ChatGPT

Proposed Architecture for Prime TS BD Intelligence Automation

Overview

Prime Technical Services (PTS) needs an integrated pipeline of five intelligence engines that run daily to gather and analyze business development data. The proposed solution uses open-source, low-cost tools orchestrated via **n8n** (a self-hostable automation platform) [1] to perform daily web scraping, data enrichment, and report generation. Each engine produces structured outputs stored in a unified backend, enabling both automated AI agents and human analysts to leverage the data. The design emphasizes **consistent data schemas**, modular components, and extensibility for future AI agent integration (e.g. CrewAI or LangGraph) with human-in-the-loop oversight. The figure below outlines the overall system architecture, with each numbered component corresponding to an engine described in detail in subsequent sections.

Main Components & Flow: Every 24 hours, the **Scraper Engine** (1) collects fresh data from target sources and normalizes it into a **Jobs Dataset**. Next, the **Program Mapping Engine** (2) enriches these job records by tagging them with the most likely DoD program and prime contractor. The enriched data feeds into the **Org Chart Reconstruction** module (3), which infers team structure and roles for each program. Simultaneously, aggregated insights flow into the **HUMINT Playbook Builder** (4) to generate talking points and outreach materials for business development. Finally, a **Program Scoring Engine** (5) computes priority scores for programs based on hiring trends and PTS's capabilities. All engines are coordinated via scheduled n8n workflows and write to a central database (with optional Airtable sync for easy viewing). This database can later be tapped by AI agents or BI tools. Below we describe each engine, the technology stack choices, data storage design, and how AI agents/humans interact with the system.

1. Scraper Engine – Daily Talent & Program Data Collection

Purpose: Gather all available cleared staffing job postings and program-related info from multiple sources (competitor career pages, ClearanceJobs, LinkedIn, SAM.gov, FPDS, USAspending) on a daily basis. This engine is the data foundation, ensuring PTS has an up-to-date feed of who is hiring for what roles on which programs.

Data Sources & Targets:
- **Competitor Websites:** For each key competitor company, the engine crawls their careers/job listings page to find new postings requiring security clearances. These pages may be static HTML or use common ATS platforms (Workday, iCIMS, etc.).
- **ClearanceJobs:** Scrape the ClearanceJobs board for postings (filter by clearance level or keywords if possible). ClearanceJobs is a niche site for cleared jobs, and postings here often include program or contract names.
- **LinkedIn:** Use LinkedIn's job search to find postings by competitors or containing program keywords. Because LinkedIn content is dynamic and behind logins, leverage available tools or APIs. For example,

Apify's open-source LinkedIn Jobs Scraper can extract structured data from LinkedIn job URLs (title, company, location, post date, # applicants, description, etc.) [2]. This yields rich metadata on each job post.
- **SAM.gov (Contract Opportunities):** Use the SAM.gov API [3] to pull newly posted DoD contract opportunities, focusing on those requiring staffing (e.g. large service contracts). While not "jobs," these can hint at upcoming programs and prime contractors.
- **FPDS/USAspending (Contract Awards Data):** Query USAspending (the official federal spending open data source) [4] for recent awards related to targeted programs or agencies. This provides funding amounts, award dates, and prime awardee names which will later feed the scoring and prime inference.
- **Other Sources:** If any "cleared staffing" feeds or social media sources (like specific LinkedIn groups or forums) are available, these can be added similarly.

Scraping Tools & Approach:
Maximize use of open-source scrapers and APIs to minimize cost:
- **Scrapy Framework:** Utilize Scrapy for custom web crawlers on competitor sites and ClearanceJobs. Scrapy is a powerful open-source Python framework for web data extraction [5]. We can build spiders for each competitor's careers page, parse job listing entries (title, location, etc.), then follow links to detail pages for full descriptions. Scrapy's built-in scheduling and pipelining will help normalize output.
- **Headless Browser/Puppeteer:** For pages requiring heavy JavaScript (if any), integrate a headless browser (e.g. Puppeteer or Playwright) via Scrapy or n8n to render and scrape. Many ATS job boards can be accessed via their REST APIs or predictable AJAX calls, reducing the need for full browser automation.
- **LinkedIn Scraping:** Given LinkedIn's anti-scraping measures, leverage existing solutions. For instance, the user's repository `linkedin-scraper` or Apify actors can be used. Apify's LinkedIn Job Detail actor specifically can fetch comprehensive job info in JSON [2]. This might require an authenticated session cookie or use of their paid proxy, but it greatly accelerates development. Another option is the open-source library `linkedin-jobs-scraper` (a Node/Python tool) which can search and scroll LinkedIn jobs.
- **Public APIs:** For SAM.gov opportunities, use the official REST API endpoints with appropriate filters (e.g. filter by last 24h and keywords for major programs). For USAspending/FPDS, use their API to pull records by keywords (program names or contract identifiers). These APIs return JSON which we can parse within n8n or a Python node.
- **Normalization:** All sources' outputs will be normalized into a **unified job posting schema**. Key fields include: `job_id` (unique per source), `source` (e.g. LinkedIn, ClearanceJobs), `company`, `title`, `location`, `clearance_required` (if mentioned), `posting_date`, `description_text`, and any `program_mentions` (list of keywords detected). The Scraper Engine ensures data cleanliness (strip HTML, convert dates, unify location names) so that downstream engines can rely on consistent fields.

Execution & Orchestration:
The Scraper Engine is triggered daily (e.g. early morning) by n8n's time-based trigger. The workflow can run multiple scraping tasks in parallel to speed up data collection, using n8n's ability to call HTTP APIs and run code. For example, one branch calls the LinkedIn scraping API, another runs a custom Python snippet (or an n8n Code node) to run Scrapy spiders for competitor sites, etc. n8n's open-source nature allows hosting this workflow on a small server with no recurring license costs [6]. After scraping, the data is consolidated (e.g. combined into one array or written into the database). Error handling is built-in: if a site is unreachable, the system logs it and continues with others, ensuring one failed source doesn't break the whole pipeline.

