:
                codes.append({
                    "code": item["code"],
                    "description": item.get("description", "")
                })

        return codes

    except Exception as e:
        print(f"Error fetching CPT codes: {e}")
        return []


# ---------------------------------------------------------------------
def get_icd_entries_for_table_id(table_id: str) -> List[Dict[str, Optional[str]]]:
    """
    Fetch ICD entries from the temporary ICD table for a given table_id.

    Expected item shape:
        { "code": "...", "table_id": "...", "description": "..." }

    Returns:
        A list of dicts: [{"code": "...", "description": "... or None"}, ...]
        - Uses a Scan with FilterExpression on table_id.
        - Handles pagination with LastEvaluatedKey.
    """
    try:
        response = icd_temp_table.scan(
            FilterExpression=Attr("table_id").eq(table_id)
        )
        items = response.get("Items", [])

        # Handle pagination
        while "LastEvaluatedKey" in response:
            response = icd_temp_table.scan(
                FilterExpression=Attr("table_id").eq(table_id),
                ExclusiveStartKey=response["LastEvaluatedKey"],
            )
            items.extend(response.get("Items", []))

        icd_entries = []
        for item in items:
            # Only accept items that actually have a 'code'
            if "code" in item:
                icd_entries.append({
                    "code": item["code"],
                    "description": item.get("description")
                })

        return icd_entries

    except Exception as e:
        print(f"Error fetching ICD entries: {e}")
        return []


# ---------------------------------------------------------------------
def get_reference_cpt_code(code: str):
    """
    Look up a CPT code in the reference_table.

    Returns:
        {"code": "...", "description": "..."} if found, else None.
        - Uses a direct GetItem with PK = code.
    """
    try:
        response = reference_table.get_item(Key={"code": code})
        item = response.get("Item")
        if not item:
            return None
        return {"code": item["code"], "description": item.get("description", "")}
    except Exception:
        # On any error (network, missing table, etc.) just treat as not found
        return None


def get_reference_icd_code(icd_code: str):
    """
    Look up an ICD code in the icd_reference_table.

    Returns:
        {"code": "...", "description": "..."} if found, else None.
        - Uses a direct GetItem with PK = code.
    """
    try:
        response = icd_reference_table.get_item(Key={"code": icd_code})
        item = response.get("Item")
        if not item:
            return None
        return {"code": item["code"], "description": item.get("description", "")}
    except Exception:
        # On any error, treat as if reference entry does not exist
        return None


# ------------------------------------------------------------------
def batch_compare_with_bedrock(pairs: List[Tuple[str, str]], label: str):
    """
    Compare multiple (description_A, description_B) pairs using a single Bedrock call.

    Args:
        pairs: list of (desc_from_bill, desc_from_reference) tuples.
        label: "CPT" or "ICD" (used in the instructions).

    Returns:
        List[Optional[bool]]:
          - True  => pair deemed equivalent
          - False => pair not equivalent
          - None  => model response could not be interpreted or an error occurred

    Design:
      - We build a numbered list of pairs.
      - Prompt the model to respond with a JSON list like ["YES","NO",...].
      - Parse and normalize each element to boolean/None.
    """
    if not pairs:
        return []

    # Build pair text to feed into the model (1-based numbered list)
    lines = []
    for idx, (d1, d2) in enumerate(pairs, start=1):
        lines.append(f"{idx}. Description A: {d1}")
        lines.append(f"   Description B: {d2}")

    prompt = f"""
You are a medical coding expert.
Determine whether each pair of {label} descriptions are equivalent. They may have
different wordings and abbreviations, but as long as they deliver similar message, it
should be valid. For example, for CPT codes: "Left Foot X-ray test" and "X-ray
examination of foot" delivers similar message and should be determined valid.
But also, do not be too lenient. For example for ICD codes: "Chest pain on breathing"
and "Heart disease" should not be determined to be giving similar message.
Respond with ["YES","NO",...] only.
Pairs:
{chr(10).join(lines)}
"""

    try:
        body = json.dumps({
            "anthropic_version": "bedrock-2023-05-31",
            "max_tokens": MAX_TOKENS,
            "messages": [{"role": "user", "content": prompt}],
            "temperature": 0,
        })

        # Single Bedrock model invocation for all pairs
        response = bedrock.invoke_model(
            modelId=os.environ.get("MODEL_ID"),
            body=body,
        )

        # Expect a JSON-encoded array of strings like ["YES","NO",...]
        response_body = json.loads(response["body"].read())
        text = response_body["content"][0]["text"].strip()

        decisions = json.loads(text)

        results = []
        for val in decisions:
            s = str(val).upper()
```

```python
            if s.startswith("Y"):
                results.append(True)
            elif s.startswith("N"):
                results.append(False)
            else:
                # Unable to interpret this element; mark as None
                results.append(None)

        return results

    except Exception:
        # If the entire call fails, we return a list of None aligned with `pairs`
        return [None] * len(pairs)


# ---------------------------------------------------------------------
def check_cpt_justification_with_bedrock(cpt_codes, icd_codes):
    """
    Ask Bedrock whether each CPT is medically justified by at least one ICD.

    Args:
        cpt_codes: list of {"code": "...", "description": "..."} from bill.
        icd_codes: list of {"code": "...", "description": "..."} for justification.

    Returns:
        (justified, message)
        - justified == True  => all CPTs justified; message is None
        - justified == False => some CPTs not justified; message is short explanation
        - justified == None  => Bedrock error; message is error string
    """
    # Construct human-readable lists for the model
    cpt_text = "\n".join([f"- CPT {c['code']}: {c['description']}" for c in
cpt_codes])
    icd_text = "\n".join([f"- ICD {i['code']}: {i['description']}" for i in
icd_codes])

    prompt = f"""
You are a medical coding expert.
ICD diagnoses:
{icd_text}

CPT services:
{cpt_text}

Determine whether EVERY CPT is medically justified by at least one ICD.
If all are justified, respond with: OK
Otherwise respond with: ISSUE: <short reason>.
"""
```

```python
    try:
        body = json.dumps({
            "anthropic_version": "bedrock-2023-05-31",
            "max_tokens": MAX_TOKENS,
            "messages": [{"role": "user", "content": prompt}],
            "temperature": 0,
        })

        response = bedrock.invoke_model(
            modelId=os.environ.get("MODEL_ID"),
            body=body
        )

        text = json.loads(response["body"].read())["content"][0]["text"].strip()

        # Normalize the decision:
        if text.upper().startswith("OK"):
            return True, None
        else:
            # Any non-OK response is treated as an issue; return text verbatim
            return False, text

    except Exception as e:
        # If Bedrock fails, surface the error but don't hard-crash the whole Lambda
        return None, str(e)


# ---------------------------------------------------------------------
def lambda_handler(event, context):
    """
    Lambda entrypoint for validation stage.

    Responsibilities:
      - Read table_id from event.
      - Fetch temporary ICD and CPT entries for that table_id.
      - Compare ICD descriptions against ICD reference table (via Bedrock).
      - Compare CPT descriptions against CPT reference table (via Bedrock).
      - Check that CPT codes are justified by (valid) ICD codes (via Bedrock).
      - Aggregate issues and all_valid flag.
      - Pass validation_result and CPT codes to NEXT_LAMBDA_NAME.

    Returns:
      bool: all_valid (True if all checks passed, False otherwise).
    """
    all_valid = True          # Global flag tracking if everything is valid
    issues = []               # Accumulates human-readable issue strings
    cpt_icd_issue = None      # Holds any CPT/ICD justification issue text
    cpt_codes = []            # CPTs found for this table_id (passed downstream)

    print("Event:", json.dumps(event))

    try:
        table_id = event.get("table_id")
        if not table_id:
            # table_id is mandatory to continue; fail and trigger next lambda with
error state
            issues.append("Missing table_id in event.")
            validation_result = {"table_id": None, "all_valid": False, "issues":
issues}
            trigger_next_lambda(event, validation_result, [])
            return False

        # --------------------- ICD extraction & validation ---------------------
        icd_entries = get_icd_entries_for_table_id(table_id)

        # This will hold ICDs (with chosen descriptions) that we use for CPT
justification
        icd_for_justification = []

        if not icd_entries:
            issues.append(f"No ICD entries found for table_id {table_id}")
        else:
            icd_pairs = []                # (bill_description, ref_description) for
Bedrock
            icd_items_for_validation = []  # structured tracking of each pair

            for entry in icd_entries:
                icd_code = entry["code"]
                icd_desc = entry.get("description")

                # Look up the ICD code in the reference table
                ref_icd = get_reference_icd_code(icd_code)

                if ref_icd:
                    ref_desc = ref_icd["description"]

                    if icd_desc:
                        # Will be compared via Bedrock
                        icd_items_for_validation.append({
                            "code": icd_code,
                            "icd_desc": icd_desc,
                            "ref_desc": ref_desc
                        })
                        icd_pairs.append((icd_desc, ref_desc))
                else:
                        # No temp description; rely entirely on reference description
                        icd_for_justification.append({
                            "code": icd_code,
                            "description": ref_desc
                        })
                else:
                    # ICD is not in reference table
                    if icd_desc:
                        issues.append(
                            f"ICD {icd_code} not found in reference; using temporary
description."
                        )
                        icd_for_justification.append({
                            "code": icd_code,
                            "description": icd_desc
                        })
                    else:
                        # No ref entry and no temp description → cannot use this ICD
for justification
                        issues.append(
                            f"ICD {icd_code} missing description and not in reference;
skipping."
                        )

            # Bedrock validation for ICD descriptions (only those with both
temp+reference descriptions)
            if icd_pairs:
                results = batch_compare_with_bedrock(icd_pairs, "ICD")
                for item, result in zip(icd_items_for_validation, results):
                    icd_code = item["code"]
                    ref_desc = item["ref_desc"]

                    if result is False:
                        # Description mismatch between bill and reference
                        issues.append(f"ICD {icd_code} description mismatch.")
                        all_valid = False
                        # Still use the reference description for justification checks
                        icd_for_justification.append({"code": icd_code, "description":
ref_desc})
                    else:
                        # If result True or None, we fall back to reference
description for justification
                        icd_for_justification.append({"code": icd_code, "description":
ref_desc})

        # --------------------- CPT extraction & ICD justification -------------------
--
        cpt_codes = get_cpt_codes_for_table_id(table_id)
```

```python
    if not cpt_codes:
        issues.append("No CPT codes found.")
        all_valid = False
    else:
        if icd_for_justification:
            # Ask Bedrock if each CPT is justified by at least one ICD
            justified, msg = check_cpt_justification_with_bedrock(
                cpt_codes, icd_for_justification
            )
            if justified is False:
                # Model explicitly says some CPTs are not justified
                issues.append(msg)
                cpt_icd_issue = msg
                all_valid = False
            elif justified is None:
                # Model call failed
                issues.append("Bedrock error during CPT justification.")
        else:
            # No ICDs at all that we can trust/use
            issues.append("No ICD codes available to justify CPTs.")

