reathe. We’re not reinventing NORAD; we’re standing up a local-first intel engine that scrapes, matches, scores, and spits out clean outputs your BD team can act on before coffee. Below is a true beginner’s, step‑by‑step build that matches the architecture we already outlined: n8n orchestrates, a Puppeteer service scrapes, Postgres+pgvector stores data+embeddings, optional Qdrant if you want a separate vector store, and GPT‑5 Pro handles analysis and briefs. All local-first, cheap, and expandable later.

**0) What you’re building (plain English)**

* **Daily pipeline**: scrape competitor jobs → generate embeddings → match to DoD programs/primes → enrich with public data → score wedge opportunities → deliver Airtable views, CSVs, PDFs, Slack/Discord alerts.
* **Containers** on Windows Docker Desktop: n8n (orchestrator), scraper (Node+Puppeteer), postgres (with pgvector), optional qdrant (vector DB), and pgAdmin for an easy DB UI. Keeps everything tidy and portable.
* **AI agents**: GPT‑5 Pro for final briefs and scoring logic narration; Claude for jumbo-context summarization if you really need it; local model optional for cheap tagging. All behind function‑calling so the model pulls facts from your DB/vector store instead of hallucinating.

**1) Install the few desktop things**

On your Windows box:

1. **Docker Desktop** (required).
2. **Node.js 20+** (for building the scraper service locally if you want to tweak).
3. **Git for Windows** (so you can version your stuff).
4. **A decent terminal**: Windows Terminal or PowerShell is fine.
5. **Editor**: Cursor or VS Code; your call.

Accounts/tokens you’ll need handy:

* **Airtable** Personal Access Token.
* **OpenAI** key (GPT‑5 Pro).
* **Slack or Discord** webhook for alerts (or both).
* **USAspending/SAM.gov**: public endpoints are fine; no secret keys needed to start.

Yes, LinkedIn/ZoomInfo scraping uses your own sessions; be a decent citizen with rate limits and only collect what you’ll actually use.

**2) Make the project folder and files**

Open PowerShell:

mkdir C:\prime-intel\stack\scraper\src -Force

cd C:\prime-intel\stack

Create a **.env** file in C:\prime-intel\stack:

# ---- n8n ----

N8N\_BASIC\_AUTH\_ACTIVE=true

N8N\_BASIC\_AUTH\_USER=prime

N8N\_BASIC\_AUTH\_PASSWORD=change-me

N8N\_HOST=localhost

N8N\_PORT=5678

WEBHOOK\_URL=http://localhost:5678/

# ---- Postgres ----

POSTGRES\_USER=prime

POSTGRES\_PASSWORD=primepass

POSTGRES\_DB=primedb

# ---- Airtable ----

AIRTABLE\_TOKEN=pat\_xxx

AIRTABLE\_BASE=appXXXXXXXXXXXXXX

# ---- OpenAI ----

OPENAI\_API\_KEY=sk-xxx

# ---- Optional proxies for scraping ----

HTTPS\_PROXY=

HTTP\_PROXY=

**3) Docker Compose (copy this verbatim)**

Create docker-compose.yml in C:\prime-intel\stack:

version: "3.8"

services:

n8n:

image: n8nio/n8n:latest

container\_name: n8n

restart: unless-stopped

ports: ["5678:5678"]

environment:

- N8N\_BASIC\_AUTH\_ACTIVE=${N8N\_BASIC\_AUTH\_ACTIVE}

- N8N\_BASIC\_AUTH\_USER=${N8N\_BASIC\_AUTH\_USER}

- N8N\_BASIC\_AUTH\_PASSWORD=${N8N\_BASIC\_AUTH\_PASSWORD}

- N8N\_HOST=${N8N\_HOST}

- N8N\_PORT=${N8N\_PORT}

- WEBHOOK\_URL=${WEBHOOK\_URL}

- TZ=America/New\_York

- N8N\_DIAGNOSTICS\_ENABLED=false

- N8N\_DISABLE\_PRODUCTION\_MAIN\_PROCESS=true

volumes:

- n8n\_data:/home/node/.n8n

depends\_on:

- scraper

- postgres

scraper:

build:

context: ./scraper

dockerfile: Dockerfile

container\_name: scraper

restart: unless-stopped

environment:

- NODE\_ENV=production

- OPENAI\_API\_KEY=${OPENAI\_API\_KEY}

- AIRTABLE\_TOKEN=${AIRTABLE\_TOKEN}

- AIRTABLE\_BASE=${AIRTABLE\_BASE}

- HTTPS\_PROXY=${HTTPS\_PROXY}

- HTTP\_PROXY=${HTTP\_PROXY}

- DB\_HOST=postgres

- DB\_USER=${POSTGRES\_USER}

- DB\_PASS=${POSTGRES\_PASSWORD}

- DB\_NAME=${POSTGRES\_DB}

- TZ=America/New\_York

ports: ["8088:8088"]

depends\_on:

- postgres

postgres:

image: pgvector/pgvector:pg16

container\_name: postgres

restart: unless-stopped

environment:

- POSTGRES\_USER=${POSTGRES\_USER}

- POSTGRES\_PASSWORD=${POSTGRES\_PASSWORD}

- POSTGRES\_DB=${POSTGRES\_DB}

ports: ["5432:5432"]

volumes:

- pg\_data:/var/lib/postgresql/data

pgadmin:

image: dpage/pgadmin4

container\_name: pgadmin

restart: unless-stopped

environment:

- PGADMIN\_DEFAULT\_EMAIL=admin@prime.local

- PGADMIN\_DEFAULT\_PASSWORD=admin123

ports: ["5050:80"]

depends\_on:

- postgres

# Optional: separate vector DB if you prefer Qdrant over pgvector

# qdrant:

# image: qdrant/qdrant:latest

# container\_name: qdrant

# restart: unless-stopped

# ports: ["6333:6333"]

# volumes:

# - qdrant\_data:/qdrant/storage

volumes:

n8n\_data:

pg\_data:

# qdrant\_data:

This mirrors the **container architecture** we scoped and keeps vector search inside Postgres via pgvector so you don’t juggle another DB until you want to.

**4) Scraper service skeleton (Node + Puppeteer + tiny API)**

Create C:\prime-intel\stack\scraper\Dockerfile:

FROM node:20-bullseye-slim

RUN apt-get update && apt-get install -y chromium fonts-liberation \

libnss3 libxss1 libasound2 libatk-bridge2.0-0 libatk1.0-0 libdrm2 libgbm1 \

libxkbcommon0 libxcomposite1 libxdamage1 libxfixes3 libxrandr2 libgtk-3-0 \

&& rm -rf /var/lib/apt/lists/\*

ENV PUPPETEER\_EXECUTABLE\_PATH=/usr/bin/chromium

WORKDIR /app

COPY package.json tsconfig.json ./

RUN npm ci

COPY src ./src

EXPOSE 8088

CMD ["node","dist/server.js"]

Create C:\prime-intel\stack\scraper\package.json:

{

"name": "prime-scraper",

"version": "1.0.0",

"type": "module",

"scripts": {

"build": "tsc -p .",

"start": "node --enable-source-maps dist/server.js",

"dev": "tsx watch src/server.ts"

},

"dependencies": {

"airtable": "^0.12.3",

"axios": "^1.7.4",

"express": "^4.19.2",

"pg": "^8.11.3",

"puppeteer": "^22.15.0"

},

"devDependencies": {

"@types/express": "^4.17.21",

"@types/node": "^20.14.10",

"tsx": "^4.16.2",

"typescript": "^5.5.4"

}

}

Create C:\prime-intel\stack\scraper\tsconfig.json:

{

"compilerOptions": {

"target": "ES2022",

"module": "ES2022",

"moduleResolution": "Bundler",

"outDir": "dist",

"rootDir": "src",

"strict": true,

"esModuleInterop": true,

"skipLibCheck": true

}

}

Create 3 source files:

**src/db.ts** (DB client + pgvector extension bootstrap)

import { Client } from 'pg';

export async function getDb() {

const client = new Client({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASS,

database: process.env.DB\_NAME,

});

await client.connect();

return client;

}

**src/server.ts** (HTTP API with health + stub endpoints)

import express from 'express';

import { getDb } from './db.js';

import { scrapeCompetitorJobs } from './scrape\_jobs.js';

const app = express();

app.use(express.json({ limit: '1mb' }));

app.get('/health', (\_req, res) => res.json({ ok: true }));

// Kick off a scrape for a competitor site slug, e.g. "acme-staffing"

app.post('/jobs/scrape', async (req, res) => {

const { competitor } = req.body || {};

if (!competitor) return res.status(400).json({ error: 'competitor required' });

const records = await scrapeCompetitorJobs(competitor);