Output: A **Jobs Dataset** updated daily. Initially this may be stored as JSON files or directly inserted into a **"job_postings"** table in the backend database. Each record has all relevant fields and a timestamp. This dataset feeds the next engine.

2. Program Mapping Engine – Job-to-Program & Prime Matching

Purpose: Automatically match each scraped job posting to a specific DoD program (e.g. Sentinel GBSD, SMORS, IBCS) and infer the likely prime contractor for that program. This provides context on *what project* the hiring is for and *who* the major players are, turning raw job data into actionable intelligence.

Mapping Methodology:
This engine uses a combination of **dictionary-based tagging**, **text analytics**, and **external data cross-referencing**:

- **Program Dictionaries:** We will maintain a curated list of known program names, code names, and common acronyms (e.g. "Sentinel" and "GBSD" both map to the Ground-Based Strategic Deterrent program). This dictionary may also include known subprograms or nicknames gleaned from job posts. When a job's title or description contains one of these keywords, we flag it as a potential program match. For example, if a posting mentions "Sentinel program" or "GBSD", it's tagged to **Sentinel GBSD**. We'll also include context words (e.g. "ICBM" or "Minuteman replacement" hint at GBSD).
- **Toolchain & Tech Mentions:** Certain programs are associated with unique technologies or toolchains. For instance, IBCS (Army's Integrated Air and Missile Defense) might mention radar integration or specific battle command systems; a software-heavy program might list a specific software stack. By mapping tools/skills to programs (in a supporting dictionary), we boost confidence in matches. E.g. a job requiring experience with "Ground Control Station software" could indicate a UAV program.
- **Geography Clues:** The job location can be a strong hint. Many DoD programs center around particular bases or contractor facilities. For example, jobs in **Huntsville, AL** that involve missile defense are likely related to Army programs like IBCS or SMDC efforts; jobs in **Roy, UT** referencing GBSD likely tie to Northrop's Sentinel work. The engine will incorporate a geo-to-program heuristic: a mapping of locations to major programs in that area.
- **Role Family:** The type of role can provide context. A cluster of openings for e.g. test engineers, software developers, and a project manager all at one location for one company likely belong to the same program team. By grouping postings by company + location + time, then examining titles, we can infer if they belong to a single program ramp-up. The mapping engine will group related postings and assign the same program tag if other signals align.
- **External Contract Data for Prime Inference:** Once a job is mapped to a program, the engine infers the prime contractor. A simple approach uses a lookup table of known primes for key programs (e.g. Sentinel GBSD → Northrop Grumman). For less obvious cases, the engine queries **USAspending/FPDS data** by program keywords to see recent contract awards. Since USAspending is the official source of federal spending data [4], it can reveal which company received the main contract for that program. For example, searching USAspending for "IBCS" would show major awards to Northrop Grumman (prime) and identify key subcontractors. The engine can automatically assign the prime based on that result. If a competitor (subcontractor) is hiring, the prime is still Northrop, which is important context for BD outreach.
- **Machine Learning (optional):** Down the road, we can introduce an ML classifier (trained on historical tagged data) to predict program from job text. However, initially a rules-based approach with dictionaries and fuzzy matching (using libraries like **fuzzywuzzy/TheFuzz** for acronym similarity) will be used for transparency and ease of QA.

Confidence & Human QA: Each job-program match will include a confidence score (e.g., exact keyword match in title = high confidence, only tool/location hints = lower confidence). For low-confidence mappings or unknown keywords, the system flags them for review. These could be output to an Airtable view or a simple report for a human analyst to verify, embodying the human-in-the-loop principle. This design allows a human to quickly correct or confirm mappings, and those corrections can feed back into expanding the dictionaries.

Implementation: The Program Mapping Engine can run as a Python script (triggered by n8n after scraping). It will read new job records, iterate through each, and apply the logic above. Maintaining the dictionaries as JSON or database tables (e.g. a table of `program_keywords`) will make updates easy. The script can also call the USAspending API for prime lookup when needed (caching results to avoid repeated calls). The result is an **enhanced jobs dataset** where each job record now has fields like `program_name` and `prime_contractor` (plus a confidence or source of mapping).

Output: Updated job records written back to the database (or Airtable). Additionally, we create or update a **"Programs"** table that inventories each program observed, with fields: `program_name`, `prime_contractor`, `total_openings` (count of related jobs), `companies_hiring` (list of competitors with openings), etc. This provides a snapshot of program activity. These program records will be used by later engines (Playbook Builder and Scoring).

3. Org Chart Reconstruction – Team Composition & Hierarchy Inference

Purpose: Reconstruct an approximate organizational chart for each program's team, based on job postings (and any other available breadcrumbs). This engine answers: "What might the program's team look like? Who are they hiring and who leads those teams?"

