ChatGPT

# AI-Driven Competitor Intelligence Automation Blueprint

## Orchestration Workflows

Prime Technical Services' automation engine is orchestrated through **n8n** workflows that run on a daily schedule. The workflows coordinate scraping, data enrichment, analysis, and output delivery in a staged pipeline. Key workflows include:

- **Daily Job Scrape Workflow:** A scheduled n8n workflow (e.g. every morning) triggers **Puppeteer/ Playwright** scraping for each target competitor's careers page or job board. The workflow uses parallel scraping tasks (or sequential tasks with concurrency control) to navigate competitor job postings that require security clearances. Each job posting's details (title, description, location, date, etc.) are extracted and sent to the data pipeline. New postings are identified (by comparing with previously stored jobs) to focus processing on changes.

- **Cross-Reference & Enrichment Workflow:** After scraping, a follow-on workflow cross-references the collected job descriptions (JDs) against known **prime contractors**, overlapping job roles, and relevant **DoD programs**. This involves both semantic matching and rule-based matching:

- A **semantic matcher** uses a vector database to find similarities between a competitor's JD and a library of program descriptions or other companies' JDs. For example, the system generates embeddings for each JD and searches for nearest neighbors in a corpus of DoD program documents and past postings [1]. This finds conceptually related programs even if specific keywords differ.
- A **rules-based matcher** applies heuristic rules or keyword tagging. For instance, if a job posting explicitly names a program or contract (e.g. "SEWP" or a contract code), or mentions a prime contractor ("supporting Prime XYZ on Project ABC"), the workflow tags that job with the identified program/prime. These rules draw from Prime's **cleared labor capture playbook** (i.e. known terms, program acronyms, competitor aliases, etc.).

- The workflow then enriches each job entry with the cross-referenced info: likely program name, incumbent prime contractor, and whether the competitor appears to be a **prime or subcontractor** on that program.

- **Public Data Enrichment Workflow:** This workflow integrates **LinkedIn and ZoomInfo** data to add organizational context. For each competitor company (or even for specific job postings/team), the system gathers public data:

- Company profiles from LinkedIn/ZoomInfo to assess company size, divisions, and contract focus. Using LinkedIn's public info, the system can infer if the competitor is a small subcontractor (e.g. few hundred employees) or a large prime. It flags organizational layers – for example, a small firm hiring

for a role on a big program likely means they are a **subcontractor** under a prime, whereas the prime's own postings indicate top-level ownership of the program.

- Org chart clues: If available, the engine notes key personnel changes or repeated patterns (ZoomInfo might show org charts or employee counts in certain departments). This helps identify **staffing gaps and pain points** – e.g. if a competitor's profiles show many vacancies in a particular skill area or recent departures (implying backfills).

- The workflow attaches these insights to the relevant program or competitor record (e.g. tagging a competitor as "prime contractor on Program X" vs "sub on Program Y", and noting potential gap areas).

- **Federal Contract Data Workflow:** To contextualize the job postings against contract opportunities, the engine pulls data from **USAspending, FPDS, and SAM.gov**:

- It uses the USAspending API and FPDS (via beta.SAM.gov) to retrieve federal award records for programs of interest. USAspending.gov is the official open data source for federal spending [2], and current/historical FPDS procurement data reports can be generated through SAM.gov [3]. The workflow might run overnight (or early morning) to fetch updates on relevant contracts – focusing on **high-labor, long-period programs** that align with the job postings. For example, if a competitor JD is linked to "Program ABC," the system pulls the contract details: total award amount, period of performance, incumbent awardee (prime), major subcontractors, recent **modifications** (mods).

- The workflow flags contracts with attributes indicating opportunity: large labor components (many FTEs), nearing recompete dates, or frequent funding mods (which could signal scope changes or contractor performance issues). These data points are stored in the database and linked to the programs/jobs identified earlier.

- **Analysis & Briefing Workflow:** This is an AI-driven workflow that synthesizes all gathered data into human-intelligible insights. It runs after data collection is complete (e.g. every morning after scraping/enrichment):

- An **AI agent** (using GPT-5 or Claude) processes each program or opportunity, taking the compiled data (open roles, competitor info, contract stats, etc.) and generating a **"program brief"**. Each brief includes a summary of the program, the current staffing situation, and inferred pain points. The agent is prompted to produce **personalized talking points** for that program's context – e.g. if a competitor has had a cyber engineer role open for 90+ days on Program XYZ, a talking point might be how Prime Technical Services has a pipeline of cleared cyber engineers ready to step in.
- The workflow also generates **human-intelligence call sheets**. These are one-page digests for BD teams making calls to either the government customer or the prime contractor. A call sheet might list: key contract details (agency, value, end date), incumbent (competitor) performance indicators (e.g. "5 key roles open for 3+ months"), and tailored discussion points (for example, "Ask about delays in deliverables due to understaffing; emphasize our ability to rapidly fill cleared positions"). The AI ensures the tone and content align with Prime's competitive BD strategy and existing call flow scripts.

- The AI analysis uses both templated structure and dynamic insight. It can fill in structured data (like contract facts) and add narrative analysis on top.

- **Output & Notification Workflow:** Finally, the engine disseminates the results:

- It compiles the fresh insights into **CSV summaries** and updates **Airtable views** (for easy internal browsing). For instance, an Airtable base may have a "Opportunities" table with each program opportunity, its score, and key data; and a "Jobs" table with all scraped job entries. The workflow uses Airtable's API (via n8n's Airtable nodes) to upsert records so that the BD team can view the latest data in familiar Airtable dashboards.
- The workflow posts alerts to Slack/email. A Slack message might go to a "BD Intelligence" channel every morning highlighting top 3–5 opportunities with brief info and a link to more details or the Airtable view. Emails (or Slack file attachments) deliver the full **PDF briefs** for each major program to relevant team members. The PDFs are generated automatically (see Output Modules below) and provide a polished format for printing or sharing.
- All workflows include logging and error-handling as recommended in n8n best practices [4]. If any step fails (e.g. a scraper error or an API limit issue), n8n's error workflows capture it and either retry or alert the admin, ensuring the pipeline is resilient.

These orchestrated workflows together create a **daily intelligence cycle** – from data acquisition to insight delivery – all automated within n8n. The use of multiple sub-workflows and triggers keeps the system modular and easier to maintain (e.g. the scraping workflow can be updated independently of the analysis workflow). This design follows automation best practices to keep workflows **optimized, scalable, and resilient** [4] for real-world use.

## Container Architecture

The entire system runs on a Windows machine using **Docker Desktop**, with each major component encapsulated in a container for easy deployment and scalability. Docker Desktop supports running these containers on Windows with minimal setup [5]. The containerized components are:

- **n8n Orchestrator Container:** Runs the n8n automation server (via the official n8n Docker image). This container schedules and executes the workflows described above. It connects to all other services via network links – e.g. database, vector store – and uses mounted volumes for any persistent state (like logs or encryption keys). The n8n container is configured with needed credentials (API keys for LinkedIn, ZoomInfo, OpenAI, etc.) as environment variables. We use Docker to easily map port 5678 for n8n's UI and persist data in a volume (or bind mount) so workflow definitions and execution history survive restarts [6].

- **Scraping Service Container:** A custom Node.js container with **Puppeteer/Playwright** installed (including a headless Chrome/Chromium). This container is responsible for actually visiting competitor websites and scraping job postings. It may expose an API endpoint (so n8n can send it a target URL or company name to scrape), or n8n might execute commands inside the container. Using a separate container ensures all browser dependencies (Chrome libraries, fonts, etc.) are included and isolates the scraping process. On Windows Docker, this container will run using Linux containers (for Chrome headless) which Docker Desktop handles seamlessly. We include appropriate launch flags (like `--no-sandbox`) for Puppeteer in Docker [7]. The container writes scraped data either directly to the database or returns it to n8n via HTTP for further processing. (Alternatively, one could integrate Puppeteer in the n8n container by installing it there, but maintaining a separate service is cleaner and avoids bloating the n8n image.)