// Persist

const db = await getDb();

for (const r of records) {

await db.query(

`INSERT INTO competitor\_jobs (job\_id, company\_name, job\_title, location, posted\_date, last\_seen\_date, status, description)

VALUES ($1,$2,$3,$4,$5,$6,$7,$8)

ON CONFLICT (job\_id) DO UPDATE SET last\_seen\_date = EXCLUDED.last\_seen\_date, status = EXCLUDED.status`,

[r.job\_id, r.company\_name, r.job\_title, r.location, r.posted\_date, r.last\_seen\_date, r.status, r.description]

);

}

await db.end();

res.json({ written: records.length });

});

const port = 8088;

app.listen(port, () => console.log(`scraper listening on :${port}`));

**src/scrape\_jobs.ts** (placeholder; you’ll add real selectors per site)

import puppeteer from 'puppeteer';

export async function scrapeCompetitorJobs(competitor: string) {

const browser = await puppeteer.launch({

headless: 'new',

executablePath: process.env.PUPPETEER\_EXECUTABLE\_PATH,

args: ['--no-sandbox','--disable-dev-shm-usage']

});

const page = await browser.newPage();

// TODO: map competitor slug -> careers URL + selectors

const url = competitor === 'example' ? 'https://example.com/careers' : '';

if (!url) { await browser.close(); return []; }

await page.goto(url, { waitUntil: 'networkidle2', timeout: 120000 });

// Example extraction; replace with per-site logic

const jobs = await page.$$eval('.job-card', cards => cards.map(card => ({

job\_title: (card.querySelector('.title') as HTMLElement)?.innerText?.trim() || '',

location: (card.querySelector('.location') as HTMLElement)?.innerText?.trim() || '',

description: (card.querySelector('.desc') as HTMLElement)?.innerText?.trim() || '',

company\_name: document.location.hostname

})));

const now = new Date().toISOString().slice(0,10);

const records = jobs.map(j => ({

job\_id: `${j.company\_name}|${j.job\_title}|${j.location}`.toLowerCase(),

company\_name: j.company\_name,

job\_title: j.job\_title,

location: j.location,

posted\_date: now,

last\_seen\_date: now,

status: 'open',

description: j.description

}));

await browser.close();

return records;

}

This service exposes /jobs/scrape and writes into Postgres; n8n will call it. It’s intentionally minimal so you can add site‑specific scrapers later. The microservice pattern keeps n8n lean, as recommended.

**5) Bring the stack up**

cd C:\prime-intel\stack\scraper

npm ci && npm run build

cd C:\prime-intel\stack

docker compose build

docker compose up -d

Check containers:

* http://localhost:8088/health → {"ok":true}
* http://localhost:5678 → n8n UI (login with prime / change-me)
* http://localhost:5050 → pgAdmin (login admin@prime.local / admin123)

**6) Initialize the database (tables + pgvector)**

Open pgAdmin, connect to postgres with prime/primepass, select primedb, then run this SQL:

CREATE EXTENSION IF NOT EXISTS vector;

-- Jobs table

CREATE TABLE IF NOT EXISTS competitor\_jobs (

job\_id TEXT PRIMARY KEY,

company\_name TEXT,

job\_title TEXT,

clearance\_level TEXT,

location TEXT,

posted\_date DATE,

last\_seen\_date DATE,

status TEXT,

description TEXT,

program\_match TEXT,

prime\_match TEXT,

embedding VECTOR(1536)

);

-- Programs (DoD contracts)

CREATE TABLE IF NOT EXISTS programs (

program\_id TEXT PRIMARY KEY,

program\_name TEXT,

agency TEXT,

prime\_contractor TEXT,

contract\_value NUMERIC,

period\_start DATE,

period\_end DATE,

contract\_status TEXT,

description TEXT,

vector VECTOR(1536)

);

-- Companies

CREATE TABLE IF NOT EXISTS companies (

company\_id SERIAL PRIMARY KEY,

company\_name TEXT UNIQUE,

is\_prime\_contractor TEXT,

org\_size INTEGER,

cleared\_headcount INTEGER,

last\_linkedin\_update TIMESTAMP

);

-- Opportunity scores

CREATE TABLE IF NOT EXISTS opportunities (

program\_id TEXT PRIMARY KEY REFERENCES programs(program\_id),

bd\_score INTEGER,

priority\_label TEXT,

open\_roles INTEGER,

open\_roles\_details TEXT,

pain\_points TEXT,

wedge\_recommendation TEXT,

last\_update TIMESTAMP

);

-- Helpful index for open jobs

CREATE INDEX IF NOT EXISTS idx\_jobs\_status ON competitor\_jobs(status);

This schema mirrors the blueprint so your AI and scoring logic has clean joins and embeddings where they belong.