Approach: Even without direct org charts, we infer structure by analyzing roles and their relationships:

- **Clustering by Program:** Using the program tags from Engine 2, group job postings by program and company. For each program X at company Y, consider we are trying to map out that contractor's team for X.
- **Role Classification:** Classify each job title into a **level or role category**. For example, identify **Management roles** (titles like "Program Manager", "Project Lead", "Team Lead", "Director", etc.), **Senior IC (individual contributor)** roles (e.g. "Senior Software Engineer", often mentors or tech leads), and **Junior/Mid IC roles** ("Engineer I/II", "Analyst", etc.). Also flag any **Administrative or Support** roles (e.g. "Program Scheduler", "Recruiter for Program X"). This can be done via keyword rules or a small NLP model. We'll build a "macro position" library similar to what some LinkedIn org scripts use [7]. Essentially, we map titles to a hierarchy level.
- **Location Grouping:** Within each program team, group positions by location or department if applicable. If the program spans multiple sites, likely each site has its own sub-team. For instance, a program might have a dev team in one city and a test team elsewhere, each with their own lead. The engine will group jobs by location and see if similar titles exist across locations.
- **Reporting Structure Inference:** Use the above information to infer a tree structure. For each program:

- Identify the highest-level manager role available (e.g. “Program Manager” or “Site Lead”). This becomes the top of the org chart for that program at that company.
- Attach other roles under likely supervisors. For example, if we see multiple “Software Engineer” and “Systems Engineer” roles and one “Engineering Manager” role in the same location, we infer those engineers report to that manager. Similarly, if there’s a “Chief Engineer” and several “Lead Engineer” postings, the “Leads” might report to the Chief, who reports to the PM.
- Use counts to gauge team size. E.g., 10 open junior positions under one manager suggests that manager’s span is ~10 (which is plausible for a team lead). If two manager roles are listed (e.g. Software Manager and Test Manager), likely both report to the PM, indicating two subteams.
- Leverage any clues in job descriptions: sometimes postings mention “reports to the Program Director” or “leading a team of 5”. Incorporate such text if present. This is rare in postings but valuable when found.
- **Historical/External Breadcrumbs:** If available, incorporate **LinkedIn people data**. For example, using a LinkedIn scraping tool, search for people at competitor companies with that program in their title. If someone on LinkedIn is “Program Manager – Sentinel” at Competitor Co., that confirms the top role. Their LinkedIn might show their direct reports or at least job titles of reports (Sales Navigator Org chart features could be mimicked via scraping if we have access). For now, we assume minimal external people data, but the system is designed to plug this in later for refinement.

This process mirrors a Python script shared on Reddit, where an algorithm used job titles and locations to build org charts and make a “best guess” of reporting lines [7] . We follow a similar heuristic approach.

Visualization & Data Model: The engine will produce a structured representation of the inferred org chart for each program. A suitable model is a tree (or list of edges). We can represent it as: for each program, list each role (node) with attributes (title, level, location) and a `reports_to` field (linking to the manager node). For storage, a simple **OrgChart table** could have columns: `program_name`, `role_title`, `reports_to_title`, `location`, `level_rank`, `count` (number of such roles). Alternatively, we store as a JSON blob (hierarchical) in a field of the Program record. To visualize, we might later export this to a tool like draw.io or use a JavaScript library for org charts. Initially, a textual outline (like bullet indents) can be generated for quick review.

Quality Assurance: Org inference is imperfect, so this engine too will flag uncertainties: e.g., if it’s unclear which manager a role belongs under, or if multiple possible “top” nodes exist. Those can be presented for human review. Because this doesn’t directly feed external outputs (it’s mostly internal intel), the risk is lower, but we still allow a BD analyst to adjust assumptions. Over time, user feedback (e.g. confirming that Program X likely has two teams reporting to one director) can be fed back (perhaps via editing a config file of org assumptions).

Output: For each program, an **inferred org structure**. Example output for a hypothetical program “Sentinel GBSD” (for a subcontractor):

- Program Manager (Sentinel) – **Location: Roy, UT** – 1 position
- Systems Engineering Lead – Location: Roy, UT – 1 position
    - Systems Engineer (mid) – Location: Roy, UT – 3 positions
    - Systems Engineer (junior) – Location: Roy, UT – 2 positions
- Software Development Lead – Location: Roy, UT – 1 position
    - Software Engineer (mid) – Location: Roy, UT – 4 positions

    ◦ Software Tester – Location: Roy, UT – 2 positions
• Logistics/Support Manager – Location: Huntsville, AL – 1 position
    ◦ Field Support Technician – Location: Huntsville, AL – 2 positions

This illustrates how the engine might break down a team by roles and sites. The actual output could be stored in a table or delivered in a Markdown/HTML format for easy reading.

4. HUMINT Playbook Builder – Pain Points & Outreach Scripts

Purpose: Aggregate insights from the data (engines 1–3) to generate human-intelligence (HUMINT) playbooks for PTS's BD team. These playbooks include talking points, call scripts, and email openers tailored to each target program's hiring pain points, staffing gaps, and challenges. Essentially, this engine translates raw data into narrative form that a BD professional can immediately use to engage hiring managers.

Key Insights to Aggregate: For each program (and associated company or hiring manager), compile:
- **Chronic Openings / Labor Gaps:** Identify roles that have been open for a long time or posted repeatedly over weeks. If our daily scrape shows the same job still open for 60+ days or re-listed multiple times, that's a red flag that the company struggles to fill it. The playbook will highlight these hard-to-fill roles (e.g. "Multiple attempts to hire a cleared DevOps Engineer – indicates a talent gap").
- **Volume and Urgency:** If a program suddenly posts many jobs at once (hiring surge), it suggests a new contract award or ramp-up. Conversely, consistently high volume over months indicates ongoing needs. The playbook quantifies this ("XYZ Corp has posted 15 jobs for IBCS in Q3, suggesting aggressive expansion").
- **Clearance Mismatches:** Note if positions require unusually high clearances for the salary or role. E.g., a Help Desk role requiring TS/SCI – perhaps indicating they can't find people willing to take a lower-level job with such a high clearance. Also highlight if many openings need **polygraph** (especially hard to find). This info is valuable for BD: "Hiring for polygraphed engineers – likely facing an extremely limited talent pool." Indeed, the cleared labor pool is limited; it's reported that there are tens of thousands more cleared jobs than cleared candidates to fill them [8]. This shortage forces contractors to compete harder and could be a pain point we exploit by offering ready candidates.
- **Location & Churn Challenges:** Emphasize any geographic difficulties. If jobs are in remote or expensive areas (e.g. rural base towns or high cost cities) and require on-site work, note that as a hurdle for the company. Also, if multiple companies are hiring in the same small market, talent might be jumping around (churn). For example, "High turnover likely at Offutt AFB – 5 companies competing for cleared cyber analysts in Omaha." Such context can be used in conversation ("We know it's tough to keep talent in that area; we have strategies to help with retention.").
- **Compensation Clues:** While many postings won't list salaries, any that do can be gold. Extract any salary ranges found. If they seem low for the clearance/skill (maybe gleaned from general market knowledge or sites like Glassdoor), that's a talking point ("Your offered salary may be below market for TS/SCI in that region, contributing to hiring delays"). If we have no direct salary info, we can mention general market rates from public sources to show we're informed.
- **Past Performance Alignment:** Cross-link what PTS can offer. For the playbook, we'll maintain notes on PTS's own past performance and talent pool. For example, if PTS has placed network engineers on similar Air Force projects, the script should mention that experience. We can tag programs with internal **capability fit** (High/Medium/Low) based on PTS's expertise. High-fit programs get more aggressive pitches (since we have credibility there).