    # --------------------- CPT reference validation ---------------------
    cpt_pairs = []                # (bill_desc, reference_desc)
    cpt_items_for_validation = [] # structured metadata for each pair

    for item in cpt_codes:
        code = item["code"]
        desc = item.get("description", "")

        # Look up CPT in reference table
        ref = get_reference_cpt_code(code)
        if not ref:
            issues.append(f"CPT {code} not found in CPT reference table.")
            continue

        ref_desc = ref["description"]
        cpt_items_for_validation.append({
            "code": code,
            "description": desc,
            "ref_description": ref_desc
        })
        cpt_pairs.append((desc, ref_desc))

    # Only call Bedrock if there are pairs to validate
    if cpt_pairs:
        results = batch_compare_with_bedrock(cpt_pairs, "CPT")
        for item, result in zip(cpt_items_for_validation, results):
                if result is False:
                    # Description mismatch between claim and reference for this CPT
                    issues.append(f"CPT {item['code']} description mismatch.")
                    all_valid = False

        # --------------------- Final aggregation & chaining ---------------------
        validation_result = {
            "table_id": table_id,
            "all_valid": all_valid,
            "issues": issues,
            "cpt_icd_justification_issue": cpt_icd_issue,
        }

        # Hand off results + CPT codes to the next Lambda
        trigger_next_lambda(event, validation_result, cpt_codes)
        return all_valid

    except Exception as e:
        # Any unexpected exception is treated as a failed validation
        issues.append(str(e))
        # table_id may not exist if the failure happened very early
        trigger_next_lambda(event, {"table_id": table_id, "all_valid": False,
"issues": issues}, [])
        return False
```

Lambda 3

9./fetchPatientData.py

```
import boto3
import os
import logging
from typing import List, Dict, Optional
from boto3.dynamodb.conditions import Key
from datetime import datetime
from boto3.dynamodb.conditions import Attr



logger = logging.getLogger()
AWS_REGION = os.environ.get("AWS_REGION", "us-east-2")

dynamodb = boto3.resource('dynamodb', region_name=AWS_REGION)
```

```python
# Table names from environment variables
LAMBDA2_TRACKING_TABLE = os.environ.get("LAMBDA2_TRACKING_TABLE",
"lambda2_executions")
PATIENT_INFO_TABLE = os.environ.get("PATIENT_INFO_TABLE", "Patient_info")
MED_BILL_TABLE = os.environ.get("MED_BILL_TABLE", "temporary_med_bill2")
PROVIDER_TABLE = os.environ.get("PROVIDER_TABLE", "Provider_info")
TEMP_MED_BILL_TABLE = os.environ.get("TEMP_MED_BILL_TABLE",
"temporary_patientMedicalBillInfo")
TEMP_ICD = os.environ.get("TEMP_ICD", "temporary_med_icd")

tracking_table = dynamodb.Table(LAMBDA2_TRACKING_TABLE)
patient_table = dynamodb.Table(PATIENT_INFO_TABLE)
med_bill_table = dynamodb.Table(MED_BILL_TABLE)
provider_table = dynamodb.Table(PROVIDER_TABLE)  # or whatever your table name is
medical_bill_table = dynamodb.Table(TEMP_MED_BILL_TABLE)
icd_temp_table = dynamodb.Table(TEMP_ICD)

# def get_latest_table_id() -> Optional[str]:
#     """
#     Fetch the most recent table_id from Lambda 2's tracking table.
#     Assumes the tracking table has a timestamp attribute for sorting.
#     """
#     try:
#         # Query or scan to get the most recent execution
#         response = tracking_table.scan(
#             Limit=50  # Get recent items
#         )

#         items = response.get('Items', [])

#         if not items:
#             logger.warning("No executions found in Lambda 2 tracking table")
#             return None

#         # Sort by timestamp (assuming timestamp field exists)
#         sorted_items = sorted(
#             items,
#             key=lambda x: x.get('timestamp', ''),
#             reverse=True
#         )

#         latest_item = sorted_items[0]
#         table_id = latest_item.get('table_id')

#         logger.info(f"Latest table_id from Lambda 2: {table_id}")
#         return table_id

#     except Exception as e:
```

```python
#         logger.error(f"Error fetching latest table_id: {e}")
#         return None

def get_patient_info_by_table_id(table_id: str):
    """
    Fetch patient info from Patient_info table using table_id (non-key).
    Uses scan because table_id is not the PK.
    """
    try:
        response = patient_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        items = response.get('Items', [])

        if not items:
            logger.warning(f"No patient info found for table_id: {table_id}")
            return None

        return items[0]

    except Exception as e:
        logger.error(f"Error scanning patient info by table_id: {e}")
        return None

def get_provider_info_by_table_id(table_id: str):
    """
    Fetch provider info from provider_info table using table_id (non-key).
    Uses scan because table_id is not the PK.
    """
    try:
        response = provider_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        items = response.get('Items', [])

        if not items:
            logger.warning(f"No provider info found for table_id: {table_id}")
            return None

        return items[0]

    except Exception as e:
        logger.error(f"Error scanning provider info by table_id: {e}")
        return None

def get_medical_bill_info_by_table_id(table_id: str):
    """
    Fetch medical bill info from temporary_patientMedicalBillInfo table using table_id (PK).
    """
    try:
        response = medical_bill_table.get_item(
            Key={'table_id': table_id}
        )

        item = response.get('Item')

        if not item:
            logger.warning(f"No medical bill info found for table_id: {table_id}")
            return None

        return item

    except Exception as e:
        logger.error(f"Error getting medical bill info by table_id: {e}")
        return None

def get_charge_by_code(code: str, table_id: str):
    """
    Fetch charge amount from temporary_med_bill2 table using code (PK).
    """
    try:
        response = med_bill_table.get_item(
            Key={
                'code': code,  # partition key
                'table_id': table_id            # sort key
            }
        )

        item = response.get('Item')

        if not item:
            logger.warning(f"No charge found for code: {code}")
            return None

        return item

    except Exception as e:
        logger.error(f"Error getting charge by code: {e}")
        return None


def get_patient_info(table_id: str) -> Optional[Dict]:
    """
    Fetch patient information from Patient_info table using table_id as FK.
    """
    try:
        # Query using table_id as the key
        # response = patient_table.query(
        #     KeyConditionExpression=Key('table_id').eq(table_id)
        # )
        response = patient_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        print(f'response = {response}')
        items = response.get('Items', [])

        if not items:
            logger.warning(f"No patient info found for table_id: {table_id}")
            return None

        # Return the first matching record
        patient_data = items[0]

        logger.info(f"Patient data retrieved for table_id {table_id}: "
                    f"{patient_data.get('patient_name', 'Unknown')}")

        return patient_data

    except Exception as e:
        logger.error(f"Error fetching patient info: {e}")
        return None


def get_codes_for_table_id(table_id: str) -> List[Dict[str, str]]:
    """
    Fetch all CPT codes associated with a table_id from temporary_med_bill2.
    Returns a list of dicts with 'code' and 'description' keys.
    """
    try:
        response = med_bill_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        items = response.get('Items', [])

        # Handle pagination if there are many items
        while 'LastEvaluatedKey' in response:
            response = med_bill_table.query(
                KeyConditionExpression=Key('table_id').eq(table_id),
                ExclusiveStartKey=response['LastEvaluatedKey']
            )
            items.extend(response.get('Items', []))

        # Extract code and description
        codes = []
        for item in items:
            if 'code' in item:
                codes.append({
                    'code': item['code'],
                    'description': item.get('description', ''),
                    'charge_amount': item.get('charge_amount', 0)
                })

        logger.info(f"Found {len(codes)} codes for table_id {table_id}")
        return codes

    except Exception as e:
        logger.error(f"Error fetching codes from DynamoDB: {e}")
        return []


##################################
# DELETION LOGIC
def delete_patient_info(table_id: str) -> bool:
    """
    Delete patient info record by table_id.
    Assumes patient_id is the primary key.
    """
    try:
        # First, find the record to get the primary key
        response = patient_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        items = response.get('Items', [])

        if not items:
            logger.warning(f"No patient info found for table_id: {table_id}")
            return False

        # Delete each matching record (should be only one)
        for item in items:
            patient_id = item.get('patient_id')  # Assuming patient_id is the PK
            if patient_id:
                patient_table.delete_item(Key={'patient_id': patient_id})
                logger.info(f"Deleted patient_id: {patient_id} (table_id:
{table_id})")
```

