⚕️ NHCX Insurance FHIR Utility

A production-ready microservice that converts Indian health insurance policy PDFs into NHCX-compliant FHIR R4 bundles using AI-powered extraction.

Python 3.9+ | FastAPI 0.110+ | React 18 | FHIR R4 | NHCX Compliant

---

📋 Table of Contents

🎯 Overview

Indian health insurance PDFs (sold under IRDAI guidelines and processed through the **NHCX** — National Health Claim Exchange) contain dense, unstructured text about plan types, benefit limits, exclusions, co-pays, and TPA details. Translating this into a machine-readable, interoperable format requires:

1. High-quality OCR that retains document structure
2. AI that understands Indian insurance terminology
3. Clinical mapping that aligns extracted medical conditions and benefits directly to **SNOMED CT** codes
4. A standards-compliant FHIR mapper that captures every clinical and financial nuance

This utility automates all four stages, exposing the result via a REST API and a premium browser UI.

🔄 End-to-End Pipeline

```
┌──────────────────────────────────────────────────────────────────────┐
│                         USER (Browser / API)                         │
└──────────────────────────────┬───────────────────────────────────────┘
                               │  1. Upload PDF
                               ▼
┌──────────────────────────────────────────────────────────────────────┐
│  STAGE 1 — PDF → Structured Markdown (OCR)                          │
│                                                                      │
│  Fast path: pdftext (digital PDFs — all IRDAI/NHCX filed docs)      │
│  Slow path: Marker ML models (scanned / image-only PDFs)            │
│  • Preserves tables, headings, and benefit schedules                 │
│  • Runs on CPU (fp32) or GPU (fp16), configurable via config.yaml   │
└──────────────────────────────┬───────────────────────────────────────┘
                               │  2. Structured Markdown
                               ▼
┌──────────────────────────────────────────────────────────────────────┐
│  STAGE 2 — Markdown Pruning (policy_pruner)                         │
│                                                                      │
│  • Removes boilerplate sections (ToC, glossary, arbitration, etc.)  │
│  • Keeps only clinically and financially relevant content            │
│  • Reduces LLM token usage by ~40–60%, cutting cost and latency     │
└──────────────────────────────┬───────────────────────────────────────┘
                               │  3. Pruned Markdown
                               ▼
┌──────────────────────────────────────────────────────────────────────┐
│  STAGE 3 — Structured Data Extraction (LLM)                         │
│                                                                      │
│  • System prompt instructs the LLM to fill a strict JSON schema     │
│    (insurance_fhir_mapping.json) — no invented keys allowed         │
│  • Extracts: plan name, type, aliases, period, insurer, TPA,        │
│    networks, coverages, benefits, limits, exclusions, costs         │
│  • Swappable across 5 providers via a single config.yaml change     │
│    (OpenAI · Gemini · Ollama · Groq · AWS Bedrock)                  │
└──────────────────────────────┬───────────────────────────────────────┘
                               │  4. Extracted JSON
                               ▼
┌──────────────────────────────────────────────────────────────────────┐
│  STAGE 4 — FHIR R4 Bundle Generation (insurance_plan_fhir_mapper)  │
│                                                                      │
│  Resources built:                                                    │
│  • Organization (Insurer)  — meta.profile, IRDAI identifier         │
│  • Organization (TPA)      — IRDAI licence number                   │
│  • Organization[] (Network hospitals)                                │
│  • InsurancePlan           — alias, language, narrative, period,    │
│                              networks, coverages, benefit limits,    │
│                              exclusion & condition extensions,       │
│                              plan-level costs & applicability        │
│  • Bundle (collection)     — language=en-IN, timestamp              │
└──────────────────────────────┬───────────────────────────────────────┘
                               │  5. FHIR Bundle (JSON)
                               ▼
┌──────────────────────────────────────────────────────────────────────┐
│  RESPONSE                                                            │
│  • extracted_data  — structured intermediate JSON (auditable)       │
│  • fhir_bundle     — NHCX-compliant R4 Bundle ready for submission  │
└──────────────────────────────────────────────────────────────────────┘
```

🔬 Why These Tools?

📄 Dual-Path PDF Extraction

Most PDF parsing libraries (`pdfplumber`, `PyMuPDF`) struggle with complex, multi-column insurance documents. The service uses a tiered approach:

- **Fast path** — `pdftext` extracts text from digital PDFs instantly with no ML overhead. All standard IRDAI/NHCX-filed documents take this path.
- **Slow path** — `marker-pdf` (a deep-learning OCR pipeline built on `transformers` and `torch`) fires only for scanned / image-only PDFs. It detects document layout, preserves table structure as Markdown, and is configurable via `marker.*` settings in `config.yaml`.