Content Generation: Using the above data points, the engine drafts tailored outreach content:
- **Call Script Outline:** A short briefing a BD person can use in a phone call with a hiring manager. For example: - *Introduction:* "We've been following *Program X*'s hiring activity – congrats on the new contract expansion."
- *Pain Point Highlight:* "Noticed several openings for cleared software engineers have stayed unfilled for 3+ months [8], likely due to the TS/SCI clearance requirement and competition for talent. That's a challenge we frequently solve."
- *Value Proposition:* "Prime TS has a pipeline of cleared candidates experienced in [relevant tech]. We also have past performance on similar DoD programs, so we understand *Program X*'s domain."
- *Call to Action:* "We'd love to help fill those 5 roles quickly so your team can meet its deadlines."
- **Email Openers:** One or two punchy sentences that can kick off an email. E.g., "Hi [Name], I saw that *Project Y* is looking for 10 cleared analysts in Denver – tough market! We've identified a few cleared analysts interested in that work. Could we discuss helping your team?"
- **Meeting Briefs:** A one-page summary combining the data: program overview, recent hiring trends (# of open roles, required clearances), likely pain points ("security clearance backlog delaying onboarding", "salary constraints", etc.), and how PTS can address them (our candidate pool, quicker hiring process, etc.). This acts as a leave-behind for any meeting.

Technology & Tools:
We can leverage AI NLP to generate polished text while ensuring the factual points are from our data. A practical approach is to use **templates + GPT-based refinement**: - Create a template with placeholders (for counts, specific roles, etc.) for each section of the script. Fill in placeholders with data (numbers, program names, etc.).
- Then use a language model (like GPT-4 via API, or an open-source LLM like Llama 2 locally for cost savings) to refine the text for tone and clarity. Provide a prompt like: "Here are bullet points about Program X's hiring challenges and our company's strengths... Write a concise, professional call script highlighting these." The model will produce a nicely worded paragraph. This approach ensures consistency but uses AI to make it more compelling and natural.
- Use **OpenAI** if allowed for high-quality output, but since we prioritize low-cost, consider open-source alternatives. There are models tuned for business communications that we could host (depending on infrastructure). The volume of generation is small (a few paragraphs per program per day), so even using an API might be economically feasible.
- **Human Review:** Because these outputs will be directly used by humans in communication, have a human QA each generated script at least initially. n8n can route the draft outputs to an Airtable or email, where a BD lead reviews and tweaks them if necessary. Over time, as trust in the system grows, this could be more automated.

Output: A **"Playbook"** for each program, updated as needed (probably weekly or when significant changes occur, since daily emails might be overkill). This could be in the form of a document or an entry in a database. For instance, an Airtable base could have a table "Program Playbooks" where each record has Program, Pain Points (bulleted list), Suggested Script, Email Intro, etc., all ready to copy-paste. The playbooks will explicitly cite the evidence: e.g., "(Role X has been posted since June 1)" or "(70,000 more cleared positions than candidates in the market [8])" to show we've done our homework. By centralizing this info, PTS's BD team can quickly get up to speed on any program before a call.

5. Program Scoring Engine – BD Priority Ranking

Purpose: Quantitatively score and rank DoD programs to prioritize business development efforts. PTS can't pursue all opportunities at once; this engine surfaces which programs (among those being tracked) present the best opportunity **and** alignment for PTS. The score can guide where to focus outreach and resources.

Scoring Criteria: We propose a composite score (e.g. 0–100 scale) for each program, based on:

- **Hiring Demand (30% weight):** How many open positions and hiring companies are associated with the program. More openings = greater staffing need = potentially easier entry for PTS. A program with 50 open jobs across industry gets a higher score than one with 2 openings. We can normalize this (e.g. 10 points for each quantile of volume). Recent spikes in hiring demand could further boost the score.
- **Pain Level (20% weight):** Based on Engine 4's analysis – how severe are the hiring challenges? If a program shows multiple pain indicators (many high-clearance roles, long vacancy times, niche skills), then presumably the program team is "in pain" and more likely to welcome outside help. We can assign points for factors like ">X% of jobs require TS/SCI", "average job age > 60 days", etc. A program where "positions outnumber cleared people" (as highlighted by industry stats [8]) would score high here.
- **Funding & Contract Size (20% weight):** A program's importance correlates with its budget. Using USAspending/FPDS data, we find the total obligated funds or contract value for the program (for primes and major subs). Larger programs (e.g. multi-billion dollar programs like GBSD) score higher because they have more scope (and likely more staffing contracts to go around). Also, if additional future funding is appropriated (from budget documents or news), that's a positive indicator. This data is accessible via the USAspending API by filtering on program references or award titles.
- **PTS Past Performance Fit (20% weight):** We factor in how well PTS is positioned to deliver. If PTS has prior experience or existing cleared talent in the program's domain (e.g. cyber, aerospace engineering, software development, etc.), then the score should be higher because PTS is more likely to win subcontracts or make successful placements. This could be a manual rating: for each program, assign a fit score (High, Medium, Low mapped to numeric values) maintained in a config file by PTS leadership. The engine reads that and converts to points.
- **Competitive Intensity (10% weight):** If every competitor is already heavily involved or if the program is locked up by incumbents, it might be harder for PTS to break in, lowering priority. We can proxy this by the number of **different companies hiring** for the program. If many companies (including small ones) are getting work, it might be more open. If only the prime and one known sub are hiring, it might be tougher. Fewer hiring companies could either mean an opportunity (nobody else has filled the gap we can) or a closed shop – this is tricky, but we will assume that a moderate number of hiring firms is healthiest. We can incorporate a simple formula or let BD experts tweak this parameter.

