ChatGPT

# Folder/File Setup Instructions

Organize your project files into a clear structure so you know where everything lives. Create a main project folder (e.g. `ProgramMappingEngine/`) with subfolders for data, code, outputs, etc. For example:

```
ProgramMappingEngine/
├── .acproject               # Auto Claude project config (tasks, settings)
├── data/                    # Source-of-truth databases and knowledge files
│   ├── Programs_KB.csv      # Program Knowledge Base export (DoD/IC programs)
│   ├── Contractors.csv      # Contractors/companies database
│   ├── Contract_Vehicles.csv# Contract vehicles master list
│   └── Contacts.csv         # (Optional) Contacts/Org chart info
├── scripts/                 # Code modules generated for each task
│   ├── job_mapping.py       # Script for program tagging logic
│   ├── contact_lookup.py    # Script for contact retrieval logic
│   └── ...                  # (Additional scripts per task as needed)
├── prompts/                 # Prompt templates (for reference or reuse)
│   ├── job_mapping_prompt.md
│   ├── briefing_prompt.md
│   └── ...
├── outputs/                 # Outputs for review
│   ├── BD_Briefings/        # Generated BD briefing docs (Markdown/PDF)
│   └── Logs/                # Logs or changelogs from runs
└── agents/                  # (Optional) Custom sub-agent definitions
    └── *.md                 # e.g. specialized agent roles if needed
```

- `.acproject`: This is the Auto Claude project configuration file. It will list your tasks (with titles and descriptions) and project-wide settings. You generally don't edit this by hand – the Auto Claude app manages it – but know that it tracks your tasks and progress. Think of it as the project's master checklist.

- `data/` **folder**: Put your reference data here. For example, export your Notion **Federal Programs** database to a CSV (`Programs_KB.csv`) and place it here as the knowledge base of programs (Program Name, Agency, Prime, Description, etc.). Do the same for any Contractors list or Contract Vehicles list if available. These files will serve as input for the AI to reference (you can also have the AI access Notion via API, but CSV/JSON snapshots make it easy for a beginner). If you update the Notion databases later, you can re-export or have a sync mechanism, but to start, static files are simplest.

- `scripts/` **folder**: This will house any code files the AI writes (e.g. Python scripts or workflow code). For instance, the program mapping logic might be implemented in `job_mapping.py` (or as

multiple smaller modules). Keep each major piece of logic in a separate file so it's easy to manage. Using clear filenames helps (e.g. `contact_lookup.py` for the org chart contact finder). The AI agents will create/edit these files as they complete tasks.

- `prompts/` **folder** (optional): You can store any prompt templates or examples here. This isn't strictly required, but it's a nice way to keep track of how you're instructing Claude. For example, you might save the prompt you use to generate a BD briefing in `briefing_prompt.md`. This way you have a library of prompts you can reuse or tweak without rewriting from scratch.

- `outputs/` **folder**: Use this to save the results of your pipeline for review. For instance, when the system generates a Business Development briefing document for a program, you can have it saved as `outputs/BD_Briefings/ProgramXYZ_Briefing.md`. Later, you can convert or print these to PDF as needed. Also, you might log important run information (like a processing log or the Auto Claude Changelog) in an `outputs/Logs/` subfolder.

- `agents/` **folder** (advanced, optional): Auto Claude allows **sub-agents** (specialized AI personas) if you want to break the work among "expert" agents (for example, a "Research Agent" vs a "Coder Agent"). Each sub-agent can be defined in a markdown file here (with YAML front-matter specifying its role, tools, etc.) [1]. For a beginner, you don't need to create custom agents immediately – the system's default approach will suffice. But as you grow, you could define agents like `planner.md` (for orchestrating sequence) or `analyst.md` (for summarizing results). These files would contain descriptions of the agent's role and any special instructions. Auto Claude will then delegate matching tasks to them automatically [2]. For now, you can skip this, but remember it's here for more advanced workflows.

In summary, this structure keeps your assets organized: **data** for inputs, **scripts** for logic, **outputs** for results, and the **project file/tasks** to tie it all together. This makes it easier to navigate your project as it grows – much like keeping school subjects in separate folders so you can quickly find your math homework versus your science project.

# ⚙ Configuration Checklist (Claude, GitHub, Notion, etc.)

Before diving into tasks, make sure all the key pieces are set up and talking to each other. Use this checklist to configure your environment:

- **Claude API Access:** Obtain your Anthropic Claude API key. In the Auto Claude app, plug this key into the settings (or set it in an environment file if required). This key lets the Auto Claude agents actually use Claude's brain via the API. Ensure you have a sufficient quota (the tasks may involve heavy analysis – ideally use Claude's 100k context version for big data, if available). In settings, select a model like *Claude 2 100k* for large context or GPT-4 if using OpenAI for some parts. (If Claude isn't enabled for code on your account, ensure you have Claude Code access.)

- **Notion Integration:** Set up a Notion integration token. In Notion, create an internal integration and note the **Notion API Key** (a long secret starting with `secret_...`). Share the relevant databases with this integration: e.g. your **Job Postings** database and **Program Knowledge Base** database. For each database, copy its **Database ID** (you can get this from the URL or by using the integration docs). You will use these IDs so the AI or your code knows which tables to query or update. *(Example:* In the snippet we have, the Program Mapping DB ID was `0a0d7e463d8840b6853a3c9680347644` [3].) Store the token and IDs in your project's configuration (somewhere accessible to agents, e.g. in a secure config file or environment variables).

- **n8n Orchestration Environment:** Install n8n (either locally via Docker/Desktop or use n8n cloud). n8n will be our "pipeline coordinator" [4] – it glues everything together. In n8n, set up a **Webhook node** to receive incoming job data. Configure Apify (or whichever scraping tool) to call this webhook URL with the job JSON. For example, Apify can be set to do a POST request to `https://your-n8n-instance/webhook/job-data-intake` after a run [5] [6]. Make sure n8n has the credentials it needs: add your Notion credentials in an n8n Notion node (so n8n can create/update pages), and any other API credentials (OpenAI, etc.) if you plan to call them from n8n.

- **Apify/Scraper Setup:** Ensure your job scraping agent (Apify or custom Puppeteer script) is configured to run on a schedule (e.g. daily or hourly) and output the expected JSON. The JSON should include fields like *job title, company, location, description, clearance, technologies, URL*, etc. (Your provided **Webhook Payload Template** shows the expected format [7] – confirm your scraper sends data in that structure.) Test the scraper manually to see that it grabs a few job postings and triggers the n8n webhook successfully. This will be the **trigger** to start the whole mapping workflow.

- **GitHub Repository (optional but recommended):** It's good practice to version control your project. Initialize a Git repository in your project folder. You can connect Auto Claude to push code changes to GitHub. If you create a repo on GitHub and set up git in your project, Auto Claude can automatically commit changes after tasks, and even generate a **Changelog** of what was done [8] [9]. This provides backup and traceability. If you prefer not to use GitHub, at least regularly back up your `scripts/` and `.acproject` files.

- **External AI/Embedding Services:** If you plan to use OpenAI for embeddings or GPT calls, get an OpenAI API key and plug it into your environment. For semantic search, OpenAI's `text-embedding-ada-002` is recommended [10]. Alternatively, set up a local embedding model or a vector database service:

- If using **Postgres/pgvector**: Install Postgres and the pgvector extension, and create a table for program embeddings. Prepare connection details (host, DB name, credentials) so the AI can use them.
- If using **Qdrant or Pinecone**: Get an API URL/key for the vector DB.

- *Note:* As a beginner, you might skip a dedicated vector DB initially and let the AI just compute similarities in-memory for a small number of programs. But if you have hundreds of program entries, a vector index is very useful for speed [11]. We'll include steps for both approaches in the tasks.

- **Environment Variables & Keys:** Collect all your keys (Claude API, OpenAI API, Notion token, etc.) in one place. Auto Claude might have a `.env` file or a settings UI where you can store these. For example, you might have:

- `ANTHROPIC_API_KEY=<your Claude key>`
- `OPENAI_API_KEY=<your OpenAI key>`
- `NOTION_TOKEN=<your Notion integration token>`
- `NOTION_DB_JOBS=<your Job Postings DB ID>`
- `NOTION_DB_PROGRAMS=<your Program DB ID>`

- and so on. Ensure your tasks can access these (Auto Claude's agents can be configured to load env vars if needed). **Never hard-code secrets** in the task descriptions or code; reference the env variables instead for security.

