ChatGPT

Detailed Plan for AI-Powered Job Application Automation

Tech Stack and Architecture Overview

We will use a **multi-language stack** to leverage each language's strengths while ensuring efficiency and speed. The core implementation will be in **Python** for its rich ecosystem of AI libraries and quick development cycle. Python offers excellent support for machine learning (HuggingFace, PyTorch Lightning) and browser automation (Selenium, Playwright). For performance-critical components, we can integrate **Rust** or **Go** as needed: - **Python** – Main orchestration, AI model training/inference, and high-level logic (fast prototyping and access to ML libraries). - **Rust** – (Optional) For modules that require low-level efficiency or heavy parallel processing. Rust can be used to speed up text parsing or handle concurrency if Python becomes a bottleneck. - **Go** – (Optional) For building a lightweight compiled utility, such as the system tray application or background services, given Go's ease of cross-platform support and low resource usage.

Using multiple languages is feasible by structuring the project into independent modules or services. For instance, a Python service could handle AI tasks, while a small Go binary manages the tray UI. However, to start, we might implement everything in Python for simplicity and then optimize specific parts with Rust/Go once the system is functional. This approach ensures we get both **rapid development** and **optimized performance** where necessary.

Overall, the architecture will be component-based: - **Core Controller (Python):** Coordinates the job search, filtering, ranking, and application process. - **AI Services (Python):** Handles model training, embeddings, and any AI-driven features. - **Browser Automation Module (Python + Playwright):** Executes form filling and web interactions. - **Tray Interface (Go or Python):** Runs as a desktop application for user settings and controls. - **Data Storage (SQLite/Files):** Manages logs, histories, and user preferences on disk.

This design ensures each part can be developed and tested in isolation, and it allows swapping out components (e.g., using a Rust module for scraping later if needed) without rewriting the entire system.

Browser Automation Strategy

For automating the job application process on websites, we will use **browser automation libraries** like Selenium or Playwright. The preference is **Playwright** (with its Python API) for modern features and reliability: - **Playwright** supports multiple browsers (Chromium, Firefox, WebKit) and has a robust Python library. It's known for faster execution and better stealth capabilities than older Selenium in many cases. - We will run Playwright in **headless mode** for speed, but can fall back to headful mode for debugging or when human-like behavior is needed. - To avoid detection (which can trigger captchas or bans), we'll incorporate **stealth techniques**. This includes using tools like Playwright Stealth (or modifying navigator variables), randomizing delays between actions, and mimicking human input patterns. Similar projects have successfully used stealth plugins and careful automation to perform one-click job applications [1]. - If

needed, **Selenium** with undetected-chromedriver is an alternative. Selenium has a large community and plugins for stealth; we remain open to it if Playwright faces issues on certain sites. - We may also use **Chrome DevTools Protocol** directly for lower-level control if we need to simulate user behavior more granularly or handle sites that detect typical automation libraries.

The browser automation module will handle tasks like: - **Navigating to job listings:** Searching and iterating through job posts on platforms (LinkedIn, Handshake, Indeed, etc.). - **Form auto-fill:** Populating fields (name, email, resume upload, answers to common questions) using stored profile info. For LinkedIn "Easy Apply", for example, the bot can automatically fill forms and submit [1]. - **Document attachment:** Uploading resumes or cover letters as needed (ensuring the correct file for each application). - **Handling multi-step forms:** Some applications have several pages of questions. The automation will detect progress and continue filling until completion. - **Error handling:** If a page fails to load or a selector is not found, the system will log the error and skip or retry gracefully.

Crucially, the automation will be designed to **not overload or spam** applications: - It will work at a reasonable pace with random delays to mimic a human applying. - It can include logic to stop or slow down if too many applications are done in a short time (to avoid rate limits or detection).

By using these robust browser automation practices, the system can reliably submit applications while minimizing the chance of being flagged by anti-bot measures.

AI Model Training Plan

We plan to incorporate **multiple AI models** to enhance filtering, ranking, and personalization. This includes both fine-tuning existing models and potentially training new ones from scratch to deepen theoretical understanding.

Types of Models to Train: - **Natural Language Understanding (NLU) models:** e.g., a transformer-based classifier (like DistilBERT or TinyBERT) fine-tuned to classify job descriptions as “suitable” or “not suitable” for the user. This model will help in filtering and ranking by predicting fit based on job text vs. the user’s profile. - **Semantic Embedding models:** We may use Sentence Transformers or similar to create embeddings for job postings and personal documents (resume, etc.). This will allow semantic similarity matching beyond keyword overlap [2]. For instance, the model would understand that *“Django” implies Python experience* even if Python isn’t mentioned, addressing vocabulary mismatches [3]. - **Generative models:** If we include personalized cover letter or question-answer generation, a fine-tuned language model (like a smaller GPT-2 or a LLaMA-2 7B fine-tuned on Q&A) can be used to craft answers or tailor resumes. This could be done via a local model to keep it cost-free (using HuggingFace Transformers with our GPU for inference).