These weights can be adjusted, but the idea is to combine **data-driven factors** (hiring numbers, funding) with **strategic factors** (PTS fit).

Implementation: The Program Scoring Engine runs after all prior data is updated (so it has the latest numbers). It can be implemented as a Python function or even a database stored procedure that calculates the score for each program record. Steps: 1. For each program in the Programs table, gather metrics: count of open jobs (from jobs table), average job age or % with high clearance (from jobs data), total funding

(from an aggregated contracts table or API call), PTS fit score (from a manual list or table), and number of hiring orgs (from jobs data).
2. Normalize each metric to a common scale (0-10 or 0-1) to avoid any single factor dominating due to raw scale differences. For example, if the max open jobs among programs is 50, that gets a 10, and others get proportional scores. Funding might be scaled logarithmically (since budgets vary widely).
3. Apply weights and sum up to get final score.
4. Rank programs by score and perhaps bucket them (High Priority, Medium, Low).

The engine could also produce a brief rationale alongside the score for transparency, e.g., *"Program X: High priority due to 20 open positions and strong PTS fit, despite moderate funding."*

Output & Usage: The scores are written to the Programs table (new fields: `score`, `priority_level`). We then output a **"Program Priority Dashboard"** – possibly an Airtable view or a simple web page – that lists all tracked programs with their score, breakdown of factors, and recommended action (e.g. "Engage immediately" for top scorers, "Monitor" for mid-tier). This dashboard can be visualized easily using open-source BI tools (Metabase or Superset) or even Google Sheets for simplicity. The main goal is to give PTS leadership a quick glance at where to focus.

For example, a top entry might read:

- **Sentinel GBSD – Score: 88** (High Priority)
  Factors: Very high hiring demand (30+ openings), TS/SCI roles unfilled, PTS has relevant past performance. Prime: Northrop. *Action:* Reach out to Northrop team and key subcontractors (e.g. XYZ Corp).

Whereas a low priority might be a program with few openings or outside our wheelhouse.

PTS can revisit the scoring formula over time. The engine is transparent and easily tunable – since it's based on data we store, adjusting weights or thresholds is straightforward (no black-box ML here). This flexibility is important as BD strategy might shift.

Orchestration & Workflow Integration

All five engines are interconnected and run in sequence using **n8n** workflows. n8n's visual workflow editor and scheduling capabilities make it ideal here [1]. We will set up multiple workflows (or a single complex one with sub-workflows) as follows:

- **Daily Scrape Workflow:** Triggers at a low-traffic hour (e.g. 3 AM) every day. Nodes: trigger → multiple "HTTP Request" or custom nodes for each data source → data aggregation node → "Database" node to upsert job records. Each scraping node can run in parallel to speed up execution. After all complete, it can signal the next workflow.
- **Post-Scrape Processing Workflow:** Either triggered automatically by the above (via webhook or a simple time buffer), or scheduled shortly after. This workflow runs the Program Mapping Engine (could be a single Code node running our Python mapping script, or a series of function nodes). It updates records with program info and primes, logging any unmapped cases. Next, it calls the Org Chart engine (another code node) to update the OrgChart data. Then it calls the Program Scoring calculation and writes scores. Essentially, this workflow takes raw data to enriched analytics. If any

part fails (e.g. mapping script error), n8n can catch it and notify via email/Slack to the admin, so issues are resolved promptly.

- **Weekly Playbook Workflow:** The HUMINT playbook might not need daily updates; weekly or on-demand is sufficient (unless a specific meeting is upcoming). We schedule this workflow every Monday 6 AM, for instance. It pulls the latest data, calls an LLM API or local model to generate updated scripts for each high-priority program, and then delivers the output. Delivery could be sending an email to BD team with each program's mini-playbook, or updating an Airtable/Notion page that the team can consult. Human reviewers can be CC'd or included in the loop here. If using GPT via n8n, we'd use an "HTTP Request" node to OpenAI's API with our prompt and data. n8n can handle secrets (API keys) securely via environment variables.

All workflows are **containerized** (we can run n8n in Docker [9] along with any custom services needed) and deployable on a cloud VM or on-prem server. Monitoring features (like n8n's execution logs and error notifications) will be enabled to ensure reliability.

By structuring it into clear workflows, we allow easy maintenance: e.g. if a new data source needs adding, we update the Scrape workflow; if a new program comes into scope, we just add it to the dictionary and possibly adjust the playbook template, without touching the rest.