- **Notion Database Prep:** Double-check your Notion databases have the right **properties** (columns) to receive the data. For the Job Postings database, have properties for *Job Title, Company, Location, Clearance Level, Job Description, Source URL* (these will get filled from the scraper) and properties for the enrichment results: *Program Name, Prime Contractor, Customer Agency, AI Confidence Score*, etc [12]. You might also include *Status* (to track state like "raw_import", "pending_enrichment", "enriched", "needs_review") as mentioned in the integration plan [13] [14]. For Program Knowledge Base, ensure fields like *Program Name, Agency, Prime, Description, Key Locations, Keywords, Priority* exist [15] [16]. If you imported from CSV, you may need to tweak types (e.g. make Program Name the primary title property, maybe add relations between Job DB and Program DB if using Notion relations).

- **Batch and QA Settings:** Decide on your batching and QA strategy upfront and configure it. For example, n8n could be set to trigger enrichment every X minutes or when a certain number of new jobs are collected [17]. In the integration doc, they chose *every 15 minutes or >10 pending jobs* as a trigger [18]. Also, plan how to enforce the "manual QA after 10 records" rule. You might, for instance, run the first batch of 10 jobs and then **pause** the workflow (or not schedule the next batch) until you manually verify the outputs. This isn't a literal software switch but a process choice – as a beginner, you can simply not automate beyond 10 on day one. Then after checking quality, you can let it run continuously. We'll also mark low-confidence cases to a review queue in Notion so you can catch them easily.

Once all these pieces are configured, you have the **infrastructure ready**. Think of this like setting up your kitchen before cooking: you've got the oven preheated (Claude API ready), all ingredients on the counter (data in place), and the recipe laid out (n8n workflow scaffold). Now you're ready to let the AI chefs loose on the tasks!

# Claude Task Definitions

Below are the defined tasks that Auto Claude will perform to build the Program Mapping Engine. Each task has a **Title**, a detailed **Description** of what needs to be done (including requirements, constraints, and which files/data to use), and a recommended **Agent Profile** (which Claude model/settings to use for best results). You can enter these tasks into Auto Claude's Kanban (or task list) one by one. The descriptions are

written so that even a new developer (or a 10-year-old ) can understand what's needed. The AI agents will use these to generate code, configure workflows, or produce documentation as specified.

## Task 1: New Job Ingestion & Deduplication Pipeline

**Description:** Set up the initial pipeline that handles incoming job postings. When a new batch of job JSON arrives (via Apify webhook to n8n), the system should parse it and insert the jobs into the "Job Postings" Notion database. Implement the following in n8n (or as a small script if needed):
- **Validate Input:** Ensure the incoming JSON matches the expected schema (fields: title, company, location, description, clearance, technologies, URL, etc.) [7]. If the JSON is malformed or missing key fields, log an error or push it to a "validation failed" list.
- **Deduplicate Entries:** Avoid inserting duplicate job postings. A simple rule is to check if a job with the same *Company* + *Job Title* + *Location* was already processed in the last ~7 days [14]. Implement a lookup in the Notion Job database (or maintain a cache list) to skip duplicates. *Example:* if "Software Engineer at Acme Corp in DC" was inserted yesterday, don't insert it again today. You can use an n8n **Function** node or Notion search to find existing entries with those fields.
- **Create Notion Records:** For each unique job, create a new entry in the Notion Job Postings database with the basic fields. Map JSON fields to Notion properties one-to-one (e.g., *title* → **Job Title**, *company* → **Company**, etc.) [19]. Set an initial **Status** property to "raw_import" (meaning it's just ingested, not yet enriched) [20]. Also, if the Source URL contains certain indicators (like "sam.gov" or a known federal site), you might auto-tag a **"Source"** or **"Category"** property as "FEDERAL" vs "Commercial" as a hint (this was noted as an example in the workflow doc) [21].
- **Batch Handling:** The workflow should be ready to handle multiple jobs at once (the JSON "jobs" array). However, to avoid overload, process them in small batches (e.g. 5-10 jobs at a time). If more jobs come in, you can queue them by leaving their Status as "raw_import" for the next round. This task doesn't actually do the enrichment yet – it's just intake.
- **Output/Next Step:** The end result is that new jobs are in the Notion database with Status = *raw_import*. The next step (Task 3) will pick them up for enrichment. You should also update n8n's workflow to trigger that next step: for example, after creating records, set their Status to "pending_enrichment" or trigger a separate workflow that starts the enrichment for those entries [14]. We will design that enrichment trigger in another task.
**Key Files/Data:** This task primarily involves Notion and n8n configuration – no custom code file is expected. However, you might create a small script or function for deduplication logic if Notion's API filtering isn't used directly (e.g., an n8n Function node containing JavaScript to filter out dupes). If so, document that in `scripts/` (perhaps a snippet in a README). The Notion database ID for jobs (e.g. `@file:JobDB_ID` in config) will be used in the Notion nodes/API calls.
**Agent Profile:** *Claude 2 (100k)* with **Standard** reasoning is sufficient (the logic is straightforward). Emphasize reliability: use a deterministic approach (no creative writing needed). Set **Auto-Optimize** off (not a complex task) and a moderate temperature (the output should be precise). The agent should focus on producing a correct n8n workflow or script rather than long explanations.

## Task 2: Program Knowledge Base Prep (Keywords & Embeddings)

**Description:** Prepare the **Program Knowledge Base** so that the mapping engine can use it for matching. This involves two main parts: (A) building curated keyword lists for programs, and (B) setting up semantic embeddings for program descriptions. We'll create a script (e.g., `@file:scripts/program_kb_setup.py`) to perform these steps and save the processed data for use in

the mapping task. Requirements:
- **Load Program Data:** Read the programs CSV (`@file:data/Programs_KB.csv`) which contains fields like Program Name, Customer Agency, Prime Contractor, Description, Key Locations, etc. [15] [16]. Parse this into a list or dictionary in Python. Each program entry in memory should hold those attributes. (If the data is accessible via Notion API, you can fetch it directly – but for now, assume we use the CSV for simplicity.)
- **Build Keyword Dictionary:** For each program, compile a set of keywords/phrases that are strong clues for that program [22] [16]. This should include:
- Program names and common abbreviations (e.g. "Ground-Based Strategic Deterrent" and "GBSD" for the Sentinel program).
- Any project code names or nicknames from the data.
- Prime contractor names (e.g. if a job's company matches the program's prime, that's a hint) [23] [16].
- Key locations (if provided, e.g. "Huntsville" for a missile defense program) [24] [25].
- Technical keywords unique to the program (from the Description or a "Technologies" field – e.g. a program involving satellites might have "SIGINT" or a system name) [16] [25].
- Known subcontractors or partner companies involved (if listed, those names can be clues) [26].
Assemble these into a master dictionary structure, e.g. a Python dict where keys are program IDs and values are sets of keywords. Also create reverse indexes if helpful (like a dict mapping each keyword to relevant program(s)). This will be used for quick lookups during rule-based matching. Make sure to handle case variations and maybe common suffixes (e.g. "Program" vs no "Program" in name).
- **Compute Program Embeddings:** Using an embedding model (like OpenAI ADA or similar) generate a vector for each program's description [27] [28]. The description text is crucial for semantic matching, as it provides context. If the description is very long, consider summarizing it first or truncating to the most relevant parts (to stay under token limits). Store the resulting vector for each program. You can simply keep them in memory for now or save to disk (`program_embeddings.npy` or a JSON file). If using a **vector database** (e.g., Postgres with pgvector or Qdrant), this script should also handle uploading these vectors to that DB. For example, connect to the DB and upsert each program with its vector. Ensure you also store an identifier (like Program Name or ID) alongside the vector so we can retrieve which program a similar vector corresponds to.
- **Testing the Setup:** After building the dictionary and vectors, include a small test in the script: pick a sample program and verify that some known keywords are in the dictionary, and that an embedding vector was created (maybe print the vector's length or first few components to confirm). This ensures our data is ready.
- **Efficiency Consideration:** This task is run occasionally (whenever the program list updates). It's fine if it takes a bit of time (embedding can be slow if many programs). We just need to run it before processing jobs. For now, assume maybe a few dozen or hundred programs – manageable to embed in a single run.
**Key Files/Data:** Input is `Programs_KB.csv` (and possibly other reference files like `Contractors.csv` if needed for company names). Output can be two things: `@file:data/program_keywords.json` (storing the keyword dictionary) and `@file:data/program_embeddings.json` (or `.npy` if binary) storing program vectors. If using a real vector DB, then the "output" is data loaded into that DB – still, log what you did. We will reference these outputs in the mapping task.
**Agent Profile:** *Claude 2 100k* with **UltraThink** mode ON – because we want Claude to carefully analyze potentially lots of text (program descriptions) and generate a comprehensive dictionary. UltraThink gives it a higher "thinking budget" to not miss details [29]. Enable **Auto-Optimize Phases**, because this task can be split: one phase to generate code that builds dictionaries, another to generate code for embeddings, etc., which Claude can optimize step by step. We expect code output (Python script), so use **Claude Code** style. Temperature low (we want consistent results, not creative variance). This is a setup task that benefits from thoroughness and error-checking.