- **PostgreSQL Database Container:** A Postgres 15+ container serves as the system's structured data warehouse. It stores the scraped job data, enrichment information, contract records, scores, etc. The database is configured with persistent storage (Docker volume) to retain data. We also enable the **pgvector** extension in this Postgres container if we choose to use Postgres for vector embeddings storage. This allows storing embedding vectors and performing similarity search via SQL. (If pgvector is used, it effectively doubles as the vector DB; if not, see the separate Qdrant container below.) The schema design is described in a later section. The n8n container connects to this DB via the Docker network, and credentials are passed securely via env variables. Regular backups can be scheduled from this container or via an n8n workflow to export data (e.g. to CSV or dump files).

- **Vector Database Container (Qdrant – optional):** For advanced semantic search, a Qdrant container may be deployed (if we opt not to use pgvector). **Qdrant** is an open-source high-performance vector search engine that can run as a Docker container [8]. It stores neural embeddings along with metadata (payloads) for each vector [1]. The Qdrant container exposes a RESTful API (on a port, e.g. 6333) that the n8n workflows or AI agents call to insert vectors and query nearest neighbors. We can store each job posting and each program description as vectors in Qdrant, tagged with metadata like `company`, `program_id`, etc. Qdrant's ability to filter results by metadata at query time is useful – e.g. "find similar postings where `company != X`" to identify overlaps across competitors [1]. If using Qdrant, we likely use OpenAI or local model to compute embeddings and then interact with Qdrant via its API. Docker makes it easy to include Qdrant; if target load is moderate, a single container instance is sufficient for daily use [8].

- **Airtable Integration:** Airtable itself is a cloud service (no container), but we include it as a logical part of the architecture. The n8n workflows connect to Airtable's API to read/write data. No local container is needed; instead, we use Airtable API keys in n8n and the built-in Airtable nodes to push records. In deployment, ensure the host has internet access for n8n to reach Airtable's servers.

- **AI Models/Agents:** The heavy AI processing is done via API calls (for OpenAI GPT-4/5, Anthropic Claude, etc.) or via a local LLM container:

- *External AI APIs:* We utilize the **OpenAI API** (for ChatGPT-5 or GPT-4) and Anthropic's Claude API. These do not run in our Docker network but are accessed over the internet. Thus, no container for the model itself, but the logic to call them is within n8n (e.g. using HTTP Request nodes or a custom node).

- *Local LLM Container (if needed):* Optionally, to incorporate a local large language model, we might deploy an open-source model server. For instance, a container running a REST API around a local LLM (such as a LLaMA 2 variant via llama.cpp or a HuggingFace model server). This container would load the model weights and provide an endpoint for generating text or embeddings. It's only necessary if we need to avoid external API usage for some data due to sensitivity or cost. If included, n8n's agent routing (described later) will decide when to call this local service. The container would be on the same Docker network so n8n can call it at `http://local-llm:port/…`.

- **Utility Containers:** Additional supporting containers can be part of the stack:

- A container for **Chrome PDF generation** (though Puppeteer can handle PDF creation as well, so the scraping container might double for this purpose).

- A lightweight **SMTP server** for sending emails (if using an internal mail service instead of a third-party SMTP) – this could be included if needed for on-prem email notifications.
- Monitoring/Logging container: e.g. Filebeat/ELK if we decide to aggregate logs from n8n and other services for debugging. This is optional; n8n itself logs workflow executions, which can also be captured.

All these containers are orchestrated via Docker Compose for deployment simplicity. The Compose file defines the services (n8n, db, qdrant, scraper, etc.), their environment variables (e.g. API keys, DB credentials), volumes, and network. For example, the Compose ensures n8n can reach Postgres and Qdrant by naming those services and linking networks. The Windows host running Docker Desktop manages these Linux-based containers in the background, abstracting away OS differences [5].

This containerized setup allows the engine to be easily portable and reproducible. Developers can run the entire stack on a Windows laptop for testing, and it can equally be deployed to a cloud VM or server if needed by using the same Docker configuration. Scaling can be achieved by adjusting container resources or replicating services (for instance, if scraping load grows, one could scale out the scraping service containers behind a queue, though initially one instance is sufficient).

## AI Prompt Agents and LLM Orchestration

The automation engine leverages multiple **AI prompt agents** (LLMs) to perform semantic analysis, summarization, and reasoning tasks throughout the workflows. Rather than a single monolithic model call, we use specialized models (ChatGPT-5, Claude "Opus", and potentially local models) for different functions – this improves both performance and cost-efficiency. The integration of these agents is done through the OpenAI function-calling paradigm and n8n's workflow logic. Key aspects of the AI agent design:

- **Function-Calling & Tool Use:** We harness OpenAI's function-calling capabilities to let the AI agents dynamically fetch information and take actions [9]. Each agent is given a **toolkit** of functions it can call – for example, a function to query the vector database for similar text, a function to look up a competitor's record in our Postgres DB, or a function to initiate a web search (if ever needed). The prompt instructs the model that these functions are available and encourages it to use them to get accurate data instead of guessing. This structured approach injects predictability and reliability into the AI's output [10] [11]. In practice, when an agent gets a task (say, "Analyze Competitor X's challenges on Program Y and suggest talking points"), the conversation might go like:
- The agent receives the request along with function definitions (e.g. `search_vectors(query)`, `get_contract(program)`, etc.).
- It decides it needs more info on Program Y, so it outputs a function call like `get_contract("Program Y")`. The n8n workflow or custom code intercepts this, executes the actual function (e.g. queries Postgres or FPDS data), and returns the result to the agent.
- The agent incorporates that info into context and maybe then calls `search_vectors("Job descriptions Program Y")` to see if other companies have similar openings, etc. It can chain multiple function calls to gather all necessary data [12].
- Finally, with all info, it composes the answer: the program brief or call sheet content.

Using this **tool-using agent** approach means the AI's output is grounded in real data we provide, drastically reducing hallucinations and increasing relevancy. It also allows multi-step reasoning within one workflow

run – the model can effectively conduct research by calling our tools, which is then monitored and fed back (all within seconds).

- **Prompt Engineering:** Each AI agent prompt is carefully crafted to align with Prime's BD strategy and to elicit the desired output format. For example, the prompt template for the program brief generation agent might be: *"You are an AI business analyst assisting a federal contracting BD team. Using the data provided (below), generate a one-page brief with the following sections..."* followed by explicit sections (Overview, Pain Points, Recommended Talking Points, etc.). We provide the agent with structured context (via function calls or direct context injection from the DB/vector search) so that it has all relevant facts – contract values, dates, competitor's open roles count, etc. We also include instructions to follow any **embedded call flow** guidelines from Prime's playbook (e.g. always start the talking points by referencing mission impact, use a confident tone, etc.). By defining **structured function outputs and schemas**, we ensure the model's responses are well-organized. This harnesses deep function-calling prompts to get structured results that are easier to parse and turn into PDF or Airtable fields [13] [14].

- **Multi-Model Ensemble:** Different models are used for different tasks, playing to their strengths (see the **Agent Routing Plan** section for specifics). For instance:

- We use **ChatGPT-5** for the final composition of briefs and call sheets because of its strength in coherent, strategic language generation. It excels at following the detailed formatting and tone requirements for client-ready documents.
- We use **Claude (Anthropic)**, especially a version with extended context (codenamed "Opus"), for tasks like reading or summarizing large documents. Claude's 100k+ token context window is ideal if we want to feed in multiple lengthy job descriptions or an entire contract SOW and get a summary or extract commonalities. For example, if 10 job postings across 3 companies all seem related to a certain program, we could give Claude a concatenated list of those postings and ask it to summarize the common theme or requirements – something GPT-4 (8k or 32k context) might not handle in one go.
- The workflow might first invoke Claude to do heavy-duty summarization (due to its context size), then pass that summary to GPT-5 for final analysis and writing (for style and nuance).