Fine-Tuning vs Building from Scratch: - We will **fine-tune pre-trained models** for most tasks. Fine-tuning leverages existing knowledge in large models (like BERT, GPT) and adapts it to our specific domain. This is efficient and requires less data – for example, fine-tuning a model to identify relevant skills in a job description can be done with a modest dataset of labeled job posts. - Training from scratch is far more resource-intensive and usually not necessary unless we need a very specialized model. Given our goal of practical implementation, we’ll use pre-trained weights (such as TinyLlama or DistilBERT) and fine-tune them. This approach gives high performance quickly [4], since the model already understands general language and we just teach it domain-specific patterns. - However, to satisfy the learning aspect, we can incorporate **theoretical exercises**: for instance, building a simple neural network from scratch (with

PyTorch Lightning) to classify jobs vs. profile match. This can be done on a small scale to understand the process, even if its results might not outperform the fine-tuned transformers. - We may mix both approaches: use fine-tuned models for actual functionality, and maintain a separate experimental module where we build models from scratch to compare results. This provides a learning opportunity without compromising the application's effectiveness.

Hardware Utilization: - We will leverage the **local GPU** for both training and inference. Training transformer models on CPU would be extremely slow; using a GPU (even a single consumer GPU) accelerates fine-tuning by orders of magnitude. All deep learning frameworks we use (PyTorch, TensorFlow, HuggingFace) will be configured to use GPU when available. - During inference (actual application run), the GPU can also speed up tasks like embedding generation or running the classifier model to filter jobs. This ensures the added AI features don't introduce significant lag when scanning through many job listings.

In summary, the plan is to incorporate **AI at every critical decision point**: filtering out irrelevant jobs, ranking the best matches, and personalizing application content. We'll primarily fine-tune existing transformer models to do so, ensuring we get powerful capabilities without prohibitive training time. All model development will be done locally to avoid costs, using the GPU for efficiency.

Personal Data Management (Resume and Documents)

We will maintain a **data folder** containing the user's personal files – likely ~10–20 documents including resumes, cover letters, transcripts, project portfolios, etc. Managing these properly is key for tailoring applications. The design considerations are: - **Formats:** The files may be in various formats (PDFs for resumes/transcripts, DOCX for cover letters, Markdown or TXT for personal notes, JSON for structured data). Our system will incorporate parsers for each: - PDFs: use a PDF parsing library (like PyMuPDF or pdfplumber) to extract text. - DOCX: use `python-docx` to extract text. - Markdown/TXT: open directly as text files. - JSON: if present (e.g., a structured profile), load with Python's json module. - **Static vs Dynamic:** These files are mostly static, but can occasionally be updated. We will treat this folder as a **source of truth** that can be re-read each time the program runs (or when a manual refresh is triggered). Since the number of files is small, re-indexing them is quick. We could also implement a simple timestamp check – if a file's modified date changes, update our data store for it. - **Embedding for Semantic Search:** To allow semantic matching and quick retrieval of relevant info from these personal documents, we will embed their content in a **vector database**. Using a vector store like **FAISS** (Facebook AI Similarity Search) is ideal as it's free, local, and efficient for similarity queries. With FAISS, we can convert each document (or even sections of documents) into vectors and store them: - We'll use a pre-trained embedding model (like Sentence-BERT) to encode documents. Each file might produce several vectors (e.g., one per section or paragraph for granular search). - These vectors will be stored in memory or on disk via FAISS indexes. For our scale (10–20 files), this is trivially small; but using FAISS sets the stage for scaling up if needed. - When a job description requires a certain skill or experience, we can semantically search the user's docs to see if there's a relevant snippet (for use in a cover letter or form answer). For example, if a job asks for *"machine learning projects"*, the system can quickly retrieve the portion of the resume or portfolio that describes such a project using vector similarity [2]. - **Alternatives:** We are open to other local vector DBs like **Qdrant** or **Chroma**. These can run locally and offer advanced features. Qdrant, for instance, could run as an embedded service and provide persistent storage of vectors. However, given the small scale and desire for simplicity, FAISS (which operates as a library in-process) is likely sufficient and keeps everything in Python. - **Cost Consideration:** All these approaches are free and run locally, aligning with the goal of zero external cost. We avoid cloud-based vector services or paid APIs by using open-source tools.

In summary, the data folder will be the repository of all personal info that can be used in applications. Our system will parse and index these files so that we can *instantly access relevant personal details* when customizing a job application, ensuring each application is as personalized and effective as possible.

