ChatGPT

Building Your End-to-End BD Automation Engine (Step-by-Step Guide)

Overview: We'll set up an end-to-end Business Development (BD) automation pipeline composed of five "engines" – **Scraper**, **Program Mapping**, **Org Chart**, **Playbook Generator**, and **Scoring** – that work together. Each engine will handle a part of the process (from collecting job listings to generating outreach plans), and all will be orchestrated to run automatically via **n8n** (a workflow automation tool). The data will be stored and updated in Notion databases as the "source of truth." Ultimately, every 24 hours this system will scrape new jobs, enrich them with program info, map contacts/org charts, generate BD playbook insights, and score opportunities [1] [2]. Don't worry if that sounds complex – we'll break it down into simple steps. Just follow along in order, **like a recipe**, and you'll get everything configured from start to finish!

Step 1: Prepare All Required Tools and Accounts

Let's start by getting all the necessary tools and services ready. Make sure you have the following set up **before** writing any code or building workflows:

- **Claude Pro/Max subscription:** You mentioned you have this (Claude "Max"). This gives you access to Claude's 200k-token context and the ability to use Claude's coding features. 👍
- **Claude Code CLI:** Install Claude's command-line tool for coding. Open a terminal on your computer and run: `npm install -g @anthropic-ai/claude-code`. (You need Node.js/NPM installed first.) This is required for the Auto Claude app to run code with Claude.
- **Auto Claude Desktop App:** Download and install the Auto Claude app for your platform from the GitHub releases page [3]. They have installers for Windows, macOS, etc. Once installed, open the app.
- **Git & GitHub (optional):** Auto Claude works with a local Git repository. If you haven't already, create a new local project folder for this and run `git init` in it (or you can let Auto Claude initialize it for you when creating a project). You don't *have* to push it to GitHub yet, but having git setup helps Auto Claude manage file changes.
- **Notion account with API access:** We'll use Notion to store data (like program info and scraped jobs). Ensure you have a Notion database (or two) set up: one for **Programs** (the master list of DoD programs, with fields like Program Name, Prime Contractor, Contract Value, etc.) and one for **Job Postings/Opportunities** (to store scraped job info and the tags we'll add). If you already have a Notion "Programs" database, great. If not, create a new table in Notion for it. Also create a Notion integration (at developers.notion.com) to get an **API key**, and share the databases with that integration. We'll need the API key and the database IDs in our config.
- **Apify account:** Apify is a cloud scraping service. We'll use it (with Puppeteer actors) to scrape job listings and possibly LinkedIn. Sign up for Apify (if you haven't), and get an **API token** from your Apify account settings. Also, find the **Actor IDs** for any scrapers you plan to use (for example, an Apify actor for ClearanceJobs or LinkedIn). You mentioned you've run an "Insight Global Clearance Jobs" scraper – if that was via Apify, note the actor name/ID.

- **n8n (workflow automation tool):** Decide how you'll run n8n. Easiest for testing is to use n8n cloud or run the Desktop app. For production/continuous running, you might set up n8n on a server or Docker. For now, install n8n Desktop (from n8n.io) or use their cloud trial, so you can start building the workflow. We'll integrate everything in n8n later.

Plain English check: *At this point, you should have all the pieces: an Auto Claude app ready to go, a folder on your computer for the project, access to Claude (through your subscription) to power Auto Claude, a Notion database (for storing data), an Apify account (for scraping), and n8n (for automation). Once these are in place, we can start building!*

Step 2: Create a New Project in Auto Claude and Connect Claude

Next, we'll set up your project in the Auto Claude app:

1. **Open Auto Claude** and select "New Project" (or it might prompt you to open a folder). Choose the folder that will hold your project (the Git repo folder you prepared). If it's a fresh folder, that's fine – Auto Claude will treat it as the project workspace.
2. **(Optional Template):** Auto Claude might ask if you want to use a template. If you don't see any templates or are unsure, it's fine to proceed with a blank project. (You mentioned you left it blank – that's okay. We'll define everything ourselves.)
3. **Connect to Claude:** The app will guide you to connect your Claude account. Typically, it will open a browser window for Anthropic OAuth – log in with your Claude credentials and authorize the app. After this, Auto Claude is linked to Claude and can start autonomous coding agents for you [4].
4. **Verify Claude is ready:** In the Auto Claude interface, you should see something indicating Claude is connected (often your Claude username or a status icon). Also ensure the Claude **"coding" mode is enabled** (Auto Claude relies on the Claude Code capabilities to run multiple agents). Since you have Claude Max, you're all set on the subscription side.
5. **Important Setting – Enable Claude code execution:** In your Claude Desktop app (if you use it separately for chatting), go to **Settings > Feature Preview** and enable the **"Analysis tool"** (code execution) option. This allows Claude to run code and manage long conversations. It's not directly part of Auto Claude, but enabling this can help Claude handle big tasks without getting stuck (it allows Claude to summarize intermediate results automatically).

   Note: The last step (enabling the Analysis Tool) helps avoid Claude's 60-second timeout and ensures Claude can "organize its thoughts" in long tasks [5]. Essentially, it prevents the AI from hitting a wall if a task is too long by letting it summarize and continue. This is a one-time setting in Claude Desktop. In Auto Claude, tasks are usually broken into sub-tasks to avoid timeouts, but it's good to have Claude's full capabilities enabled just in case.

Now your project is created and Claude is connected. Think of Auto Claude as your **AI project manager + development team**. You'll create tasks (in plain language) describing what you need, and Claude's agents will plan and generate code for those tasks. We'll use this to build our engines step by step. 🛠

Step 3: Organize Your Project Folder Structure

Before diving into coding tasks, let's set up a clear folder structure in your project. Organizing by engine will make it easier to manage files and for the AI agents (and you) to know where things go:

- **Create folders for each engine:** In your project directory, make folders for each main component:
- `Engine1_Scraper/` – for the Scraper engine (job scraping related scripts or configs)
- `Engine2_ProgramMapping/` – for Program Mapping logic (scripts, data like keyword lists)
- `Engine3_OrgChart/` – for Org Chart & contact mapping files
- `Engine4_Playbook/` – for Playbook generation content or templates
- `Engine5_Scoring/` – for Scoring logic (e.g., scoring scripts or config)

(If you prefer shorter names, you can do `scraper/`, `program_mapping/`, etc. The key is each engine has its own space.)

- **Add a data folder:** Create a folder called `data/` or within each engine folder have a `data/` subfolder. This is where you'll put input datasets like your scraped jobs CSV, program list CSV, etc. For example, put your existing **InsightGlobal jobs scrape CSV** into `Engine1_Scraper/data/` (or a unified `data/` folder) and maybe your **programs database export** (if you have one in CSV/Excel form) into `Engine2_ProgramMapping/data/`. This way the AI agents can easily access those files when coding (Auto Claude agents can open and read files from the repo to understand data formats).
- **Create configuration files:** We'll define some JSON config files for things like scraper settings or program mapping rules. For now, just create empty placeholder files so we know where they are:
- `Engine1_Scraper/Configurations/ScraperEngine_Config.json` – this will hold settings for the scraper (like which sources to scrape, any search keywords, etc.). Go ahead and create an empty file with this name (Auto Claude can fill it in later).
- `Engine2_ProgramMapping/Configurations/ProgramMapping_Config.json` – a config for program mapping, such as a list of program keywords to look for. Create this file empty for now.