### Task 3: Job→Program Mapping Engine (Enrichment Logic)

**Description:** This is the core task – create the logic that takes a job posting and determines which program it most likely belongs to. We will implement this as a Python script (`@file:scripts/job_mapping.py`) or module that can be called for each job (either by n8n or manually). The expected behavior: for a given job (with title, company, location, description, etc.), output the best-fit Program Name, Prime Contractor, Customer Agency, and a confidence score, plus any other tags. Key requirements:

- **Input & Preprocessing:** The function (e.g. `map_job_to_program(job)`) should accept the job's details (either as separate parameters or a dict). Begin by preprocessing the text: clean the job title + description by lowercasing, removing stopwords or irrelevant symbols, and standardizing terms (e.g., "U.S." -> "US", unify capitalization of acronyms) [30]. Also note down important attributes like required clearance level; for example, if the job requires "TS/SCI with poly," that strongly hints it's an intel community program, which might narrow the search space (you can use clearance as a filtering heuristic – maybe only compare against programs in agencies that use that clearance level).

- **Dictionary Keyword Matching:** Use the **keyword dictionary** prepared in Task 2 to scan the job text for any direct hits [31] [25]. For each program's keyword set, check if any keyword appears in the job title or description. Implement a scoring system: e.g., if a program name or acronym is mentioned exactly, that's a big score (say +5) [32] [33]. If a prime contractor name matches the job's company, that's a clue (maybe +3) [34]. If a key location matches the job location, +2 [25]. Technical terms, +1 each. You can get creative with the scoring but keep it simple and based on counts or presence. Accumulate scores for each candidate program. The result will be a shortlist of programs that got any score, with their total points. If one program stands out with a high score (especially if it had a direct name match), you might even decide to select it immediately as a confident match [35] [36] (perhaps still do the vector step for confirmation).

- **Semantic Vector Search:** In parallel or after scoring, perform the **embedding similarity** search [28] [37]. Take the embedded vectors from Task 2 for all programs. Embed the job posting text (you can combine title + description into one string for embedding). Compare this job vector to each program vector to find the top N closest programs (e.g., top 3 matches) [24] [28]. If using a vector database (like pgvector/Qdrant), query it with the job vector to get the top hits quickly. If not, just compute cosine similarity in Python for each and sort. Each retrieved program comes with a similarity score (0 to 1). Translate these into a semantic confidence score. For instance, if similarity > 0.85 that's very high confidence semantically. Lower scores are weaker.

- **Combine Signals:** Now merge the insights from the keyword and the vector approaches. Ideally, one program will be strong in both [38]. Use logic like: if a program appears in both lists, boost its confidence significantly. If the top keyword match and top vector match are the same program, that's your winner (with high confidence) [39]. If they differ, compare their scores. For example, maybe no keywords matched Program X, but the vector similarity for Program X is 0.9 – that suggests Program X strongly, whereas the keyword method might have flagged Program Y with a couple of weak hits (like a contractor name) [40]. In that case, Program X likely is the better choice. Conversely, if keywords clearly mention Program Y by name, but vector match wasn't as high (maybe the description phrased differently), lean towards Program Y due to the direct clue. Implement a simple decision: choose the program with the highest "combined score." You can create a composite score (e.g., weight keyword points + semantic score*10 or something comparable). Document your*

*method in comments so it's explainable. If the scores are close or ambiguous (e.g., two programs tie or it's not obvious), you might pick the top one but mark the result as low confidence. Also consider if multiple programs** could apply: in most cases we want one best guess, but if two are very close, maybe note the second as a backup suggestion (this could be stored in a "Tags" field or ignored for now) [41].

- **LLM Validation (optional):** As an extra safety net, if the confidence is below a threshold (say < 0.5 on a 0-1 scale), consider calling Claude or GPT-4 to double-check [42]. For example, have a prompt that says: *"We have a job posting text: … The top two candidate programs are X (description…) and Y (description…). Which program does this job most likely belong to?"* [43]. Parsing the LLM's answer can give a final decision. This is optional and uses more tokens; you might implement it later if needed. The task for now can leave a hook for this (like a function `validate_with_llm(candidates)` that can be filled in or called conditionally).

- **Confidence Scoring:** Whichever program is chosen, assign a final confidence score. You can normalize your composite score to 0–1 or use labels High/Med/Low. For instance: High if it was a direct match or very high vector similarity (>0.8 and keywords hit), Low if it was a stretch or only weak clues. This confidence will be stored in the Notion field "AI Confidence Score" [44]. It helps in QA (low confidence ones will go to manual review).

- **Program Details Lookup:** Once the program is determined, retrieve its official data from the knowledge base. From our program list (or via Notion API), get the exact Program Name (to ensure consistent naming), the Prime Contractor name, and Customer Agency [45]. These are the fields we'll write back to the job's record. If the program was not found at all (no match), then the script should return a result indicating "No match" or set a flag for review. Similarly, if the program exists but, say, prime contractor or agency info is missing in the DB, handle that gracefully (leave blank or fill "Unknown").

- **Output:** The script should output a structured result, e.g. a Python dictionary or JSON like:
`{ "Program Name": "...", "Prime Contractor": "...", "Customer Agency": "...", "Confidence": 0.87, "Secondary Candidates": ["..."], "Notes": "Direct keyword match on acronym; high semantic similarity." }`.
We will use this output in the next task to update Notion, so ensure keys align with Notion property names. Also, log or print some rationale if possible (for debugging, it's useful to know why it chose that program – you could include a brief "Notes" as shown). The logic itself can also be tested with a few sample jobs to see if it picks reasonable programs.

**Key Files/Data:** This will use the outputs from Task 2: the `program_keywords` dictionary and program `embeddings`. So import those from file or module. It will also refer to environment variables for any API calls (OpenAI API for embedding new job text, or vector DB connection if used). The code goes into `job_mapping.py`. We might also maintain a config section at top (like a threshold for confidence, or list of top-priority programs to boost as mentioned in the blueprint for Top 20 programs [46] [47] – optionally, if a matched program is in "Top 20" we can boost it slightly as per strategy). Make sure the code is well-commented, since as a beginner you'll rely on those comments to understand the flow.

**Agent Profile:** *Claude 2 100k*, **UltraThink** mode. This is a complex, multi-step coding task that benefits from Claude planning carefully [2]. The agent may break the solution into parts (parsing, keyword match, vector search, etc.), which is good. Allow it to auto-optimize in phases if it chooses (it might first outline the

approach, then fill in code). The context window should be large enough to hold some sample data as examples. Temperature low-medium (0.2–0.3) to ensure determinism in code, but we do want Claude to be a bit flexible in reasoning (it's combining heuristic and algorithmic approach). Encourage it to test the code within its environment if possible. We want a correct, efficient implementation more than a quick one – so the profile should favor accuracy and thorough reasoning. (In practice: Claude Code with default or slightly extended reasoning is fine, as long as UltraThink is on to avoid shallow answers.)