Job Listing Data and Company List

We will have a **Company List CSV** that contains target companies or relevant data about companies. Initially, this list will be maintained manually by the user: - The CSV might include fields like company name, company career page URL, whether they sponsor international visas, etc., depending on what information the user gathers. We will define a clear format (for example: `Company Name, CareersPageURL, Sponsorship(Y/N), OtherNotes`). - On startup, the program will load this CSV to get the list of companies. This list can serve multiple purposes: - Filter job searches to only include these companies (if the user only wants to apply to known preferred companies). - Conversely, use it to **blacklist** companies (if it's a list of companies the user has already applied to or does not want to apply again). - Use company data (like the sponsorship column) as an input to filtering logic. For example, if a company is known not to sponsor, skip it outright since the user is an international student who needs visa sponsorship. - **Manual vs Automatic Updates:** The initial plan is manual maintenance. However, we will design with extensibility in mind: - Future Feature – *Automatic Updates:* We can integrate web scraping or APIs to update this list. For example, using LinkedIn or Indeed to search for new companies hiring, or a Handshake API (if available) to fetch partner companies. We could periodically scrape a site like Handshake for "new employers" or use LinkedIn's company search to discover similar companies. - Another approach is integrating job boards: rather than updating the company list, directly query job listings from Handshake/Indeed/LinkedIn by keywords. But this may require API keys or scraping logic. We'll likely start simple (user-provided list) and later incorporate such features as needed. - **Data Usage:** Our filtering logic (next section) will heavily reference the company list: - If "must sponsor international students" is a requirement, we'll check the company entry. If the company is not known to sponsor (and the job description doesn't mention it either), we skip those. - If certain companies are highly preferred, we could up-rank jobs from those companies in our results. - If the CSV contains a record of application status (say a column for "Already Applied"), the system can avoid reapplying or duplicating efforts.

By structuring company information in a simple CSV, we give the user full control and transparency. They can add or remove companies as their preferences change. The system will simply load and respect that data. As the project evolves, this mechanism can become smarter – potentially transforming into a small database or automatically populating from online sources – but the manual CSV approach is a robust starting point.

Filtering Logic for Job Postings

The filtering logic will be a two-tier system: first **hard-coded rules**, then an **AI-powered filter** for nuanced decisions. This ensures we never miss obvious criteria, but also have flexibility to include borderline cases.

1. Rule-Based Filtering (Primary):
We will implement a set of deterministic filters that quickly eliminate jobs not meeting basic criteria. These rules reflect absolute requirements or deal-breakers: - *Sponsorship Requirement:* As an international student, the user requires visa sponsorship. So any job that explicitly states **"no sponsorship available"** or **"must be authorized to work without sponsorship"** will be filtered out immediately. We can maintain a small list of

key phrases to detect this in job descriptions (e.g., *"U.S. Citizenship required"* is a no-go). The company list can also provide a hint; if a company is known to sponsor (or not), that informs this filter. - *Location/Remote:* If the user only wants **remote** positions, we filter out on-site jobs. The config may allow options like remote-only, hybrid acceptable, or specific locations. Initially, if `remote_only=true`, any job not marked as remote (or whose location isn't one of the user's specified cities) is skipped. - *Job Title/Role:* We will have a list of target job titles or keywords (from user preferences, e.g., *"Software Engineer", "Data Scientist"*). The filter will drop listings that don't match these desired roles. This can be a simple keyword search in the job title field. - *Experience Level:* If the user is entry-level and only wants junior positions, filter out jobs requiring, say, 5+ years of experience. We can detect numbers of years or keywords like "Senior" or "Manager" in the title/description and exclude if they don't fit the desired experience range configured. - *Skill Must-haves:* Optionally, the user could specify must-have technologies (for example, if they only want jobs involving Python, not other languages). A rule can check the presence of certain keywords in the job description. However, this might be handled better by ranking or AI, since a missing keyword doesn't always mean irrelevant (semantic understanding is better here).

These hard filters are conservative to ensure we only spend time on plausible matches. They are also fast – simple string or regex checks.

2. AI-Based Filtering (Secondary):
After applying the hard rules, we'll examine how many jobs remain: - If the list is too large (e.g., hundreds of postings), we might still apply the AI filter to narrow down to the top candidates. - If the list is too small (say less than 10% of the original set, as per your threshold), it suggests the rules might be overly strict. In that case, we invoke a more flexible AI filter to catch some positions that may be worth considering despite failing a rule.