```python
        return True

    except Exception as e:
        logger.error(f"Error deleting patient info for table_id {table_id}: {e}")
        return False


def delete_provider_info(table_id: str) -> bool:
    """
    Delete provider info record by table_id.
    Assumes provider_id is the primary key.
    """
    try:
        response = provider_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        items = response.get('Items', [])

        if not items:
            logger.warning(f"No provider info found for table_id: {table_id}")
            return False

        for item in items:
            provider_id = item.get('provider_id')  # Assuming provider_id is the PK
            if provider_id:
                provider_table.delete_item(Key={'provider_id': provider_id})
                logger.info(f"Deleted provider_id: {provider_id} (table_id:
{table_id})")

        return True

    except Exception as e:
        logger.error(f"Error deleting provider info for table_id {table_id}: {e}")
        return False


def delete_medical_bill_info(table_id: str) -> bool:
    """
    Delete medical bill info record by table_id (PK).
    """
    try:
        response = medical_bill_table.delete_item(
            Key={'table_id': table_id}
        )

        logger.info(f"Deleted medical bill info for table_id: {table_id}")
        return True
    except Exception as e:
        logger.error(f"Error deleting medical bill info for table_id {table_id}: {e}")
        return False


def delete_cpt_codes(table_id: str) -> bool:
    """
    Delete all CPT codes (from temporary_med_bill2) for a given table_id.
    Composite key: (code, table_id)
    """
    try:
        # Scan for all items with this table_id
        response = med_bill_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        items = response.get('Items', [])
        deleted_count = 0

        # Handle pagination
        while True:
            for item in items:
                code = item.get('code')
                if code:
                    med_bill_table.delete_item(
                        Key={
                            'code': code,
                            'table_id': table_id
                        }
                    )
                    deleted_count += 1

            # Check for more pages
            if 'LastEvaluatedKey' in response:
                response = med_bill_table.scan(
                    FilterExpression=Attr('table_id').eq(table_id),
                    ExclusiveStartKey=response['LastEvaluatedKey']
                )
                items = response.get('Items', [])
            else:
                break

        logger.info(f"Deleted {deleted_count} CPT codes for table_id: {table_id}")
        return True

    except Exception as e:
        logger.error(f"Error deleting CPT codes for table_id {table_id}: {e}")
        return False


def delete_icd_codes(table_id: str) -> bool:
    """
    Delete all ICD codes (from temporary_med_icd) for a given table_id.
    Composite key: (code, table_id)
    """
    try:
        response = icd_temp_table.scan(
            FilterExpression=Attr('table_id').eq(table_id)
        )

        items = response.get('Items', [])
        deleted_count = 0

        # Handle pagination
        while True:
            for item in items:
                code = item.get('code')
                if code:
                    icd_temp_table.delete_item(
                        Key={
                            'code': code,
                            'table_id': table_id
                        }
                    )
                    deleted_count += 1

            if 'LastEvaluatedKey' in response:
                response = icd_temp_table.scan(
                    FilterExpression=Attr('table_id').eq(table_id),
                    ExclusiveStartKey=response['LastEvaluatedKey']
                )
                items = response.get('Items', [])
            else:
                break

        logger.info(f"Deleted {deleted_count} ICD codes for table_id: {table_id}")
        return True

    except Exception as e:
        logger.error(f"Error deleting ICD codes for table_id {table_id}: {e}")
        return False


def cleanup_all_data_for_table_id(table_id: str) -> Dict[str, bool]:
    """
    Delete ALL data associated with a table_id from all temporary tables.
    Returns a dict showing success/failure for each table.
    """
    results = {
        'patient_info': False,
        'provider_info': False,
        'medical_bill_info': False,
        'cpt_codes': False,
        'icd_codes': False
    }

    logger.info(f"Starting cleanup for table_id: {table_id}")

    # Delete from all tables
    results['patient_info'] = delete_patient_info(table_id)
    results['provider_info'] = delete_provider_info(table_id)
    results['medical_bill_info'] = delete_medical_bill_info(table_id)
    results['cpt_codes'] = delete_cpt_codes(table_id)
    results['icd_codes'] = delete_icd_codes(table_id)

    # Log summary
    success_count = sum(results.values())
    logger.info(f"Cleanup complete for table_id {table_id}: "
                f"{success_count}/5 operations successful")

    return results
```

10. Index.py

```
import os, json, base64, logging, shutil, uuid
from datetime import datetime
from typing import Any, Dict, Optional
from urllib.parse import unquote_plus
import boto3
from parsingPolicies import parse_policy_pdf
from parsingBills import parse_bill_pdf
from fetchPatientData import *
from decimal import Decimal

# ---------- Logging ----------
logger = logging.getLogger()
logger.setLevel(logging.INFO)
_MAX_LOG_CHUNK = 60000

def _log_big(label: str, text: str | None):
    if not text:
        logger.info(f"{label}: <empty>")
```

```python
        return
    logger.info(f"{label} (len={len(text)}):")
    for i in range(0, len(text), _MAX_LOG_CHUNK):
        logger.info(text[i:i+_MAX_LOG_CHUNK])

# ---------- AWS Clients ----------
AWS_REGION = os.environ.get("AWS_REGION", "us-east-2")
MAIN_BUCKET = os.environ.get("MAIN_BUCKET", "chitest02")  # <-- add this
s3 = boto3.client("s3")
bedrock_rt = boto3.client("bedrock-runtime", region_name=AWS_REGION)

# ---------- Configuration ----------
POLICY_PREFIX = "policies/"
PARSED_PREFIX = "parsed/"
MODEL_ID = os.environ.get("MODEL_ID")


# ---------- Helper Functions ----------
def _delete_object(bucket: str, key: str) -> bool:
    """Delete an S3 object"""
    try:
        resp = s3.delete_object(Bucket=bucket, Key=key)
        logger.info(f"Deleted: s3://{bucket}/{key}")
        return True
    except Exception as e:
        logger.error(f"Delete failed for s3://{bucket}/{key}: {e}")
        return False

def decimal_to_number(obj):
    """Convert Decimal objects to int or float for JSON serialization"""
    if isinstance(obj, Decimal):
        return int(obj) if obj % 1 == 0 else float(obj)
    return obj


def _check_parsed_policy_exists(bucket: str, policy_key: str) -> bool:
    """Check if parsed policy JSON already exists"""
    try:
        filename = policy_key.split('/')[-1].replace('.pdf', '.json')
        parsed_key = f"{PARSED_PREFIX}policies/{filename}"

        s3.head_object(Bucket=bucket, Key=parsed_key)
        logger.info(f"Parsed policy already exists: s3://{bucket}/{parsed_key}")
        return True
    except s3.exceptions.ClientError:
        return False

def _get_latest_policy(bucket: str) -> Optional[Dict[str, Any]]:
    """Fetch the most recent parsed policy JSON"""
    try:
        response = s3.list_objects_v2(
            Bucket=bucket,
            Prefix=f"{PARSED_PREFIX}policies/",
            MaxKeys=100
        )

        if 'Contents' not in response or not response['Contents']:
            logger.warning("No parsed policies found")
            return None

        # Sort by LastModified descending
        objects = sorted(response['Contents'], key=lambda x: x['LastModified'],
reverse=True)
        latest_key = objects[0]['Key']

        logger.info(f"Fetching latest policy: s3://{bucket}/{latest_key}")
        obj = s3.get_object(Bucket=bucket, Key=latest_key)
        policy_data = json.loads(obj['Body'].read().decode('utf-8'))

        return policy_data
    except Exception as e:
        logger.error(f"Failed to fetch latest policy: {e}", exc_info=True)
        return None

def _build_bill_data_from_dynamodb(table_id: str) -> Optional[Dict[str, Any]]:
    """
    Construct bill data structure from DynamoDB tables.
    Returns a dictionary matching the expected bill_data format,
    or None if something is missing.
    """
    logger.info(f"Building bill_data from DynamoDB for table_id={table_id}")

    # Patient info
    patient_info = get_patient_info_by_table_id(table_id)
    if not patient_info:
        logger.warning(f"No patient info found for table_id={table_id}")
        return None

    # Fetch provider info
    provider_info = get_provider_info_by_table_id(table_id)
    if not provider_info:
        logger.warning(f"No provider info found for table_id={table_id}")
        return None

    # Fetch medical bill info
    bill_info = get_medical_bill_info_by_table_id(table_id)
    if not bill_info:
        logger.warning(f"No medical bill info found for table_id={table_id}")
        return None


    # CPT / procedure codes
    cpt_codes = get_codes_for_table_id(table_id)
    if not cpt_codes:
        logger.warning(f"No CPT codes found for table_id={table_id}")
        cpt_codes = []

    # Base structure
    bill_data: Dict[str, Any] = {
        "table_id": table_id,
        "patient_id": patient_info.get("patient_id", ""),
        "provider_id": provider_info.get("provider_id", ""),
        "created_at": patient_info.get("created_at", ""),
        "source_bucket": patient_info.get("source_bucket", MAIN_BUCKET),
        "source_key": patient_info.get("source_key", f"dynamodb:{table_id}"),
        "patient_info": {
            "firstname": patient_info.get("firstname", ""),
            "lastname": patient_info.get("lastname", ""),
            "phone": patient_info.get("phone", ""),
            "address": patient_info.get("address", ""),
            "city": patient_info.get("city", ""),
        },
        "hospital_info": {
            "name": provider_info.get("name", ""),
            "phone": provider_info.get("phone", ""),
            "address": provider_info.get("address", ""),
            "city": provider_info.get("city", ""),
        },
        "medical_bill_info": {
            "subtotal": decimal_to_number(bill_info.get("subtotal")),
            "tax_rate_percent": decimal_to_number(bill_info.get("tax_rate_percent")),
            "total_tax": decimal_to_number(bill_info.get("total_tax")),
            "balance_due": decimal_to_number(bill_info.get("balance_due"))

        },
        "items": []
    }

    # Add items (procedures/codes)
    for code_item in cpt_codes:
        code = code_item.get("code", "")

        # Fetch charge from temporary_med_bill2 table
        charge_data = get_charge_by_code(code, table_id)
        if charge_data:
            charge = decimal_to_number(charge_data.get("bill"))
        else:
            # Fallback to code_item if not found in temporary_med_bill2
            charge = float(code_item.get("charge_amount", 0) or 0)

        item = {
            "code": code,
            "description": code_item.get("description", ""),
            "bill": charge
        }
        bill_data["items"].append(item)

    logger.info(
        f"Built bill_data from DynamoDB for table_id={table_id} "
        f"with {len(bill_data['items'])} items, "
        f"balance_due={bill_info.get('balance_due')}"
    )
    print(f"Debug: This is bill_data: {bill_data}")
    return bill_data


# def _compare_bill_to_policy(bill_data: dict, policy_data: dict, bucket: str) ->
Optional[str]:
def _compare_bill_to_policy(bill_data: dict, policy_data: dict, bucket: str, key: str,
metadata) -> Optional[str]:
    """Compare bill against policy and save comparison results"""
    max_retries = 5
    base_delay = 2  # seconds

    try:
        prompt = f"""You are an insurance coverage analyst. Compare this medical bill
against the insurance policy and identify what's covered and how much is covered.