- You can make similar config placeholders for other engines if needed (not mandatory right now, but for completeness: e.g. an OrgChart config file if you want to list known contacts, etc.).

- **Include a README:** It's helpful to have a `README.md` in your project root explaining the project goals and structure (even if just a few lines). This is not only for human reference, but also giving the AI context. You can write something like: *"This project is Prime TS BD Automation. It has 5 engines: Scraper, Program Mapping, Org Chart, Playbook, Scoring. The goal is to scrape job listings, map them to DoD programs, find org contacts, generate outreach playbooks, and score opportunities, updating Notion databases."* Keep it simple and clear. Auto Claude's agents will read this and **understand the big picture**, which guides their coding decisions.

After this step, your folder structure should look something like:

```
BD-Automation-Project/
├── Engine1_Scraper/
│   ├── Configurations/
│   │   └── ScraperEngine_Config.json
│   ├── data/
│   │   └── InsightGlobal_Scrape.csv  (example job data)
│   └── ... (other subfolders like Scripts, etc.)
├── Engine2_ProgramMapping/
│   ├── Configurations/
│   │   └── ProgramMapping_Config.json
│   ├── data/
│   │   └── Programs_List.csv (export of program DB, if available)
│   └── ...
├── Engine3_OrgChart/ ... (similar structure)
├── Engine4_Playbook/ ...
├── Engine5_Scoring/ ...
└── README.md
```

Don’t worry if you don’t have all the data files yet – we’ll generate or fetch them as we go. The key is that now **Auto Claude knows the lay of the land** in your repo and we have logical places to put each part of the system.

Step 4: Set Up Notion Databases and API Integration

Since Notion will be our “database” for programs and jobs, we need to configure that now so our automation can read/write to it:

- **Identify your Program database in Notion:** Make sure you have a table (database) in Notion that lists the programs. For example, a database named “DoD Programs” or similar, with columns for things like **Program Name**, **Description**, **Prime Contractor**, **Contract Value**, **Status**, etc. Include any fields you think are important (you can always add more later). If you already exported 388 programs into a CSV [6], that suggests you have these in Notion. Good! If not, you may need to input them or import that CSV to create the database.
- **Set up a Jobs/Opportunities database:** Similarly, have a Notion database for the scraped job postings (or BD opportunities). It can have fields like **Job Title**, **Company**, **Location**, **Clearance Level**, **Posting URL**, **Scraped Date**, **Matched Program**, **Match Confidence**, **BD Score**, **Priority Level**, etc. Basically all the info we will capture for each job. Don’t worry if you’re not 100% sure on fields – create what makes sense (you can modify later). For now, at least have Title, Company, Program (even if blank initially), and some kind of status/score fields.
- **Notion API Key:** You should have an integration token from Notion (from Step 1). In Notion, ensure both the Program DB and Jobs DB are shared with the integration (so the API can access them). Copy your secret API key – we’ll need it in n8n or in code.
- **Collect Database IDs:** Each Notion database has a unique ID (in the URL when you open the database or via the Notion API docs). Grab the ID for the Program database and the Jobs database. We’ll plug these into n8n nodes or our scripts so we can update the right place. For example, in the n8n workflow JSON you have, I see a databaseId like `5ab80c40-31f0-48b5-ba1d-c4dae1af6fc6` for saving jobs [7] – that would be the Jobs DB. Another ID for maybe a “Hot Leads” database appears as well [8]. Identify which is which in your case. If unsure, you can use the Notion API or the official docs to confirm the ID corresponds to your database.

- **Test the connection (optional):** It's a good idea to test your Notion API connection. In n8n, for instance, you can create a credential for Notion by providing the API key, and then use a quick Node (Notion > Get Database or similar) to see if it fetches your schema. If you aren't in n8n yet, you can also test by writing a small Python script using `requests` to GET a database (if you're comfortable), but this is optional. The goal is just to be sure your token and IDs are correct.

By having Notion set up now, when we move to n8n or coding, we won't hit permission issues. The plan is: **scraped jobs will be added to the Jobs DB**, enriched with program info, and perhaps we'll also update the Program DB itself with any new info we discover (like if we scrape a program's budget, we'd put it in the Program DB). So keep those database IDs handy.

Step 5: Build the Scraper Engine (Job Data Collection)

Now the real fun begins – let's build the **Scraper Engine** to gather cleared job postings daily. Since you want an *automated* solution, we'll rely on tools (Apify, etc.) rather than manual scraping. Here's how to proceed:

1. **Plan what to scrape:** Decide which sources you want to scrape regularly. Common ones:
2. Competitor companies' career pages (you likely have a list of competitors and their job board URLs – e.g., Northrop Grumman careers, Lockheed Martin jobs, etc.).

3. Job boards like **ClearanceJobs** (which you already did a scrape for), **LinkedIn** (filtered for certain clearance keywords or companies), and possibly **SAM.gov** for contract opportunities.
   Make a list of these target sources. You might have a config file like `Competitor_Job_Board_URLs.xlsx` (as mentioned in your files) – that's perfect [9]. Ensure it's up-to-date with the URLs or API endpoints for each source.

4. **Set up Apify actors for each source:** For each target source, you can use Apify's ready-made actors or create your own:

5. *If Apify has a public actor:* Search Apify's actor library. For example, Apify has a **LinkedIn Jobs Scraper** actor that can take a LinkedIn search URL or job ID and return structured data [10] [11]. It also has actors for other job boards. If you find actors for your needs, note their names.
6. *If not, create a custom actor:* You might need to create a custom Apify actor using Puppeteer for some sites. Apify's interface allows you to write a little JavaScript to navigate a page and extract data. However, since you have many sources, it might be easier to use a code-based approach or something like **JobSpy** library in Python [12] [13]. But given you have Apify set up, let's stick with that to minimize coding for scraping.

7. In summary, ensure **each source is covered** either by an Apify actor or another method. For now, let's assume Apify actors handle it.

8. **Define ScraperEngine_Config.json:** Open the placeholder file we made (`Engine1_Scraper/Configurations/ScraperEngine_Config.json`) and fill it with the information about what to scrape. This can be a simple JSON structure like:

```
{
  "sources": [
    { "name": "ClearanceJobs", "actorId": "apify/clearance-jobs-scraper",
"searchTerm": "TS/SCI" },
    { "name": "LinkedIn", "actorId": "apify/linkedin-jobs-scraper",
"query": "security clearance" },
    { "name": "CompetitorSites", "list": "Competitor_Job_Board_URLs.xlsx" }
  ],
  "output": "Engine1_Scraper/data/new_jobs.json"
}
```

This is just an example. The idea is to list out the sources and any parameters. You might include keywords to search (like “TS/SCI”, “Secret”, etc.), locations (like major hubs), or other filters. If you have a CSV of competitor URLs, reference it. This config file will guide our code or workflow on what to scrape.