The AI filter will use the **NLU model** we discussed: - We'll feed the job description (and possibly company info, if available) and the user's resume/profile into a classifier or similarity model. The model will output a score or label indicating how well the job suits the user. - This model can understand context and synonyms. For example, even if a job post doesn't explicitly mention a skill the user has, it might still be a fit (maybe it uses a different term). The AI model can catch those cases, overcoming simple keyword checks [3] [5]. - We might implement this as a **BERT-based binary classifier** (fit vs not fit) or a **similarity ranking model**. Another approach is to generate an "embedding" for the job description and one for the user's resume and compute cosine similarity – if the score is above a threshold, we consider it a potential match. - The threshold for AI inclusion can be adaptive: for instance, normally we only take jobs that passed all hard filters. But if too few, we take the top N% of jobs as per the AI model's score even if they failed a non-critical rule.

Personalization & Weighting:
We will allow the user to configure how strict filters are. For example: - The sponsorship filter might be non-negotiable (always on, given legal necessity). - Remote filter could be optional: the user might mark it as "must be remote" or just "prefer remote." If it's a preference, the rule might not hard-eliminate on-site jobs but the ranking will score remote higher. - Each criterion could have a weight in the AI model's decision. If the user cares 70% about role fit and 30% about location, the model or scoring system can reflect that (this is an advanced feature for later). - Initially, we will implement basic toggles (must-have vs not) and gradually move to weighted preferences as we incorporate the AI model which can handle numeric scores.

In practice, the filtering pipeline might work like: 1. **Apply hard rules** to get initial candidate list. 2. If the resulting list is < X% of all found jobs, relax some rules or use AI to score all jobs and pick additional ones that score highly. 3. Conversely, if the initial list is very large, we might use the AI model to pick the top-scoring subset for further processing, saving time on clearly less relevant ones.

This two-step approach ensures efficiency (fast elimination) and intelligence (semantic inclusion of good fits). It maximizes the chances of finding at least ~10–20 strong opportunities even from a large search, and ensures we don't miss hidden gems that a simple filter might wrongly drop.

Ranking Logic for Matched Jobs

After filtering, we will typically have a set of jobs that are all potential fits. The next step is to **rank these job opportunities** so the system knows which ones to apply to first (and perhaps which ones to prioritize if time is limited).

Approach to Ranking: - **Rule-Based Scoring:** We can start with a straightforward scoring system. For each job, assign points for each match: - For each key skill the user has that appears in the job description, add points. - If the job is at a preferred company (from the CSV list), add points. - If the location or remote status matches perfectly, add points (or subtract if not ideal but still allowed). - If the job is exactly the title/role the user desires, add a significant weight. - Subtract points for any minor mismatch (for example, job asks for 5 years experience and user has 3 – if we still passed the filter, we could subtract a bit to rank it lower than a job asking for 2 years). - This results in a numerical score for each job. We then sort jobs by score descending. - **AI-Based Scoring:** In parallel, or as an evolution, use the AI model to produce a fit score: - A more sophisticated method is to have a model (like a regression or a classification probability) indicate the fit. For instance, a BERT classifier might output a probability between 0 and 1 of "good fit." This can directly serve as a score. - Alternatively, use an embedding approach: compute the similarity between job posting text and the user's resume text. The cosine similarity value (from -1 to 1) can be used to rank. Jobs with higher similarity are likely more relevant. - The semantic approach might reveal subtleties. For example, it could identify that a job heavily emphasizes teamwork and the user's profile has many team projects, boosting that match. - **Hybrid Scoring:** The most robust approach is combining both: - Use AI to get an initial fit score. - Use rule-based scoring to adjust for user explicit preferences. For instance, if two jobs are equally rated by AI in terms of skills, but one is remote (which the user prefers) and the other is on-site, we boost the remote one above the other. - This hybrid ensures the ranking isn't solely driven by the model (which might, say, favor a job at a non-preferred company if the skills match well). It injects user's priority weights into the final ordering.

Handling Ambiguity: Despite all scoring, there may be cases where it's not clear if a job is a good fit (e.g., limited description, or the user has a unique skill that may or may not be applicable). The user indicated they want to be **prompted if there's ambiguity**. We will implement this as follows: - If a job's score falls into a middle range where the system is uncertain (for example, it barely passed the threshold or the AI model gives it a moderate probability around 50%), we flag it as "ambiguous." - The system can then **prompt the user for confirmation** before applying. In the UI (floating taskbar or a popup), it might list these ambiguous jobs with a question: *"Apply to this job at Company X requiring skill Y? (Not sure if you meet preference Z)."* The user can then confirm or skip. - Implementation-wise, we could have the automation open the job posting in a browser tab for the user if manual review is needed, while it continues with other applications in the background. For example, it might defer ambiguous ones to the end or pause and wait for user input. Given the user's note, we might open an external browser tab with the job details while the

main process moves on to the next definite job – then circle back to it after completing others, depending on user's decision. - Another way is to queue these jobs separately: continue auto-applying to the clear high-score jobs, but list ambiguous ones in the tray interface under a "Review Needed" section. The user can then address them (approve/reject) and the system will act on those inputs (apply to approved ones).