Data Storage & Schema Design

Choosing the right backend is critical for a "single source of truth" that all engines (and future AI agents) can draw from. We have two main options: **Airtable** or a **SQL database (PostgreSQL)** – or a combination. Below is the recommendation:

- **Primary Database – PostgreSQL:** Use PostgreSQL as the core data store for all structured data. Postgres is free, open-source, and well-understood, and can easily handle the kind of relational data we have (jobs, programs, etc.) with the volumes expected. It also allows complex querying (useful if we want to do analysis or if an AI agent needs to query). Compared to Airtable, Postgres scales to millions of records with no issue [10], which is important as daily scraping accumulates data over months. We can host Postgres on a small cloud instance or even use a managed free tier for minimal cost. Additionally, if we later need vector search (for semantic queries on job descriptions), Postgres can be extended with the pgvector plugin – an open-source way to store embeddings – avoiding the need for an external vector DB. This aligns with our open-source, self-host philosophy [11].

- **Airtable (Supplementary):** Airtable shines as a quick front-end and is very user-friendly for non-technical team members [10]. We recommend **syncing key data to Airtable** for the BD team's consumption, but not relying on it as the system of record. For example, an Airtable base could have read-only synced tables for "Jobs", "Programs", and "Playbooks" from the Postgres DB (this can be done via the Airtable API or third-party sync tools). The BD team can then view and even tweak data in Airtable (those edits could be captured and fed back if needed). Airtable will handle a few thousand records fine, but if the job postings table grows too large, we might limit the sync to recent items or summary info. Another advantage is Airtable's interface designer – we could make a simple dashboard or form for human QA tasks (e.g., a view filtering jobs that have low confidence mapping, where a person can fill in the correct program from a dropdown). This gives the "human-in-the-loop" an easy way to intervene without digging into code. We note Airtable's limitations (record limits, performance) [12], so it's a convenience, not a dependency.

- **Schema Design:** We'll define explicit schemas for each entity:

- **JobPosting:** (id, source, company, title, location, clearance_level, posted_date, description_text, program_name (FK to Program), prime_contractor, scrape_date, status). We include program_name as a foreign key linking to a Program table. `status` could indicate if filled or stale (if a job disappears from the source, we might mark it filled).
- **Program:** (name, prime_contractor, total_openings, last_update, pts_fit (score/category), funding_amount, clearance_level_required (e.g. highest clearance among its jobs), score, priority). Name is primary key (or an ID). We also store any static info like customer agency (Air Force, Army, etc.) if known.
- **OrgChart:** We can have a table with one row per inferred role: (program_name FK, role_title, level, location, reports_to (role_title or an ID reference within this table), count_openings). Or simply store a JSON structure in the Program table in a field `org_chart`. A normalized approach is trickier to query but allows more detail if needed.
- **Playbook/Insights:** Possibly a **ProgramInsights** table: (program_name FK, pain_points_text, recommended_script_text, last_generated). Or we embed the playbook in a text field in Program table (if we don't need a separate table). For email templates, etc., storing as text or markdown is fine.
- **Users/Contacts:** Not mentioned in requirements, but eventually we might integrate contact data (hiring managers names, etc.). The architecture can accommodate an additional table for contacts (with program or company association), which the BD team could populate as they identify targets. Our playbook engine could then personalize the email opener with a name, etc.

All data in Postgres can be exposed to the AI layer or BI tools. If we decide to incorporate a vector database for semantic search or agent queries, an open-source solution like **Qdrant or Weaviate** could be used as an alternative to Pinecone [11] – but this is optional. For example, we could embed each job description using Sentence Transformers and store vectors to enable queries like "find similar positions we filled before" for an AI assistant. However, until the AI agent is a reality, this might be over-engineering. PostgreSQL with full-text search might suffice for basic needs.

Storage Cost Efficiency: All tools chosen are free or low-cost to run. Postgres on a small instance (or even a free tier of ElephantSQL or similar) covers our needs initially. Airtable's free plan might handle the limited synced data (if not, PTS might upgrade to a paid plan, but only for the convenience features). We avoid proprietary data warehouses or expensive SaaS tools, aligning with budget constraints.

AI Agent Integration & Human-in-the-Loop Readiness

While the immediate design works with deterministic logic and human oversight, it is "AI-ready" by design for future enhancements. We anticipate integrating agentic AI frameworks like **CrewAI** or **LangGraph** to automate more complex decision-making while keeping humans in control:

- **CrewAI for Multi-Agent Orchestration:** In the future, we could have multiple AI agents collaborating on tasks like: monitoring new sources, extracting specific info (using tools like the ScrapeGraph AI scraper via CrewAI [13]), or handling interactive queries from users. CrewAI provides a way to coordinate such agents with tools, as seen with its ScrapeGraph integration which allows AI to intelligently scrape content based on natural language instructions [14]. For example, an agent could be tasked with "Find any emerging programs related to space systems" – it could use web search tools to find news, then ScrapeGraph to extract details, feeding back into our system. Our architecture will make it easy to plug this in because data storage and APIs are already in place. We would simply grant the agents controlled access (perhaps via an API layer on our DB).

- **LangGraph for Workflow Agents:** LangGraph is an open-source framework by LangChain for building complex, stateful agent workflows [15]. We can leverage LangGraph to create an AI **"BD Analyst Agent"** that can answer questions or generate custom reports from our data. For instance, a user might ask the agent, "Which programs hiring for cloud engineers should we target this month?" The agent would break this down: find programs with cloud-related jobs, filter by our scoring, etc., using the database and perhaps vector search as tools. LangGraph's graph-based orchestration would let us define this reasoning flow explicitly, ensuring reliability. Importantly, LangGraph supports **human-in-the-loop (HITL)** interactions natively [16]. We can configure the agent to pause and request human approval when confidence is low or a decision is critical (much like our current system flags items for review). For example, an agent doing program mapping could ask a human "I found a new acronym 'ABC-X' in a job post. Should I map it to Program Alpha?" – a human can confirm or correct, and the agent learns. Our consistent data schemas make it straightforward for an agent to retrieve info; and by incorporating HITL at key points, we ensure the AI's actions remain supervised and correct.