9. **Use Auto Claude to code the scraping logic (optional):** You can have Claude help write a script to orchestrate these scrapers. For example, create a **new task in Auto Claude** called **“Implement Scraper Engine to collect job postings from all sources”**. In the task description, write something like: *“Use the sources listed in ScraperEngine_Config.json to fetch new job postings. For each source: if it’s an Apify actor, call the Apify API with the actor ID and required input (e.g., search term or URLs). Collect all results into a unified JSON or CSV. Ensure the output has fields: job title, company, location, clearance, posting URL, etc., normalized. Save the consolidated results to* `Engine1_Scraper/data/JobsDataset.json`*.”*
   Claude will likely plan steps and generate a Python (or Node.js) script to do this. It might use `requests` (for Apify API calls) and then merge the results. Let it create the script (for instance `Engine1_Scraper/scrape_jobs.py`). Review the code it produces: it should read the config JSON, loop through sources, make HTTP requests to Apify’s API endpoints (using your Apify token for auth), wait for the actor run to finish, fetch the results, and combine them.

10. **Tip:** You will need to provide your Apify API token to the script. The agent might ask where to store it – you can suggest using an environment variable or include it in the config JSON (but be careful not to hard-code secrets in code). For simplicity, you might put `"apifyToken": "YOUR_API_TOKEN"` in the config for now, or instruct the script to read from an environment variable.

11. **Testing the script:** Once Auto Claude writes the code, you can run it (maybe via the Claude Desktop “analysis tool” or manually in your terminal) to see if it works. If any part fails (e.g., wrong actor ID or slow response causing a timeout), adjust and retry. This might take a couple of iterations. (Remember, you can always run smaller tests – like only one source at first – to verify the pipeline works without hitting timeouts.)

12. **Normalize and store the data:** The output of the scraping should be a unified dataset (like an array of jobs). Ensure each job entry has a unique ID or a combination of fields to uniquely identify it, so we can avoid duplicates on consecutive runs. Fields to include:

13. `jobId` (maybe generate one or use the source’s ID), `title`, `company`, `location`, `clearanceLevel` (if available), `postedDate`, `url`, and maybe a snippet of `description` or `program_keywords` if you can extract them.

The **Scraper Engine's job** is to produce this **Jobs Dataset** (as a JSON or CSV) every day [14] [15]. Save it to a file (and also we will later push each entry into the Notion Jobs DB via n8n).

14. **Set up daily schedule in n8n:** We will integrate this with n8n soon, but keep in mind the plan: n8n will trigger this Scraper engine every 24 hours (say at 6 AM daily). The n8n workflow could either **call the script** we just made (using an "Execute Command" node or a Code node) or directly call Apify actors via HTTP nodes and skip our script.

15. If you prefer not to rely on an external script, you can replicate the logic in n8n with multiple HTTP Request nodes (one per source) and some Function nodes to merge results. That's equally fine. In fact, the final system diagram suggests using n8n's HTTP and code nodes to parallelize scraping [16].
16. For simplicity, you might use n8n to call the Python script (fewer nodes to configure). If doing so, ensure the environment running n8n has Python and required libraries (requests) installed. Alternatively, you can containerize the script or turn it into an AWS Lambda, etc. But let's not complicate – calling a local script in n8n is okay for now.

After this step, you have a working **Scraper Engine** that can fetch the latest job listings from various sources. If you run it now, it would output a consolidated list of, say, the 190 Insight Global jobs plus others (depending on sources). This provides fresh data for the next engines.

(Quick checkpoint: At this point, we haven't yet put anything into Notion or mapped programs – we just have raw job data. Next, we'll enrich that data by linking each job to a program.)

Step 6: Build the Program Mapping Engine (Job-to-Program Tagging)

The **Program Mapping Engine** takes each job posting and figures out **which DoD program** (if any) it is associated with. This is crucial because it connects a generic job ad to a specific contract or project opportunity you care about. We'll implement this in a simple way first (keyword matching), and you can later enhance it with AI if needed.

1. **Prepare a Program Keywords list:** For each program in your Program database, come up with keywords or clues that might appear in a job posting if it's related. For example, if a program is "GPS III Satellite Program," postings might mention "GPS III" or the satellite name. If a program is "JEDI Cloud Contract," maybe jobs mention "JEDI" or the agency. Many cleared job postings include hints like the contract name or the customer organization. Create a list or table of program names and associated keywords. You might store this in the `ProgramMapping_Config.json` we made. For example:

```
{
  "programKeywords": {
    "Program Alpha": ["Program Alpha", "Alpha Contract", "PA-X"],
    "Sentinel (GBSD)": ["GBSD", "Ground Based Strategic Deterrent",
"Sentinel"]
    // ... add for all key programs
  },
  "unmappedTag": "Unmatched"
}
```

This JSON maps program names to an array of keywords/phrases. Keep them lowercase in both the data and when checking for easier matching. The `unmappedTag` is just a label for jobs that don't match any known program (we can mark them as "Unmatched" for follow-up).

2. **Auto Claude helps code the mapper:** Now, create a **new task in Auto Claude** titled **"Implement Program Mapping Engine to tag jobs with program names"**. In the description, outline what we need: *"For each job record (from the Jobs Dataset JSON), compare the job title and description with our programKeywords list (from ProgramMapping_Config.json). If any keyword is found, assign the corresponding Program Name to that job's* `matchedProgram` *field and maybe a confidence score. If multiple programs match, pick the one with most keyword hits or mark as ambiguous. If none match, set* `matchedProgram` *to 'Unmatched'."* Claude will then likely create a script (maybe `Engine2_ProgramMapping/map_jobs_to_programs.py`). It should:

3. Load the latest jobs JSON (from Engine1's output).
4. Load the ProgramMapping config (with keywords).
5. Loop through each job's text (title, maybe description). For each program in the list, check if any keyword appears. This can be a simple substring search (e.g., case-insensitive).
6. If found, add `matchedProgram = "Program Name"` and possibly `matchConfidence`. (Confidence can be a simple metric like number of keywords found or length of match – or even just 100% if found, 0% if not. You can refine this later.)
7. If multiple programs' keywords appear in one job (unlikely but possible if the job description is long), the script could either choose one (maybe the one with more matches) or mark the job for manual review. To keep it simple, you might decide on one (like highest count or first match) and proceed.
8. Output an **enriched jobs dataset**. Essentially, add new fields to each job entry for `matchedProgram` and `matchConfidence`.

Let Auto Claude generate this code and then review it. Ensure it correctly reads the JSON and does the string matching. If you have a list of top 20 programs [17], focus on those – that covers the majority of cases. You can always expand the list later.

1. **Test program mapping logic:** Take a few sample job entries (perhaps ones you know the program for) and run them through the script. For example, if a job description mentions "GBSD", does it correctly tag "Sentinel (GBSD)" program? If not, tweak the keywords. This might be iterative: update the keywords list to catch variations. For instance, if a program name is commonly abbreviated, include the abbreviation.

2. You can log or print jobs that remain "Unmatched" – these are ones to manually analyze or use AI for. In your earlier analysis, you had 185 unmatched out of 190 [18], which indicates the need to improve this mapping. Likely, many job postings don't explicitly mention program names, or we might need to infer from context (like customer name, which company is hiring, etc.).

3. **(Optional AI enhancement):** For a more advanced approach, you can integrate an AI call here. For example, use the OpenAI node in n8n or an API call to GPT-4: *"Given this job description, which known DoD program is it likely associated with (from this list)?"* However, to keep things simple and not

dependent on the AI every run, start with keywords. You can flag unmatched jobs for a separate AI processing step if needed (maybe once a week, go through unmatched and have GPT suggest programs, then add those to your keyword list).