By ranking jobs and involving the user for tough calls, we ensure the automation remains under control and aligned with the user's true interests. The top-ranked jobs (which we apply to first) are likely to yield the best chances, and the user's time/attention is only needed for borderline cases, making the process efficient but still flexible.

Automated Form Filling and CAPTCHA Handling

A critical component is the **automation of form filling** on application pages, as well as dealing with CAPTCHAs or other anti-bot measures.

Form Filling Process: - We will create a mapping of common form fields to the user's profile data. For example: - "Full Name" → user's name (from resume or config). - "Email" → user's email. - "Phone" → user's phone. - "Resume Upload" → attach the appropriate resume file. - "Cover Letter" → either attach a file or paste text (we can generate a tailored cover letter text if required). - "Yes/No Questions" → some forms have questions like "Are you legally authorized to work in X country?" which we can automatically answer based on user's input (e.g., if international student, answer truthfully and indicate need for sponsorship if asked). - The browser automation (Playwright) will identify fields by HTML element attributes. We'll maintain a library of selectors for known platforms: - For LinkedIn Easy Apply, the steps and fields are known (we can refer to existing automation like AIHawk which interacts with LinkedIn's form [1]). - For company career pages, forms might vary. We will rely on HTML `name` or `aria-label` attributes to guess fields. For instance, any `<input>` with `type=email` likely is the email field; anything with labels containing "Name" is name, etc. We can implement heuristics. - We'll also allow site-specific plugins: e.g., if we know a certain company uses Greenhouse or Workday forms, we can write a handler for those systems (since many companies use common ATS platforms). - **Multi-Step Navigation:** If an application form spans multiple pages (common in ATS like Workday), the script will detect the "Next" or "Continue" button, click it, then fill the next set of fields. This continues until a "Submit" button is clicked. We'll ensure to handle timeouts and waits appropriately for each step to load.

CAPTCHA Handling: - We aim to **avoid CAPTCHAs** by using stealth techniques (as discussed) and not performing actions that trigger them (like too many rapid requests). However, inevitably some sites might present a CAPTCHA (especially if logging in to LinkedIn or if an IP is flagged). - When a CAPTCHA is encountered, fully automated solving is challenging without external services: - Using third-party solving services (like 2Captcha) would violate the cost-free requirement and possibly terms of service, so we prefer not to. - We can attempt basic solutions: for example, reCAPTCHA v2 has known APIs and maybe a computer vision approach (using something like Tesseract OCR on simple image CAPTCHAs). But modern CAPTCHAs are often too advanced for built-in solving. - **Human-in-the-loop:** The practical solution is to involve the user: - If a CAPTCHA appears, pause the automation and alert the user through the UI (e.g., flashing the tray icon or a popup saying "Action required: please solve CAPTCHA"). - The automation can open the page in a regular browser window (or use Playwright's controlled browser but visible to user) so the user can solve the CAPTCHA manually. Once solved, the user can signal the program to continue. - This way, the flow isn't completely broken. We'll also log that a CAPTCHA was encountered, for future reference. - **Auto-Retry and Failover:** If a particular application fails (due to a form issue or a CAPTCHA we can't bypass

and user isn't available), the system will: - Save a **snapshot** (screenshot) of the page [1] at failure for debugging. - Log the job ID or details to a "failed attempts" log with the reason (if known). - Possibly retry once after a short delay (in case it was a transient issue). We might limit retries to avoid infinite loops. - If still failing, mark that application as skipped/failed and move on, to keep the pipeline flowing.

Logging and Snapshots: Every application attempt (success or fail) will produce log entries: - We'll log the job title, company, timestamp of attempt, and result (applied successfully, skipped, error, etc.). - On failure or any unusual event, a detailed message (exception trace or description) is logged. - Screenshots: using Playwright's screenshot capability, we can capture the state of the form at key points (especially on error). These images will be stored in a `logs/screenshots` folder with identifiable names (like `CompanyX_Job123_failed.png`). - This logging helps in debugging issues and also provides a history for the user to review (and ensures we don't double-apply to the same job, since applied jobs will be recorded).

By implementing robust form handling and carefully addressing CAPTCHAs, the system will be resilient. The use of automation libraries combined with fallbacks to user intervention when needed ensures that we can tackle a wide variety of application processes without getting completely stuck.

User Interface – Floating Taskbar Utility

To give the user control and visibility into the process, we will implement a **floating taskbar (tray) application**. This will likely be a small GUI that sits in the system tray (notification area) with an icon, running alongside the automation script.