### Task 4: Update Notion with Enrichment Results

**Description:** Now that we have a mapping result for a job, we need to update the Notion database entry with this information. This task is about integrating the output of Task 3 back into Notion (or whichever database you use). Essentially, for each job record that was "pending_enrichment", fill in the Program Name, Prime Contractor, Agency, Confidence, etc., and mark it as "enriched". We will implement this as part of the n8n workflow or as a separate script. Steps:
- **Receive Result:** Assume Task 3's code (the program mapping script) will be called by n8n for each job (or batch of jobs). n8n can execute a code node that runs `job_mapping.py` on a job and gets the result. Now, within n8n, take that result (the JSON/dict with program info) and map it to the Notion update. If not using n8n, we could write a small function in `job_mapping.py` to call Notion API directly – but using n8n's Notion nodes is simpler.
- **Map Fields:** Ensure the keys from the result align to Notion properties. For example: - Program Name (text or relation property in Notion) ← result["Program Name"]
- Prime Contractor (text property) ← result["Prime Contractor"]
- Customer Agency (text or multi-select property) ← result["Customer Agency"]
- AI Confidence Score (number property) ← result["Confidence"] (likely multiply by 100 if you want a percentage or just store 0–1) [48].
- Tags/Secondary (maybe a multi-select or text list) ← result["Secondary Candidates"] if you want to store those (optional).
- Any flags or notes (if "No match" or low confidence, perhaps set a **Status** to "needs_review" as opposed to "enriched"). Use the Notion *Update Page* endpoint or n8n's Notion Update node to write these fields for the correct page (you need the Page ID of the job entry – n8n will have it from when it created the page initially).
- **Update Status:** Change the job's **Status** property from "pending_enrichment" to "enriched" or "needs_review" accordingly [49]. Our logic: if confidence is high or medium, mark enriched; if low or no match, mark needs_review so it surfaces in the human review queue [50]. If an error occurred during mapping (like the script failed), you might set Status = "error" for that entry and handle it later.
- **Batch Processing:** If processing in batch (say 10 jobs), loop through each job's result and update each page. This could be done with multiple Notion update calls. It's fine if it's a bit slow (a few seconds per update). If using n8n, you might have a loop node or simply feed the array of results into a Notion node configured to update many pages.
- **Verification:** After updating, maybe read back one of the pages to ensure the fields were set correctly (this can be done manually via Notion UI as well). It's important to double-check that the names match exactly. For instance, if "Program Name" is a relation property to the Programs database, you might need to supply the Page ID of the program entry rather than just text. As a beginner, if that's too complex, you can keep Program Name as a plain text property for now. (Relations are nicer long-term, but require finding the right relation id for each program). So for simplicity, consider Program Name a text field in Notion that you just write the name into.
- **Logging:** Have the system log what it updated. For example, print a line: "Updated Job ID X with Program Y

(Confidence 0.9)". These logs could later compile into a **Changelog** automatically by Auto Claude's features (which we'll see in the Changelog panel).

**Key Files/Data:** No new file for this logic if using n8n's built-in nodes. If we write a script, it could be part of `job_mapping.py` or a separate small script `update_notion.py`. It will use the Notion API – meaning it needs the Notion database IDs and the job's page ID. The page ID can be passed from the ingestion step. In n8n, it's straightforward since the page ID comes from the Create node and can be passed along. Make sure the integration token and database permissions are correct; test updating one field on a test page first.

**Agent Profile:** Use *Claude Instant* or a standard Claude model here (doesn't require huge context or intense reasoning). The task is more about making no mistakes in field mapping. A **Profile** like "Claude 1 with high reliability" or even GPT-3.5 could do, but to keep things consistent, use *Claude 2* with a **focused profile** (don't need UltraThink). Set temperature to 0 (we want a precise mapping code or instructions, not creative text). Essentially, this agent will either write a short piece of glue code or just confirm the n8n node configuration. It's a straightforward step, so no long reasoning needed.

### Task 5: Org Chart Contact Extraction

**Description:** Automate the retrieval of relevant contacts (people) for the program associated with each job. The idea is once we tag a job with Program X and Prime Contractor Y, we want to find who are the key players related to that program or company – e.g., Program Managers, Hiring Managers, capture leads, etc. This task will likely produce a script (`@file:scripts/contact_lookup.py`) that, given a Program name and Prime contractor, queries a contacts database or external API (like a CRM) to get names and roles. Requirements:
- **Data Source for Contacts:** Identify where the contacts are stored. Possibilities: an internal CSV (`Contacts.csv`) you provided (maybe from ZoomInfo or similar) or an external system like Bullhorn CRM. We have files like `GDIT PTS Contacts.csv` and `DCGS Contacts Full.csv` in the data – these seem like specific lists of people for certain programs or companies. For now, use whatever data you have. For example, if `Contacts.csv` contains entries with fields [Name, Title, Company, Program, Email, ...], we can filter that. If using an API (Bullhorn), the script would call the API with the program or company as query. Since this can be complex, start with the offline data approach for a beginner.
- **Lookup Logic:** When run for a particular job's program tag, perform two lookups:
1. **By Program Name:** Find any contacts in the Contacts data whose "Program" field matches the program (or related keywords). For instance, if the program is "Sentinel GBSD", look for contacts where program or project field has "GBSD" or "Sentinel".
2. **By Prime Contractor:** Find contacts whose current or past company is the prime (e.g., "Northrop Grumman"). Specifically target those likely associated with that program (their title might contain the program name or relevant division).
Combine results and then filter for relevance – prioritize people with titles like "Program Manager", "Project Lead", "Recruiter for X program", or similar. If the contacts data doesn't have program info, you might rely on titles and company. *Example:* For Program X at Company Y, good contacts might be: anyone at Company Y with a senior title in that technical domain, or if program name is mentioned in their profile.
- **Output Contacts:** Structure the output as a short list of key contacts with basic info. For example, the script could output 2-5 names along with their title and maybe a LinkedIn or email if available. If no specific contacts are found, think of roles to target instead (like "No direct contacts found; consider reaching out to Company's Director of Programs or LinkedIn search for Program X managers"). But since we want automation, let's assume some will be found in our data.

- **Integration with Notion:** You might have a People/Contacts database in Notion to store these. The integration plan mentioned populating a "People/Org-Chart database" with results [51]. If such a database exists (perhaps you have one for org charts), the script can create new entries there or link existing ones. To keep it simple: the script can just return the list of contacts, and you can manually review or insert them. For automation, you could have another Notion relation on the Job entry linking to contacts, but that's advanced. We'll focus on gathering the data first.
- **API Option (Advanced):** Optionally, outline how to query an API like Bullhorn: e.g., use Bullhorn's REST API with a query like "program name OR prime contractor name" in candidates or contacts search. Given this may require API credentials and specific knowledge, we can leave hooks (like a function `query_bullhorn(program, company)` that returns dummy data for now or is left to be filled). The blueprint suggests this integration in Phase 4, using an existing extraction plan [51] – but as a first implementation, the CSV lookup is fine.
- **Safety Checks:** Ensure not to pull personal data that isn't allowed. If using internal data, it's fine. If calling external API, abide by their rate limits and privacy rules. As a beginner, stick to provided data.
- **Result Formatting:** For each job (or program), we might attach the top 1-3 contacts. The output could be like: "Contacts: Jane Doe – Program Manager at CompanyY (jane.doe@companyy.com); John Smith – Technical Lead, CompanyY (LinkedIn: ...)" etc. This info will feed into the BD briefing in the next task, so either store it in Notion (maybe as a text field or relation) or just include it in the briefing generation step directly.

**Key Files/Data:** Likely uses `@file:data/Contractors.csv` (to verify company names), and `@file:data/Contacts.csv` or similar files (like the ZoomInfo exports). For example, if `DCGS Contacts Full.csv` is a list of contacts related to the DCGS program, our script can use that when program = "DCGS". If there are multiple contact files per program, consider merging them or choosing based on program name. We might create a unified `Contacts.csv` combining these or load multiple files. Document assumptions in the code comments (e.g., "using DCGS contacts file for DCGS program lookup"). The output of this task may be directly used by the next task (briefing generation) rather than stored, depending on how you design it.

**Agent Profile:** *Claude 2*, standard mode (no need for UltraThink unless the data is very large). The complexity here is moderate – mostly filtering and matching text fields. Use a **focused coding** style. Temperature low. The agent should produce a Python script that is straightforward (read CSV, filter, return list). Use of pandas library is acceptable to simplify CSV handling (but not required; standard CSV reader and loops are fine). The key is correctness. We also might want the agent to generate some example output for a test program to ensure it works. So encourage it to include a simple `if __name__ == "__main__":` block to test the function with a known program (if feasible). This way, as a beginner, you can run that script independently to see if it prints expected contacts.