POLICY:
{json.dumps(policy_data, indent=2)}

MEDICAL BILL:
{json.dumps(bill_data, indent=2)}

Analyze and return ONLY a single JSON object with this structure:
{{
  "bill_summary": {{
    "patient": "...",
    "provider": "...",
    "total_billed": 0.00,
    "procedure_count": 0
  }},
  "coverage_analysis": [
    {{
      "procedure": "...",
      "cpt_code": "...",
      "billed_amount": 0.00,
      "covered": true/false,
      "coverage_type": "in-network/out-of-network/not covered",
      "deductible_applies": true/false,
      "deductible_amount": 0.00,
      "coinsurance_rate": "xx%",
      "patient_responsibility": 0.00,
      "insurance_pays": 0.00,
      "explanation": "..."
    }}
  ],
  "totals": {{
    "total_billed": 0.00,
    "total_covered": 0.00,
    "total_patient_owes": 0.00,
    "total_insurance_pays": 0.00,
    "breakdown": {{
      "deductible": 0.00,
      "coinsurance": 0.00,
      "copay": 0.00,
      "not_covered": 0.00
    }}
  }},
  "notes": ["Any important details about coverage limits, exclusions, etc."]
}}"""

        messages = [{
            "role": "user",
            "content": [{"type": "text", "text": prompt}]
        }]

        payload = {
            "anthropic_version": "bedrock-2023-05-31",
            "messages": messages,
            "max_tokens": 4000,
            "temperature": 0
        }

        resp = bedrock_rt.invoke_model(modelId=MODEL_ID, body=json.dumps(payload))
        body = json.loads(resp["body"].read())

        content = body.get("content", [])
        text_blocks = [c.get("text", "") for c in content if c.get("type") == "text"]
        comparison_text = "\n".join([t for t in text_blocks if t])

        logger.info("=== COVERAGE COMPARISON ===")
        _log_big("Comparison Result", comparison_text)

        # Parse JSON from response
        import re
        json_match = re.search(r'\{.*\}', comparison_text, re.DOTALL)
        if json_match:
            comparison_result = json.loads(json_match.group(0))
        else:
            comparison_result = {"raw_response": comparison_text}

        # Save comparison results
        comparison_id = str(uuid.uuid4())
        # timestamp = datetime.datetime.utcnow().isoformat(timespec="seconds") + "Z"
        timestamp = datetime.utcnow().isoformat(timespec="seconds") + "Z"


        comparison_output = {
            "comparison_id": comparison_id,
            "timestamp": timestamp,
            "bill_table_id": bill_data.get("table_id"),
            "bill_source_key": bill_data.get("source_key"),
            "policy_id": policy_data.get("policy_id", "unknown"),
            "comparison": comparison_result
        }

        # Save to S3
        filename = f"comparison_{comparison_id}.json"
        comparison_key = f"{PARSED_PREFIX}comparisons/{filename}"

        s3.put_object(
            Bucket=bucket,
            Key=comparison_key,
            Body=json.dumps(comparison_output, indent=2, default=str),
            ContentType="application/json",
            Metadata=metadata
        )

        logger.info(f"Saved comparison to s3://{bucket}/{comparison_key}")
        return comparison_key

    except ClientError as e:
        if e.response['Error']['Code'] == 'ThrottlingException':
            if attempt < max_retries - 1:
                delay = base_delay * (2 ** attempt)  # Exponential backoff: 2, 4, 8,
16, 32 seconds
```

```python
                logger.warning(f"Throttled. Retrying in {delay}s (attempt {attempt +
1}/{max_retries})")
                time.sleep(delay)
            else:
                logger.error(f"Comparison failed after {max_retries} retries: {e}",
exc_info=True)
                return None
        else:
            logger.error(f"Comparison failed: {e}", exc_info=True)
            return None
    except Exception as e:
        logger.error(f"Comparison failed: {e}", exc_info=True)
        return None


def handler(event, context):
    tmp_root = "/tmp"
    results = []

    job_id = event.get('job_id')
    print(job_id)

    metadata = {}
    if job_id:
        metadata['job_id'] = job_id

    try:
        records = event.get("Records", [])

        # --------- MODE 1: S3-triggered invocation ---------
        if records:
            if not records:
                logger.info("No S3 records in event.")
                return {"results": []}

            for idx, rec in enumerate(records, start=1):
                bucket = rec["s3"]["bucket"]["name"]
                key = unquote_plus(rec["s3"]["object"]["key"])
                size = rec["s3"]["object"].get("size")
                logger.info(f"[{idx}] Start s3://{bucket}/{key} size={size}")

                deleted = False
                result = {
                    "bucket": bucket,
                    "key": key,
                    "type": None,
                    "action": None,
                    "deleted": False,
                    "comparison_key": None
                }

                try:
                    # 1) Determine document type
                    is_policy = key.startswith(POLICY_PREFIX)
                    is_json = key.lower().endswith('.json')
                    is_pdf = key.lower().endswith('.pdf')

                    doc_type = "POLICY" if is_policy else "BILL"
                    result["type"] = doc_type

                    logger.info(f"[{idx}] Document type: {doc_type},
is_json={is_json}, is_pdf={is_pdf}")

                    # Skip if it's already a parsed JSON file
                    # if key.startswith(PARSED_PREFIX):
                    if key.startswith(f"{PARSED_PREFIX}policies/") or
key.startswith(f"{PARSED_PREFIX}comparisons/"):
                        logger.info(f"[{idx}] Skipping parsed file: {key}")
                        result["action"] = "skipped_parsed"
                        results.append(result)
                        continue

                    # 2) Handle POLICY
                    if is_policy:
                        if is_pdf:
                            # Check if already parsed
                            if _check_parsed_policy_exists(bucket, key):
                                logger.info(f"[{idx}] Policy already parsed,
skipping")

                                result["action"] = "skipped_exists"
                                results.append(result)
                                continue

                            # Parse new policy PDF
                            logger.info(f"[{idx}] Parsing policy PDF")
                            parsed_key = parse_policy_pdf(bucket, key, MODEL_ID)
                            result["action"] = "parsed"
                            result["parsed_key"] = parsed_key
                        else:
                            logger.info(f"[{idx}] Skipping non-PDF policy file")
                            result["action"] = "skipped_format"

                    # 3) Handle BILL
                    elif not is_policy:
                        # ---------- Handle BILL documents ----------
                        bill_data = None
                        # S3-triggered bill path: parse PDF or load JSON
                        parsed_bill_data = None
                        if is_pdf:
                            logger.info(f"[{idx}] Parsing bill PDF")
                            parsed_key = parse_bill_pdf(bucket, key, MODEL_ID)
                            result["action"] = "parsed"
                            result["parsed_key"] = parsed_key

                            # Fetch the parsed data
                            obj = s3.get_object(Bucket=bucket, Key=parsed_key)
                            parsed_bill_data =
json.loads(obj['Body'].read().decode('utf-8'))

                        elif is_json:
                            logger.info(f"[{idx}] Loading existing bill JSON")
                            obj = s3.get_object(Bucket=bucket, Key=key)
                            parsed_bill_data =
json.loads(obj['Body'].read().decode('utf-8'))
                            result["action"] = "loaded_json"

                        bill_data = parsed_bill_data

                        # 4) Compare with policy if we have bill data
                        if bill_data:
                            logger.info(f"[{idx}] Fetching latest policy for
comparison")
                            policy_data = _get_latest_policy(bucket)

                            if policy_data:
                                logger.info(f"[{idx}] Comparing bill to policy")
                                comparison_key = _compare_bill_to_policy(
                                    bill_data,
                                    policy_data,
                                    bucket,
                                    key,
                                    metadata
                                )
                                result["comparison_key"] = comparison_key
                                result["compared"] = comparison_key is not None
                            else:
                                logger.warning(f"[{idx}] No policy found for
comparison")
                                result["compared"] = False
                        else:
                            logger.warning(f"[{idx}] Could not build bill_data from
any source")
                            result["compared"] = False
```

```python
                        # 5) Delete original bill PDF after processing
                        if is_pdf:
                            deleted = _delete_object(bucket, key)
                            result["deleted"] = deleted

                except Exception as e:
                    logger.error(f"[{idx}] Error processing {key}: {e}",
exc_info=True)
                    result["error"] = str(e)

                finally:
                    results.append(result)
                    logger.info(f"[{idx}] Done.")

            logger.info(f"Processed {len(records)} record(s).")
            return {"results": results}

        # ---------- MODE 2: Direct invocation with table_id (no S3 Records) ---------
        table_id = event.get("table_id")
        if table_id:
            logger.info(f"Direct invocation with table_id={table_id}")
            bucket = event.get("bucket_name", MAIN_BUCKET)

            # 🔹 1) Look at validation result from Lambda 2 (if present)
            validation = event.get("validation")
            if validation is not None:
                all_valid = validation.get("all_valid")
                if all_valid is False:
                    # Build human-readable reason
                    reason_parts = []

                    # Main issue from CPT vs ICD if present
                    main_issue = validation.get("cpt_icd_justification_issue")
                    if main_issue:
                        reason_parts.append(main_issue)

                    # Any additional issues list
                    extra_issues = validation.get("issues") or []
                    if extra_issues:
                        reason_parts.append("; ".join(extra_issues))

                    reason = " ".join(reason_parts) or "See validation details."
                    message = f"The bill is invalid because: {reason}"