📄 `insurance_fhir_mapping.json` — Schema-Guided Extraction

Rather than writing fragile regex patterns, the LLM is guided by a **strict JSON schema template** stored in `config/insurance_fhir_mapping.json`. The system prompt instructs the model:

```
You MUST NOT invent new JSON keys, group data under new headers, or change the schema.
Replace the instruction text inside the template values with extracted data.
If any field is not found, set it to null or [] — do not omit the key.
```

This approach:

- Makes the output **deterministic and parseable** — the mapper always receives the same shape of JSON
- Lets you **extend the schema** without touching Python code — just add a new key with an instruction string
- Provides a **clear audit trail** — the `extracted_data` field in the API response contains the raw LLM output before FHIR mapping

⚙️ `config.yaml` + `.env` — Layered Configuration

The settings system uses a **priority chain** (highest → lowest):

```
Environment Variables  →  .env file  →  config.yaml  →  Pydantic defaults
```

Layer	Purpose
`config.yaml`	Non-secret settings: LLM provider, model names, Marker worker counts
`.env`	Secrets: API keys for OpenAI, Gemini, Groq, AWS credentials
Environment variables	CI/CD overrides without file changes

Switching providers requires **one line change** in `config.yaml`:

```yaml
llm:
  provider: "gemini"   # ← change to "openai", "ollama", "grok", or "bedrock"
```

🏥 SNOMED CT Integration

The FHIR mapping engine integrates a local **SNOMED CT Dictionary** located at `src/core/snomed_dictionary.json`. During resource building, extracted clinical terms are cross-referenced against this dictionary. When a match is found, the mapper automatically assigns the official SNOMED code and applies the `http://snomed.info/sct` system URI to the resulting FHIR `CodeableConcept` elements.

🤖 Multi-LLM Architecture

The service is designed to be **LLM-agnostic** from day one. A single abstract interface drives all providers:

```
LLMService (Abstract Base Class)
│
├── _OpenAICompatibleService  ← shared async chat.completions logic
│   ├── OpenAILLMService      — GPT-4 Turbo via OpenAI API
│   ├── OllamaLLMService      — any open model (Llama 3.1, Mistral…) running locally
│   └── GrokLLMService        — Llama 3 70B via Groq's ultra-fast inference API
│
├── GeminiLLMService          — Gemini Flash via Google AI SDK (non-OpenAI protocol)
└── BedrockLLMService         — Claude / Nova / Llama on AWS Bedrock via boto3
```

Why five providers?

Provider	Best For
OpenAI (GPT-4 Turbo)	Highest accuracy for complex policy language
Gemini Flash	High speed + long context window at low cost
Groq (Llama 3 70B)	Sub-second latency for real-time demos
Ollama (local)	Air-gapped / on-premise deployments — zero data leaves your machine
AWS Bedrock (Claude / Nova)	Enterprise compliance, existing AWS infrastructure

Health Checks on Startup

Every provider has a **dedicated health check** (`src/health_check.py`) that runs when the server starts. If the configured provider is unreachable or misconfigured, the application **refuses to start** with a clear error message — preventing silent failures in production.

