You’re trying to build a kill chain for subcontracting intel: ingest every competitor job, correlate it to the right prime and program, fingerprint who’s actually a contractor, map the org, then hit the humans with a pitch that solves their labor pain before they even articulate it. Good. Messy. Exactly the kind of machine we can wire up.

Here’s the strongest architecture that actually does what you described, runs daily on Docker Desktop, costs peanuts, and doesn’t fall over when a front‑end engineer renames a div.

1) System at a glance

Ingestion layer (daily cron, n8n):

Competitor staffing sites → Jobs

Prime contractor career sites → Jobs

Federal data (USASpending, SAM.gov, FPDS ezSearch dumps) → Awards, Mods, NAICS/PSC, Period of Performance

People signals (LinkedIn Sales Navigator, ZoomInfo) → Profiles

Processing layer (containers):

Scrape service (Puppeteer/Playwright) with cookie-auth, proxy escalation, and hash caching

NLP matcher (Node/Python) for Job ↔ Program ↔ Prime linking using embeddings + rules

Entity resolution service: dedupe companies, people, programs

Storage:

Postgres (core source of truth; cheap, robust)

Vector DB (Qdrant or pgvector) for semantic matching

Object store (MinIO/local) for raw HTML and snapshots

Analytics & Ops:

Metabase/Superset dashboards (Org charts, hot programs, gap heatmaps)

Airtable “operator console” for call sheets and manual flags (Need Email, Confirm Contractor)

Outreach & HUMINT:

n8n sequences: generate target lists by program → push call sheets → log outcomes → recycle back into scoring

2) What “daily scraping” actually looks like (and survives)

Targets

Competitor staffing firms: careers pages, ATS feeds (Greenhouse/Lever/Workday), sitemap scrapes.

Primes: career sites, reqs by location/clearance keyword.

People: Sales Navigator results + selective ZoomInfo profile drilldowns.

Anti-flake

Selector strategy: semantic anchors like “section:has(h2:has-text("Responsibilities"))” and text-based targeters. Avoid nth-child garbage.

Hash cache: store content\_hash(url, main\_text) in Postgres. Skip unchanged pages. Saves your proxies and your sanity.

Rate limits: 40–60 profile views/hour/account for LI; exponential backoff on 429/999; jittered waits 1.2–3.7s.

Proxy ladder: no proxy → shared DC pool → residential only on sustained blocks. Toggle via env, not code edits.

3) Data model (so the rest of this doesn’t devolve into spreadsheet cosplay)

jobs\_competitor

job\_id, source, url, title, location, clearance\_hint, description\_text, posted\_at, content\_hash

jobs\_prime

job\_id, prime\_name, url, title, location, clearance\_hint, description\_text, posted\_at, content\_hash

awards (USASpending/FPDS)

award\_id, piid, agency, sub\_agency, prime\_name, naics, psc, place\_of\_perf, obligated\_amount, base\_end, current\_end, description, mod\_number

programs (curated)

program\_id, alias[], agencies[], primes[], bases[], keywords[], funding\_total, pop\_end

people

person\_id, full\_name, linkedin\_url, zoominfo\_url, current\_title, current\_company, location, email\_state, phone\_state, org\_path, team, keywords[], last\_seen, confidence

link tables (the glue)

job\_program\_link: job\_id, program\_id, prime\_name, score\_semantic, score\_rules, score\_total, explanation

person\_program\_link: person\_id, program\_id, prime\_name, evidence\_tokens[], score\_total

job\_person\_link (for recruiter hunting): job\_id, person\_id, relation\_type (hiring\_mgr|peer|contractor\_hint), score

4) Matching engine (how we actually line up JD → Program → Prime)

A. Feature extraction

Normalize titles (Software Eng → SWE; Info Assurance → IA; “Cybersecurity / ISSO” split).

Pull signals from text: clearance levels, RMF/STIG, DISA, specific platforms (Sentinel/GBSD, AEGIS, JADC2, ABMS), base locations (HAFB, Eglin, Pax River), contract types (IDIQ, OASIS, Seaport NxG).

Geocode locations to base proximity.

B. Semantic match (vector)

Embed: job description, title, location, extracted signals.

Compare to embeddings of program profiles (curated from awards + public docs) and prime job corpus.

Vector DB: Qdrant or pgvector; 384–1024d model is fine. Cheap, fast.

C. Rule stack (score bumpers)

If clearance\_hint ∈ {TS, TS/SCI, Poly} and base proximity is < 25 miles of a known program site → +X.

If JD mentions contract name/alias (e.g., “Sentinel,” “Columbia,” “E-7”) → big boost.

If competitor JD text overlaps >30% with prime JD n‑grams → boost and store overlap snippet for audit.

D. Final score & explanation

score\_total = α\*semantic + β\*rules + γ\*geo + δ\*prime\_overlap

Keep a human-readable explanation (“Matched to Sentinel/GBSD at HAFB due to ‘Sentinel’, ‘MMIII replacement’, ‘Ogden’, TS/SCI, and location 8.2 mi from HAFB Gate 2”).