                    # 🔹 2) Save an 'invalid bill' comparison object to S3
                    comparison_id = str(uuid.uuid4())
                    timestamp = datetime.utcnow().isoformat(timespec="seconds") + "Z"
                    comparison_output = {
                        "comparison_id": comparison_id,
                        "timestamp": timestamp,
                        "bill_table_id": table_id,
                        "bill_source_key": f"dynamodb:{table_id}",
                        "policy_id": None,
                        "comparison": {
                            "status": "invalid_bill",
                            "message": message,
                            "validation": validation,
                        },
                    }

                    filename = f"comparison_{comparison_id}.json"
                    comparison_key = f"{PARSED_PREFIX}comparisons/{filename}"

                    s3.put_object(
                        Bucket=bucket,
                        Key=comparison_key,
                        Body=json.dumps(comparison_output, indent=2),
                        ContentType="application/json",
                        Metadata=metadata,
                    )

                    logger.info(
                        f"Bill marked invalid; saved comparison to
s3://{bucket}/{comparison_key}"
                    )

                    # 🔹 CLEANUP AFTER SAVING INVALID COMPARISON
                    cleanup_results = cleanup_all_data_for_table_id(table_id)
                    logger.info(f"Cleanup results: {cleanup_results}")

                    # 🔹 3) Return the "The bill is invalid because ..." message
                    return {
                        "mode": "dynamodb",
                        "table_id": table_id,
                        "success": False,
                        "comparison_key": comparison_key,
                        "reason": message,
                    }

            # 🔹 If we reach here: either validation is missing or all_valid is True →
proceed normally


            # Build bill_data from DynamoDB
            bill_data = _build_bill_data_from_dynamodb(table_id)
            if not bill_data:
                logger.warning(f"Could not build bill_data for table_id={table_id}")
                return {
                    "mode": "dynamodb",
                    "table_id": table_id,
                    "success": False,
                    "reason": "no_bill_data"
                }

            # Fetch latest policy
            policy_data = _get_latest_policy(bucket)
            if not policy_data:
                logger.warning("No parsed policies found for direct invocation")
                return {
                    "mode": "dynamodb",
                    "table_id": table_id,
                    "success": False,
                    "reason": "no_policy"
                }

            # Compare
            logger.info(f"Comparing DynamoDB bill for table_id={table_id} against
latest policy")
            comparison_key = _compare_bill_to_policy(
                bill_data=bill_data,
                policy_data=policy_data,
                bucket=bucket,
                key=f"dynamodb:{table_id}",
                metadata=metadata
            )

            # 🔹 CLEANUP AFTER SUCCESSFUL COMPARISON
            if comparison_key:
                logger.info(f"Comparison successful. Starting cleanup for
table_id={table_id}")
                cleanup_results = cleanup_all_data_for_table_id(table_id)
                logger.info(f"Cleanup results: {cleanup_results}")
            else:
                logger.warning(f"Comparison failed, skipping cleanup for
table_id={table_id}")
                cleanup_results = None


            return {
                "mode": "dynamodb",
                "table_id": table_id,
                "success": comparison_key is not None,
                "comparison_key": comparison_key
            }

        # If neither Records nor table_id → nothing to do
        logger.info("Event has neither S3 Records nor table_id; nothing to process.")
        return {"results": []}

    finally:
        # Clean Lambda /tmp
        try:
            for name in os.listdir(tmp_root):
                path = os.path.join(tmp_root, name)
                try:
                    if os.path.isdir(path):
                        shutil.rmtree(path, ignore_errors=True)
                    else:
                        os.remove(path)
                except Exception as e:
                    logger.warning(f"Failed to remove {path}: {e}")
            logger.info("Cleaned /tmp.")
        except FileNotFoundError:
            pass
```

11. parsingBills.py

```
import os, json, base64, logging, re, uuid, datetime
from typing import Optional, Dict, Any
from decimal import Decimal
import boto3

logger = logging.getLogger()
logger.setLevel(logging.INFO)

s3 = boto3.client("s3")
bedrock_rt = boto3.client(
    "bedrock-runtime",
    region_name=os.environ.get("AWS_REGION", "us-east-2")
)

# ---------- Output bucket for parsed bills ----------
OUTPUT_BUCKET = os.environ.get("OUTPUT_BUCKET", "chitest02")  # <- new: write here
PARSED_PREFIX = "parsed/"                                     # keep key structure
under this prefix
```

```
JSON_BILL_PROMPT = """
You are extracting structured data from a medical receipt.

Return ONLY a JSON object matching EXACTLY this schema:

{
  "patient_info": {
    "firstname": string|null,
    "lastname": string|null,
    "age": number|null,
    "phone": string|null,
    "address": string|null,
    "city": string|null,
    "state": string|null,
    "zipcode": string|null
  },
  "hospital_info": {
    "name": string|null,
    "phone": string|null,
    "address": string|null,
    "city": string|null,
    "state": string|null,
    "zipcode": string|null
  },
  "medical_bill_info": {
    "items": [
      {"code": string|null, "description": string|null, "bill": number|null}
    ],
    "subtotal": number|null,
    "discount": number|null,
    "tax_rate_percent": number|null,
    "total_tax": number|null,
    "balance_due": number|null
  }
}

Rules:
- Output MUST be valid JSON only. No backticks, no markdown, no extra commentary.
- Amounts (bill, subtotal, discount, taxes, balance_due) MUST be numbers (no $ or
commas).
- If a field is missing on the document, set it to null.
- If a ZIP code appears inside an address, also copy it to zipcode.
- Do not invent data. Be conservative.
- Go through EACH page systematically (page 1, 2, 3, etc.)
- Extract EVERY line item - medications, procedures, supplies, services, etc.
- Count and report how many pages you processed
- Count and report total number of line items extracted
```

```
"""

# ---------- JSON Extraction Helper ----------
def _extract_json(text: str):
    if not text or not text.strip():
        raise ValueError("Model returned empty text.")
    m = re.search(r"```json\s*(\{[\s\S]*?\})\s*```", text, re.IGNORECASE)
    if m:
        return json.loads(m.group(1))
    m = re.search(r"```\s*(\{[\s\S]*?\})\s*```", text)
    if m:
        return json.loads(m.group(1))
    m = re.search(r"\{.*\}", text, re.DOTALL)
    if not m:
        raise ValueError("Model did not return JSON.")
    return json.loads(m.group(0))

def _to_number(x):
    if x is None:
        return None
    if isinstance(x, (int, float)):
        return float(x)
    if isinstance(x, str):
        cleaned = x.replace("$", "").replace(",", "").strip()
        try:
            return float(cleaned)
        except ValueError:
            return None
    return None

def _extract_zip_from_address(addr):
    if not addr or not isinstance(addr, str):
        return None
    m = re.search(r"\b(\d{5})(?:-\d{4})?\b", addr)
    return m.group(0) if m else None

# ---------- Main Function ----------
def parse_bill_pdf(bucket: str, key: str, model_id: str) -> str:
    """
    Parse a bill PDF and save the extracted JSON to S3.
    Returns the S3 key (under parsed/bills/) where the JSON was saved, in
OUTPUT_BUCKET.
    """
    try:
        logger.info(f"Parsing bill PDF: s3://{bucket}/{key}")

        # Read PDF from S3
        obj = s3.get_object(Bucket=bucket, Key=key)
```

```
        file_bytes = obj["Body"].read()

        b64 = base64.b64encode(file_bytes).decode("utf-8")

        # Build Bedrock messages
        messages = [{
            "role": "user",
            "content": [
                {"type": "text", "text": JSON_BILL_PROMPT},
                {
                    "type": "document",
                    "source": {
                        "type": "base64",
                        "media_type": "application/pdf",
                        "data": b64
                    }
                }
            ]
        }]

        # Build Bedrock request payload
        payload = {
            "anthropic_version": "bedrock-2023-05-31",
            "messages": messages,
            "max_tokens": 4000,
            "temperature": 0
        }

        logger.info(f"Invoking Bedrock model: {model_id}") # log before call
        resp = bedrock_rt.invoke_model(modelId=model_id, body=json.dumps(payload)) #
call Bedrock
        body = json.loads(resp["body"].read()) # read and parse response

        content = body.get("content", []) # extract content
        text_blocks = [c.get("text", "") for c in content if c.get("type") == "text"]
# extract text parts
        out_text = "\n".join([t for t in text_blocks if t])
        logger.info(f"Model response length: {len(out_text)}")

        data = _extract_json(out_text) # may raise

        # ---------- Extract Fields ----------
        p = data.get("patient_info", {}) or {}
        h = data.get("hospital_info", {}) or {}
        m = data.get("medical_bill_info", {}) or {}

        patient_info = {
            "firstname": p.get("firstname"),
```

```python
            "lastname": p.get("lastname"),
            "age": _to_number(p.get("age")),
            "phone": p.get("phone"),
            "address": p.get("address"),
            "city": p.get("city"),
            "state": p.get("state"),
            "zipcode": p.get("zipcode") or
_extract_zip_from_address(p.get("address")),
        }

        # take hospital info
        hospital_info = {
            "name": h.get("name"),
            "phone": h.get("phone"),
            "address": h.get("address"),
            "city": h.get("city"),
            "state": h.get("state"),
            "zipcode": h.get("zipcode") or
_extract_zip_from_address(h.get("address")),
        }

        items = []
        for it in m.get("items", []) or []:
            items.append({
                "code": it.get("code"),
                "description": it.get("description"),
                "bill": _to_number(it.get("bill"))
            })

        medical_bill_info = {
            "subtotal": _to_number(m.get("subtotal")),
            "discount": _to_number(m.get("discount")),
            "tax_rate_percent": _to_number(m.get("tax_rate_percent")),
            "total_tax": _to_number(m.get("total_tax")),
            "balance_due": _to_number(m.get("balance_due")),
        }

        # ---------- Generate IDs and Timestamps ----------
        table_id = str(uuid.uuid4())
        patient_id = str(uuid.uuid4())
        provider_id = str(uuid.uuid4())
        created_at = datetime.datetime.utcnow().isoformat(timespec="seconds") + "Z"

        # ---------- Construct Parsed Output ----------
        parsed_output = {
            "table_id": table_id,
            "patient_id": patient_id,
            "provider_id": provider_id,
            "created_at": created_at,
            "source_bucket": bucket,
            "source_key": key,
            "patient_info": patient_info,
            "hospital_info": hospital_info,
            "medical_bill_info": medical_bill_info,
            "items": items
        }