4. **Update Notion Program DB if needed:** If during this process you discover new programs or new information (for example, a job clearly is for a program not yet in your database), you might want to update the Program database. Initially, your Program DB has 388 programs (the "master" list). If the scraper finds something outside that list, consider adding it. You had a file "New_Programs_Identification.xlsx" [19] which suggests finding programs not in the database. For now, just note this – you can append new programs to Notion via the API as they arise. Perhaps mark them for review.

5. We won't automate adding programs just yet (avoid accidentally adding noise), but keep an eye on it.

Now you have an **enriched jobs dataset** where each job posting is tagged with a Program (or marked Unmatched). This output can be written to a new JSON, say `Engine2_ProgramMapping/data/Enriched_Jobs.json`. More importantly, we will push these results into Notion next, so your "Jobs" database in Notion gets populated with each job and its matched program.

Step 7: Update Notion with New Jobs and Program Links

With scraping and mapping done, we need to **write the results to Notion** so that your data is centralized. We'll create steps in n8n to do this (since n8n has a Notion node and can handle multiple records nicely):

- **In n8n, add a Notion node for Jobs:** After the scraping and mapping in your workflow, add a node (Notion > Create or Update Page in Database). Configure it to use your **Jobs/Opportunities database** in Notion. Map each field from the enriched job data to the corresponding Notion property. For example, in n8n's Notion node you'll set:
- Title = {{$json["title"]}} (or however n8n references the incoming data)
- Company = {{$json["company"]}}
- Location = {{$json["location"]}}
- Clearance Level = {{$json["clearanceLevel"]}} (assuming you made that a select or text property)
- Job URL = {{$json["url"]}}
- Scraped Date = {{$json["postedDate"]}} (make sure to format as date)
- **Matched Program** = {{$json["matchedProgram"]}} (this could be a relation or just text – if your Notion Program DB is separate and you want a relation, you'll have to provide the Page ID of the program. To keep it simple, you might just store the program name as text for now. Later you can turn it into a relation.)
- **Match Confidence** = {{$json["matchConfidence"]}} (number property)
- Status = "Open" (you might set a default status like Open or New)
- etc., including BD Score and Priority Level (which we'll fill in after scoring).

In the JSON from our mapping script, we don't have BD score yet (that comes in next engine), so you can leave BD Score blank or skip it here. We will update it later once scoring is done, or we incorporate scoring in the same run.

- **Prevent duplicates:** When adding to Notion, ensure you're not duplicating entries every day. You likely want to *update if exists, otherwise create*. One strategy is to use an ID or unique key. If you have a `jobId` or the combination of (Title+Company+Location) as unique, you could configure the Notion node to search or filter. Unfortunately, n8n's Notion node doesn't do upsert directly. You might need to first search if that job exists (maybe store jobId in a property in Notion), then update, else create. This can be done by linking nodes (e.g., Notion Search, then conditional Create/Update). If that's too advanced for now, you can cheat by clearing the Jobs DB daily (not ideal) or trusting that the same job won't appear twice (which might not hold). A simple approach: add a "Job ID" property in Notion and **use the job's URL or a hash as the unique ID** – since postings URLs are unique. Then, in n8n, use a Filter or IF node: if the job URL already exists in the database (you can query Notion by URL), skip it; if not, create a new entry.
- For example, use a Notion Search node with filter "URL == current job URL", see if results > 0. If yes, skip; if no, create new.

- **Write to Program DB if needed:** We should also consider updating the Program database with any new info gleaned. In this step, after tagging jobs to programs, you might update a "Last Hiring Activity" date or "Open Roles Count" for each program in the Program DB. This is a nice-to-have: e.g., each time we tag a job with Program X, increment a counter or update a property in the Program's page (like "# of Open Jobs" or "Last Scrape Date"). If you want to do this:

- Add another Notion node (Update Page in Database) configured for the Program DB.
- For each job's matchedProgram, you'd find that Program's page (maybe store the Program Name as the key, or use a relation).
- Then update fields like "Open Roles" (could increment by 1 for each job).
- This might require first retrieving the current value (to increment) unless you maintain it elsewhere. For a beginner, this might be a bit complex, so you can skip it initially. The main goal is to get all jobs in the Jobs DB with program tags. Analysis on programs (like counting jobs per program) can be done later or within Notion itself.

At this stage, after running the Scraper + Program mapping and Notion update, your Notion **Jobs database** should now be filled with entries (each job listing as a row, showing which program it's tied to). If you open Notion, you'd see new pages created for each job, and for example *"Software Engineer – Lockheed Martin – Program XYZ – TS/SCI – Score: (pending)"* etc. This is a big milestone: you've turned raw scraped data into structured intel in your system of record (Notion).

Step 8: Build the Org Chart & Contact Mapping Engine (Inferring People and Structure)

Now we move to Engine 3, the **Org Chart reconstruction and contact mapping**. This one is a bit conceptual, and may require data from your internal systems (like Bullhorn or any CRM of contacts you have). The idea is to figure out **who** the key people are for each program and how they relate (who is program manager, who are team leads, etc.), using a combination of internal data and external scraping:

1. **Gather internal contact data (if available):** If your company uses something like Bullhorn (a staffing CRM) and has contacts associated with programs or past placements, export that data. For example, a list of **contacts (names, titles, company, program, etc.)** you have in your database. Perhaps you have an export like "Lockheed Contact Export.xlsx". Collect those into a CSV. These contacts can serve as a starting point (they might include program managers, hiring managers, etc., that you've interacted with). Place this file into `Engine3_OrgChart/data/` (e.g., `Internal_Contacts.csv`).

2. **Decide which contacts to focus on:** Likely, you want to map org charts for key programs (the ones that are hot or important). You might start with the Top 20 programs identified earlier [17]. For each of those, the goal is to identify roles like **Program Manager**, **Lead Engineer**, **Recruiter**, etc., and the team composition (maybe number of open roles gives a clue to team size).

3. Also identify if you already know some of these people from your contacts list. For example, if Program Alpha is run by Colonel John Smith at Company X, and you have John Smith in your contacts, that's a link.

4. **LinkedIn scraping for missing contacts:** For people you don't know, we can use LinkedIn. The strategy: take each program and try searches like "<Program Name> Program Manager <Company Name>" on LinkedIn. Apify has LinkedIn Profile scrapers and even a LinkedIn Search actor. To keep it automated:

5. Use Apify's **LinkedIn Profile Scraper** actor: it can fetch a profile by URL. If you have a list of profile URLs (not likely yet).
6. Alternatively, use **LinkedIn People Search** actor: give it a search query and it returns profiles. You might use queries derived from program info.
7. This could be complex to automate fully, but you can do it in steps. Perhaps maintain a list of "target roles to find" in a config file, like `OrgChart_Config.json`:

```
{
  "targets": [
    { "program": "Program Alpha", "company": "Lockheed Martin", "role":
"Program Manager" },
    { "program": "Program Alpha", "company": "Lockheed Martin", "role":
"Lead Engineer" },
    ...
  ]
}
```

   List out the key roles for each important program. Then an agent or script can iterate and perform a LinkedIn search query for each.
8. **Use Auto Claude to script this (optional):** You can ask Auto Claude in a new task: *"Create a script to search LinkedIn via Apify for key contacts for each program. Use OrgChart_Config.json targets: for each*

target, perform a LinkedIn search (if Apify has an actor or perhaps use the LinkedIn API if available) and retrieve at least one result (name, title, profile URL). Save results to Engine3_OrgChart/data/Found_Contacts.json." The AI might come up with using Apify's actor by making an API call similar to how we did for jobs. You'll need to provide your Apify token again.