- **Local LLMs** are integrated for specific utility tasks or sensitive data handling. For example, if we have proprietary text from an internal playbook or past performance that we don't want sent to external APIs, a fine-tuned local model could be used to analyze or paraphrase that content for inclusion. Local models could also handle quick classification tasks (like tagging whether a job description is a backfill or new role, based on subtle wording) without incurring API calls. While local models may be less capable in general, we can design prompts that leverage them for contained tasks where they perform reliably.

- **Retrieval-Augmented Generation (RAG):** Our agent design follows the RAG pattern – combining vector database retrieval with generation. Before an LLM agent answers a query (like generating a program brief), the workflow ensures relevant documents are retrieved (via semantic search on the vector DB) and provided either via function calls or direct context. This means the LLM is not generating answers purely from its trained memory, but with the latest factual data from our knowledge base [15]. For example, when preparing talking points, the agent will have in context the actual stats from FPDS (like "Contract X has $100M remaining and 2 years left") and recent job stats

("Competitor has 3 openings for software engineers unfilled for 4 months") so that the suggestions it gives are evidence-based. This significantly improves the credibility and usefulness of the output.

- **Error Handling and Oversight:** The workflows include checks on the AI outputs. If an agent's response fails to meet the expected format (say the function-calling agent returns an error or the content is missing a section), n8n can detect this (with validators or simple regex checks) and either retry with adjusted prompt or route to a fallback. We also log every prompt and response (possibly using a tool like PromptLayer or just to the database) so we can audit what the AI is doing. This is aligned with best practices of monitoring AI agent interactions for reliability [16] [17]. Over time, these logs help refine prompts and function definitions to continually improve the agent's performance.

In summary, the AI layer of this engine is **dynamic and tool-aware**. By assigning the right model to each job and giving the models the ability to call functions for data, we ensure that the automation's intelligence is both cutting-edge and grounded in reality. The next section details how we decide which model handles which task (Agent Routing Plan), to complete this intelligent orchestration.

## Vector Database and Semantic Matching Flow

Semantic matching is at the heart of cross-referencing competitor job postings with programs and other companies' postings. The engine uses a **vector database** to perform these matches efficiently:

- **Embeddings Generation:** When a new job description is scraped, the system generates a numeric embedding vector for the text. We use a high-dimensional embedding model (such as OpenAI's `text-embedding-ada-002` or an equivalent local model) that converts the job posting text into a vector that captures its semantic meaning [18]. Similarly, we pre-compute embeddings for other relevant text corpora:
- **Program descriptions:** e.g. summaries of DoD programs from public sources or the descriptions from contract award announcements.
- **Historical job postings:** past scraped JDs (so we can find if a new posting is similar to something seen before, which might indicate the same role or program).
- **Taxonomy terms:** any key phrases or concept definitions from Prime's playbooks (like definitions of roles or technologies) could also be embedded to aid matching (optional).

- **Vector Database (Qdrant or pgvector):** All these embeddings are stored in a vector store. If using **Qdrant**, we store each vector with metadata like `id`, `text_source` (job, program, etc.), `company` (for jobs), `program_name` (if known for a program description), etc. Qdrant allows us to filter search results by metadata at query time [1], which is extremely useful. For example:

- To find **job overlaps**, when processing a competitor's new job, we can query the vector DB for the nearest neighbors *excluding posts from the same company*. That filter (e.g. `company != "Competitor A"`) ensures we retrieve similar job postings by other companies [1]. If we find, say, two other vendors with very similar postings, it's a clue that multiple contractors are staffing the same program (or all chasing the same hard-to-fill role), which we flag as a competitive insight.
- To match a job to a **DoD program**, we query the nearest program description vectors. If, for instance, a job posting for a "Satellite Communications Engineer" at Competitor X has top vector hits

that include a program description for a certain Air Force SATCOM program, the system links that posting to the program with high confidence.

- If using **pgvector** (Postgres extension) instead: The approach is similar – we store vectors in a Postgres table with a `VECTOR` column and use SQL cosine distance queries or the `<->` operator to find nearest neighbors. We can still use metadata by including columns for company, etc., and filtering in the WHERE clause of the SQL query.

- **Semantic Search Workflow:** Within n8n, after inserting a new job's embedding, a search step runs:

- **Job-to-Program Search:** The vector DB is queried for the closest program description vectors to the job posting vector. We retrieve the top N results (e.g. top 5 programs). If one is significantly closer than others (above a similarity threshold), we mark that as the likely program context. If multiple are close, we might keep a few and allow the AI agent to consider them all when writing briefs.
- **Job-to-Job Search:** The DB is also queried for similar job postings from other companies. We might retrieve top matches and see if those matches share a program tag or prime. This can reveal if, say, Competitor Y had a nearly identical job posting last month – indicating they're both for the same contract. The engine can cluster postings by similarity; if a new posting is very similar to an existing cluster associated with Program Z, we infer it's also for Program Z.

- **Program-to-Job Search:** Conversely, when analyzing a known program (from the contract data side), we can query for all job postings that semantically match that program's description. This gives a picture of how many and which companies are hiring for that program, and what roles are in demand. This info feeds into the BD strategy (e.g. if Program Z has 15 open reqs across 4 contractors, it suggests big staffing needs and possibly delivery risk on that program).

- **Metadata Utilization:** Each vector point's metadata payload helps refine results. For instance, if we're matching by program name or keyword rule already, we might boost those results. Or we might use a combination of filters: *find similar text AND filter where job title contains "Engineer" if the target posting is an engineer role*, etc. The vector DB supports such complex filtering built into the similarity search for efficiency [1].

- **Clustering and Trend Analysis:** Beyond one-to-one searches, the engine can perform periodic clustering on the vector data:

- Using techniques like K-means or HDBSCAN on the job posting vectors to identify clusters of related roles. Often, a cluster might correspond to a program or a technical domain. We then label clusters if possible (e.g. cluster of vectors all align with "Cloud modernization program").
- We track cluster sizes over time – if a cluster (program) suddenly sees more and more postings, that indicates ramp-up (or trouble staffing) on that program.

- This unsupervised analysis acts as a "Radar" for emerging areas of opportunity that may not be explicitly named.

- **Integration with AI Agents:** The results from vector searches are passed into the AI workflows:

- When the AI agent is formulating a program brief, we include snippets from the most similar documents (job postings or program descriptions) as reference context. For example, an agent

might see: *"Context: Competitor X job posting excerpt – '...experience with Platform ABC and active TS/SCI clearance..."'* which might be a clue linking to a specific program. The agent uses that to enrich the narrative ("The need for multiple TS/SCI-cleared engineers suggests Program ABC is expanding...").

- If using function-calling, we literally allow the agent to invoke a `search_vectors` function. The function returns the top matches with brief info, which the agent can incorporate. This way, the **LLM dynamically queries the vector DB** as needed during its reasoning process, rather than only relying on what we pre-fed. This dynamic querying is powerful, as the model might ask for clarification (e.g., *Agent thinking:* "I see open roles for cyber engineers; let me search the vector DB for similar cyber roles to see if it's widespread." It calls `search_vectors("cyber engineer Program Y")` and gets results which it then uses to conclude something).

- **Continuous Learning:** As more data accumulates (more job postings over weeks/months), the vector database becomes richer. The engine could periodically retrain or update embedding logic (for example, if a domain-specific embedding model becomes available, it could re-embed all texts for better accuracy). All new data is indexed, so the semantic searches improve over time. We also might integrate feedback: if a user corrects a program match (says "No, this job is actually for Program Q, not Program P"), we can store that and in future adjust the matching (perhaps even fine-tune an internal classifier or add that example to a training set for a model).

Using this vector-based semantic layer implements the "brain" of the system that **understands relationships** beyond simple keyword overlap. It's a practical application of retrieval-augmented AI: using vector search to supply relevant context to our LLMs and logic. This ensures that, for instance, even if a competitor's job post is vague (no direct program name), our system can still connect the dots via similarity to known programs or other posts – giving Prime Technical Services a significant intelligence advantage.