        # ---- Save to chitest02/parsed/bills/ ----
        base_name = os.path.splitext(os.path.basename(key))[0]
        filename = f"{base_name}_{table_id}.json"
        parsed_key = f"{PARSED_PREFIX}bills/{filename}"

        # this function saves the parsed data to s3
        s3.put_object(
            Bucket=OUTPUT_BUCKET,          # write to chitest02
            Key=parsed_key,                # parsed/bills/<file>.json
            Body=json.dumps(parsed_output, indent=2, default=str),
            ContentType="application/json"
        )
        logger.info(f"Parsed JSON saved to s3://{OUTPUT_BUCKET}/{parsed_key}")
        return parsed_key

    # except block to handle errors
    except Exception as e:
        logger.error(f"Failed to parse bill PDF: {e}", exc_info=True)
        raise
```

12. parsingPolicies.py

```
import os
import re
import json
import base64
import logging
from datetime import datetime
from typing import Any, Dict, Optional

import boto3
from botocore.exceptions import ClientError

# ---------- Logging ----------
logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

```
# ---------- AWS Clients ----------
AWS_REGION = os.environ.get("AWS_REGION", "us-east-2")
s3 = boto3.client("s3")
bedrock_rt = boto3.client("bedrock-runtime", region_name=AWS_REGION)

# ---------- Constants ----------
PARSED_PREFIX = "parsed/"
MAX_TOKENS = 3000

JSON_POLICY_PROMPT = """
You are an expert insurance policy parser. Extract key coverage details from the
attached PDF. Return JSON with this structure:
{
  "schema_version": "policy.v1",
  "policy_id": "ShieldPlus_USF_2025-2026",
  "plan": {
    "name": "Shield Plus",
    "policy_number": "string|null",
    "policy_year": "2025–2026",
    "manager": "ISO",
    "underwriter": "BHSI (Bermuda) reinsured by BHSI",
    "network": {"name": "Aetna", "url": "string|null"}
  },
  "limits": {
    "medical_expense_per_injury_or_sickness_usd": 250000,
    "evacuation_usd": 50000,
    "repatriation_usd": 25000,
    "source_page": "p.3"
  },
  "deductibles": {
    "student_health_center_usd": 0,
    "elsewhere_usd": 100,
    "policy_year_max_usd": 100,
    "source_page": "p.3"
  },
  "coinsurance": {
    "in_network": "80%",
    "out_of_network": "60%",
    "source_page": "p.3"
  },
  "copays": {
    "primary_care_usd": 20,
    "specialist_usd": 35,
    "urgent_care_usd": 50,
    "emergency_room_usd": 200,
    "source_page": "p.3"
  },
  "coverage_highlights": {
    "maternity": "Covered per policy",
    "preexisting_wait_months": 12,
    "wellness_preventive": "As stated",
    "source_page": "p.3"
  },
  "rates": [
    {"age_range": "18–24", "monthly_usd": 123},
    {"age_range": "25–40", "monthly_usd": 145},
    {"age_range": "Dependent", "monthly_usd": 200}
  ],
  "eligibility": "F-1/J-1 full-time, etc.",
  "exclusions": ["..."],
  "claims": {
    "mail_to": "…",
    "deadline": "90 days",
    "phone": "…",
    "email": "…",
    "source_page": "p.X"
  },
  "refund_policy": {
    "conditions": ["…"],
    "processing_fee_usd": 50,
    "source_page": "p.X"
  },
  "assistance_services": {
    "provider": "On Call International",
    "phones": ["…"],
    "email": "…",
    "source_page": "p.X"
  },
  "artifacts": {
    "pdf_s3": "s3://…/raw/…/policy.pdf",
    "text_jsonl_s3": "s3://…/text/…/pages.jsonl"
  }
}
Rules:
- Go through EACH page systematically (page 1, 2, 3, etc.)
- Extract EVERY line item - medications, procedures, supplies, services, etc.
- Count and report how many pages you processed
- Count and report total number of line items extracted
- Amounts MUST be numbers (no $ or commas)
- If a field is missing, set it to null
- Output MUST be valid JSON only - no markdown, no commentary
"""


# ---------- Helpers ----------
```

```python
def _extract_json(text: str) -> Any:
    """
    Extract a JSON object or array from a model response.
    Tries in order:
      1) ```json ... ``` fenced block
      2) ``` ... ``` fenced block
      3) First balanced {...} or [...]
    """
    if not text or not text.strip():
        raise ValueError("Model returned empty text; no JSON to parse.")

    # 1) ```json ... ```
    m = re.search(
        r"```json\s*(\{[\s\S]*?\}|\[[\s\S]*?\])\s*```",
        text,
        flags=re.IGNORECASE,
    )
    if m:
        return json.loads(m.group(1))

    # 2) ``` ... ```
    m = re.search(r"```\s*(\{[\s\S]*?\}|\[[\s\S]*?\])\s*```", text)
    if m:
        return json.loads(m.group(1))

    # 3) First balanced {...} or [...]
    def _balanced_slice(s: str, open_ch: str, close_ch: str) -> Optional[str]:
        start = s.find(open_ch)
        if start < 0:
            return None
        depth = 0
        in_str = False
        esc = False
        for i in range(start, len(s)):
            ch = s[i]
            if in_str:
                if esc:
                    esc = False
                elif ch == "\\":
                    esc = True
                elif ch == '"':
                    in_str = False
            else:
                if ch == '"':
                    in_str = True
                elif ch == open_ch:
                    depth += 1
                elif ch == close_ch:
```

```
                    depth -= 1
                    if depth == 0:
                        return s[start : i + 1]
            return None

    for o, c in (("{", "}"), ("[", "]")):
        cand = _balanced_slice(text, o, c)
        if cand:
            return json.loads(cand)

    raise ValueError("Could not locate JSON block in model output.")


# ---------- Core ----------


def parse_policy_pdf(bucket: str, key: str, model_id: str) -> str:
    """
    Parse a policy PDF from S3 using an Amazon Bedrock model and save extracted JSON
back to S3.

    Args:
        bucket: S3 bucket name.
        key: S3 object key for the PDF.
        model_id: Bedrock model ID.

    Returns:
        The S3 key where the parsed JSON was saved.
    """
    try:
        logger.info("Parsing policy PDF: s3://%s/%s", bucket, key)

        # Fetch PDF from S3
        obj = s3.get_object(Bucket=bucket, Key=key)
        file_bytes = obj["Body"].read()

        # Encode to base64 for Bedrock document input
        b64 = base64.b64encode(file_bytes).decode("utf-8")

        # Build Bedrock request
        messages = [
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": JSON_POLICY_PROMPT},
                    {
                        "type": "document",
                        "source": {
                            "type": "base64",
                            "media_type": "application/pdf",
                            "data": b64,
                        },
                    },
                ],
            }
        ]

        payload: Dict[str, Any] = {
            "anthropic_version": "bedrock-2023-05-31",
            "messages": messages,
            "max_tokens": MAX_TOKENS,
            "temperature": 0,
        }

        logger.info("Invoking Bedrock model: %s", model_id)
        resp = bedrock_rt.invoke_model(modelId=model_id, body=json.dumps(payload))

        # Bedrock returns a StreamingBody under resp["body"]
        raw = resp["body"].read()
        body = json.loads(raw)

        # Extract text from response content
        content = body.get("content", [])
        text_blocks = [c.get("text", "") for c in content if c.get("type") == "text"]
        out_text = "\n".join(t for t in text_blocks if t)

        logger.info("Model response length: %d", len(out_text))

        # Parse JSON
        parsed_data = _extract_json(out_text)

        # Add metadata
        artifacts = parsed_data.get("artifacts", {}) if isinstance(parsed_data, dict)
else {}
        artifacts.update(
            {
                "pdf_s3": f"s3://{bucket}/{key}",
                "parsed_at": datetime.utcnow().isoformat(timespec="seconds") + "Z",
            }
        )
        if isinstance(parsed_data, dict):
            parsed_data["artifacts"] = artifacts

        # Save to S3
        filename = os.path.basename(key).rsplit(".", 1)[0] + ".json"
        parsed_key = f"{PARSED_PREFIX}policies/{filename}"
        # this function saves the parsed data to s3
        s3.put_object(
            Bucket=bucket,
            Key=parsed_key,
            Body=json.dumps(parsed_data, indent=2),
            ContentType="application/json",
        )

        logger.info("Saved parsed policy to s3://%s/%s", bucket, parsed_key)
        return parsed_key

    # except block to handle errors
    except Exception as e:
        logger.error("Failed to parse policy PDF: %s", e, exc_info=True)
        raise
```