9. **Alternate approach:** If coding this is tricky, you can do a one-time manual search for a few key programs to gather contacts. Since our goal is to outline the system, it's okay to do some things manually initially (like find the Program Manager's name for Program Alpha and input it). Automation can be added later.

10. **Compile an Org Chart structure:** Once you have a list of key contacts for a program (from internal + LinkedIn data), you need to structure it. An **Org chart** could simply be a list of roles and names under the program. For example:

11. Program Alpha (Prime: Lockheed Martin)
    - Program Manager: John Smith (Lockheed Martin)
    - Lead Engineer: Alice Doe (Lockheed Martin)
    - Senior Analyst: (open role) [or Bob Jones if identified]
    - etc. One way to store this is to create a Markdown or OPML (outline format) file. I see you had something like `GBSD_Sentinel_OrgChart.opml` [20] – OPML is an outline format which could represent an org chart hierarchy. If you like, you can use that. Or simply create a Markdown file per program in `Engine3_OrgChart/` with a bullet list structure.

For example, you could have `Engine3_OrgChart/OrgCharts/ProgramAlpha.md` containing the structure above. Auto Claude can assist: for each program, prompt it with known contacts and roles to format a nice org chart outline. You might give it an input of names/titles and ask it to arrange by seniority.

1. **Link contacts back to Notion:** If you have a Notion database for **Contacts** or you use the Program DB to list key people, update that with the found contacts. Perhaps your Program database in Notion has properties like "Program Manager Name", "Program Manager Contact Info", etc. You can use n8n or a script to fill those in for the programs you found info for. For instance, after gathering John Smith for Program Alpha, use the Notion API to update Program Alpha's page with "Program Manager = John Smith". This can be done with a Notion Update node similarly to how we considered updating counts.

If you don't have a dedicated contact DB or field, you might just keep this info in the org chart files for now. It depends how you want to access it later. Storing in Notion is nice for quick reference and for the Playbook engine to use.

1. **Note on Org Chart automation:** This part might feel less automated than others – and that's okay. Org charts often require piecing together info that isn't explicitly stated anywhere. We're using a mix of internal data and external search. It might not be 100% complete every run. A pragmatic approach is to run this engine periodically (maybe weekly or when a program becomes hot) rather than daily. Also, you might maintain a manual oversight: e.g., review the found LinkedIn profiles to ensure they're correct before trusting the data.

At the end of this step, you should have at least some key contacts identified for top programs and a structure of who's who in each program's team. This will feed the next step, the **Playbook Generator**, which uses these people and roles to tailor outreach. But even if the org charts are not fully complete, don't worry – you can still generate useful playbook content with what you have. The system can always update org info as new clues come (like if new job postings list a hiring manager name, you'd capture that).

Step 9: Build the Playbook Generation Engine (HUMINT & Outreach Content)

The **Playbook Generator Engine** creates customized outreach materials (emails, talking points, call scripts) for engaging the opportunity. It combines **HUMINT** (human intelligence – e.g., insights from your team or news about the program) with the data we've collected (program name, open roles, pain points, contacts). We will leverage AI (Claude or ChatGPT) heavily here to generate human-like content.

1. **Gather playbook templates and content:** You mentioned you compiled a **Prime TS Sales Prospecting Playbook v1** in August [21] – which likely includes email templates, call scripts, rebuttal lists, etc. Locate that content (perhaps in a PDF or document). If it's a PDF, consider converting key parts into text or a Markdown file. For instance, create `Engine4_Playbook/MasterPlaybook.md` and copy in the general templates (e.g., a generic intro email template, a follow-up call script outline, etc.). This will serve as base material.

2. Also include any **competitive intelligence** or **event info** that might be relevant. E.g., if you have a section on "Competitive Insertion Tactics" or upcoming defense conferences [22], have that available too. The more context the AI has, the more tailored the output can be. But keep it concise enough for the model to handle.

3. **Define the output you want:** For each program (or each hot opportunity), what should we generate? Possibly:

4. A **brief summary** of the opportunity (e.g., "5 open roles for Program X at Company Y indicate they need cloud engineers – likely pain point in scaling infrastructure.").
5. An **intro email draft** from your company to a stakeholder (e.g., a hiring manager or program manager), referencing the program and how you can help.
6. A **call script** for a BD rep calling about the program.
7. Key **talking points** or value propositions tailored to the program's context (like referencing something known about the program's mission or recent news).

8. Any **social engineering angles** (if appropriate), like mentioning you saw them hiring at a rapid pace, etc. Decide which of these you need. You can start with one (e.g., just an email template) to keep it simple, and add others later.

9. **Use Claude or ChatGPT to generate content:** This can be done within n8n using an **OpenAI node** (for GPT-4) or you could attempt to use Claude via API if you have access (Anthropic has an API, but not sure if available to you). Easiest is likely GPT-4 via the OpenAI node, since you have that in n8n.

10. In n8n, after the scoring engine determines a "Hot" lead, you can branch and add an **OpenAI** node.
11. Feed it a prompt that includes the relevant info: Program name, company, any known contacts (like program manager's name), and possibly some context from the Program DB (like contract value or known issues if you have any in notes), plus your *playbook template content*.

12. For example, prompt: *"You are a business development specialist. Here is our sales playbook template for outreach: [insert template text]. Now, generate a customized outreach email for Program X at Company Y. Program X details: [maybe a one-liner from Notion Program DB if you have a description]. They currently have Z open positions (according to our data). Our company offers [your service offering]. Compose a friendly email introducing our services to the Program Manager (Mr. John Smith) that references their current hiring need and proposes a meeting."* The model will then output an email draft tailored to Program X.
13. Similarly, you can do a call script: *"Also provide 3 talking points for a phone call, anticipating one possible objection and a rebuttal."* Use the content from your Master Playbook to guide style and structure.
14. **Incorporate HUMINT:** If you have any notes or human-collected intel on the program (e.g., "Heard that budget got cut" or "They struggled with last contract transition"), include that in the prompt or have it stored in Notion and pulled in. For instance, maybe your Program DB has a "Notes" field where you jot such intel. You can append that: *"Note: This program had a recent leadership change, which might affect contracting."* The AI can then weave that in appropriately.
15. **Decide output location:** Where do you want these generated playbook materials to go? Options:
16. **Notion:** You could have a Notion database for "Outreach Plans" or even attach the content to the Program's page in Notion (e.g., update a "Latest Outreach Draft" property with the text).
17. **Email:** You could have n8n automatically send the drafted email to you or your team for review via an Email node. That way every morning you get an email: "Here's an outreach draft for Program X opportunity." Then you can decide to send it or tweak it.
18. **Markdown files:** Alternatively, Auto Claude or your code could simply save these to files (like `Engine4_Playbook/Outputs/ProgramX_email.txt`). But it's probably more useful in Notion or email for easy access.

Let's choose one for now: let's have n8n email the BD team (or yourself) the content. In the **n8n workflow**, after generating the content with OpenAI, add an **Email Send node** (like SMTP node). Compose an email that includes the AI output. For example, subject: "Daily Outreach Draft – [Program Name]", body containing the text. You can use HTML or Markdown for formatting if needed. If you prefer, also log it to Notion for record-keeping (maybe create a page in an "Outreach" database with the content).