### Task 6: BD Briefing Document Generation

**Description:** Create an automated **Business Development briefing** for each enriched job/program. This is a Markdown report that consolidates everything we've gathered: the job details, the program context, relevant contacts, and any BD "signals" or recommendations. The goal is a 1-2 page brief that a BD person can quickly read to understand the opportunity. Steps to implement:
- **Gather Inputs:** For a given job entry (now enriched with program info), collect the pieces needed for the briefing. This includes: - Basic job info: Job Title, Company, Location, Clearance level, perhaps a brief summary of the role (you can use the job description but usually it's long; maybe just first few lines or a

summary of it). - Program information: Program Name, Customer Agency, Prime Contractor, and a short description of the program's mission or scope. You have the program description in the knowledge base – you might want to include a short blurb on what the program does (to give context *why this job exists*). - Contract vehicle or type (if known): If your data or logic identified a contract vehicle or contract number (sometimes job postings mention the contract name/number). If the job Source was a federal site or if we have a guess at the contract vehicle (maybe from matching in the program DB), include it. If not, omit or mark as unknown. - Contacts: The key contacts identified (from Task 5). List their names and roles. - Business signals: This is more analytical – include things like: *"This is a new opening on Program X, which indicates the prime contractor is staffing up. Program X is a high-priority program for Agency Y (e.g., Air Force) with an upcoming recompete in 2027."* If you have data like contract end dates or incumbent info from the program DB or elsewhere, mention it. (E.g., if Program X contract is ending in a year, that's a recompete opportunity). - Competitive landscape: If known, mention if this job is with the incumbent prime or a subcontractor. For example, *"Prime Contractor: CompanyY (incumbent). Known competitors: CompanyZ also has roles on this program."* You might get competitor info from the program DB (maybe a field of known subcontractors or competition). - BD Recommendations: A short list of next steps or tips for the BD team, such as: "Reach out to Jane Doe (Program Manager) for a introduction", "Emphasize Company's past performance in similar programs during outreach", "Monitor SAM.gov for RFPs related to Program X's next phase", etc. These can be somewhat general suggestions, fine-tuned if you know specifics. For a start, a few boilerplate next steps are okay. - **Markdown Formatting:** Structure the briefing in Markdown with clear sections: - **Opportunity Overview:** A paragraph summarizing the job and what it means (e.g., "Company Y is hiring a Software Engineer for Program X (Air Force's _ ***program) in Colorado Springs – indicating new work on that program."). - Program Background: A paragraph about Program X (what it is, who runs it, prime contractor). - Key Details: A bullet list of key info (Prime: _,*** **Agency:** , Location: **,** **Clearance:** , Contract Vehicle: __, etc.). - **Org Chart/Contacts:** A bullet list of names and titles (and maybe contact info). - **Business Development Notes:** Another list or paragraph with the signals and recommendations (e.g., timeline of contract, suggestion to meet contacts, etc.). - **Automation:** Use Claude (or GPT) to generate parts of this text. We can feed it the structured data and have it produce nicely worded sentences. You'll create a prompt that instructs the AI to act as a BD analyst and produce the write-up (we will include a template in the Starter Prompt Library). The script for this task could be minimal: essentially format the data into the prompt and call the LLM API (Anthropic or OpenAI) to get the output. Or, since we have Claude in the loop via Auto Claude, you might directly use Claude in the agent to write the doc. However, for end-to-end automation, coding it as an API call is useful. For now, outline the approach. If not coding the call, you might just rely on manually running a prompt in Claude using the template – but we want to integrate it. - **Output to File/Notion:** Decide where the briefing goes. Two options: 1. **Notion**: Create a new Notion page (in a Briefings database or as a sub-page in the Job entry) and fill it with the markdown content. Notion can then export to PDF if needed. This keeps everything in one place for the team. 2. **Markdown file**: Save it in `outputs/BD_Briefings/Job123_Brief.md`. This is straightforward and you can later manually copy it or convert to PDF. As a beginner, writing to a file may be easier to implement. We can do that, and also copy the content into a Notion page manually if desired. - **Quality Check:** Ensure the generated text is accurate and not hallucinating details. Because the AI might try to infer things not in input. To mitigate this, provide it with facts (like contract end date if known, or explicitly say "if unknown, state unknown"). Use a consistent tone: professional and concise. Maybe 3-5 sentences per section, bullet points for lists – easy to skim (remember the user wanted readable output). - **PDF conversion:** If you need PDF, you can use a tool or Notion's export. But that's outside the scope of AI tasks – likely you'll do that manually or via a script later. We'll mention it but not implement conversion in code right now.

**Key Files/Data:** The input comes from the Notion enriched job record and the program DB, which we already have. The contacts from Task 5 are also input. If you created an `outputs/BD_Briefings` directory, the script will write a file there. For example, `ProgramX_Briefing.md`. No new data files required, but ensure the markdown includes relevant sources (maybe embed a link to the job posting URL or program page for reference). If we have a template prompt, you might store it in `prompts/briefing_prompt.md` and have the script read it – or just include it in code. Up to you; storing separately makes tweaking easier.

**Agent Profile:** *Claude 2 100k*, possibly with **Creative** or **Business** tone profile. This task involves natural language generation, so we want Claude's strength in writing. We'll give it a lot of context (job info, program info, contacts), which could fit in the context window easily. Use a slightly higher temperature (maybe 0.5) to allow a more fluent narrative, but not too high to avoid made-up info. UltraThink not required; the heavy analysis was done in mapping. We want this to be more of a "report writing" mode. If Auto Claude has a preset for documentation or report generation, use that. Otherwise, instruct Claude to be thorough yet concise. This agent might not produce a lot of code – more likely it will produce the markdown text itself or a function that prints markdown. We can actually let Claude output the final briefing content as the "task result". But since we might want to do it for many jobs, coding it might be better. Perhaps the agent can generate a template script that formats a given job's data into markdown. However, since writing is Claude's forte, an alternative is to have an interactive session for each briefing. Considering automation: let's plan to script it. The agent will thus write code that takes structured data and returns a markdown string, possibly calling itself (the AI) for the heavy lifting via API. This is complex to code (calls back into AI), so maybe simpler: skip the code and just use Claude directly when needed. For now, we define the task such that the AI knows the structure and can be prompted accordingly. **In summary**: the agent should deliver either a filled template or instructions for how to prompt Claude to get the briefing. We'll supply a prompt in the library for manual use too.

### Task 7: Workflow Orchestration & Sequencing

**Description:** Ensure all the pieces work together in sequence, mostly through the n8n workflow (or any orchestrator). This task is about designing and implementing the control flow: how do we go from scraping to final briefing with minimal human intervention. It's partly planning/documentation and partly implementation in n8n. Steps/considerations:
- **n8n Workflow Design:** Lay out the nodes and triggers in n8n to cover the entire pipeline: 1. **Trigger:** Webhook start (from Apify) with new jobs (Task 1 handles this). 2. **Ingestion:** Nodes to validate and create Notion records (Task 1). 3. **Set Status to pending_enrichment:** done in Task 1 after insert. 4. **Enrichment Trigger:** Either a schedule (Cron node every 15 min) or a workflow trigger node that watches the Notion DB for new "pending_enrichment" entries. Decide on one (the plan was every 15 min or if >10 jobs waiting) [17]. Cron is easier to start with. 5. **Enrichment Loop:** For each pending job (you can use a Notion "search/read" node to get all pages with Status=pending_enrichment), pass them one by one (or in batches) into a function node that calls our `job_mapping.py` logic (Task 3). This could be done by an **Execute Command** node (to run a local Python script) or by an **HTTP Request** node if you deploy the logic as an API service. Simpler: use an n8n Code node to replicate what our Python does (since our code is Python and n8n code nodes are JavaScript, you might actually prefer using the Python script via command line). Alternatively, we integrate the Python logic into the Auto Claude agent pipeline and not have n8n call it – but then hooking back to Notion is needed. To keep on track: use Execute Command to run `python job_mapping.py` with the job data as input (maybe passing an ID and the script itself fetches from Notion). 6. **Update Step:** After getting mapping results, use Notion Update nodes to write the data back (Task 4). 7. **Org Chart Step:**

Optionally, add a node that triggers `contact_lookup.py` after enrichment, and adds the found contacts to the output or maybe updates a field "Contacts Identified" in Notion with a short text or count. This could also feed into the briefing. 8. **Briefing Step:** Finally, for each enriched job, trigger the briefing generation. You might do this on a separate schedule or immediately. Perhaps a simpler approach: have a manual trigger or a separate workflow that you run at the end of the day to generate briefings for all new enriched jobs. This way you can QA the tags first. But if you want full automation, you can integrate it: e.g., once a job is enriched, fire a "Call Claude to generate briefing" step. This could be an HTTP node calling the Claude API with our assembled prompt (Task 6). 9. **Output Delivery:** Decide how you get the briefing to the user. If in Notion, it's already there. If in a file, maybe email it or notify someone. n8n can send an email with the briefing attached or Slack message, etc. For now, just store it. 10. **End/Repeat:** The workflow would mark the job as done. The Cron will catch the next batch next time.