📦 FHIR R4 Bundle Structure

A typical output bundle contains the following resources:

```
Bundle (collection, language: en-IN)
├── Organization  [Insurer]
│   ├── meta.profile → ABDM StructureDefinition/Organization
│   ├── identifier   → IRDAI insurer registry (use: "official")
│   └── telecom      → phone · email · website
│
├── Organization  [TPA]
│   ├── meta.profile → ABDM StructureDefinition/Organization
│   └── identifier   → IRDAI TPA licence number
│
├── Organization[]  [Network Hospitals]   (one per network)
│   └── type → "Healthcare Provider Network"
│
└── InsurancePlan
    ├── meta.profile → ABDM StructureDefinition/InsurancePlan
    ├── text         → auto-generated XHTML Narrative
    ├── language     → "en-IN"
    ├── identifier   → UUID (use: "official")
    ├── alias[]      → alternate product / marketing names
    ├── status       → "active"
    ├── type[]       → ABDM ValueSet/ndhm-insuranceplan-type
    ├── ownedBy      → ref → Organization [Insurer]
    ├── administeredBy → ref → Organization [TPA]
    ├── period       → start / end dates
    ├── network[]    → ref → Organization[] [Networks]
    ├── coverageArea[] → geographic strings
    ├── contact[]
    │   ├── purpose  → contactentity-type system
    │   ├── name     → HumanName.text
    │   └── telecom  → phone · email
    ├── extension[]
    │   ├── Claim-SupportingInfoRequirement (POI / POA documents)
    │   └── Claim-Exclusion (pre-existing diseases, waiting periods)
    ├── coverage[]
    │   └── benefit[]
    │       ├── type → CodeableConcept (SNOMED CT where available)
    │       └── limit[]
    │           ├── value  → Quantity (INR / Days / %)
    │           └── code   → benefit-unit CodeableConcept
    └── plan[]
        ├── identifier → UUID
        ├── type        → ndhm-plan-type (Individual / Family Floater)
        └── specificCost[]
            └── benefit[]
                └── cost[]
                    ├── type          → benefit-cost-type
                    ├── applicability → in-network / out-of-network
                    └── value         → Quantity
```

🌐 API Reference

All endpoints are prefixed with `/api/v1`.

Insurance Processing

Method	Endpoint	Description
`POST`	`/insurance/process`	Full pipeline: PDF → OCR → LLM → FHIR bundle
`POST`	`/insurance/extract-only`	PDF → OCR → LLM extraction only (no FHIR mapping)
`POST`	`/insurance/generate-fhir`	JSON → FHIR bundle (when you already have extracted data)

System Health

Method	Endpoint	Description
`GET`	`/insurance/health`	Service health — LLM + PDF processor status, API version

FHIR Utilities

Method	Endpoint	Description
`POST`	`/fhir/validate`	Structural validation of a FHIR bundle (error/warning/info issues)
`POST`	`/fhir/bundle-summary`	Human-readable summary card from a FHIR bundle

Interactive API docs available at `http://localhost:8082/docs` (Swagger UI).

🛠️ Technology Stack

Layer	Technology	Why
Frontend	React 18, Vanilla CSS	Lightweight; no UI framework overhead
API Server	FastAPI + Uvicorn	Async-native, auto OpenAPI docs, Pydantic validation
PDF → Text	`pdftext` (fast) + `marker-pdf` (fallback OCR)	Fast path for digital PDFs; ML fallback for scans
LLM Abstraction	Abstract base class + Factory	Swap any of 5 providers with one config line
LLM Providers	OpenAI · Gemini · Ollama · Groq · Bedrock	Cloud + on-premise coverage
FHIR Building	`fhir.resources` (R4)	Type-safe FHIR resource construction; Pydantic-backed
Config	`config.yaml` + `.env` + Pydantic Settings	Layered; secrets stay out of YAML
Health Checks	Per-provider sync functions	Fail-fast on startup if LLM is misconfigured

---

⚙️ Configuration

`config.yaml` — Non-secret settings

```
llm:
  provider: "gemini"          # openai | ollama | gemini | grok | bedrock

  openai:
    model_name: "gpt-4-turbo"

  ollama:
    base_url: "http://localhost:11434/v1"
    model_name: "llama3.1"

  gemini:
    model_name: "gemini-3-flash"

  grok:
    model_name: "llama3-70b-8192"

  bedrock:
    region_name: "us-east-2"
    model_id: "global.amazon.nova-2-lite-v1:0"

marker:
  workers: 2
  pdftext_workers: 2
  batch_multiplier: 2
  model_precision: "fp32"
  exclude_images: true

pdf_processor:
  min_chars_for_text_pdf: 200   # characters threshold for fast-path selection

### `config/insurance_fhir_mapping.json` — Mapping Schema

The extraction logic is driven by the prompt template and JSON schema defined in [config/insurance_fhir_mapping.json](file:///config/in
```

`.env` — Secrets (copy `.env.example` → `.env`)

```
OPENAI_API_KEY="sk-..."
GOOGLE_API_KEY="AIza..."
GROK_API_KEY="gsk_..."
AWS_ACCESS_KEY_ID="AKIA..."
AWS_SECRET_ACCESS_KEY="..."
```

> **Ollama** requires no API key — just have the Ollama service running locally.