1. **Automate HUMINT parsing (optional):** You mentioned "HUMINT parsing" – possibly meaning analyzing unstructured intel (like a PDF or an email from a human with notes). If you have such sources (maybe meeting notes from conferences, or Slack messages about a program), you could use an AI to extract key points and feed them into the Playbook. This is an advanced step; an example might be: if you had a transcript from a conference panel about Program X, you could run it through a summarizer and store the highlights in the Program DB. That way, the outreach content can say "As mentioned at last month's defense symposium, Program X is focusing on AI integration…". Implementing this fully would require identifying where HUMINT data lives and using an AI to parse it. Since this can be a whole project in itself, for now you might just keep a manual process: whenever you get such intel, put it into the Program's Notion page manually so it's available for the AI to use. Later, you could set up something like: monitor an email inbox or a folder for new intel docs, trigger an AI parser when something is added, and update Notion automatically. But let's leave that as a future enhancement.

Now, you have a capability to **generate tailored outreach content** whenever an opportunity is identified. The Playbook engine doesn't necessarily run for every job – it should focus on high-priority ones (we'll determine those via the scoring engine next). Typically, you'd run this for "Hot" leads daily. In practice, you can integrate it so that if any job's BD score is Hot, an outreach draft is created automatically. That's exactly what we'll set up with the Scoring engine and filters.

Step 10: Build the Scoring Engine (Prioritize Opportunities)

The **Scoring Engine** assigns each opportunity a score (0–100, for example) to gauge how attractive it is for pursuit. You already have a weighting model in mind [23] and even a code snippet for calculating BD Score [24] [25]. We'll implement this logic so that every job (opportunity) gets a **BD Score** and a **Priority Level** (Hot, Warm, Cold).

1. **Define scoring criteria:** Based on your plan, scoring might include factors like:
2. **Hiring Demand (need)** – e.g., number of open roles for that program or difficulty filling them. (If a program has many jobs open, higher score.)
3. **Pain Level** – maybe clearance difficulty or skill rarity. (If the job requires a high clearance or niche skill, that indicates pain for the customer, hence an opportunity for you – higher score.)
4. **Contract Value** – if the program value is high or if a lot of money is at stake, you'd prioritize it.
5. **Company Fit** – how well does it align with your company's capabilities. (If it's in an area you excel, score higher.)
6. **Competitive Intensity** – are there many competitors or few? (If few can do it or it's in a remote location, the customer might be desperate – higher score.)

You mentioned some weights like 30% demand, 20% pain, 20% contract value, 20% fit, 10% competition [23]. We can use those or adjust. The code in the n8n workflow used slightly different weights (it summed to 100% differently, but close enough).

1. **Implement scoring function:** You can do this in a **Code node in n8n** or as a small script using Auto Claude. Since you already have a working example (in the JSON, the Code node "Calculate BD Score" with JavaScript [24] [25]), we can reuse that. It basically:
2. Looked at clearance level to gauge *pain* (higher clearance -> more pain -> higher score).
3. Looked at if title contains certain IT keywords to gauge *company fit* (since PTS is an IT services company, IT roles = better fit).
4. Looked at location for competitive intensity (some locations are harder to staff, so if job is in those, score higher).
5. Possibly considered contract value by checking if the program's contract value was known (maybe through Program DB).
6. It then combined them with weights and got `bdScore`.
7. Then assigned `priorityLevel` = Hot (score $\geq$ 80), Warm ($\geq$ 50), Cold (< 50) [26] [27].

To implement: If using n8n, add a **Function (Code) node** after the jobs are enriched with program. Input each job JSON to it. In JavaScript, replicate the logic:

```
// Example scoring inside n8n Function node
const job = item.json;
let score = 0;
```

```
// Hiring Demand: could use a placeholder or # of open jobs in that program. If
you have a count of jobs per program, use it, else maybe use 50 as baseline.
let hiringDemand = 50;
// If clearance level is very high:
let pain = 0;
if (job.clearanceLevel && job.clearanceLevel.includes("Poly")) pain += 20;
else if (job.clearanceLevel && job.clearanceLevel.includes("TS/SCI")) pain +=
10;
// Company Fit: if role is in your specialty (let's say IT related)
const itKeywords = ["engineer", "developer", "analyst", "cyber", "software"];
let fit = itKeywords.some(k => job.title.toLowerCase().includes(k)) ? 20 : 0;
// Contract Value: if Program DB has a value field, use it. For now, maybe
assume medium.
let contractValue = job.programValue ? (job.programValue > 100e6 ? 100 : 60) :
60;
// Competitive Intensity: if location is in low-talent area, add points
const lowCompLocs = ["Huntsville", "Colorado Springs", "Augusta", "Hawaii"];
let compIntensity = lowCompLocs.some(loc => job.location.includes(loc)) ? 80 :
50;
// Combine weights (adjust weights as desired):
score = 0.25*hiringDemand + 0.20*pain + 0.20*contractValue + 0.20*fit +
0.15*compIntensity;
score = Math.round(score);
let priority = score >= 80 ? "Hot" : (score >= 50 ? "Warm" : "Cold");
// Attach to output
item.json.bdScore = score;
item.json.priorityLevel = priority;
return item;
```

Note: This is just illustrative. You should customize the criteria based on what data you actually have. For example, if you can determine the number of job openings for the program (hiringDemand), use that. Or if certain skills align with your core competencies, factor that in. The weights can be tweaked anytime.

1. **Run scoring on each job:** Whether in n8n or a separate script, apply the above to each job record. Now each job will have a `bdScore` (0–100) and a `priorityLevel`.

2. **Update Notion with score:** Add those fields into the Notion update step (from Step 7). If you hadn't added BD Score and Priority in the initial creation, now you can **update** each page in the Jobs DB with the new values. In n8n, you can either include it in the creation (if you calculate score earlier), or run another Notion node in the flow after scoring to update the existing entries. Since it might be easier to calculate score right after mapping (before writing to Notion), you could rearrange the workflow: Scrape -> Map -> Score -> then write to Notion *with* all fields including score.

3. That's probably best: connect the output of Program Mapping to the Scoring Function node, then its output goes to the Notion Create node. So by the time we write to Notion, the entry has a BD Score and a Priority tag.

4. Ensure your Notion database has properties for **BD Score** (number) and **Priority Level** (select or text). Create those if not already.

5. **Filter Hot leads:** In the workflow, after scoring, you can put a Filter or IF node to check `priorityLevel == "Hot"` [28] [29]. For each Hot lead, you might take additional actions:

6. For example, branch to the **Playbook Generator** steps (call the OpenAI node to create outreach content for those hot ones).
7. Also, maybe post a Slack message or email to alert the team about the hot lead (if you have Slack or just rely on the summary email).

8. In your n8n JSON, I saw a "Filter Hot Leads" node [30] connected to something that likely creates a page in another database or triggers the email. That matches our plan.

9. **Summarize and notify:** Finally, set up a daily **summary email** or dashboard. You had an Email node crafting an HTML report [31] [32] – which is great. You can stick with that: basically compile stats like number of hot leads, warm leads, etc., and list the top 10 hot jobs with their program and score. This makes it easy to digest each morning.