- **Error Handling:** Incorporate error paths. For example, if the mapping script fails for a job, catch that error (n8n has a failure branch) and update the job Status to "error" with a note. Maybe send yourself an alert. Also, if an API call fails due to rate limit, you might retry (n8n can retry, or put in a Wait and retry).
- **Manual QA Checkpoints:** The user wanted a checkpoint after 10 records. To implement: one approach is not purely technical but procedural – e.g., run the first part of the workflow manually for 10 jobs, then pause. But we can bake something in: maybe a counter of how many have been processed today and if >10, route to a pause. n8n doesn't have a built-in "pause until manual" except the workflow can stop. Alternatively, use a **flag** file or variable. A simple idea: the Cron that triggers enrichment only runs if a "QA Approved" flag is true. Initially false after 10. This might be overkill. Instead, consider: **first run mode** – process 10, then disable the Cron (which you do manually). The system itself can list how many it enriched; you can check Notion dashboard to see count. (In the **Monitoring Dashboard**, you might track total enriched [52] [53].) For now, plan to handle this manually: after 10 enrichments, review them in Notion's *Needs Human Review* view if any [54], adjust anything, then continue. We mention it so the team knows to do it.
- **Testing the Orchestration:** Simulate the flow with a dummy input. Use an example job JSON, run it through the workflow step by step watching each node's output. This will highlight any integration bugs (like incorrect field names or script path issues). Iterate until smooth.
- **Documentation:** Within the .acproject or an attached Markdown, document the workflow with a diagram or list of steps (like above). Auto Claude's **Roadmap** feature can help visualize it. Also, the **MCP (Master Control Panel) Overview** might show all agents and tasks status at a glance – ensure our tasks are marked complete in the project once done, and any ongoing processes (like Cron triggers) are noted somewhere.

**Key Files/Data:** No new code file except possibly an n8n workflow export (which is a JSON defining the workflow). If possible, export the n8n workflow and save as `@file:ProgramMappingWorkflow.json` for reference. Otherwise, capturing the design in text is fine. This task uses all previous components, so it doesn't produce new artifacts but ties them together.

**Agent Profile:** *Claude (any)* – This is more of a **planning/documentation** task than heavy coding. You might run this in a **Planner/Architect agent** persona. For example, a profile that excels in outlining and verifying workflows. UltraThink could be useful to cover all edge cases. The agent should output either a clear written plan or even pseudo-code for the orchestration. Temperature low (accuracy). Essentially, we want to be sure nothing is missed – so instruct Claude to be meticulous. Possibly use the **Ideation** feature here: have Claude review if there are any steps or dependencies we forgot. (This is a good place to apply Auto Claude's *Ideation* to double-check our approach – see the features section later.)

## Task 8: Quality Assurance & Feedback Loop

**Description:** Implement mechanisms to maintain and improve the system's accuracy over time. This task focuses on two things: (A) the immediate QA gating for low-confidence outputs, and (B) setting up a feedback loop for continuous learning. The goal is to prevent bad data from slipping through and to make the engine smarter with each batch. Steps:

- **Confidence Threshold & Human Review Queue:** Use the confidence score from Task 3 to decide if a job needs human attention. For example, if Confidence < 0.5 or the mapping script was unsure, mark the job's Status as **"needs_review"** [49]. In Notion, create a filtered view (perhaps called *Needs Human Review*) that automatically shows any job with that status or with a low AI score [54]. This way, a BD analyst can periodically check that view and see all items that need manual tagging or verification. Also, define other **criteria for human review**: e.g., if the job's required clearance is very high (TS/SCI w/ poly) or if a new company appears that isn't in our Contractors DB, those might be flagged too [55]. We can implement some of these: the mapping script could recognize "new prime contractor" (not in Contractors.csv) and set a flag field "NewCompany=True", and your Notion view can filter on that. Outline these rules clearly (maybe as a comment in code or a Notion database guide).
- **Batch QA Pause:** As discussed, after the first batch (10 jobs) and occasionally thereafter, do a manual QA pass. The system can facilitate this by producing a **summary of the latest batch**. For instance, an n8n step or a script can count how many jobs were enriched and how many went to review, and output a short summary: "Batch of 10 processed: 7 tagged, 3 flagged for review." This summary could be posted somewhere (Notion log or even emailed to the team). In Notion, you might have an "Enrichment Runs Log" database to which you add an entry each day with stats [52]. Automating that is nice-to-have. You could implement a node that counts and writes to such a log.
- **Collect Human Feedback:** When a human corrects something (e.g., they change the Program Name for a job, or they fill in a field the AI left blank), capture that. How? One approach: a "Feedback" table where analysts can note errors, or simpler, just periodically filter the jobs where the Program Name was manually changed and examine those. We can automate detection: the system could store the originally suggested program in a hidden field, and if later the Program Name field is different, that implies a human override. The script can scan for such discrepancies. For now, document that as a process: *"Review any cases where Program mapping was changed by a user, and add those instances to the dictionary or adjust logic."* This may not be automated in code initially, but we should plan it.
- **Update Knowledge Base:** Use feedback to update `Programs_KB.csv` or the Notion Program DB. For example, if a job was tagged "unknown" but a human identified it as Program Z which wasn't in our database, we must add Program Z to the knowledge base (with whatever info we can gather) so next time it's recognized [56]. Likewise, if our keyword dictionary missed a term that the human noticed, add it. Maybe maintain a *"FAQ"* or *"Known Issues"* doc to track these until you have enough to retrain or update systematically.
- **Model Fine-tuning (future):** Mention that over time, if you log enough corrected mappings, you could train a small model or fine-tune an existing one on this classification task [57] [56]. That's beyond the current scope (and not needed with powerful APIs available), but it's an option for long-term improvement without always relying on expensive API calls.
- **Monitoring Dashboard:** As an optional subtask, set up a Notion dashboard with key metrics (some were suggested: success rate, avg confidence, error rate, etc.) [53] [58]. This isn't coding per se, more of configuring Notion views and maybe using rollup formulas. For example, a formula to calculate % of jobs with Confidence > 0.8, or a rollup count of jobs by Program to see which programs are tagged most. Doing this helps spot anomalies (e.g., if one program is getting a lot of tags suddenly, maybe investigate if false positive). Document some metrics you decide to track, so stakeholders know if the engine is performing.

**Key Files/Data:** Mostly uses existing structures. Possibly create a `feedback_log.csv` or add to the `Enrichment Runs Log` in Notion if one is used. In the code, maybe an update to `program_keywords.json` if new terms are found. This task might produce documentation rather than code – e.g., a Markdown file "QA_Plan.md" where you write these rules out for the record. That could be useful for onboarding others.

**Agent Profile:** *Claude 2* with **analytical** tone. This is partly a thinking task. Use **UltraThink** to have Claude deeply consider edge cases and QA rules. It might output both some code (for e.g. checking for new companies) and a document describing the QA approach. Encourage a thorough approach: we want to catch mistakes early and often. The agent could be an "Analyst" persona. Temperature low, since we want factual, decisive plans, not whimsical ideas. Essentially, treat this like asking Claude to be a strategy consultant on improving data quality – it should give structured, clear recommendations (which we have enumerated above).

---

Each of these tasks will be "tickets" in Auto Claude. As you complete each, test and verify the piece works before moving on. By the end, you'll have a fully functional pipeline: from scraping a job posting to getting a rich BD briefing document, with minimal manual work in between.

# Starter Prompt Library

To help you interact with Claude for various parts of this project, here are some **starter prompt templates**. You can copy-paste these into Claude (or into your Auto Claude agent prompts) and fill in the specific details as needed. They are written in a way to be beginner-friendly: you just replace placeholders (like `{...}`) with your data, and ask Claude to perform the task.