13. testingKnowledgeBase.py

```
import boto3
import os
import logging
from typing import List, Dict

logger = logging.getLogger()

AWS_REGION = os.environ.get("AWS_REGION", "us-east-2")
KNOWLEDGE_BASE_ID = os.environ["KNOWLEDGE_BASE_ID"]

kb_rt = boto3.client("bedrock-agent-runtime", region_name=AWS_REGION)


def retrieve_policy(cpt_codes: List[Dict] = None) -> List[Dict]:
    """
    Retrieve relevant policy clauses from Bedrock Knowledge Base.

    Args:
        plan_id: Insurance plan identifier
        cpt_codes: Optional list of CPT procedure codes

    Returns:
        List of retrieval results with policy snippets
    """
    try:
        # Build comprehensive query including CPT codes if available
        # If no codes, just return empty
        if not cpt_codes:
            logger.warning("No CPT codes provided to retrieve_policy; skipping KB 
retrieval.")
            return []

        query_parts = [
            f"For insurance plan, what policy sections describe coverage, limitations, 
or exclusions"
        ]

        # if icd_codes:
        #     query_parts.append(f"for diagnoses {', '.join(icd_codes)}")

        if cpt_codes:
            cpt_code_list = [c.get('code', '') for c in cpt_codes if c.get('code')]
            if cpt_code_list:
                query_parts.append(f"and procedures {', '.join(cpt_code_list[:10])}")
# Limit to first 10

        query_text = " ".join(query_parts) + "?"

        logger.info(f"KB query: {query_text}")

        # Call Bedrock Knowledge Base retrieve API
        response = kb_rt.retrieve(
            knowledgeBaseId=KNOWLEDGE_BASE_ID,
            retrievalQuery={"text": query_text},
            retrievalConfiguration={
                "vectorSearchConfiguration": {
                    "numberOfResults": 5  # top-K results
                }
            }
        )

        results = response.get("retrievalResults", [])
        logger.info(f"KB returned {len(results)} policy snippets")

        # Log snippet sources for debugging
        for i, result in enumerate(results, 1):
            score = result.get("score", 0)
            source = result.get("location", {}).get("s3Location", {}).get("uri", 
"unknown")
            logger.info(f"Snippet {i} - Score: {score:.3f}, Source: {source}")

        return results

    except Exception as e:
        logger.error(f"Error retrieving from Knowledge Base: {e}")
```

```python
        return []


def format_snippets(snippets: List[Dict]) -> str:
    """
    Format retrieved policy snippets into a readable context string.

    Args:
        snippets: List of retrieval results from Knowledge Base

    Returns:
        Formatted string with all policy snippets
    """
    if not snippets:
        return "NO_POLICY_SNIPPETS_FOUND"

    parts = []
    for i, snippet in enumerate(snippets, start=1):
        # Extract text content
        text = snippet.get("content", {}).get("text", "")

        # Extract source location
        location = snippet.get("location", {})
        s3_location = location.get("s3Location", {})
        source_uri = s3_location.get("uri", "unknown")

        # Extract relevance score if available
        score = snippet.get("score", 0)

        # Format snippet with metadata
        part = f"[Policy Snippet {i}] (Relevance: {score:.3f})\n"
        part += f"Source: {source_uri}\n"
        part += f"Content:\n{text}\n"

        parts.append(part)

    formatted = "\n" + "="*80 + "\n"
    formatted += "\n".join(parts)
    formatted += "="*80 + "\n"

    return formatted
```

Lambda 4

```python
import boto3
import json
from botocore.exceptions import ClientError

s3 = boto3.client('s3')
BUCKET_NAME = 'chitest02'

# Prefix (folder path)
PREFIX = 'parsed/comparisons/'


def lambda_handler(event, context):
    try:

        job_id = event.get('queryStringParameters', {}).get('jobId')

        if not job_id:
            return {
                'statusCode': 400,
                'headers': {"Content-Type": "application/json"},
                'body': json.dumps({"error": "Missing jobId parameter"})
            }
        # A dictionary that contains all objects with the given prefix
        response = s3.list_objects_v2(Bucket=BUCKET_NAME, Prefix=PREFIX)

        # if statement to check if there are no objects in response
        if 'Contents' not in response or len(response['Contents']) == 0:
            return {
                'statusCode': 404,
                'headers': {"Content-Type": "application/json"},
                'body': json.dumps({"error": "no files found"})
            }



        matched_key = None

        # Loop through each object in the response
        for obj in response['Contents']:
            key = obj['Key']

            if key.endswith("/"):
                continue

            head = s3.head_object(Bucket=BUCKET_NAME, Key=key)
            metadata = head.get('Metadata', {})
            if metadata.get('job_id') == job_id:
                matched_key = key
                break

        # If statement to check if there is no files exist
        if not matched_key:
            return {
                'statusCode': 404,
                'headers': {"Content-Type": "application/json"},
                'body': json.dumps({"error": "No file matches jobId"})
            }


        # Retrieve the matched S3 object
        obj = s3.get_object(Bucket=BUCKET_NAME, Key=matched_key) # Get the S3 object
        file_data = obj['Body'].read().decode('utf-8') # Read and decode the file
contents
        json_data = json.loads(file_data) # Parses the content as JSON

        s3.delete_object(Bucket=BUCKET_NAME, Key=matched_key) # Delete the S3 object

        return {
            'statusCode': 200,
            'headers': {"Content-Type": "application/json"},
            'body': json.dumps(json_data)
        }

    # Handle AWS ClientError exceptions
    except ClientError as e:
        # if statement to check if the error is NoSuchKey
        if e.response['Error']['Code'] == 'NoSuchKey':
            return {
                'statusCode': 404,
                'headers': {"Content-Type": "application/json"},
                'body': json.dumps({"error": "file not ready yet"})
            }
        # Handle other ClientError exceptions
        else:
            return {
                    'statusCode': 500,
                    'headers': {"Content-Type": "application/json"},
                    'body': json.dumps({"error": str(e)})
                }
```

Configurations

~~~
# Medical Bill Adjudication System - Libraries & Dependencies

## 📚 Complete Dependency List

This document provides a comprehensive list of all libraries, packages, and
dependencies used in the Medical Bill Adjudication System across all components.

---

## 🐍 Python Dependencies (Backend/Lambda)

### AWS SDK & Services
```
boto3                          ^3.26.0      # AWS SDK for Python
botocore                       ^1.29.0      # Low-level AWS service interface
```

### Core Libraries
```
python                         ^3.11        # Python runtime (required)
json                           built-in     # JSON parsing
os                             built-in     # Operating system interface
io                             built-in     # Input/output operations
base64                         built-in     # Base64 encoding/decoding
logging                        built-in     # Logging framework
time                           built-in     # Time functions
datetime                       built-in     # Date and time
re                             built-in     # Regular expressions
uuid                           built-in     # UUID generation
shutil                         built-in     # File operations
```

### Type Hints & Validation
```
typing                         built-in     # Type hints
decimal                        built-in     # Decimal arithmetic (for DynamoDB)
Enum                           built-in     # Enumeration support
```

### Data Processing
```
json                           built-in     # JSON serialization
~~~

~~~
decimal.Decimal                built-in     # DynamoDB numeric handling
```

### Error Handling
```
botocore.exceptions
  - ClientError               # AWS API errors
  - EndpointConnectionError   # Connection issues
  - ReadTimeoutError          # Read timeouts
  - ConnectTimeoutError       # Connection timeouts
  - ConnectionClosedError     # Connection closed
```

### Security & Encryption
```
boto3.client('kms')           # AWS Key Management Service
boto3.client('secretsmanager')# AWS Secrets Manager
```

---

## 🟢 Node.js/JavaScript Dependencies (Frontend)

### Core Framework
```
react                          ^19.1.1      # UI library
react-dom                      ^19.1.1      # React DOM renderer
react-router-dom               ^7.9.4       # Client-side routing
react-scripts                  5.0.1        # Build scripts and configuration
```

### AWS Integration
```
@aws-sdk/client-s3             ^3.896.0     # AWS S3 client
```

### Utilities
```
uuid                           ^13.0.0      # UUID generation
web-vitals                     ^2.1.4       # Performance metrics
```

### Testing & Development
```
@testing-library/react         ^16.3.0      # React component testing
@testing-library/jest-dom      ^6.8.0       # Jest DOM matchers
@testing-library/dom           ^10.4.1      # DOM testing utilities
@testing-library/user-event    ^13.5.0      # User interaction simulation
~~~

~~~
ajv                             ^8.17.1      # JSON Schema validator
```

### Build & Development Tools
```
npm                             ^10.0.0      # Node package manager
node                            ^18.0.0      # Node.js runtime
```

---

## 🗄️ AWS Services (Infrastructure)

### Compute
```
AWS Lambda                      # Serverless compute
  - Runtime: Python 3.11
  - Memory: 256-1024 MB
  - Timeout: 60-300 seconds
  - Concurrency: Auto-scaling
```

### Storage
```
Amazon S3                       # Object storage
  - Versioning: Enabled
  - Encryption: AES256 or KMS
  - Lifecycle: Auto-archive
```

### Database
```
Amazon DynamoDB                 # NoSQL database
  - Billing Mode: Pay-per-request
  - TTL: Enabled on temp tables
  - Global Tables: Optional
```

### AI/ML
```
Amazon Bedrock                  # Managed AI service
  - Model: Claude 3.5 Sonnet (anthropic.claude-3-5-sonnet-20241022-v2:0)
  - Alternative: Claude 3 Opus, Claude 3 Haiku
```

### API & Integration
```
Amazon API Gateway              # REST API service
~~~

```
  - Protocol: HTTPS/REST
  - Rate Limiting: Enabled
  - CORS: Configurable
```

### Monitoring & Logging
```
Amazon CloudWatch              # Monitoring and logging
  - Logs: Real-time
  - Metrics: Custom metrics
  - Alarms: Configurable
```

### Tracing
```
AWS X-Ray                      # Distributed tracing
  - Service Map: Enabled
  - Sampling: Configurable
```

### Secrets Management
```
AWS Secrets Manager            # Secret storage
  - Rotation: Automatic (optional)
```

Configuration
```
AWS Systems Manager
  Parameter Store              # Configuration storage
  Secrets Manager              # Secret storage
```

### Identity & Access
```
AWS IAM                        # Identity and access management
  - Roles: Lambda execution
  - Policies: Least privilege
```

## 📦 Development Dependencies (Optional)

### Testing
```
pytest                         ^7.0.0       # Python testing framework
pytest-cov                     ^4.0.0       # Coverage plugin
```

~~~
unittest                       built-in     # Python unit testing
jest                           ^29.0.0      # JavaScript testing framework
```

### Code Quality
```
pylint                         ^2.17.0      # Python linter
flake8                         ^6.0.0       # Python style checker
black                          ^23.0.0      # Python code formatter
eslint                         ^8.0.0       # JavaScript linter
prettier                       ^3.0.0       # Code formatter
```

### Type Checking
```
mypy                           ^1.0.0       # Python static type checker
```

### Documentation
```
sphinx                         ^6.0.0       # Documentation generator
mkdocs                         ^1.4.0       # Static site generator
```

### Local Development
```
sam                            ^1.80.0      # AWS SAM CLI
serverless                     ^3.26.0      # Serverless framework
localstack                     ^2.0.0       # Local AWS stack
```

### Version Control
```
git                            latest       # Version control
pre-commit                     ^3.0.0       # Git hooks
```

---

## 🔄 Environment & Configuration

### Environment Variables
```
ENVIRONMENT                    # dev, staging, production
AWS_REGION                     # AWS region
AWS_ACCESS_KEY_ID              # AWS credentials
AWS_SECRET_ACCESS_KEY          # AWS credentials
AWS_PROFILE                    # AWS CLI profile
~~~