🚀 Installation & Setup

Prerequisites

- Python 3.9+
- Node.js 18+
- `pip`, `venv`, `npm`

1. Clone & configure secrets

```
git clone <repo-url>
cd InsuranceService

cp .env.example .env
# Edit .env — add the API key for your chosen LLM provider
```

2. Frontend dependencies

```
cd frontend && npm install && cd ..
```

▶️ Running the Application

Using Docker (Recommended)

```
docker-compose up -d --build
```

Service	URL
Frontend	`http://localhost:8001`
Backend API	`http://localhost:8082`
Swagger UI	`http://localhost:8082/docs`

Local Development — One Command

Each script creates the `.venv` (first run only), installs dependencies, and starts the backend with hot-reload:

Windows (PowerShell)

```
.\dev.ps1
```

> If execution policy blocks it, run once: `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned`

macOS / Linux

```
chmod +x dev.sh && ./dev.sh
```

The backend starts at `http://localhost:8082`. The `.venv` is reused on subsequent runs so only changed packages are reinstalled.

Frontend (separate terminal)

```
cd frontend && npm start
```

The React dev server starts at `http://localhost:3000` and proxies all `/api/v1/*` calls to `localhost:8082`.

The backend runs the LLM health check on startup — it exits immediately if the configured provider is unreachable.

Batch Processing

Process multiple PDFs in one go:

```
python scripts/batch_process.py --input data/input --output data/output
```

📂 Project Structure

```
InsuranceService/
│
├── app.py                          # FastAPI entry point, lifespan, middleware
├── config.yaml                     # Non-secret configuration (LLM, Marker, PDF)
├── .env.example                    # Template for secrets (copy → .env)
├── requirements.txt                # All Python dependencies (single file)
├── dev.ps1                         # One-command backend setup & run (Windows)
├── dev.sh                          # One-command backend setup & run (macOS/Linux)
│
├── config/
│   └── insurance_fhir_mapping.json     # JSON schema template used in LLM prompt
│
├── scripts/
│   └── batch_process.py            # CLI tool: batch PDF → FHIR bundle
│
├── src/
│   ├── config.py                   # Pydantic Settings — YAML + .env + env var layers
│   ├── constants.py                # All log messages, error codes, string literals
│   ├── health_check.py             # Per-provider LLM health check functions
│   ├── logging_config.py           # Structured logging setup
│   ├── middleware.py               # Request/response logging middleware
│   │
│   ├── core/
│   │   ├── pdf_processor.py        # Dual-path OCR: pdftext fast-path + Marker fallback
│   │   ├── prompts.py              # Loads insurance_fhir_mapping.json into system prompt
│   │   └── snomed_dictionary.json  # Local SNOMED CT terminology dictionary
│   │
│   ├── routes/
│   │   ├── claims.py               # Insurance processing endpoints (process, extract, generate-fhir)
│   │   ├── health.py               # GET /insurance/health
│   │   └── fhir.py                 # FHIR utilities (validate, bundle-summary)
│   │
│   ├── services/
│   │   ├── policy_pruner.py        # Strips boilerplate sections from Markdown
│   │   ├── fhir/
│   │   │   ├── fhir_constants.py   # ABDM/HL7 URLs, system codes, profile URLs
│   │   │   └── insurance_plan_fhir_mapper.py  # Builds FHIR R4 bundle from dict
│   │   └── llm/
│   │       ├── llm_service.py      # Abstract base + 5 concrete LLM implementations
│   │       └── llm_factory.py      # Reads config and returns the right LLMService
│   │
│   └── schemas/
│       └── insurance_schemas.py    # Pydantic request/response schemas
│
├── tests/
│   └── test_fhir_mapper.py         # Unit tests for FHIR R4 parameters
│
└── frontend/
    ├── package.json                # React app; proxy → localhost:8082
    ├── nginx.conf                  # Nginx config (listens :8001, proxies /api/ → backend:8082)
    ├── Dockerfile                  # Multi-stage build — node builder + nginx:alpine
    └── src/
        ├── App.js                  # Layout, state, API wiring
        ├── index.js                # Global CSS, keyframes, responsive grid
        ├── api/
        │   └── claimService.js     # All API call functions
        └── components/
            ├── Upload.js           # Drag-and-drop PDF zone + FHIR toggle
            ├── Result.js           # Tabbed JSON viewer, copy/download
            ├── StatusBar.js        # Live /health polling (pauses during requests)
            └── SummaryCard.js      # Plan summary card with FHIR validation issues
```

📄 License

This project is developed as part of the **NHCX Hackathon**. All FHIR structure definitions and ValueSet URLs follow HL7 FHIR R4 and ABDM NDHM specifications.