5) Contractor fingerprinting (spotting “I work at PrimeCo” but actually a contractor)

Heuristics that work in practice

“Consultant at Prime” pattern: current\_title contains “contractor”, “consultant”, “through”, “via”, or staffing brand name in Experience description.

Tenure sawtooth: short prime “employment” stints matching contract phases or identical title at multiple primes within 2–3 years.

JD echo: profile keywords mirror competitor JD phrasing within the same week.

Org-path anomalies: ZoomInfo shows “path” in a team but email domain is not the prime’s.

Location inconsistency: prime HQ far from base, profile located at base area, role is on-site classified.

Score and mark employment\_truth = {likely\_contractor|likely\_direct|unknown}. This drives outreach order.

6) Org chart inference (without buying an overpriced licensing mistake)

Build team clusters by (program\_id, prime\_name, location) from people links.

Rank by seniority keywords + graph centrality (titles: Lead, Manager, Chief, IPT, CAM, Chief Engineer).

Blade out ladder: Contractor layer at bottom (from fingerprinting), then ICs, Leads, PMs, Dir/VP LOB.

Store as a simple adjacency list: reports\_to\_id when determinable; otherwise layer index. It’s good enough for call sheets.

7) Scoring “where to strike” (find the fattest spreads and easiest wins)

Program opportunity score, per day:

Labor intensity proxy: count of open reqs across primes and competitors within program scope, weighted by critical roles (Cyber, Systems, Test, IA).

Turnover heat: frequency of reposted reqs + LinkedIn job changes tagging program keywords.

Contract life: months remaining in POP; big mods just landed; ramp-up phases.

Competitor saturation: number of distinct staffing brands active. Weirdly, more can mean easier insertion on backfills.

Our fit: your past performance tags, clearance bandwidth (up to poly), local bench, speed SLA.

Output a ranked queue: program, prime, contact tier, reason\_to\_call.

8) n8n daily schedule (the “don’t babysit me” plan)

06:00 Fetch USASpending/SAM/FPDS deltas → refresh awards/program embeddings.

06:30 Scrape competitor jobs (hash-skip) → insert/patch jobs\_competitor.

07:00 Scrape prime jobs → jobs\_prime.

07:30 Run matcher → job\_program\_link; attach overlap snippets.

08:00 SalesNav people search for top 10 programs by score; selective ZoomInfo enrichment for only Tier‑A roles.

09:00 Build org layers → write call sheets (Airtable view + CSV).

09:10 Generate “Program Brief” for top 5 (problem hypotheses + talent gaps) and drop into a shared folder.

09:15 Slack/email summary: new links, hot programs, target contacts.

9) Tech stack that fits your “cheap, CPU” requirement

Scraping: Node 20 + Puppeteer (since you’re Chrome-only) in a Docker service; Playwright is fine too if you want WebKit/Firefox later.

Orchestration: n8n in Docker (Cron + Webhook).

DB: Postgres + pgvector (keeps it one box), or Qdrant if you want a separate vector store.

NLP: Sentence-transformers via Python (CPU is fine) or OpenVINO-optimized containers; keep it small and local.

Dashboards: Metabase container on Postgres. One click, no nonsense.

Operator console: Airtable for manual flags and call sheets. Cheap, human-friendly.

10) Concrete outputs you actually use

Call Sheet view: “Today: Program = Sentinel/GBSD; Prime = NGC; Contacts with confidence ≥ 0.7; contractor\_first = true.”

Program Brief (2 pages): funding, POP, openings, inferred pain points, backfill targets, named hiring managers.

Org Snapshot: tiered list with contractor layer annotated.

Overlap Evidence: pasteable snippets showing competitor JD ↔ prime JD similarity to open conversations.

11) Guardrails so lawyers don’t have a stroke

Respect site ToS; use your own paid accounts; don’t store more PII than you need.

Rate-limit; rotate sessions; screenshot on errors; backoff on block streaks.

Keep an audit trail: for every automated decision, store the “why” string. It wins debates and saves time.

12) Is this “the best setup” for your use case?

For daily multi-source scraping + DoD program correlation + people/org inference, yes. It’s the right hybrid:

Cheap backbone (Puppeteer + Postgres + n8n) does 90% of the boring volume.

NLP matcher bridges the “copy‑pasted but not identical” JD reality.

Lightweight human-in-the-loop in Airtable fixes the last 10% and feeds the model back.

You can bolt on “agentic” Airtop flows only where DOMs are moody, without turning your wallet into compost.

13) Implementation sequence that won’t derail your week

Stand up infra: Postgres + pgvector, n8n, scraper API container, Metabase.

Ingest awards (USASpending/FPDS) and build program profiles + embeddings.