~~~~
LOG_LEVEL                      # DEBUG, INFO, WARNING, ERROR
```

### Configuration Files
```
.env                          # Local environment variables
.env.development              # Development config
.env.staging                  # Staging config
.env.production               # Production config
config.py                     # Python configuration module
```

---

## 📋 Lambda Layer Dependencies

If using Lambda Layers for shared dependencies:

```
Layer: Python-Common
  ├── boto3 (AWS SDK)
  ├── botocore (AWS core)
  └── custom utilities

Layer: Bedrock-AI
  ├── boto3 (with Bedrock support)
  └── JSON processing
```

---

## 🔗 Third-Party Integrations

### AI/ML Services
```
AWS Bedrock                    # Claude AI models
  - Claude 3.5 Sonnet
  - Claude 3 Opus
  - Claude 3 Haiku
```

### Medical Data Standards
```
CPT (Current Procedural Terminology)    # Medical procedure codes
ICD-10 (International Classification)   # Diagnosis codes
HL7 FHIR (Health Information Exchange)  # Healthcare data standards
```
~~~~

~~~markdown
### Insurance Standards
```
X12 EDI (Electronic Data Interchange)   # Insurance claim format
NCPDP (Pharmacy standards)              # Pharmacy-specific standards
```

---

## 🐳 Docker/Container Dependencies (Optional)

If containerizing Lambda functions:

```dockerfile
FROM public.ecr.aws/lambda/python:3.11

RUN pip install \
    boto3==1.26.* \
    botocore==1.29.* \
    requests==2.31.* \
    urllib3==2.0.*

COPY app.py ${LAMBDA_TASK_ROOT}
CMD [ "app.lambda_handler" ]
```

---

## 📦 Package Managers & Tools

### Python
```
pip                             # Python package manager
virtualenv                      # Virtual environment tool
poetry                          # Dependency management
pipenv                          # Python project management
```

### JavaScript/Node
```
npm                             ^10.0.0      # Node package manager
yarn                            ^4.0.0       # Alternative package manager
pnpm                            ^8.0.0       # Performance-optimized package manager
```

### Build & Deploy
```
AWS CDK                         # Infrastructure as Code
AWS SAM                         # Serverless Application Model
~~~

```
Terraform                      # Infrastructure as Code (alternative)
CloudFormation                 # AWS infrastructure templates
```

## 🔐 Security Dependencies

### Authentication & Authorization
```
boto3 IAM                      # AWS Identity management
AWS Cognito (optional)         # User authentication
JWT (JSON Web Tokens)          # Token-based auth
```

### Encryption
```
AWS KMS                        # Key management
SSL/TLS                        # Transport security
boto3 encryption               # Built-in encryption
```

## 🧪 Testing Dependencies

### Python Testing
```
pytest                         ^7.0.0
pytest-cov                     ^4.0.0
pytest-mock                    ^3.10.0
moto                           ^4.1.0       # Mock AWS services
localstack                     ^2.0.0       # Local AWS
```

### JavaScript Testing
```
jest                           ^29.0.0
@testing-library/react         ^16.3.0
@testing-library/jest-dom      ^6.8.0
react-test-renderer            ^19.0.0
```

## 📊 Monitoring & Analytics Dependencies

### Logging

~~~
```
Python logging              built-in
AWS CloudWatch              # Log aggregation
AWS CloudWatch Insights     # Log analysis
```

### Metrics
```
CloudWatch Custom Metrics   # Application metrics
X-Ray Tracing               # Distributed tracing
```

### Alerting
```
CloudWatch Alarms           # Alert configuration
SNS (Simple Notification)   # Alert delivery
```

---

## 🔄 CI/CD Dependencies

### GitHub Actions (Recommended)
```
actions/checkout            # Git operations
actions/setup-python        # Python setup
actions/setup-node          # Node setup
actions/deploy-github-pages # Deploy docs
```

### CI/CD Tools
```
GitHub Actions              # Built-in CI/CD
Jenkins                     # Alternative CI/CD
GitLab CI                   # Alternative CI/CD
CircleCI                    # Alternative CI/CD
```

---

## 📱 Frontend Production Dependencies

### Web Framework
```
react                       ^19.1.1      # Core UI framework
react-dom                   ^19.1.1      # DOM rendering
```
~~~

~~~
### Routing
```
react-router-dom               ^7.9.4       # Client routing
```

### State Management (Optional)
```
redux                          ^4.2.0       # State management
zustand                        ^4.3.0       # Alternative state manager
```

### HTTP Client (Optional)
```
axios                          ^1.4.0       # HTTP requests
fetch                          built-in     # Native HTTP
```

### UI Components (Optional)
```
material-ui                    ^5.13.0      # Material Design
bootstrap                      ^5.3.0       # Bootstrap framework
tailwind                       ^3.3.0       # Utility CSS
```

---

## 🛠 Development Tools

### Code Editors
```
Visual Studio Code             # Recommended IDE
Extensions:
  - Python
  - JavaScript/TypeScript
  - AWS Toolkit
  - REST Client
  - Thunder Client
```

### Git Tools
```
Git                            ^2.40.0      # Version control
GitHub Desktop                 # GUI for Git
GitKraken                      # Alternative GUI
```

### API Testing
```
~~~

~~~
Postman                        # API testing
Insomnia                       # Alternative API client
Thunder Client                 # VS Code extension
```

### Database Tools
```
DynamoDB Local                 # Local testing
NoSQL Workbench               # DynamoDB GUI
AWS Console                    # Web interface
```

---

## 📚 Documentation Dependencies

### Python Docs
```
Sphinx                         ^6.0.0       # Python documentation
Sphinx RTD Theme              ^1.2.0       # ReadTheDocs theme
```

### JavaScript Docs
```
JSDoc                          ^4.0.0       # JavaScript documentation
TypeDoc                        ^0.23.0      # TypeScript documentation
```

### API Docs
```
Swagger/OpenAPI                ^3.0.0       # API specification
ReDoc                          ^2.0.0       # API documentation
Postman Collections            # API documentation
```

---

## 🔄 Deployment & Infrastructure Dependencies

### AWS Deployment
```
AWS CLI                        ^2.13.0      # AWS command line
AWS SAM CLI                    ^1.80.0      # Serverless deployment
AWS CDK                        ^2.80.0      # Infrastructure as Code
```

### Container Orchestration (If Used)
```
~~~

~~~~
Docker                         ^24.0.0      # Container runtime
Docker Compose                 ^2.20.0      # Multi-container
ECS                            # AWS container service
ECR                            # AWS container registry
```

---

## 📈 Performance & Optimization

### Frontend
```
Webpack                        5.0.0        # Module bundler
Babel                          ^7.23.0      # JavaScript transpiler
react-lazy-load               ^3.0.0        # Code splitting
```

### Backend
```
Connection pooling             built-in     # DB connection management
caching                        optional     # Performance caching
compression                    optional     # Response compression
```

---

## 🔗 Version Compatibility Matrix

| Component | Version | Python | Node |
|-----------|---------|--------|------|
| boto3 | ^3.26.0 | 3.8+ | - |
| React | ^19.1.1 | - | 18+ |
| AWS Lambda | - | 3.11 | - |
| Node.js | - | - | 18+ |
| TypeScript | ^5.0.0 | - | 18+ |

---

## 📦 Installation Guide

### Python Setup
```bash
# Create virtual environment
python3 -m venv venv
source venv/bin/activate

# Install AWS SDK
pip install boto3 botocore
~~~~

~~~
# Install all from requirements.txt (if exists)
pip install -r requirements.txt

# Verify installation
python -c "import boto3; print(boto3.__version__)"
```

### Node.js Setup
```bash
# Install dependencies
cd front-end
npm install

# Or with yarn
yarn install

# Verify installation
npm list

# Check Node version
node --version
npm --version
```

### AWS Setup
```bash
# Install AWS CLI
pip install awscli

# Configure credentials
aws configure

# Verify credentials
aws sts get-caller-identity
```

---

## 🔄 Dependency Management Best Practices

### Python
- [ ] Use virtual environments
- [ ] Pin versions in requirements.txt
- [ ] Use pip-tools for dependency management
- [ ] Regular security updates: `pip audit`
- [ ] Document breaking changes
~~~

~~~markdown
### JavaScript
- [ ] Lock package versions with package-lock.json
- [ ] Use npm audit for security
- [ ] Regular dependency updates
- [ ] Test before major upgrades
- [ ] Document breaking changes

### AWS
- [ ] Keep SDKs updated
- [ ] Monitor service updates
- [ ] Test in staging before production
- [ ] Plan for deprecations

---

## 🛡️ Security Considerations

### Dependency Security
```
- Use pip-audit for Python security
- Use npm audit for JavaScript security
- Keep all dependencies updated
- Scan for vulnerabilities regularly
- Use private packages for internal code
- Verify package signatures
```

### AWS SDK Security
```
- Use IAM roles instead of access keys
- Rotate credentials regularly
- Use AWS Secrets Manager
- Enable MFA
- Monitor API calls
```

---

## 📞 Support Resources

- **Python Package Index**: https://pypi.org/
- **NPM Package Registry**: https://www.npmjs.com/
- **AWS SDK Documentation**: https://docs.aws.amazon.com/
- **GitHub Packages**: https://github.com/features/packages
- **AWS CDK**: https://aws.amazon.com/cdk/

---
~~~

```markdown
## 🔄 Update Schedule

### Recommended Update Cycle
- **Critical Security**: Immediate (same day)
- **Major Security**: 1 week
- **Minor Updates**: Monthly
- **Patch Updates**: Quarterly
- **Major Versions**: Quarterly review

---

**Last Updated**: December 7, 2025
**Version**: 1.0
**Status**: Complete
```