## Output Modules and Delivery Channels

The ultimate purpose of this engine is to deliver actionable insights to the business development (BD) team in formats they can readily use. The system supports multiple output modules to meet different user needs: structured data for analysis, real-time alerts for urgency, and polished briefs for deep dives. Below are the output channels and their implementation:

- **Airtable Dashboards:** The engine pushes key results into **Airtable**, creating a living dashboard of opportunities. We design an Airtable base with relevant tables (see Database Schema section) – for example, a table for "Active Program Opportunities". Each record might represent a program (or contract) that has notable competitor staffing activity. Fields include the prime contractor, list of competitors involved, count of open roles (and aging of those roles), our calculated BD score, next recompete date, etc. Another table could list "Scraped Jobs" with details and a link to the program/opportunity table via a relationship. The n8n workflow uses the Airtable API (with an API key) to **upsert** records daily – if a program already exists, update its fields; if a new potential program appears from today's scrapes, create a new record. Views are configured in Airtable to help the team focus, for example:
- A view filtering opportunities with BD Score above a threshold, labeled "High Priority".
- A calendar view of upcoming contract end dates or expected solicitations.
- Perhaps a Kanban or status view if the team wants to track progress (e.g. "Researching", "In Contact", "Proposal Stage" for each opportunity – these could be manual fields the team updates).

Using Airtable gives a no-code front-end for non-technical team members to explore and even comment on the data. They can add manual notes (which we preserve since automation only updates designated fields). This bridges the AI output with human input. *Example:* the system adds an opportunity "Program Falcon" with Score 85. A BD manager opens Airtable, sees it, and adds a note "Reached out to Col. Smith's office, awaiting feedback" – that note remains attached. The next day, the automation updates open roles count and score, but does not overwrite the note.

- **CSV/Excel Exports:** For users who prefer working with spreadsheets or for archival, the system can output CSV files. An n8n workflow can compile a CSV of the day's significant data – e.g. a list of all open competitor job reqs with columns for company, title, program (if matched), days open, etc. Or a summary CSV of opportunities with their scores. These CSVs can be generated by n8n's **Spreadsheet node** or by custom script, and then automatically emailed out or saved to a shared drive (or even attached to a Slack message). The blueprint includes an option to run a weekly full export (say every Friday) that compiles all data for that week, giving a snapshot that can be opened in Excel for any ad-hoc analysis by analysts. The CSV output ensures compatibility with other systems too – e.g. if the team wants to pull the data into a Power BI or Tableau, they can consume the CSV or connect directly to the Postgres DB.

- **Slack Notifications:** Quick dissemination of critical insights is done via Slack. We integrate with Slack using either n8n's Slack node or incoming webhooks. Each day after analysis, the engine sends a message to a designated channel (e.g. #bd-intel). The message is concise and prioritized:

- It might start with a headline like: "**Daily GovCon Intel Update:** 3 High-Priority Opportunities Identified".
- Then bullet points for each, e.g.: *Program Falcon (Air Force) – Prime: XYZ Corp – 5 open cleared positions for 90+ days, Likely Pain Point (score 88)*; *Program Orion (Army) – Prime: CompetitorA – new hiring surge (10 postings) indicates ramp-up, recompete in 6 months (score 75)*, etc.
- Each bullet can include a link to more info: perhaps a deep link to the Airtable record or a SharePoint where the PDF brief is stored, or even triggering an n8n webhook for a "slash command" that returns details.

Additionally, Slack can be used for real-time alerts outside the daily cycle. For instance, if during scraping we find a **critical event** (maybe a competitor just posted a very senior position unexpectedly – indicating someone senior left, or a key role that's a linchpin to a project), the workflow can send an immediate Slack alert like ":rotating_light: *Urgent:* Competitor X posted a new Chief Architect role on Program Y. This could signal instability." This way, the BD team can react quickly, not just wait for the next daily summary.

- **Email Reports:** Some stakeholders might prefer email (or may not be checking Slack/Airtable). The system can send out an email report daily or weekly. Using n8n's Email node (configured with SMTP or a service like SendGrid), it can send a nicely formatted email (HTML content) that mirrors the Slack summary but with more detail. It can embed tables for top opportunities, and attach the PDF briefs as attachments. For example, each morning an email goes out to the capture team with subject "Daily BD Intelligence Brief – [Date]" and includes a summary and attachments for each program brief prepared. The email can be configured to only go to certain people (maybe high-level execs get a weekly digest instead of daily).

- **PDF Program Briefs:** One of the most important deliverables is the in-depth **program brief** PDF for each identified opportunity. These PDFs are generated automatically by the workflow:

- We use a template (perhaps an HTML/CSS template or a Markdown to PDF pipeline). The AI agent's output for the brief is structured (with headings, bullet points, etc. as per our prompt). We can feed that into an HTML template that adds branding (Prime Technical Services logo, header/footer) and nice formatting. Then using Puppeteer (the same headless Chrome can be used) or a library like pdfkit, we render that HTML to a PDF file. Puppeteer's print to PDF in headless Chrome yields a nicely formatted document with minimal hassle, so we likely use the scraping container to perform this conversion (since it already has Chrome).
- Each PDF includes sections such as: **Program Overview**, **Opportunity Analysis** (staffing situation, competitor positions), **Contract Details** (funding, timeline), **Recommended Action** (talking points for BD, suggested "wedge" strategy). We ensure any data points are up-to-date by regenerating the content each time (or updating the template fields).

- The PDFs are then delivered via Slack (could upload the PDF to Slack with an API call – Slack will show an inline preview), and/or via email attachments. We might also keep an archive of PDFs on a file server or Google Drive for reference. The file naming convention could be like "ProgramBrief_<ProgramName>_<Date>.pdf" for easy retrieval.

- **Business Intelligence Integration:** Although not explicitly asked, it's worth noting that because all data is in Postgres and Airtable, the company can use BI tools on it. We mention this because the CSV and direct DB access allow tools like Power BI, Tableau, or even custom dashboards to live-update with this data. In a deployment blueprint, we highlight that future integration with such tools is straightforward – the data model is accessible, so the team could create visualizations (e.g. a bar chart of open roles per program over time, etc.).

Each output module is designed with **readability and usability** in mind. The combination ensures that quick insights (Slack alerts) and deep insights (PDF briefs) are both provided. By automating the generation of these artifacts, we free the BD team from manual report creation, allowing them to spend time strategizing responses. Moreover, by delivering information in multiple channels, we meet people where they are – whether they live in email, Slack, or Airtable – increasing the chance that the intelligence actually gets acted upon.

Importantly, all outputs cite data sources or evidence where appropriate (e.g. the PDF might have footnotes like "Source: USAspending.gov" for a contract figure, or "Based on postings on clearancejobs.com" for staffing numbers), lending credibility when the team uses these in conversations. This way, Prime Technical Services can confidently leverage the outputs in meetings and calls, knowing they have the data to back up claims.

## Business Development Scoring Logic and Opportunity Ranking

To help the BD team prioritize where to focus, the engine assigns a **Business Development (BD) score** to each program opportunity. This scoring logic distills a variety of data points into a single indicator of how

ripe an opportunity is – essentially highlighting the ideal "wedge points" and timing windows to intervene. The scoring and logic work as follows:

- **Long-Open Roles (Labor Gaps):** One of the strongest signals in the score is how many positions a competitor has been unable to fill and for how long. For each program (or for each competitor on a program), the engine calculates metrics like:
- Number of job postings still open beyond a threshold (e.g. >30 days, >60 days).
- The average age of open reqs (how long since posted).
- Key roles open – especially if critical positions (e.g. lead engineers, project managers) are open for a long time. These contribute more to the score because a languishing key role = pain for the project.

- We weight these such that, for example, 1 role open for 90 days might add more points than 3 roles open for 10 days each, since the former indicates a specific chronic gap.

- **Backfill Indicators:** The system looks for signs that postings are backfills (replacements) rather than new positions:

- Clues such as job descriptions that mention "seeking replacement" or that appear identical to a filled role from a few months ago.
- If a job was posted, taken down, then re-posted with a slightly different title – implies the first hire didn't work out or left. This counts as a churn signal.

- Each backfill signal will increase the score because turnover on a program can signal dissatisfaction or instability, which a new vendor could exploit by offering stability.