- **Job Mapping Prompt (Match Job to Program):** Use this when you want Claude to analyze a job posting against your list of programs. For example, if you're troubleshooting the mapping or doing it manually:

```
You are an expert in U.S. defense programs and job analysis. I will give you a
job posting and a list of known programs. Identify which program the job most
likely supports.

Job Posting:
- **Title:** {Job Title}
- **Company:** {Company Name}
- **Location:** {Location}
- **Clearance:** {Clearance Requirement}
- **Description:** {Job Description (you can give a summary if very long)}

Known Programs:
{List of Program Names with brief info, e.g. "Program A – Air Force cyber
defense program (Prime: X Corp)", "Program B – Army intel system (Prime: Y
Inc)", ...}
```

```
**Task:**
1. Read the job details and compare with the known programs.
2. Pick the program that best matches the job's context (mission, tech,
clearance, company).
3. Explain briefly why you chose that program.
4. If unsure, list top 2 possibilities and what info is needed to confirm.

Respond with the program name and a one-line rationale.
```

*(This prompt guides Claude to act like the mapping engine: it compares job info to program list and selects a match. The explanation helps you understand the reasoning. It's useful for double-checking the AI's choice or when you don't fully trust the automated result.)*

- **Program Enrichment Prompt (Summarize Program Info):** Use this to have Claude summarize or format program details for use in outputs. For instance, preparing a snippet about a program for the briefing:

```
You have the following information about a defense program:

Program Name: {Program Name}
Customer Agency: {Agency/Department}
Prime Contractor: {Prime Company}
Description: "{Full description of the program...}"
Key Details: {Any key locations, technologies, etc.}

Summarize this program in 2-3 sentences, as if explaining to a business
development colleague. Include the program's purpose, the customer agency, and
the prime contractor in the summary. Be concise and factual.
```

*(Claude will produce a short summary like "Program X is a U.S. Air Force initiative focused on … The prime contractor is Y Corp, delivering capabilities in …" – which you can use directly in the briefing.)*

- **Org Mapping Prompt (Find Key Contacts):** Use this prompt if you want Claude to help figure out who might be the important people for a given program, especially if you don't have a clean list from your data. This is more open-ended, but Claude can infer likely roles to contact:

```
We have identified a program of interest: **{Program Name}** (Prime: {Prime
Contractor}, Customer: {Agency}). We want to find key contacts related to this
program for business development outreach.

Based on the program and prime contractor, list 3-5 roles or people we should
look for:
- Include specific job titles (e.g., "Program Manager for {Program Name}",
"Technical Lead at {Prime Contractor} on {Program Name}", "{Agency} Contracting
Officer for {Program Name}").
```

```
- If actual names are known (from context or typical org charts), mention them;
if not, just suggest the role.

For each suggested contact or role, explain why they'd be relevant to approach.
```

*(This prompt will make Claude act like a networking advisor. It might answer with something like: "1. John Doe – Program Manager for Program Name at Prime Contractor. Likely oversees execution of the contract and would know upcoming needs… 2. Jane Smith – Agency Project Officer for Program Name. She would be involved in contract management from the customer side…" etc. You would then try to find those people in your actual contacts data or LinkedIn.)*

- **BD Playbook/Briefing Prompt:** This is the big one to generate the briefing document content. It uses all the info we have to produce a Markdown report. You'll likely feed this via an API call or in the Auto Claude interface after assembling the data. Here's a template:

```
You are a business development analyst. Generate a briefing document in Markdown
for the following opportunity.

**Job Opportunity:**
- Job Title: {Job Title}
- Company hiring: {Company Name}
- Location: {Location}
- Clearance: {Clearance Level}
- Program Tag: {Program Name (identified)}
- Customer Agency: {Agency}
- Prime Contractor: {Prime (if different from Company)}
- Contract Vehicle: {Contract vehicle or contract name if known, else
"Unknown"}
- Job Description Summary: "{One or two-line summary of what the job does}"

**Program Background:**
{Program Name} – {One sentence description of the program's purpose/mission.}
This program is for {Agency} and is primarily executed by {Prime Contractor}.
{Add one more key detail, e.g., timeline or budget if known}. *(This info can be
drawn from program DB.)*

**Key Contacts:**
{List 2-3 key contacts or roles from Task 5, e.g.:
- Jane Doe – Program Manager at {Prime Contractor} (oversees {Program Name})
- John Smith – Technical Director, {Agency} {Program Name} Office
} *(If no specific names, list roles to target.)*

**Business Opportunity Insights:**
- {Company Name} hiring for this role suggests {explain why this hire is
happening – e.g. expansion, new contract award, backfill on project, etc., based
on context}.
- The required skills ({mention any unique tech from the job}) align with
```

```
{Program Name}'s needs, indicating this position is directly supporting the
program.
- The contract is handled via {Contract Vehicle or "an unknown contract
vehicle"}, which implies {if known, e.g. "it's a task order under ABC IDIQ" or
if unknown skip}.
- Competition: {If Prime Contractor is hiring, they are incumbent; if a
subcontractor hiring, mention competition scenario. Or note any major competitor
if known from program intel.}

**Recommendations (Next Steps):**
- Reach out to the Program Manager (or appropriate contact) at {Prime
Contractor} to introduce our company's capabilities and inquire about
subcontracting opportunities.
- Connect with the agency's office (e.g., {Agency} acquisition or program
office) to understand upcoming procurement or task orders related to {Program
Name}.
- Monitor for any news or updates on {Program Name} (budget changes, new phases,
re-compete timelines).
- Prepare a capabilities statement highlighting relevant experience to {Program
Name}'s mission (especially focusing on {mention any tech or domain from the job
that we have experience in}).

Format the briefing neatly in Markdown with headings for each section. Use
bullet points where appropriate (as above). Keep it concise (1-2 pages max).
Include all key data provided. Do not fabricate information not given.
```

*(This prompt is long, but it ensures Claude has all the pieces to produce a cohesive briefing. You'd replace the placeholders with actual data from your pipeline. Claude should return a nicely formatted Markdown document, which you can then save or share. The style is meant to be factual and actionable, giving the BD team exactly what they need without fluff.)*

- **Sequencing/Orchestration Prompt:** If you want Claude to just explain or plan the whole sequence (maybe for documentation or verification), you can ask it something like:

```
Explain the end-to-end workflow of the Program Mapping Engine I've built, in 5-7
steps, assuming a new job posting just got scraped. Include all major stages
(ingestion, enrichment, contact finding, briefing) and mention where human
review comes in.
```

*(Claude would then summarize the pipeline: e.g., "1. Ingestion: The new job is captured and added to the database… 2. Enrichment: The engine matches it to a program… 3. Org Chart: Key contacts are identified… 4. Briefing: A markdown brief is generated… 5. Review: Low-confidence cases are flagged for human review… etc." This is more for communicating to others or ensuring the AI understands the flow.)*

These starter prompts can be adjusted as you see fit. They are like recipe cards – you fill in the ingredients (your data) and the AI will do the cooking (analysis/generation). As you get comfortable, you might combine

some steps or feed outputs from one prompt into another. Feel free to tweak the wording to get the best results from Claude.

# How to Use Auto Claude Features

Auto Claude comes with several powerful features to help manage this multi-step project. Here's a quick guide on how to use them in the context of our Program Mapping Engine project:

- **Agent Terminal:** This is like your command center for each AI agent (task). When you open an Agent Terminal, you get a chat-like interface where you can interact with the AI on that specific task. For example, after you create **Task 3: Job→Program Mapping Engine**, you can go to that agent's terminal and see it working on the code. The terminal will show the AI's thought process, any code it's writing, and you can intervene if needed. You have one terminal per agent, so you can run multiple agents in parallel and watch their outputs side by side [59]. Using it is straightforward: click on the task, and you should see an option to open the Agent Terminal. From there, you can type additional instructions or clarifications if the agent gets confused. Think of it like sitting with a developer at their computer – you see what they type and can say "oh, don't forget to import the library" or "test that function once you write it." It's a very hands-on way to guide each step.

- **Roadmap:** The Roadmap feature helps you plan and visualize high-level goals and future enhancements. For our project, you might use the Roadmap to note phases like "Phase 1: Core Mapping Engine", "Phase 2: Org Chart Integration", "Phase 3: Scale to more programs", etc. It's AI-assisted, meaning you can ask Claude to suggest what the roadmap should include. For instance, you could input "Plan the next improvements after this engine is running, considering more data sources or fine-tuning" and Claude might suggest ideas. It's more strategic and less about coding. In practice, after completing these tasks, open the Roadmap panel and you could prompt something like: *"Analyze the Program Mapping Engine project and suggest a 3-month roadmap for additional features or improvements."* Claude might output a mini roadmap with items like "Implement real-time alerts for new opportunities, Integrate with CRM for outreach tracking, Expand knowledge base to 100 programs," etc. [60]. This is great for seeing the big picture and ensuring alignment with business goals. As a beginner, don't worry if you don't use Roadmap heavily at first – it's there when you want to think beyond the immediate tasks.