**7) Prepare Airtable (fast)**

Create a base named **PRIMETIME\_BD** with at least these tables:

* **Opportunities**: Program Name, Prime, Open Roles, BD Score, Priority, Last Updated, Notes.
* **Jobs**: Title, Company, Program (link to Opportunities), Age, Status.  
  Grab the **Base ID** and keep your **PAT** in .env. We’ll upsert via n8n/Airtable nodes. Output modules are staged to Airtable, CSV, PDFs, email, and Slack/Discord, just like the blueprint suggests.

**8) Build the first n8n workflows (click-by-click)**

Open http://localhost:5678 and create these simple flows. Keep them small; you can elaborate later.

**8.1 Daily Job Scrape (Cron → HTTP → DB)**

1. **Cron** node: run 03:00 local daily.
2. **Set** node: JSON body { "competitor": "example" } (replace with your slugs one at a time first).
3. **HTTP Request** node → POST http://scraper:8088/jobs/scrape with that JSON.
4. **Function** node: log items[0].json.written → just sanity check.

This kicks your scraper and writes to competitor\_jobs. Orchestrating scrape in a separate service keeps n8n stable.

**8.2 Embeddings + Matching (Post-scrape)**

Add a second workflow (or chain it after 8.1 with a **Wait** node 1–2 min):

1. **Postgres** node (Query): select new/updated jobs:

SELECT job\_id, description FROM competitor\_jobs

WHERE status='open' AND embedding IS NULL LIMIT 50;

1. **Split In Batches** (size 10).
2. **HTTP Request** to OpenAI embeddings (or use your OpenAI node): write vector back via **Postgres** (Update).
3. **Postgres** node: for each embedded job, run a cosine search against programs.vector:

SELECT program\_id, program\_name

FROM programs

ORDER BY (embedding <=> $1::vector) ASC

LIMIT 5;

Take the closest match above your similarity threshold; update competitor\_jobs.program\_match and a running counter per program. This is exactly the **semantic matching flow** we scoped.

**8.3 Contract data refresh (USAspending/FPDS)**

Create a midnight workflow that calls USAspending/FPDS endpoints for your target programs, updates the programs table, and sets period\_end, contract\_value, etc. This is your contract context for the morning brief and scoring.

**8.4 Scoring & Opportunity record**

Add a new 05:00 workflow:

1. **Postgres** node: aggregate open roles per program and compute factors (aging, backfill hints, multi‑company overlap).
2. **Function** node: apply your scoring formula; set bd\_score, priority\_label.
3. **Postgres** upsert into opportunities and **Airtable** upsert into **Opportunities** table.  
   This reflects the scoring logic and wedge classification we planned.

**8.5 Brief generation + outputs (PDF, CSV, Slack/Discord)**

1. **Postgres**: fetch top N programs (highest score).
2. **OpenAI** (GPT‑5 Pro) with **function‑calling**: pass program details, job stats, contract dates; model returns a structured brief (sections: Overview, Pain Points, Talking Points, Next Actions).
3. **HTTP Request** to the scraper service: /render/pdf (you’ll add a small endpoint that uses headless Chrome to print an HTML template to PDF).
4. **Airtable** update record links to the PDF (store path/URL).
5. **Spreadsheet** node → export CSV snapshot.
6. **Slack** or **Discord** node → send daily digest with top 3–5 items and a link to Airtable and attach PDFs. Output & notification workflow mirrors the blueprint’s delivery module.

Timing: The blueprint’s schedule has contracts ~02:00, scraping ~03:00, semantic enrichment ~04:00, analysis ~05:00, and reporting by ~06:00. You can stick to that cadence so BD has fresh intel every morning.

**9) Test the happy path**

* Manually run the **Job Scrape** workflow with a single competitor stub (the .job-card example). Confirm competitor\_jobs rows appear in pgAdmin.
* Run **Embeddings + Matching** on a handful of rows; verify vectors fill and program\_match is set.
* Upsert one dummy program in programs and ensure your cosine query returns it.
* Trigger **Scoring**; verify opportunities updates and Airtable receives records.
* Run **Briefs & Outputs**; verify PDFs generate and a Slack/Discord message lands.