- **Multiple Competitors Hiring for Same Program:** If our cross-referencing finds that several subcontractors (or even the prime and a sub) are all hiring for similar roles on the same program, that program likely has a widespread talent shortage. The score for that program is boosted because it suggests the prime might be struggling to staff (if even subs are trying and failing) and the government customer might be feeling the impact. A concentrated cluster of open reqs across companies is a red flag/opportunity.

- **Contract Timeline (Recompete Window):** The engine uses data from FPDS/USAspending about contract end dates:

- If a program's contract is entering the final year or due for recompete soon, its score is elevated. This is a window where agencies consider their options and primes either scramble to staff up to look good or potentially drop the ball – an opening for competitors to propose alternatives.
- Conversely, if a program was just awarded or has many years left, the score might be slightly lower *unless* there are severe staffing issues. (However, even a new contract with immediate staffing failures is an opportunity – possibly for a takeover or intervention).

- We incorporate a timeline factor: e.g. if <12 months to recompete, +10 points; 12–24 months, smaller boost; >24, no boost (or even slight penalty, because it's farther out).

- **Agency Signals and Past Performance:** The system tracks any publicly available indicators of agency dissatisfaction or program trouble:

- For example, GAO reports or IG reports mentioning the program (if any exist), or news articles about delays.
- If the engine had access to CPARS or other performance data (perhaps not directly, but sometimes issues are hinted in press releases), those would be factored.
- Since direct integration of such signals is complex, we approximate it: a high number of long-open jobs itself is a proxy for performance issues. Additionally, if a contract mod is issued for additional funds due to "unexpected costs" or schedule slips (which might be gleaned from mod descriptions), that hints the current team might be underperforming.

- In the absence of explicit external signals, we rely on our internal data (the job postings, etc.) as the primary indicator of pain.

- **Contract Size and Importance:** High-dollar, labor-heavy programs that have lots of personnel (and thus many potential openings) get higher baseline scores because they represent bigger opportunities for Prime Technical Services if a wedge is found. We incorporate:

- Total contract value and number of FTE positions (if known) from the contract data.
- If a program has, say, 100 FTEs and we see 10 open positions, that's 10% of the workforce – which is significant. The scoring logic accounts for that proportion.

- If the program is smaller (maybe 5 FTEs total) and 1 is open, that's also significant proportionally, but overall impact is smaller due to size – however, small programs can still be strategic if they open a door to an agency, so we don't ignore them, just score appropriately.

- **Scoring Formula:** We combine the above factors into a weighted formula. For example:

- Score = (A * # of long-open critical roles) + (B * # of long-open non-critical roles) + (C * backfill indicator count) + (D * multi-company overlap factor) + (E * contract nearing end factor) + (F * contract size factor) + (G * any manual priority flag).
- The coefficients A, B, C… are tuned based on our priorities. For instance, we might set A (critical role weight) quite high. E (nearing end) also high because a recompete soon means immediate action possible.
- The formula output is normalized or capped to, say, 0–100 scale for simplicity.

- If using an AI to assist, we could have the LLM review the computed factors and adjust qualitatively. For example, an LLM could be given a summary: "Program Falcon: 5 roles open >60d (including 1 PM, 2 Engineers), contract ends in 8 months, multiple companies hiring" and asked to output a risk/opportunity rating "High/Med/Low" or adjust the numeric score. However, a rule-based approach combined with human review is often sufficient for scoring.

- **BD Wedge Opportunity Classification:** In addition to a numeric score, the engine can label opportunities with categories:

- **"High Pain – Immediate Wedge"** for those with very high scores where the current contractor clearly has an issue (e.g. critical roles unfilled for long durations). These are ideal for immediate BD calls, perhaps offering staff augmentation or emphasizing how Prime Technical can step in quickly.

- **"Monitor – Potential Opening"** for moderate scores, maybe a few issues but contract end is farther or only a couple roles open. These might be ones to keep on the radar but not invest heavily yet – unless further signals appear.
- **"Low Priority"** for those with minimal issues. Perhaps included in reports for completeness but not a focus.

The AI can help generate a short rationale for each label: e.g. *"High Pain – Immediate Wedge: Competitor appears unable to fill key roles (PM and Lead Dev unfilled 3+ months) on Program X, likely impacting deliverables. With 6 months to contract end, client may be seeking alternatives – prime opportunity for intervention."*

- **Aligning with BD Strategy:** The actual weights and threshold for "high priority" vs "low" can be aligned with Prime's internal strategy. If Prime Technical has certain strategic targets (e.g. Army programs are more desirable than Navy for their portfolio), we could introduce a factor for agency or domain to bump scores for strategic alignment. Similarly, if the company has **embedded call flows** or specific "plays" (like a play for when an incumbent is failing vs a play for when an incumbent is leaving), the scoring could be the trigger for which playbook to activate. For instance, a very high score might automatically suggest using the "Incumbent Failure" play, whereas a moderate one might be "Soft Entry" play. These playbook triggers can be noted in the program brief output, ensuring the BD team knows which strategy to apply.

- **Human Oversight:** While the engine scores opportunities, we expect BD leadership will review and possibly adjust priorities. The Airtable view could allow a user to override or tweak a score or mark something as a false alarm. The system can be configured to accept such inputs – for example, an "Ignore" flag on an Airtable record that the automation will respect (not keep flagging it daily if the team has decided it's not actually an opportunity). Our automation should be smart but also **collaborative with human judgment**.

The BD scoring logic thus acts as a compass, pointing the team to where the combination of factors suggests a competitor's weakness or a timing advantage. It encapsulates the concept of **finding the wedge** – i.e., the gap through which Prime Technical Services can insert itself to win business – and quantifies it. By having this logic automated, it ensures no opportunity is overlooked in the flood of data and that each morning the team has a clear picture of what to pounce on.

## Daily Schedule and Automation Timing

The engine operates on a **daily cycle** with a carefully planned schedule to ensure data is fresh at the start of each work day. All times can be adjusted, but here's a blueprint schedule in **Eastern Time** (assuming the team is in America/New_York, e.g. Atlanta HQ):

- **02:00 AM – Data Update (Contracts & Static Data):** In the early morning hours, a workflow triggers to update relatively static or slow-changing data:
- This includes pulling the latest **USAspending/FPDS** data for contracts (if not done continuously). Running this at 2 AM ensures we catch any overnight database refreshes on those government systems. The workflow fetches new award data or modifications for the programs we care about and updates the Postgres DB. It also can refresh any reference tables (like a list of primes, agency org info, etc.).

- This time is chosen when web traffic is low, ensuring we don't hit API rate limits during peak times. It typically finishes quickly (within minutes, depending on data volume).

- **03:00 AM – Competitor Job Scraping:** The main scraping workflow kicks off in the pre-dawn hours:

- One by one (or a few in parallel), n8n triggers the Puppeteer/Playwright scraper container to go through each competitor site or job board. For example, scrape Competitor A's careers page, then Competitor B, etc. If there are many sources, we might stagger or parallelize to finish by a certain time.
- Scraping at 3 AM has advantages: minimal chance of interfering with anyone (though these are public sites, we still prefer off-hours to be polite on traffic), and it gives time to recover from any failures. If a site was temporarily down, n8n can catch it and retry after some minutes.

- The output of scraping (raw job postings) is stored immediately to the database and/or kept in memory for the next step. This phase might take from a few minutes up to an hour or more depending on number of competitors and site slowness. Let's assume by 4:00 AM it's complete.

- **04:00 AM – Semantic Processing & Enrichment:** Right after scraping, the cross-reference workflow starts:

- Newly scraped jobs are run through the **embedding generation** and vector searches. This likely completes quickly (generating an embedding via API is sub-second per job; vector DB search is milliseconds). Even if hundreds of jobs, this is typically a few minutes. By 4:15 AM, each new posting is annotated with potential program matches and overlaps.
- Next, the LinkedIn/ZoomInfo enrichment kicks in for any new companies or updates. For example, if we haven't pulled Competitor A's LinkedIn info in a while, or if we see Competitor C (new entrant) in postings, we call the LinkedIn API or scraper. We might also quickly check if any company's headcount changed significantly (as a proxy for something). This might run till ~4:30 AM.