- **Changelog:** As tasks get completed, Auto Claude can generate a **Changelog** which is essentially release notes or a summary of what changed [8] [61]. This is automatically compiled from the tasks and code changes. For example, when Task 3 is done, the Changelog might say "Implemented job_mapping.py for program tagging; integrated dictionary and vector search." It's like magic documentation. You can access the Changelog tab to see these notes. This is super useful to review what the AI did in each task, especially if you step away for a day or collaborate with others. It's also helpful for the QA loop – you can easily recall "what was changed in the last run" via the Changelog. In our project, after completing tasks 1-8, you could have Claude generate a combined changelog or release note summarizing the entire Program Mapping Engine build. This could then be saved or shared with your team to explain what was delivered.

- **MCP Overview (Master Control Panel Overview):** This gives you a birds-eye view of all active agents, tasks, and their statuses. Think of it as the dashboard for the whole project. You'll see which

tasks are in progress, which are done, and any that are blocked or need input. For example, if Task 5 is waiting on Task 3 to finish, you might see that dependency. The MCP Overview is useful for tracking parallel work – say you have tasks 2, 3, and 4 running at the same time, you can monitor each's progress here. It ensures you don't lose track in a complex project. In practice, once you've entered all tasks, glance at the MCP to verify everything is laid out correctly and nothing is orphaned. It might also show resource usage or if an agent needs your attention. As a beginner, use this to build confidence that the AI hasn't "forgotten" about any part of the plan – everything is in one place.

- **Ideation:** This feature lets the AI analyze your codebase or project and suggest improvements, find bugs, or propose optimizations [8]. It's like having a senior engineer review your work. After you have the engine working, go to Ideation and prompt something like: *"Review the program mapping code and suggest any improvements or potential issues."* Claude might point out, for example, "You might want to handle abbreviations better in the keyword match" or "Consider caching the embeddings to improve speed." It might also highlight any performance bottlenecks. Ideation is great when you're not sure what to improve next – Claude can brainstorm for you. In our context, after an initial run, use Ideation to see if there are ways to make the tagging more accurate or the workflow more efficient (Claude might draw on the blueprint's suggestions like adding more data sources or tuning thresholds). It's like asking "How can I make this better?" and getting an expert answer. Remember, you don't have to implement everything it suggests, but it's a valuable second opinion.

- **Worktrees:** (If you're not familiar with git worktrees, the term here refers to working on multiple threads or branches in parallel.) Auto Claude allows you to spin up **sub-agents** or separate work contexts so that multiple tasks can progress without interfering [62] [63]. For example, you could have one agent working on the scraping ingestion while another works on the program matching simultaneously. Worktrees (or sub-agents) ensure each has its own "sandbox" of context, so they don't mix up each other's code [64]. In practice, if you enable parallel execution, Auto Claude will handle this under the hood – you just see different Agent Terminals doing their thing. If you needed to create a new branch of development (say, a version 2 of the engine) while retaining the original, you could use a worktree or branch for that, but as a beginner, that's advanced. For now, understand that Auto Claude is managing context to avoid confusion. Each task's agent only sees info relevant to that task, which prevents "context pollution" where the AI might get distracted by details of a different task [65]. So, you don't have to do much except let the tool do its parallel magic. If something does go wrong in parallel (rarely, like if two agents try to edit the same file), Auto Claude might notify you, and you can then handle it (usually by merging changes in the Changelog or having one agent re-run after the other).

In summary, **use Agent Terminals** for hands-on guidance of each task, **Roadmap** for big-picture planning, **Changelog** to track everything done, **MCP Overview** to monitor task statuses, **Ideation** to continuously improve your project, and trust the system's parallel processing (Worktrees/sub-agents) to make development efficient. Each feature is there to make your life easier and development faster, even if you're not coding everything yourself. You're essentially the project manager and architect, and Claude is your team of developers – these tools help you manage that "team" effectively.

# First 3 Steps to Execute Immediately

Finally, let's break down the very first things you should do right now to get this project rolling. Think of this as your quickstart guide – do these three steps and you'll be on your way:

**Step 1: Set Up Your Environment and Tools**
Begin by installing and launching the core tools. Install **Auto Claude** on your computer (follow the instructions for your OS – this typically involves installing the Claude Code app or running the backend as per the GitHub README). Once installed, open it and plug in your **Anthropic Claude API key** when prompted. Do the same for **n8n** – get it running (for example, use n8n cloud or run `docker run -p 5678:5678 n8nio/n8n` for a local instance). Log in to n8n and be ready to create a workflow. Also, ensure you have **Notion access** set up: get the Notion integration token and have your databases (Job Postings, Programs, etc.) prepared or at least a blank database ready to fill. In summary, get Claude, n8n, and Notion all talking: try a simple test like using n8n's Notion node to create a dummy page (just to verify the API key works). This step is like gathering your tools and making sure they're not broken – do it first so you don't hit snags later.

**Step 2: Organize Project Files & Import Data**
Create the project folder structure as outlined in *Folder/File Setup Instructions*. On your machine, make a new directory **ProgramMappingEngine** (or whatever name you like). Inside, create subfolders `data`, `scripts`, etc., as shown. Now take the **CSV files** you have (the ones you provided, e.g. the programs list, contractors list) and put them into the `data/` folder. Rename them for simplicity if needed (for example, `Programs_KB.csv` for the program database). Open one of them in a spreadsheet or text editor just to familiarize yourself (see what columns it has, etc.). In Notion, if you haven't already, import these CSVs to create the actual databases (you can create a Notion table then use "Merge with CSV" or manually copy data). Having the data in both CSV and Notion will help – CSV for the AI to crunch easily, Notion for the live updates. This step ensures your "knowledge base" is loaded and ready. It's akin to making sure your cookbook and ingredients are on the counter before the chef (Claude) starts cooking.

**Step 3: Define Tasks in Auto Claude & Run a Test**
Open Auto Claude and create a new project (if not already created). Start adding the tasks from the *Claude Task Definitions* section into the system. You don't have to add all at once; start with a couple to see how it works. For instance, add Task 1 (New Job Ingestion & Deduplication) and Task 3 (Job→Program Mapping Engine) first – these are key pieces. For each, input the Title and Description we provided (you can copy from this document). Also select the recommended agent profile or settings (e.g., for Task 3, choose Claude 2, enable UltraThink, etc., as described). Once Task 3 is entered, go ahead and open its **Agent Terminal** and let it start working. It will likely begin writing the `job_mapping.py` script. Monitor its progress – if it asks for clarification or seems to pause, you can nudge it by saying "continue" or asking a question in the terminal. When it finishes, review the code it produced. Then **run a quick test** of that code: you can either integrate it into n8n or just run it manually with a sample input. E.g., take one job posting (maybe create a dummy JSON or use a real one from your scraped data) and call the function in `job_mapping.py` to see what it returns. This will check that the core logic is working. If the output looks reasonable (like it identifies a program name and confidence), congrats – you have the heart of the engine beating. If not, you can troubleshoot now (maybe adjust the prompt or fix a bug with Claude's help) before moving on.

After these three steps, you will have: your environment configured, your data in place, and a prototype of the mapping logic created and tested. From there, you can proceed to flesh out the other tasks (like Task 2 for keywords, Task 4 for updating Notion, etc.) one by one, and gradually build the entire pipeline. Remember to keep using the Auto Claude features – watch the Kanban/MCP to track tasks, use Ideation if you get stuck, and do small tests at each stage. **By taking it step-by-step, you're effectively teaching the 10-year-old version of yourself how to build a complex system – and you'll see that even something advanced can be broken down into manageable, logical steps.** Good luck, and have fun building your Program Mapping Engine!

---


[1] [2] [62] [63] [64] [65] Thinking in Parallel: How Sub-Agents Change the Game with the new release of cluade and my experience. | by Atihar Hossen Mahir | Medium
https://ah-mahir.medium.com/thinking-in-parallel-how-sub-agents-change-the-game-with-the-new-release-of-cluade-and-my-63fd72c0af65

[3] [5] [6] [7] [14] [17] [18] [19] [21] [52] [53] [54] [55] [58] Integration Setup & Workflow — Program Mapping Intelligence Hub
https://www.notion.so/4871539eab684a3c90999e96538dfa18

[4] [10] [11] [12] [13] [15] [16] [20] [22] [23] [24] [25] [26] [27] [28] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [56] [57] Program Mapping Engine – Architecture & Stack Blueprint.docx
file://file-DGi75NbmsEaVFXWMUNeKwR

[8] [9] [59] [60] [61] GitHub - AndyMik90/Auto-Claude: Autonomous multi-session AI coding
https://github.com/AndyMik90/Auto-Claude

[29] What is UltraThink in Claude Code - ClaudeLog
https://www.claudelog.com/faqs/what-is-ultrathink/