Choice of Implementation: - We can build this as a **desktop application** using Python (with a GUI framework like PyQt or Tkinter) or with another technology like an Electron app or a Go-based GUI library. The user prefers a tray utility, which suggests a native feel: - In Python, libraries like `pystray` can create a tray icon with a menu, and we could use a lightweight window (PyQt5/PySide) for settings. - In Go, there are cross-platform GUI libs (like `fyne` or `webview`) or even using Tauri (Rust+JS) for a lightweight UI. However, using Python might be simpler to integrate directly with the main logic. - Considering integration, it may be easiest to have the **Python script spawn a GUI thread** for the tray. This way the same program controls both the automation and the UI.

Taskbar Features: - **Status Display:** The tray icon tooltip or menu will show a basic status (e.g., "Idle", "Applying: 3 of 10", "Paused", etc.). This helps the user know what's happening at a glance. - **Start/Pause Controls:** The user can click to pause the automation at any time. For example, a right-click menu on the tray might have "Pause" or "Resume" options, and "Quit". - **Real-time Notifications:** When an application is submitted or when user input is needed (like an ambiguous job or a CAPTCHA as discussed), the tray can flash or a notification balloon can appear. For instance, "5 applications submitted successfully" or "Action needed: review job at Company X". - **Settings Menu:** Perhaps the most important, a settings interface should be accessible from the tray: - The settings window can list all configurable filters and preferences. This includes: - Job titles/keywords of interest (editable list). - Locations of interest and remote toggle. - "Must Sponsor Visa" toggle (likely locked to true for this user, but still). - Option to select which resume file to use (if multiple versions for different roles). - Option to enable/disable cover letter usage or select a cover letter file. - A field to set the threshold for AI filtering (e.g., the 10% threshold we discussed). - Frequency of search or interval to re-run (if we have a continuous monitoring mode). - Credentials management (possibly open a secure credential store or prompt to update stored passwords – though actual credential input

might be handled on launch or config file to avoid plain text, see security section). - All these settings can be stored (when the user hits “Save”) in the config file or database, and the running automation should pick up changes (either by restarting or dynamically if possible). - **Job List/Monitoring:** An advanced feature for the UI could be showing a small live log or list of pending vs completed jobs: - E.g., a tab or section that lists jobs found, with icons for “applied”, “skipped”, “error”. Ambiguous ones could have a “Approve?” button next to them. - The user could manually trigger an application by clicking on a job in the UI if they want to override something, or remove a job from the list if they decide they aren’t interested. - This essentially turns the automation into a semi-automatic tool where the user can intervene, which might be desirable for fine control. Initially, we might not implement the full list, but it’s a consideration.

The tray application makes the automation **user-friendly**. Instead of a black-box script running in a terminal, the user has a window into the process and the ability to adjust parameters on the fly. It also makes it easy to stop the process if needed (important for safety, e.g., if the user sees something going wrong, they can quickly pause to prevent spamming applications).

In terms of design, we’ll keep the UI minimal and focused on functionality – it’s essentially a control panel for the job application bot. Over time, if needed, it could be expanded into a more polished application, but the primary goal is to expose controls and information for now.

Data Storage and Persistence

We need to store various kinds of data: logs, job history, and user preferences. All storage will be **local**, and we aim for a simple yet reliable setup (free of external dependencies).

Logs: - We will have a log file (or files) capturing runtime information. Using Python’s logging module, we can create a rotating logfile (e.g., `logs/run.log`): - Each action (starting search, filtering results count, applying to job X, errors, etc.) is appended with timestamps. - For debugging, we might have different log levels (info, debug, error). In normal use, the log can be in info mode, and we can have an optional debug mode for detailed tracing. - The log files help troubleshoot issues after the fact. They can be plain text for simplicity.

Applied Jobs History: - It’s important to remember what jobs have been applied to, to avoid duplicate applications and to track success. - We’ll maintain a **database or file** of all applied jobs. This will include details like: - Job ID or a unique identifier (could be a URL or an ID scraped from the posting). - Job title, company, date applied, and status (applied, failed, skipped). - Possibly a note if failed (from the error log) or if skipped due to user choice. - Using **SQLite** is a good solution here. SQLite is file-based, needs no separate server, and Python has built-in support for it. We can create a database (e.g., `data/job_history.db`) with a table for applied jobs. This allows querying (e.g., check if a job was applied already) easily with SQL. - Alternatively, a simpler approach is a CSV or JSON file (like `applied_jobs.csv`). That’s easier to inspect manually, but harder to query programmatically beyond simple linear search. Given the small scale, even a CSV would work (we can just read it into memory and search), but as a matter of good practice, SQLite provides more flexibility for future expansion (and the data can be easily exported if needed). - Each time before applying, the system will check this history store to ensure we haven’t applied before. After a successful application, it will insert a new record. - We will also store user preferences (from the UI or config) in the database or a config file. There are two main options: - Use a **config file** (like YAML or JSON) for settings. This is human-editable and can be under version control if needed. Indeed, the AIHawk project

uses YAML for config [6], which is easy to read and edit. We can do the same: e.g., `config.yaml` holds all filters and preferences, and `secrets.yaml` holds credentials (see Security section). - Use **SQLite** to store settings as well (maybe a table for key-value pairs or a JSON blob). This is more programmatic and less user-friendly to edit manually, but since we have a UI for settings, it could work. However, for transparency, a config file is probably better so the user can see all their settings at a glance or back them up.