- Also during this window, any **late-night Slack alerts** can be triggered if something critical was found. However, since it's 4 AM, we might queue those for the morning unless it's something extremely urgent for a 24/7 op.

- **05:00 AM – Analysis & Scoring (AI Agents):** With all data prepared, the AI-driven analysis begins around 5 AM:

- The system activates the **analysis & briefing workflow**. This may involve multiple AI calls and thus might be the slowest component time-wise (depending on model speed and how much content it generates). We use parallelism here if possible: e.g. generate briefs for different programs in parallel threads, or at least sequentially but each taking maybe 1-2 minutes with GPT-5.
- Suppose we have 5 high-interest programs to generate briefs for; each might take ~30 seconds to 1 minute of API time. This wraps up by ~5:15-5:30 AM.
- The BD scoring logic runs here as well, calculating scores for each program now that we have final tallies of open roles etc. Scoring is computationally light (sub-minute).

- By 5:30 AM, we have updated insights: briefs ready, scores computed, and all data prepared for output.

- **05:30 AM – Report Assembly:** Now the workflow compiles outputs:

- Generate PDFs from the AI-written briefs. Using Puppeteer to render PDFs might take ~5-10 seconds per brief. Let's say by 5:40 AM, all PDFs are saved.
- Create or update Airtable records with the new data (this is quick, maybe a few seconds per record via API). Airtable views are now up-to-date.

- Prepare the Slack message content and email content. Because we prefer the team to see info at start of work (~8-9 AM), we can either send now (5:45 AM) and it will be waiting in their inbox/channel when they wake up, or we could schedule delivery closer to 7 or 8 AM. If using Slack scheduled send or having the bot wait, either approach is fine. For simplicity, we send immediately after generation – the messages will be in Slack ready to read.

- **06:00 AM – Notification Delivery:** By 6:00 AM, the engine sends out:

- Slack daily digest to the channel.
- Emails with attached briefs to the designated recipients.

- Any other channels (e.g. update a SharePoint or internal portal if needed). This timing ensures that by the time team members start their day (often around 7-8 AM in defense contracting world), they have the latest intel available.

- **Throughout the Day:**

- **On-Demand Queries:** The BD team might have questions during the day. For example, "Did Competitor X ever fill that role from last month?" or "Show me all open roles on Program Y right now." While the daily run provides a static report, we can also expose an interface (maybe via an n8n webhook or simple chatbot) to answer such questions on demand. For instance, a slash command in Slack like `/intel program Y` could trigger n8n to fetch the latest from the DB and the vector search and reply with an answer. This is not a scheduled item, but a dynamic use of the system during business hours.
- **Mini-updates:** We could schedule a quick mid-day check (e.g. 12:00 PM) to see if any new jobs were posted since morning (some companies might post in the morning). If so, we could optionally scrape just those or at least flag them. However, since most cleared job boards don't update multiple times per day, this might be unnecessary. If the user group desires, a lighter noon run could be set to catch anything new and send a small update ("Noon update: 1 new posting from Competitor B (added to Airtable)").

- **Human Feedback Integration:** Also during the day, team members might update Airtable (e.g. mark an opportunity as pursued). An n8n workflow could listen (Airtable trigger) and perhaps log that or adjust things. This ensures the system stays in sync with human actions.

- **End of Day Wrap (Optional):** At 6:00 PM, we might have an optional workflow that archives the day's findings or sends a summary if needed. But likely the morning cycle is sufficient. The data remains in the system for reference any time.

- **Weekly/Monthly Routines:** On Friday evenings or weekends, we might run maintenance:

- E.g. a Sunday 2 AM job to re-index the vector DB (purge old entries that are no longer relevant or re-embed if we changed our embedding model version).
- A Monday 5 AM expanded report that includes weekly trends (maybe charts or comparisons to last week's metrics).
- These can be included as needed, following the same pattern of scheduling in n8n's cron nodes.

The schedule above ensures a **pipeline flow**: data collection → processing → analysis → dissemination, all before the workday starts. We've built in a buffer in the early hours so that if something overruns (say one site was slow at 3 AM and scraping took longer), there's cushion time to still finish by ~6-7 AM. N8n's **timezone** is configured so that scheduled nodes run at the correct local time [19]; we set the Docker container TZ to Eastern Time so that daily triggers align with our desired schedule.

In case of failures, say the scraping for one competitor completely failed at 3 AM (site down or changed HTML), the system will still proceed with what data it has and flag that competitor for manual check. An alert can be sent to the data engineer if a scrape fails. The robust scheduling and separation of steps help isolate issues (e.g. a failure in LinkedIn enrichment at 4:15 won't halt the whole process; it can time out and the rest still continues, perhaps with a warning).

Overall, this daily cadence makes the engine a **predictable routine** in the BD workflow: every day starts with fresh intel. The team can rely on it like a morning briefing newspaper, except highly tailored to their competitive landscape.

## Data Storage and Database Schema

A well-designed database schema underpins the engine, enabling both the automation and the analytical queries. We utilize both **PostgreSQL** (for structured relational data and persistence) and **Airtable** (for a user-friendly overlay and minor collaboration data). Below is the schema blueprint and how each part is used:

**PostgreSQL Schema:**

We organize the Postgres database into a few main tables and relationships:

- **CompetitorJobs Table** – stores every scraped job posting (historical and current):
- `job_id` (PK) – unique identifier (could be a hash of the URL or an auto-increment).
- `company_name` – the competitor company (e.g. "Competitor X Inc.").
- `job_title` – title of the position.
- `clearance_level` – e.g. TS/SCI, Secret, etc., if parsed from description.
- `location` – location of job (if available).
- `posted_date` – date first seen posted.
- `last_seen_date` – date we last confirmed it's still open (the scraper updates this each day it's still listed).
- `status` – open/closed (the engine can mark as closed if it disappears from site).
- `description` – full text of the job description (could be in a text column).
- `program_match` – (FK to Programs table) if we have matched this job to a specific program/contract.

- `prime_match` – (FK to Companies table or simply a text field) if we inferred who the prime contractor is for that role.
- `embedding` – vector column (if using pgvector) holding the embedding of the description for semantic search.
- Indexes: we index `company_name` for filtering by competitor, and possibly a GIN index on `embedding` for vector similarity search if pgvector.

*Usage:* This table allows analysis of individual postings. We can query, for example, all jobs by Competitor A that have been open > 60 days (using `posted_date` and `last_seen_date`). It's also the source for the vector similarity searches. Over time it will accumulate historical data (which is fine; we might archive old ones beyond a certain age if needed).

- **Programs Table** – represents DoD programs/contracts of interest:
- `program_id` (PK) – unique ID for the program or contract (could be a contract number or a slug).
- `program_name` – human name (e.g. "Army ABC Modernization Program").
- `agency` – the government agency or branch (Army, Air Force, etc).
- `prime_contractor` – main awardee (company name, FK to Companies table).
- `contract_value` – total award value (from FPDS).
- `period_start` – contract start date.
- `period_end` – contract end date (current period).
- `contract_status` – e.g. "Active", "Recompete Pending", "Expired".
- `description` – summary of the program (could be compiled from public sources or FPDS descriptions).
- `labor_count` – number of FTEs or positions (if known or estimated).
- `vector` – embedding of program description (if we vectorize program summaries).

*Usage:* This table holds the reference info for programs. When the AI writes briefs, it will pull data from here (like period_end for timeline, etc.). It's also linked to jobs to see which jobs map to which program. If multiple program_ids exist for the same overall program (like separate task orders), we might either treat them separately or have a field linking them; for simplicity, assume each major contract or task order is a "program" record.