10. If you like, use an n8n Function node to count hot/warm/cold from the dataset and format an HTML string (like in your v2.1 JSON) and send via Email node.
11. Or, simply in the email body list each Hot lead: "Hot: Software Eng for Program X at Company (Score 85)". Whatever is readable to you/execs.

Now the scoring engine will ensure you focus on the juiciest opportunities. A job that hits many of your criteria will be Hot (for example, a hard-to-fill role on a high-value program that aligns with your expertise in a location with few competitors – that should score near 90-100). Less promising ones will be Warm or Cold, so you can spend less time on those.

At this point, the core automation is built: **Data comes in -> enriched with program -> contacts identified -> scored -> stored -> and notifications/outreach generated for top priorities.**

Step 11: Orchestrate Everything in n8n (Workflow Integration)

Let's ensure all the pieces we built are connected in one coherent workflow (or a set of workflows) in **n8n**, so the process runs end-to-end automatically each day:

- **Start with a Scheduler Trigger:** Add a Cron node or trigger node in n8n set to run at your desired time (e.g., every day at 6:00 AM). This will kick off the daily pipeline.

- **Scraping sequence:** After trigger, add nodes to perform **Scraper Engine**:

- Option A: One node that executes the scraping script (if you made `scrape_jobs.py`). For example, use an "Execute Command" node to run `python Engine1_Scraper/scrape_jobs.py`. Then use a File Read node to load the output JSON.
- Option B: Multiple parallel HTTP request nodes (one per Apify actor). You can use the Split In Batches or just a series if parallel triggers might be tricky with Apify. Possibly better: one after another to

avoid rate limits. For each, call `https://api.apify.com/v2/acts/<actorId>/runs` with appropriate payload, then wait for completion and get the dataset.
- If you use separate nodes, you'll need to merge results (you can use a Merge node or simply a Function node to concatenate arrays).
- Ensure the output is a unified list of jobs.

- **Program mapping and scoring:** Pass the jobs list into a Function node (or a series: one Function to map programs if not done in script, one to score). If you already handled mapping & scoring inside the scraping script (you could combine all in one Python script if you wanted to), then you might skip separate nodes. But keeping them modular is good.

- For clarity, you could have one sub-workflow (call it "Enrich Jobs") that takes raw jobs and outputs enriched jobs. This sub-workflow could itself call the mapping script we wrote or just do it inline with code. Up to you. Using a sub-workflow (or multiple connected nodes) makes it easier to test each part.

- **Notion update nodes:** After we have enriched + scored jobs, use the Notion node to create/update pages in the Jobs database (as described in Step 7). This will likely be within a loop (n8n can automatically handle lists – the Notion node will create multiple if you send an array). Double-check the execution: you may have to toggle "Split Items" depending on how the node expects input. (If confusion arises, you can explicitly use an IF or split mechanism to iterate.)

- **Filter Hot and generate outreach:** Use an IF node to filter items where priorityLevel == Hot. Connect the true output to:

- The OpenAI node that generates outreach content (with the prompt configured as in Step 9).
- Then connect that to an Email node to send the content (to your email or team).

- **Summary email:** Separately (or as part of same email above), you might want a summary of how many hot/warm/cold leads. You can prepare that text using a Code node (as mentioned). Then feed it into the Email node as well. In your JSON, it looks like the summary email was being constructed with HTML and sent to possibly a Gmail SMTP [33] [34]. You can replicate that.

- **Error handling:** Add some basic error paths. For example, after each major step (scrape, notion update, etc.), you can connect an error branch that sends you an alert if something fails. n8n allows catching errors globally or per node. At minimum, consider a fail-safe: if Apify scraping fails (maybe due to network), have a notification (email) that tells you "Scraper failed today for X source." This way you don't silently miss data. Also, to avoid one failure stopping everything, you can use the `Continue On Fail` option on nodes like HTTP requests – so if one source fails, others still run and the workflow continues. Log the failure for review.

- **Test the workflow manually:** Before relying on the schedule, do a manual run in n8n. Watch it step through:

- Does it retrieve data from Apify correctly? (Check the output data.)

- Does the mapping function tag the program names as expected?
- Does scoring produce reasonable scores?
- Are Notion entries created/updated properly? (Look in Notion after a run – see if the data appears correctly in each property.)
- If possible, test the branch where a job is marked Hot to see if the OpenAI node and email sending work. You might temporarily mark a job as Hot (force `priorityLevel = "Hot"`) just to test that branch, then switch back.

- Fix any issues (common ones: Notion properties not mapped right, minor coding bugs, timing issues with Apify, etc.). It's normal to adjust and rerun a few times.

- **Activate scheduling:** Once a test run works end-to-end, enable the Cron trigger (in n8n you "Activate" the workflow). Now it's live! It will automatically run each day at the set time.

One more thing: ensure your n8n instance is running continuously (if using desktop, it runs when your computer is on; for truly automated, you might run n8n on a server or use their cloud). You can start with manual runs daily if needed (like a cron job on your PC to launch n8n workflow), but ultimately having it hosted (maybe on a small AWS or Heroku or Docker on a VPS) would make it hands-off. n8n's docs have guidance on deploying it.

Congratulations – at this stage, the **entire engine is built and automated**. Each morning (or whenever scheduled), it should perform all the steps: scrape new jobs, update Notion, find program matches, score them, and send out summaries and any crafted outreach content. All while you sip your coffee!

Step 12: Monitor, Maintain, and Improve

With the system up and running, treat the first few weeks as a **pilot period**. Here's how to manage it like a 10-year-old prodigy CEO :

- **Check the outputs daily:** Look at the Notion database and the summary email. Do the results make sense? Are programs being correctly tagged, or do you see a lot of "Unmatched"? If many are unmatched, you might need to add keywords or integrate the AI mapping for those. Are the BD Scores reflecting your intuition (i.e., do the "Hot" leads indeed seem hot)? Adjust scoring weights if not – this is easy to tweak in your code node.
- **Update keywords and criteria:** This system will only be as good as the knowledge base you give it. Continuously update the ProgramMapping keywords as you learn new terms. For example, if you see an unmatched job and realize it's actually for Program Z (but used a code name you didn't include), add that code name to the keywords. Similarly, if a program gets renamed or a new top program comes into play, add it. This is an ongoing process.
- **Maintain Apify actors:** Scraping is the most brittle part (websites change, cookies expire, etc.). Make sure your Apify actors are running smoothly. If one source frequently fails, consider alternatives (maybe using a different scraper library or reducing frequency). Apify might have logs – check them if something isn't pulling data.
- **Claude/ChatGPT usage limits:** Keep an eye on how often you call the OpenAI node (or Claude). If daily, and especially if you expand to parse more things, ensure it's within your subscription limits. The prompts we use are not huge, so it should be fine, but just monitor API usage in your OpenAI dashboard.

- **Utilize Auto Claude for enhancements:** Now that the basics work, you can use Auto Claude to extend features. For example:
- If you want to incorporate FPDS/USAspending data (contract awards info) automatically, you could task Auto Claude: *"Create a Python script to query USAspending API for each program in Program DB and output any recent awards (amount, date, winner)."* Then integrate that data into either Program DB or use it in scoring (contractValue).
- If you decide to add a Slack notification for hot leads, you can have Auto Claude write a small integration (though n8n has Slack nodes too).
- If the org chart engine needs improvement, you can slowly automate more of it (maybe schedule a LinkedIn scrape monthly).
- Essentially, treat Auto Claude as your helper to build new pieces of code or workflow scripts whenever you identify a need.
- **Error recovery:** If the workflow fails one day (maybe due to a transient error), n8n might alert you, or you'll notice missing email. In such cases, you might manually rerun it or fix the issue. Build in redundancy if needed (e.g., if Notion API is down briefly, catch that and retry after 5 minutes within the workflow).