- **Data Access for Agents:** To prepare, we might implement a simple internal API (REST or GraphQL) to query the Postgres database. This API can have endpoints like `/jobs?program=XYZ` or `/programs?priority=high` which an AI agent could call (with proper auth). LangChain/CrewAI agents can be given tools that call these APIs to fetch data, rather than free-form querying the DB (which could be error-prone). We could also pre-index documents for retrieval: e.g., store each program's "profile" (a summary of everything we know, including active jobs, challenges, etc.) and allow a vector-based retrieval if an agent needs to recall context to answer questions. Using an open-source vector DB (or pgvector in Postgres) ensures we stick to low-cost solutions [11].

- **Maintaining Human Oversight:** Even as we add AI, the final architecture will keep a human-in-the-loop for critical outputs (especially anything client-facing, like the playbook content). LangGraph's design inherently values transparency and state, so we can log an agent's reasoning and present it for review. For example, before the agent sends out an automated email to a potential client (if we ever go that far), we can have a human review that email. Our system already collects the needed data for a human to validate decisions (confidence scores, intermediate results in tables), which is crucial for effective HITL integration [16].

In summary, while initial implementation may use AI only for generating text, the architecture is flexible to incorporate more autonomous agents gradually. The combination of **structured data + APIs + agent frameworks with HITL** means PTS can safely experiment with AI-driven automation (like an "AI BD Assistant") when ready, without a complete redesign.

Tech Stack Summary

Bringing it all together, here is the proposed stack and tools for each part of the system:

- **Workflow Orchestrator: n8n** (self-hosted) – manages scheduling and integration of all tasks. Chosen for its visual workflow and open-source flexibility (no per-run costs) [6] .
- **Scraping & Data Collection:**
- **Scrapy** (Python framework) – custom spiders for websites (competitor pages, ClearanceJobs) [5] .
- **Apify Actors or Puppeteer** – for LinkedIn and any dynamic sites, to use existing high-level scrapers [2] and avoid re-inventing the wheel.
- **Requests/BeautifulSoup** – for simpler HTTP GET parsing (SAM.gov API, etc.).
- **User's Existing Tools:** We will also leverage any in-house scripts PTS already has (e.g., the `fpdsScraper` repo for FPDS data, LinkedIn profile scrapers from `CrossLinked` or similar). Incorporating these ensures we maximize prior work.
- **Data Processing & Enrichment:**
- **Python** – for program mapping logic, org chart inference, and scoring. Python's rich ecosystem (pandas for data handling, regex/fuzzy matching libraries for text, etc.) makes these tasks easier.
- **spaCy or NLTK** – for any NLP like named entity recognition (e.g., identifying program names or clearance terms in text).
- **TheFuzz (fuzzywuzzy)** – for approximate string matching of acronyms and program names.
- **Geopy (or similar)** – if needed to normalize location names or derive region info.
- **Database: PostgreSQL** – main data store for structured results. Consider adding the **pgVector** extension if semantic search is needed later (open-source alternative to Pinecone) [11] .
- **Frontend/Visualization:**
- **Airtable** – synced views for quick human consumption and editing of key data (small-scale, user-friendly interface) [10] .
- **Metabase or Superset** – optional, for more advanced dashboards (could show trends like "openings over time" or a leaderboard of primes by number of openings).
- **Graphviz/D3** – for drawing org charts if needed. We might use Graphviz via Python to output an org chart diagram (and embed image in reports).
- **AI/LLM:**
- **OpenAI GPT-4 or GPT-3.5 API** – to generate playbook narratives from bullet points. This ensures high-quality language. We will use it judiciously to control cost (few short prompts per week).
- **Local LLM (optional)** – if data sensitivity or cost is a concern, we can deploy a smaller model (like Llama-2 13B) on a GPU server for text generation. The quality may be lower, so initially we lean on the API.
- **LangChain/LangGraph (future)** – frameworks to build agents on top of our data.
- **CrewAI (future)** – to orchestrate multi-agent processes or integrate AI tools like ScrapeGraph for adaptive scraping.

All these components are either open-source or have free/community tiers. We'll containerize the core components (n8n, the Python processing code, Postgres) using Docker Compose for easy deployment and portability. This also simplifies scaling individual pieces (e.g., if scraping becomes heavy, we could distribute spiders across multiple containers).

Next Steps for Implementation

Finally, to move from design to build, here are clear next steps:

1. **Infrastructure Setup:** Provision a server or cloud environment for development. Install Postgres and n8n. Ensure the environment is secure (since we might be handling sensitive competitive data). Set up a Git repository for the custom code (scrapers, processors) – possibly in PTS's GitHub for version control.

2. **Define Program & Competitor Lists:** Collaborate with PTS stakeholders to enumerate the initial set of target programs (and synonyms) and competitor companies to monitor. Populate the program dictionary and competitor site URLs. This is important input for Engines 1 and 2.

3. **MVP Scraping:** Implement one scraper at a time and verify data:

4. Start with an easy source (e.g., ClearanceJobs search for a keyword) using Python + requests. Parse HTML and confirm we can extract job info.
5. Next, try one competitor's careers page. Use Scrapy to handle pagination etc. If any site is tricky (requires login or has anti-scraping JS), note it for potential alternate approach (maybe use Selenium or skip if low value).
6. For LinkedIn, obtain necessary session cookies or API credentials (Apify API token if using Apify). Test scraping a known job posting to confirm we get the needed fields [2].
7. As we scrape, build the normalization logic – ensure all outputs map to the common JobPosting schema fields.

8. Load some sample data into Postgres (perhaps as a trial run, not yet scheduled daily).

9. **Data Model Implementation:** Create the tables in Postgres for Jobs, Programs, etc., according to the schema design. Use primary keys, foreign keys, and indexes (index on program_name, for example, to speed up lookups for program mapping and analysis).