- **Companies Table** – list of companies (competitors and possibly primes):
- `company_id` (PK).
- `company_name`.
- `is_prime_contractor` – boolean or perhaps a category ("Prime", "Sub", "Both") to indicate if this company primes any contracts or is known mostly as sub.
- `org_size` – maybe number of employees (from LinkedIn/ZoomInfo).
- `cleared_headcount` – if we have an estimate of how many cleared staff they have (maybe from job counts).
- `last_linkedin_update` – timestamp when we last pulled their LinkedIn info.
- Other fields: industry focus, key contacts (if we store any contact info, though that might be separate to avoid PII).

*Usage:* Helps enrich data about companies. We join with CompetitorJobs to filter by company attributes if needed. Also used for primes – e.g. Programs.prime_contractor links to Companies to get details.

- **Opportunities/Insights Table** – this is a denormalized table that might store the output of analysis for each "opportunity" (i.e., each program from the perspective of a wedge opportunity):
- `program_id` (FK to Programs).
- `bd_score` – numeric score calculated.
- `priority_label` – text like "High", "Medium", etc.
- `open_roles` – number of open roles (maybe total across all companies) on that program.
- `open_roles_details` – could be JSON or a text summary (like "5 engineers (3 CompetitorA, 2 CompetitorB), 1 PM (CompetitorA)" etc.).
- `pain_points` – text field summarizing the pain point (could be filled by AI or template like "Key roles unfilled for X months").
- `wedge_recommendation` – text field with the recommended approach (a succinct version of talking points).
- `last_update` – timestamp of last update.

*Usage:* This table basically materializes what goes into the PDF briefs in a structured form. It's what we would sync to Airtable for the main dashboard. The reason to have this separate from Programs is to keep the raw contract info separate from dynamic analysis. One program (contract) might not always be an "opportunity" if no issues; we might only create an Opportunities record when certain criteria are met or just create for all and mark low priority accordingly. Storing the score and analysis separately also allows easy tracking of changes (e.g. if we keep historical scores in another table or archive old records for trend analysis).

- **Embedding Vector Store:** If using Qdrant instead of pgvector, the vectors aren't in Postgres but in Qdrant's store. In that case, we maintain some consistent IDs:
- For each CompetitorJobs row inserted, we also insert a point in Qdrant with an ID referencing the job_id and metadata (company, etc.).
- For each Programs record, similarly a vector point with program_id metadata.

- Qdrant's state will mirror these tables. We don't need an extra table in Postgres for vectors, but we may have a small table to track if vector sync is done, etc. (e.g. `job_id, vector_added (bool)` if needed to handle asynchronous embedding jobs).

- **Audit/Logs (optional):** We could have tables like `scrape_log` or `ai_calls_log` to record events. For example, each run of scraping, how many jobs found, etc., or each AI call's result and tokens used. These are not strictly necessary but can help monitor the system's operations over time.

**Airtable Schema:**

In Airtable, we mirror some of the Postgres data for accessibility: - **Opportunities (Airtable)** – corresponds to the Opportunities/Insights table. Fields: Program Name, Prime, # of Open Roles, BD Score, Priority, Last Updated, Notes. We allow editing of "Notes" or "Status" in Airtable by users, which do not get overwritten by automation (except we could sync status back if needed). - **Jobs (Airtable)** – we might not want to sync every job to Airtable (could be too granular), but we might have a linked field showing, for a given Program

record, a list of current open roles (maybe just title and company). Airtable supports linked records, but since we aren't manually maintaining it, an alternative is to store a concatenated summary. However, Airtable could handle a few hundred records easily, so we could put all open jobs in a table: fields: Title, Company, Program (link to Program table in Airtable), Age (formula from dates), etc. The Program table could then roll-up counts easily. - **Companies (Airtable)** – possibly a table for companies if we want a directory, but not necessary if primarily focusing on programs. It could show company size, etc. - **Schedule/ Calls (optional)** – maybe a table to log BD call attempts or outcomes, if they want to track within Airtable. That would be entirely manual or updated by BD folks, not by the automation (except maybe creating a stub entry when something becomes High priority, to prompt them to fill it in).

**Data Relationships:** - In Postgres: CompetitorJobs -> Programs (many-to-one, many jobs map to one program). Companies -> CompetitorJobs (one-to-many). Programs -> Companies (many-to-one for prime). Programs -> Opportunities (one-to-one possibly, or one-to-one for active opportunity). - In Airtable: We'll likely have Program (Opportunity) table linked to a Jobs table if we import jobs as well. Or we could skip linking and just use summary fields.

**Database Usage in Workflows:** - The scraping workflow writes to `CompetitorJobs`. If a job_id already exists (we encountered it before), we update `last_seen_date`. If it's new, we insert a new row. - After analysis, when we identify program matches, we might update the `program_match` field on CompetitorJobs for those new jobs. - The FPDS data update writes to `Programs` – if a contract we're tracking had a mod, update its values (or insert if new program emerges). - The scoring workflow writes to `Opportunities` – upserting the latest score and analysis text. - When generating outputs, the workflow reads from `Opportunities` to get the latest scores and from `Programs` for contract details, etc. It may join with `CompetitorJobs` to get detailed counts or examples for inclusion in the brief (like listing the specific roles open). - Airtable sync: We can either push directly from the in-memory data, or do a quick query from Postgres to gather what to send. Likely we push directly as we compute (for instance, after computing a program's score and summary, call Airtable API to update that record).

**Example Data Flow:** A new job "Senior Network Engineer" at Competitor X is scraped: - Insert into `CompetitorJobs` with job details. - We run semantic match and find it likely links to Program Alpha (id 123). We update that row's `program_match = 123`. - In Program 123, which corresponds to "Program Alpha" in Programs table, we update maybe a field like `open_positions_count` (not listed above, but could be computed on the fly instead of stored). Or just know that when scoring. - We calculate score for Program Alpha considering now maybe 3 open roles including this one. Update `Opportunities` record for Program Alpha with new score and insights. - That triggers updating Airtable's Program Alpha record: new open role count = 3, etc., and maybe add "Senior Network Engineer – Competitor X – open X days" to a linked list or notes. - The AI brief for Program Alpha is generated using info from both `Programs` (contract info) and `CompetitorJobs` (the open roles details). It doesn't need everything in the DB, some parts it already got during processing (like we might have assembled a context that lists all open roles which the AI then worded nicely).

**Note on PII and Sensitive Info:** We likely do not store individual person data (like candidate info or specific contacts) to avoid privacy issues. All data is about job posts and companies/programs which is fair game and not personally identifiable beyond perhaps a hiring manager's name if listed (we'd probably ignore or strip that). So our schema avoids PII concerns.

**Performance Considerations:** Postgres can easily handle the volume of jobs (even if it's thousands of postings over time). We will optimize with indexes, e.g. index on `(program_match, status)` to quickly find open jobs per program. If using pgvector, indexing the vector with an approximate search index can speed up semantic queries. Qdrant is built for vector queries so performance is good there, and it can handle many millions of vectors if needed (we're far from that scale likely).

**Backup and Retention:** Because we're running on a Dockerized DB, we set up regular backups (maybe a nightly dump via cron, or use n8n to trigger a `pg_dump` and upload to cloud storage). We also may decide to prune very old data – e.g. jobs that were closed 2 years ago might be archived to a separate table or not kept in Airtable, etc., to keep the working set clean.

In summary, the schema captures: - The raw data of job postings (CompetitorJobs). - Reference data about programs and companies (Programs, Companies). - The analytical output (Opportunities with scores). This separation of concerns makes the system maintainable and the data easily queryable for different purposes. The Airtable mirror of key tables ensures the end users interact mostly with high-level information and not raw tables, while under the hood the Postgres keeps everything consistent and is the source of truth for the automation.