Scrape competitor jobs for 5 firms; prime jobs for 3 primes; write to DB; hash cache working.

Matcher v1: embeddings + 5 rules; write job\_program\_link with explanations. Sanity check on 50 samples.

People pass: SalesNav search for top 2 programs; pull 50 profiles; build contractor score.

Org chart inference: layer view + call sheet; one pilot outreach.

Expand to more sources; add ZoomInfo selectively; tighten scoring weights.

14) KPIs so we know it’s winning and not just “busy”

Match precision @ top‑1: ≥ 0.75 on human-reviewed samples

Time to call sheet each morning: < 15 minutes after cron start

Outreach conversion: % of calls leading to scoped need within 2 weeks

Backfill wins: time‑to‑submit vs competitor benchmark

Block rate: < 3% of page loads; automated recovery working

If you want a single line verdict: Yes, this is the right architecture for what you described. It’s fast, cheap, explainable, and it keeps you lethal in the only way that matters: showing up with the right human and the right clearance before the other guy even notices the req got reposted.

You bring the accounts and the tolerance for LinkedIn’s mood swings. I’ll keep this pipeline unglamorous, durable, and mean.

Looking at your Sales Prospecting Playbook alongside the Competitive Capture & Insertion Strategy and the other files you’ve dropped in here, your core BD architecture already covers the human-side process really well — targeted prospecting, referral loops, wedge tactics, cold call frameworks.

Where I’d push changes is in the data automation & intel layer so your manual playbook can plug into a high-speed intelligence engine:

1. Integrate Real-Time Competitor Job Feed into the Prospecting Flow

Why: You already emphasize competitor reqs as the fastest path to revenue (“money sitting on the table”). Right now it’s manual. Automate scraping of Belcan, IG, TEK, Apex, SHINE, etc., and dump new reqs daily into your call sheet pipeline.

How: Puppeteer + n8n flow → parse role, clearance, location, labor category → tag with prime/program match from FPDS/USAspending.

Outcome: Every morning, your AMs see a ranked list of hot reqs with the prime, end-customer, and clearance, ready for calls.

2. Automated Org Chart Building for Program Targets

Why: Your Playbook says to know team size, structure, who the managers are. This is the hardest part to do manually.

How:

Scrape LinkedIn profiles for employees with target program keywords (Sentinel, NGGBSD, SOCOM, NGIC, etc.).

Cross-reference ZoomInfo or Clearbit to confirm contact data.

Classify as “Prime Employee” vs. “Contractor” using employer name patterns & resume content.

Auto-build a layered chart in Airtable or Neo4j.

Outcome: Live, navigable team maps showing where to start with contractors for intel and where to go for decision-makers.

3. Pain Point & Labor Gap Signal Detection

Why: You can weaponize what’s in your Playbook (“why was this role open,” “persistent vacancies”) with automated pattern detection.

How:

Monitor competitor postings for “long open” status (reposts / >60 days open).

Track skillset scarcity by clearance + location from ClearanceJobs trends.

Flag patterns: “TEK has 3 unfilled ServiceNow dev roles at Hill for 90+ days.”

Outcome: Gives you pre-call ammo that shows you’ve done homework — e.g., “We saw the Sentinel MS&A req’s been up since March, we’ve got a candidate now.”

4. Auto-Built Call Sheets

Why: Your Playbook’s “cold call strategy” is only as good as the contact list.

How:

Daily n8n job → merge job scrape + org chart + ZoomInfo data → filter to “users” (managers who’ve hired for similar roles before).

Tag with wedge-tactic notes (“Competitor turnover,” “Hard-to-fill skill,” “Recompete coming”).

Outcome: Every rep gets a prioritized hit list that’s program- and gap-specific, not generic.

5. Clearance-Aware Talent Bench Tracker

Why: You’re telling managers “we have a pipeline” — back that with an actual, searchable clearance bench.

How: Sync Bullhorn → filter for active clearances, certifications, past performance matches → store in vector DB so AMs can pull matches in real time during calls.

Outcome: In a meeting, you can instantly say, “I’ve got three TS/SCI cloud engineers available now for Huntsville.”

6. Competitive Watchboard

Why: Your Competitive Capture doc is rich intel, but static.

How:

Scrape and alert on competitor job feeds, contract wins/losses (FPDS, GovWin), key personnel changes (LinkedIn).

Dashboard in Grafana/Metabase that shows “live health” of each competitor by program.

Outcome: Lets you time wedge insertions when competitors show weakness (reposts, turnover, acquisitions).

If you want, I can map these automations to a single tech stack that runs entirely in your Windows + Docker Desktop + n8n + Puppeteer environment so you’re not paying for high-cost hosted APIs. That way your current manual playbook literally “plugs in” to a data engine and your AMs start every day with high-probability targets.

Do you want me to build that end-to-end architecture map next so you can see exactly what containers, DBs, and scripts to run for each engine? That’s where this goes from theory to plug-and-play.