Remember, it's okay if not everything is 100% automated at first. You have built a **solid foundation** that saves you tons of time by doing the heavy lifting (data gathering and initial analysis). You can always layer more intelligence on top. For example, as a future step, you might incorporate a **"learning" component: if a BD outcome was successful or not, feed that back into scoring (reinforcement)** – but that's beyond initial scope.

Finally, keep your documentation (even simple notes) about how things are set up. That README in the repo is a good place to jot down changes or how to run stuff manually if needed. It helps if you step away and come back later.

Step 13: (Optional) Example Scenario Walk-through

Let's illustrate with a concrete (hypothetical) example to ensure it's clear:

- **Day 1:** At 6 AM, n8n triggers the Scraper. Apify scrapes 5 competitor sites + ClearanceJobs + LinkedIn. It finds 50 new job postings with clearances. The script consolidates them into `JobsDataset.json`.
- The Program Mapping script runs. Out of 50, it matches 30 to known programs (e.g., finds "Sentinel" in some descriptions, tags those to **GBSD Sentinel** program). 20 remain unmatched (maybe they are generic or new projects). Those 20 get `matchedProgram = "Unmatched"`.
- Scoring runs on all 50. Three jobs come out **Hot** (score, say, 85, 80, 90) – these might be jobs requiring poly clearance (pain high) for programs you're good at (fit high). 20 are Warm (50–79), and 27 are Cold (<50).
- Notion is updated: 50 new pages in "Jobs DB" – each with all details and their score and program. The Program DB may get some updates (like Program GBSD Sentinel now has "# Open Roles = 5").
- The workflow's Hot filter triggers the Playbook engine for the 3 Hot leads. For each, the OpenAI node creates an email draft. For example, for a hot lead for Program Sentinel, it drafts something like: "*Hi John, I saw that CompanyX is actively hiring for the Sentinel program – including several roles requiring polygraph clearance. That signals a critical talent need on a high-priority program. As a staffing partner with deep pools of cleared talent, we'd love to support you. We recently helped fill similar roles for the*

Minuteman program. Could we discuss how we can help your team hit mission goals faster? …" (plus whatever closing).

- The Email node sends out a summary to your inbox: "Hot Leads: 3 (Sentinel, JEDI, Omega Program). Warm: 20. Cold: 27. Top Hot Leads listed in a table with their details." It also might attach or include the draft outreach content for each hot lead, so you have it handy.
- You (human) review that email at 7 AM. You copy the outreach text, maybe tweak the greeting or add a personal touch, and then send it to the actual Program Manager via Outlook, etc. Or if you're confident, maybe eventually you have n8n send it directly – but baby steps!
- **Later that day:** You attend a meeting and hear about a new program "Phoenix Rise". That wasn't in your database. You go to Notion Program DB, add "Phoenix Rise" as a new entry, and maybe note it's run by Raytheon. You also update ProgramMapping_Config.json to add `"Phoenix Rise": ["Phoenix Rise"]` keywords. Now your system knows about it. Tomorrow, if a job comes up mentioning Phoenix, it'll catch it.

This scenario shows how the automation handles daily tasks, while you handle strategic decisions and relationship-building (with much more intel in hand). The **AI and automation are like your tireless research assistants**, doing overnight what could take a team of humans days to do manually.

Step 14: Final Tips and Next Steps

- **Be patient and iterative:** Treat this like building a model train – assemble piece by piece, test each car on the track. Don't worry if you have to adjust things; it's normal. Start with a small test (even limit to one source or one program) to ensure the pipeline works, then expand.
- **Safety and ethics:** When scraping and emailing, ensure compliance with terms of service and ethical use. Don't scrape sites that disallow it, and protect any sensitive data (e.g., don't expose personal info of contacts without permission). Use proxies if needed on Apify to avoid IP blocks for aggressive scraping [35].
- **Scaling up:** Once comfortable, you can add more sources (maybe Glassdoor jobs, or clearance-specific forums) and more sophistication (like a machine learning model for program matching, or integrating a CRM to log outreach activity). Auto Claude can help you write code for these as you define new tasks.
- **Community and support:** If you get stuck with Auto Claude or n8n, don't hesitate to check their docs or forums. Both have active communities. For Auto Claude specifically, since it's relatively new, you might find discussions on GitHub or Reddit [36] [37] (and the developer Andy might have usage examples). But the steps above should be sufficient without needing deep dive into Auto Claude's internals – we used it mainly to assist coding.

By following these steps, you've essentially built a mini-"AI BD Assistant" that *finds opportunities, analyzes them, and even drafts your outreach*. This is cutting-edge stuff, so kudos for taking it on! Keep refining the system, and soon it will feel like a natural part of your workflow.

You did it – end-to-end BD automation engine is in place! Now, each part (contact mapping, org charts, sequencing, etc.) will keep improving as you feed it more info and logic. And you'll be free to focus on strategy and human connections, while the machine handles the grunt work. Good luck, and enjoy the newfound superpowers in your BD efforts!

1 2 10 11 14 15 16 35 Proposed Architecture for Prime TS BD Intelligence Automation.pdf
https://drive.google.com/file/d/1uj6Q8Yawt9vWAWPcYrp8bLL_nSzWhC5Q

3 4 GitHub - AndyMik90/Auto-Claude: Autonomous multi-session AI coding
https://github.com/AndyMik90/Auto-Claude

5 compass_artifact_wf-171e29ab-2cd9-4026-9d43-f7428c87ae67_text_markdown.md
https://drive.google.com/file/d/1afiJfjUuR7XW8iX6FHP3zKXxUFl1YdEb

6 18 19 33 Prime_TS_BD_Intelligence_n8n_Workflow_Handoff.md
file://file-NiAiPdGPcKxqR1zeLd3ef4

7 8 24 25 26 27 28 29 30 31 32 34 Prime TS BD Intelligence System v2.1.json
file://file-WFrEoTQ8ekF3yxeJTaYhmq

9 20 GOOGLEDRIVEINTELLIREPO_STRUCTURE.xlsx
https://drive.google.com/file/d/1g_WXvjS2bHwelf012KVtqs2Cbk10mO9u

12 13 Deep Audit – Uncovered GitHub Repositories (Comprehensive Analysis).docx
https://drive.google.com/file/d/1kcrpj-z1QC99EQel_N99GMQAPqz1Clc1

17 21 22 23 Prime TS BD Automation – Final Project Artifacts.pdf
https://drive.google.com/file/d/1xeHVOFKRTnBc8yCT4SvIxJhmLe6H3G5U

36 Parallel AI Coding Sessions & Built-In QA: Auto-Claude ... - Reddit
https://www.reddit.com/r/vibecoding/comments/1q1ojwp/parallel_ai_coding_sessions_builtin_qa_autoclaude/

37 Claude Code 2.1 NEW Update IS HUGE! Sub Agents ... - YouTube
https://www.youtube.com/watch?v=s0JCE3WCL3s