10. **Program Mapping Module:** Develop the Python module for Engine 2. Include the dictionary lookup logic and a function to query USAspending API. Test it on the sample job data collected. Adjust dictionary entries or parsing as needed when you discover variations in how programs are mentioned. Validate prime inference by checking a couple of known programs (e.g., if your sample has "Sentinel", see that it assigns Northrop correctly).

11. **Org Chart Module:** Develop Engine 3's logic using the sample data. This is more heuristic – so create a few test scenarios (you can fabricate a small set of job postings for a hypothetical program to see if the code builds a sensible org chart). Ensure the output structure is clear. Possibly write results to console or a temporary table for review. Iterate on rules (e.g., threshold of how many juniors per manager, keywords for leadership roles) until it matches intuitive expectations.

12. **HUMINT Playbook Generation:** Draft template texts for call scripts and emails. Manually write one using real data to set a standard. Then implement Engine 4:

13. Write a function to compile the bullet point data for a program (e.g., “3 roles open >60 days”, “2 companies hiring at Location X”, “clearance TS/SCI” etc.).
14. Feed that to an LLM. Initially, test with OpenAI’s GPT-4 in “temperature=0” (deterministic) mode to see if it produces a good paragraph. Adjust the prompt or few-shot examples to steer tone (professional, concise).
15. Have a human BD member review the output for one program and provide feedback. Refine prompt or templates accordingly.

16. Implement the generation in the workflow, but gate it such that outputs either go to a draft location or require a manual trigger to send, to maintain QA control.

17. **Program Scoring Calculation:** Implement Engine 5 as a SQL query or Python code. Use actual numbers from your sample (you may simulate some values if needed). Review the ranking with PTS’s BD team to see if it matches their intuition. If not, tweak weights. The goal is that the scoring mechanism is transparent, so document the formula and possibly allow weight adjustments via a config file for easy tuning.

18. **Integrate Workflows in n8n:** Now that individual pieces work in isolation, create the n8n workflows:

19. Build the Scrape workflow with nodes for each source. Use small test runs (maybe limit to 1-2 items) to ensure each node correctly fetches and passes data. Use n8n’s debugging to see the data flow.
20. Build the Processing workflow. Possibly use an n8n Function node (JavaScript) to call an HTTP endpoint of a small Flask API that runs the Python mapping & org logic (if we choose to keep Python separate). Alternatively, use the n8n Execute Command node to run a Python script on the server. Ensure that after execution, data is written to Postgres.
21. Build the Playbook workflow with an OpenAI node (or HTTP node calling the OpenAI API). Set it to manual trigger at first for testing.

22. Set up error-catching nodes: e.g., on failure of any scraping node, send an email alert with the error. Similarly for processing.

23. **Testing & Iteration:** Run the entire pipeline end-to-end on a controlled basis:

    - Do a trial “daily run” with the workflows but perhaps not at 3AM – trigger it manually and observe. Check the Postgres tables afterwards to ensure everything is populated correctly.
    - Examine a sample program’s data across the tables: Job postings should be linked to a program, that program should have a score, an org chart, and a playbook entry. See if they all make sense together.
    - Gather feedback from domain experts. They might catch mis-assignments (e.g., “This job is actually for Program Y, not X”). Use that to improve the dictionaries and logic.
    - Ensure all citations or external data we use are up-to-date (for instance, if a program was renamed, update the dictionary).

24. **Deployment and Scheduling:** Once happy with results, schedule the n8n triggers for production. Keep an eye on the first few days of runs for any issues (perhaps some site’s HTML changed, etc.).

    - Also schedule a weekly review meeting where the BD team and tech team review the program rankings and playbooks, to provide continuous feedback and make adjustments.

- Plan maintenance: e.g., update scraping scripts when websites change (Scrapy spiders occasionally need tweaks).

25. **Documentation & Handoff:** Document the entire system for maintainers. Include how to update program dictionaries, how to add a new competitor site scraper, and how to trigger re-generation of playbooks on-demand. Also document how to pause the workflows or rerun them in case of failures.

By following these steps, we'll iteratively build the system and ensure each part works before adding the next. The result will be a robust, **daily automated BD intelligence pipeline** giving Prime Technical Services a data-driven edge in pursuing DoD program opportunities.

---

[1] [6] [9] Get to Know n8n: Easy Workflow Automation for Everyone | by Developer Awam | Jun, 2025 | Medium
https://medium.com/@developerawam/get-to-know-n8n-easy-workflow-automation-for-everyone-687c5d0c6da5

[2] LinkedIn Job Details Scraper · Apify
https://apify.com/piotrv1001/linkedin-job-details-scraper

[3] SAM.gov Get Opportunities Public API - GSA Open Technology
https://open.gsa.gov/api/get-opportunities-public-api/

[4] Government Spending Open Data | USAspending
https://www.usaspending.gov/training-videos

[5] Scrapy
https://www.scrapy.org/

[7] Python script to turn linkedin into an organizational chart - is this useful? : r/consulting
https://www.reddit.com/r/consulting/comments/gc5pqo/python_script_to_turn_linkedin_into_an/

[8] Cleared Hiring Trends | Clearance Processing Times | ClearanceJobs
https://about.clearancejobs.com/employers/learn-more/hiring-trends

[10] [12] Airtable vs SQL : r/Airtable
https://www.reddit.com/r/Airtable/comments/17x2pwc/airtable_vs_sql/

[11] Pinecone Alternatives: What Are Your Vector Database Options? | by DataStax | Building Real-World, Real-Time AI | Medium
https://medium.com/building-the-open-data-stack/pinecone-alternatives-what-are-your-options-ab656bc268b8

[13] [14] Scrapegraph Scrape Tool - CrewAI
https://docs.crewai.com/en/tools/web-scraping/scrapegraphscrapetool

[15] [16] What is LangGraph? | IBM
https://www.ibm.com/think/topics/langgraph