Storage Structure: - `config.yaml` – stores user preferences (job titles, location, toggles like remote_only, etc.). The program loads this on start (or reloads if changed). - `secrets.yaml` – stores sensitive info (credentials, API keys), not to be shared [7]. This file will be encrypted or handled carefully (discussed in Security). - `data/` directory – contains the personal files (resumes etc.), the company list CSV (e.g., `companies.csv`), and possibly the SQLite DB for history (`history.db`). - `logs/` directory – contains log files and error screenshots.

We'll make sure to **persist important data**: - The history DB/CSV persists between runs, so we remember past applications. - Logs can be rotated but we might keep them for a long time for record-keeping. - If the user updates the company list CSV or personal files, those are simply reloaded (the files themselves are the source of truth).

All storage is local, meaning no data leaves the user's machine. This is good for privacy and also meets the cost-free requirement (no cloud storage needed).

By structuring the persistence layer with simple local files and a lightweight database, we ensure the system's state is maintained across sessions and the user can trust that it won't redo or forget tasks it has already done.

Security and Privacy Considerations

Security and privacy are paramount since we are dealing with personal data (resumes, credentials, application history). We will implement measures to protect this information:

Credential Management: - The tool will need login credentials for job sites (e.g., LinkedIn username/password, or other job board logins) to automate submissions. Storing these in plaintext is risky. - We will store credentials in an **encrypted form**. One approach is to use a `secrets.yaml` file (as in AIHawk [7]) but encrypted: - We can use a master password or key that the user provides at runtime to decrypt the file. For example, when the program starts, it prompts the user for a "vault password" (this could even be the user's system login password if we want to tie it, or a separate one). This password is used to decrypt the stored credentials and then discarded from memory after use. - Encryption can be done using a library like `Fernet` (symmetric encryption) from Python's cryptography package. The encrypted blob can be stored in a file, and only decrypted in memory when needed. - Another method is to integrate with the OS's **secure credential store**: - On Windows, use Credential Manager via something like the `keyring` Python library to save passwords. On Mac, use Keychain, on Linux, Gnome Keyring or similar. The `keyring` library provides a unified API to these. This way, the credentials are stored in the OS vault and our app can retrieve them with just the key name (the user might still need to unlock their keychain depending on OS settings, but it's more secure than a file). - This avoids keeping any plaintext or even custom encrypted file at all, leveraging the OS security. - Environment variables: We could allow the option for the user to set credentials in env variables (or a `.env` file not checked into code). However, environment vars are essentially plaintext in

memory, and a `.env` file is still plaintext on disk, so this is only marginally better (useful for development, but not secure for long-term storage).

Given the user's preferences, we'll likely implement **either the encrypted file or OS vault approach**. For simplicity, an encrypted YAML file might be easier to implement cross-platform. We'll document clearly for the user not to share that file and to use a strong master password.

Personal Data Privacy: - All personal files (resumes, etc.) stay local. We are not sending these to any external service (especially since we aim for free local AI, we're not even calling external APIs like OpenAI in the base plan). - If we do integrate any cloud-based features (like scraping from an API), we will ensure no personal data is inadvertently sent. For example, if using an external job search API, we only send the query (job title, location) but not the user's resume. - If the user opts to use an AI API in future (say OpenAI for better cover letter generation), we will clearly warn that their data (like a job description or parts of their resume) might be sent to that third-party and get their consent. But currently, we stick to local models to avoid this issue entirely.

Application Data and Logs: - The logs and stored data (applied jobs, etc.) could contain sensitive info (like which companies the user applied to, which might indirectly reveal things). We will keep these files on the user's machine only. If the user wants to further secure them, they could store the `data/` and `logs/` folder in an encrypted disk or use OS file permissions to restrict access. We'll mention this as a suggestion in documentation. - The system will never post or share user data to anywhere except to the target job applications as intended.

Secure Development Practices: - We'll sanitize any inputs if we integrate with system commands or SQL (to avoid injection attacks, though since this is local and single-user, risk is low). - We will also keep dependencies updated to avoid known vulnerabilities in the browser automation or ML libraries. - Regular backups of critical data (like the history database or config) might be recommended to the user, though that's more for convenience than security.