## AI Agent Routing Plan (Model Allocation Strategy)

In this automation engine, not all AI tasks are created equal – we route tasks to different AI models (agents) based on their strengths, context length needs, and cost considerations. This **agent routing plan** ensures the right model is used for the right job, optimizing performance and efficiency:

1. **ChatGPT-5 (OpenAI)** – *Primary Reasoning and Composition Agent*
   **Uses:** All high-level analysis and final content generation. When a polished, well-structured output is needed (like the program brief or call sheet), ChatGPT-5 is invoked. It excels at understanding complex instructions and producing coherent narratives or strategies. We also use it for any reasoning that requires a broad knowledge base (e.g., interpreting the implications of a set of data points, since it can draw on its training which includes vast info – while ensuring we ground it with facts).
   **Why:** We presume "ChatGPT-5" is the latest with top-notch capability, slightly surpassing GPT-4. It offers reliable function calling and high-quality language output. We use it especially when output length is moderate (a few pages of text at most, well within its context window) and where nuanced tone (persuasive BD tone) matters.
   **Example:** Generating a "Program Falcon brief" – the workflow calls GPT-5 with a structured prompt and relevant data. GPT-5 returns a well-written report with sections as requested. Another example: an ad-hoc question like "Summarize the differences between Competitor A and B's approach on Program X" – GPT-5 would handle that reasoning and comparison.

2. **Claude "Opus" (Anthropic Claude)** – *Large Context Summarizer and Analyzer*
   **Uses:** Tasks involving very large amounts of text or the need to consider many data points at once. Claude is known (by 2025) for its 100k token context window (Claude 2). We utilize Claude for ingesting multiple long job descriptions or contract documents in one go, or for clustering analysis.
   **Why:** Claude's ability to handle lengthy input without losing coherence makes it ideal to synthesize, for example, "Here are 20 job descriptions – what patterns do you see?" or to read a long contract performance report. It may not always produce as refined output as GPT, but for extraction and

summarization it's excellent. Also, Claude has a tendency to be very good at structured outputs and can sometimes be more factual (depending on prompt) for summarization.
**Example:** We might feed Claude all open job descriptions for Program Y across 3 companies and ask it for a summary of the skill needs and any indication of issues (like "all require 10+ years experience – implies senior talent needed"). Claude returns a summary which we then feed to GPT-5 or use directly in analysis. Another example: reading a long GAO report excerpt about a program and condensing it into key points for our team – Claude can do that in one shot.

3. **Local LLM (e.g. Fine-tuned Llama2)** – *Sensitive Data and Utility Tasks Agent*
   **Uses:** When we have data that we prefer not to send to external APIs (for confidentiality) or repetitive classification tasks that are cheaper to handle locally. Also as a backup if API limits are hit. For instance, Prime's internal **cleared labor capture playbook** text might be something we load into a local model so it can quote or use it without sharing externally. Or if we need to run through hundreds of job descriptions to classify their category (admin vs technical, etc.), a local model can do this without racking up token costs.
   **Why:** A fine-tuned local model (perhaps a 13B or 30B parameter model tuned on business text) can handle straightforward tasks and ensures data privacy for sensitive inputs. It's also always available (no reliance on external uptime) and can be run on local hardware if needed (with GPU support via Docker container).
   **Example:** Classifying each job posting as "Likely Backfill" vs "New Position" based on subtle textual cues. We can prompt a local model with the job description and some examples, and get a classification. Or generating variations of a cold-call script line using internal preferred language – a local model fine-tuned on corporate comms might do that well.

4. **OpenAI Function-Call Agent** – *Meta Agent Orchestrator*
   (This is more of a pattern than a separate model – it would still be GPT-4/5 under the hood, but worth noting.)
   **Uses:** We employ the OpenAI model in function-calling mode as an **agent orchestrator** for complex multi-step queries. Essentially, when a single prompt requires multiple steps (search this, then calculate that, then answer), we send it to a function-enabled GPT. This is how the tool usage described earlier is implemented. It's not a separate model, but it's a mode where GPT-4/5 acts as a *manager*, deciding which function to call and when.
   **Why:** This allows dynamic tool use within one conversation turn with GPT, rather than writing separate logic in n8n for every step. It simplifies the workflow and leverages the model's reasoning to know what it needs.
   **Example:** For generating the program brief, we give GPT-5 the option to call `lookup_contract_details(program_id)` or `query_vector_similarity(program_id)`, etc. GPT-5 might call those functions in sequence and then compile the brief. We route this kind of task to GPT-5 with function calling explicitly enabled. The "routing" in this case is internal to how we call GPT-5.

5. **Fallback / Safety Logic:**
   If one model fails or returns uncertain output, we have fallback routes. For example, if GPT-5 returns an incomplete answer (maybe hit a token limit or an error), the system could automatically retry with Claude or vice versa. Also, for any classification or generation that the local LLM does, we might have a verification by an API model for quality (depending on sensitivity – but if not sensitive, sending a double-check to GPT-4 can catch errors from the weaker local model).

**Agent Selection Criteria in Workflow:**

During the workflow, a decision node (or simply conditional logic in code) will decide which model to use: - **Input size check:** If the combined context we need to feed exceeds ~30k tokens, prefer Claude. If under that, GPT-5 is fine. For example, "Summarize these 50 pages of text" clearly goes to Claude. - **Task type check:** If the task is "creative or complex writing with strategy", lean GPT-5. If it's "extract structured info or list key points", Claude can be chosen. - **Data sensitivity check:** If input includes internal notes or anything we deem confidential, use local LLM (or possibly GPT-4 with Azure if we consider that more secure within tenant – but sticking to local for this plan). E.g., if incorporating internal strategy notes into a prompt, route to local. - **Cost/time check:** For bulk operations (like classifying hundreds of items), local might be slower per item but can be run in parallel on local hardware without cost. We consider volume: if it's a one-off summary, API is fine; if we need to do something for many records nightly, local might be chosen to save $. We budget API usage mainly for high-value outputs.

These rules can be implemented via tags in n8n: e.g. a function node that computes `model_choice` and then branches to different API call nodes.

**Examples of Routing in Action:** - The daily brief generation uses GPT-5 with function tools, as that's a complex, multi-step creative task. - The clustering of job posts by theme might be done by feeding text into Claude in one prompt (since it can handle a lot) rather than multiple GPT queries. - The identification of backfill jobs might be done by local LLM for each job, and if the local model is "not sure" (e.g. low confidence in answer), we could have GPT-5 double-check those specific cases.

**Future Adaptability:** As models evolve (say GPT-5 gets a 100k context too, or new open-source models become powerful), this routing plan can be updated. The system is designed such that the choice of model is abstracted per task. For instance, n8n could have environment configs like `USE_CLAUDE_FOR_LARGE_CONTEXT=true`, or simply we adjust conditions. So we're not hardcoding a single model everywhere, we keep it flexible.

By orchestrating multiple AI models in this way, we ensure **the best of all worlds**: the creativity and reliability of top-tier OpenAI models, the breadth of context from Anthropic, and the privacy and cost-control of local models. This collaborative ensemble of AI agents significantly enhances the engine's performance and trustworthiness, giving Prime Technical Services an edge in both intelligence quality and operational control.

---

**Sources:**


- n8n official documentation on Docker deployment and scheduling [6] [5]
- Qdrant integration for semantic search and metadata filtering [1] [8]
- AI automation best practices (Nate Herk's AI Automation Society masterclass) – emphasizing RAG, vector DB usage, and resilient workflows [15] [4]
- OpenAI function calling blog – highlights structured tool use and multi-step agent workflows [9] [10]
- USAspending.gov and FPDS as data sources for federal contract tracking [2] [3]


---

1 8 18 Using Qdrant as a vector database for OpenAI embeddings
https://cookbook.openai.com/examples/vector_databases/qdrant/getting_started_with_qdrant_and_openai

2 USAspending: Government Spending Open Data
https://www.usaspending.gov/

3 Tracking Federal Awards: USAspending.gov and Other Data Sources
https://www.congress.gov/crs-product/R44027

4 15 n8n Masterclass: Build AI Agents & Automate Workflows (Beginner to Pro) · AI Automation Society
https://www.skool.com/ai-automation-society/n8n-masterclass-build-ai-agents-automate-workflows-beginner-to-pro

5 6 19 Docker | n8n Docs
https://docs.n8n.io/hosting/installation/docker/

7 Docker - Puppeteer
https://pptr.dev/guides/docker

9 10 11 12 13 14 16 17 OpenAI Function Calling for API Integration and Automation
https://blog.promptlayer.com/openai-function-calling/