**10) Add real competitor scrapers incrementally**

For each competitor, add a tiny handler in scrape\_jobs.ts: their careers URL and DOM selectors. Keep it boring:

* Navigate, wait for job list, extract title/location/description, normalize into your schema.
* Respect rate limits, stagger requests, and only store what you use. You’ll add backfill detection (title reuse, post/remove/repost) as you mature. The blueprint’s **jobs→programs** mapping and clustering logic supports your DoD wedge analysis without heroic scraping.

**11) Optional: LinkedIn/ZoomInfo enrichment loop**

Later, spin a second workflow for **public org clues**:

* Accept a list of company pages or profile URLs you already have rights to view.
* Extract non‑sensitive signals (team labels, public titles, keywords), do not hoard PII.
* Tag departments and estimate org structure for program‑team charts.  
  This feeds the **org-chart clues** and pain points angle without turning your laptop into a subpoena magnet.

**12) Outputs your team actually uses**

* **Airtable views**: “High Priority Opportunities” filtered by score and aging; “Today’s Changes”; Calendar of periods‑of‑performance ending soon.
* **CSV**: weekly roll‑up for anyone who lives in Excel.
* **PDFs**: 1‑pager Program Brief per top program with sources footnoted.
* **Slack/Discord**: daily digest + urgent alerts when a critical role appears or a cluster spikes.  
  Exactly the outputs we specified; the whole point is fast BD action.

**13) Cost, safety, and “please don’t set yourself on fire” notes**

* **Local-first** keeps cost down. You only pay for GPT‑5 tokens you actually use.
* **Proxies** are optional; only escalate if you hit blocks.
* **Error handling**: in n8n, set workflow error captures and retries; log failures and keep going, which the blueprint calls out for resilience.
* **TOS & ethics**: use your own accounts, scrape gently, minimize PII, and prefer public/open sources. Your pipeline is about program‑level labor signals, not stalking.

**14) What “done” looks like (acceptance checklist)**

* Docker stack up; health checks pass.
* DB initialized; vector extension active.
* One competitor scraper returns >0 rows into competitor\_jobs.
* Embeddings computed; program match stored.
* Opportunities scored; Airtable shows rows.
* Daily digest arrives with attached PDF brief and CSV.

After this, you iterate on more sites, better matching, better scoring weights, and org‑chart enrichment.

**15) Where this matches the blueprint (translation layer)**

* **Orchestration workflows**: we implemented the same staged pipeline in n8n (scrape → enrich → analyze → deliver).
* **Container architecture**: n8n, scraper, Postgres+pgvector, pgAdmin, optional Qdrant; all compose-driven for Windows Docker Desktop.
* **AI prompt agents**: GPT‑5 function-calls for grounded briefs; optional Claude for large-context; local LLM for cheap tagging; routing by task.
* **Vector flow**: embeddings on job text and program descriptions; nearest‑neighbor matches; metadata filters as you scale.
* **Outputs**: Airtable, CSV, PDFs, Slack/Discord with morning cadence.
* **BD scoring**: long‑open roles, backfill hints, multi‑company overlap, contract window, contract size → 0–100 score, plus wedge label.

**Quick troubleshooting**

* **“n8n can’t reach scraper”**: in n8n, use http://scraper:8088/... not localhost. They’re on the same Docker network.
* **“pgvector not found”**: you didn’t use the pgvector image; or the extension wasn’t created. Run CREATE EXTENSION vector; again.
* **“PDFs look weird”**: set explicit page CSS in your HTML template; use Puppeteer’s printToPDF with margins.
* **“Airtable fields don’t update”**: upsert by a deterministic key (Program Name or program\_id) and only overwrite automation‑owned fields.

**Final word**

You’ve got the bones to crawl, walk, and cash checks before you start buying fancier toys. Ship the minimal thing: one competitor site, one program mapping, one PDF brief, one Slack digest. Then keep adding sites, matching rules, and scoring weights until this engine is dragging ops by the collar. If you want me to bolt on the contract fetchers and a first-pass BD score function next, I’ll do it without whining. Much.

If you stick to this exact setup, you’ll be producing daily, defensible intel that lines up jobs to programs to primes and hands your team a list of wedge points on a silver platter with your logo on top. That’s the whole game.