Conclusion (Security): By encrypting credentials and keeping all operations local, the user's privacy is well-protected. The only data going out to the internet is the intended form submissions to apply for jobs. We will document all data flows so the user is aware of what's happening. This cautious approach ensures that the convenience of automation doesn't come at the cost of personal security.

Summary and Next Steps

We have outlined an **end-to-end design** for a job application automation system that aligns with the user's preferences for a comprehensive, cost-free, and learning-rich project. To recap the key points: - **Tech Stack:** Primarily Python for its rich libraries, with potential Rust/Go components for optimization. This gives us both flexibility and performance. - **Automation:** Utilizing Playwright (or Selenium) for browser automation to navigate job boards and submit applications, enhanced with stealth tactics to avoid detection. - **AI Integration:** Training and fine-tuning models locally (leveraging the GPU) to handle intelligent filtering, matching, and possibly content generation. This ensures smarter decisions than simple keyword matching, capturing semantic relevance [3] [2]. - **Data Handling:** Personal files are parsed and embedded for quick retrieval, and a manually curated company list guides the search and filtering. Everything is stored locally (using FAISS for embeddings and SQLite/CSV for records) to avoid external costs. - **Filtering & Ranking:** A

layered filtering approach (rules first, then AI) guarantees both precision and recall in finding suitable jobs. Ranking logic (with manual scoring and AI scoring combined) prioritizes the best fits, while giving the user a chance to weigh in on uncertain cases. - **Form Filling:** The system auto-fills repetitive application forms and only stops for human help when absolutely necessary (like hard CAPTCHAs). Comprehensive logging and error capture are in place to debug and improve the process. - **User Interface:** A lightweight tray application provides control and transparency, allowing the user to adjust settings and monitor progress in real-time. This turns the tool into a user-friendly assistant rather than a hidden script. - **Security:** Credentials and personal data are handled with care – encryption and local storage ensure privacy and security are maintained throughout.

Next Steps: With this design in hand, implementation can proceed in stages: 1. **Setup Project Structure:** Create the initial code repository with modules for automation, AI, data, UI, etc., and prepare config files. 2. **Basic Automation & Filtering:** Implement a basic version: fetch jobs (perhaps from a static source or a single site like LinkedIn Easy Apply) and apply rule-based filtering, then attempt automated application on a few sample listings (without AI or UI initially). This will test the core automation loop. 3. **Integrate AI Models:** Train or fine-tune a simple classifier for filtering and see how it improves results. Incorporate an embedding-based similarity for ranking. This can be iterative – start with a very small model or even a heuristic simulation, then replace with actual ML. 4. **Develop the Tray UI:** Create the tray application and link it to the running process (e.g., start/stop, display status). Ensure settings from UI correctly flow into the automation logic (which might require restarting parts of the process or being designed to read settings dynamically). 5. **Testing & Refinement:** Run the tool in a dry-run mode (not actually submitting applications, or using a test site) to fine-tune form filling, logging, and to identify any website-specific challenges. Adjust stealth delays and add any needed exceptions. 6. **Security Hardening:** Before real use, implement the credential encryption and verify that no sensitive info is printed or left in logs. Test that the system handles wrong passwords, expired sessions, etc., gracefully. 7. **Expansion:** Once the core is working for one or two job sources, expand to others (if desired) like Indeed or Handshake by adding new scrapers or adapting the form filling to those platforms. Also gradually improve the AI models with more data or fine-tuning, and integrate more of the “nice-to-have” features like dynamic resume tailoring (which could use an AI agent as shown in similar projects [8]).

By following this phased approach, we will build a robust application that not only automates job applications but also intelligently targets the right opportunities and learns from the user’s preferences. The end result will be a powerful personal AI job assistant that operates efficiently, securely, and with a high degree of customization, helping the user land their dream job faster [9] [10].

---

[1] [6] [7] [9] [10] GitHub - jomacs/linkedIn_auto_jobs_applier_with_AI: LinkedIn_AIHawk is a tool that automates the jobs application process on LinkedIn. Utilizing artificial intelligence, it enables users to apply for multiple job offers in an automated and personalized way.
https://github.com/jomacs/linkedIn_auto_jobs_applier_with_AI

[2] [3] [4] [5] Talent Matching with Vector Embeddings - ingedata
https://www.ingedata.ai/blog/2025/04/01/talent-matching-with-vector-embeddings/

[8] GitHub - AloysJehwin/job-app: AI-powered automation that extracts resume data, matches job listings, rewrites resumes to fit job descriptions, and manages storage using Google Drive and Sheets via n8n.
https://github.com/AloysJehwin/job-app