Job AI Agent — Full Workflow & Techniques (Workbook)

> A practical, production-ready workbook describing nodes, techniques, data schemas, prompts, and example code snippets to build a job-search + auto-apply AI agent. Use this as a single-source design and implementation guide.

---

1. Goals & Scope

Primary goal: Build an autonomous/semi-autonomous agent that: finds job postings matching user preferences, extracts and normalizes job data, ranks jobs, personalizes resume sections & cover letters, fills application forms (where allowed), logs applications, and surfaces results via a dashboard with human-in-the-loop checkpoints.

Constraints & safety: Respect terms-of-service for job portals. Prefer official APIs / third-party aggregators (SerpAPI, JSearch, Indeed API). Use Selenium/Playwright only for internal/personal accounts and rate-limit actions. Never bypass captchas.

---

2. High-level Architecture

```
[User Profile & Resume JSON] --> [Job Search Node] --> [Job Fetch Node] --> [Extractor Node]
     --> [Matcher & Ranker Node] --> [Resume Personalizer Node] --> [Cover Letter Node]
     --> [Form Filler Node] --> [Application Tracker DB] --> [Dashboard]

Human-in-loop checkpoints: Review Matches -> Confirm Apply -> Solve Captcha

Supporting infra: FastAPI backend, LangGraph / LangChain orchestration, Vector DB (Milvus/Pinecone/Weaviate), Redis for queues, Postgres for audit logs, S3 for files.
```

---

3. Nodes (Detailed)

3.1 Job Search Node

Purpose: Discover job listings given keywords, location, filters.

Techniques: - Use SerpAPI (LinkedIn/Indeed engines) or official Indeed API - Google Custom Search as fallback - Search query templates with synonyms expansion - Rate limiting and pagination

Inputs: user_keywords, location, remote_flag, experience_level **Outputs:** list of job-result objects: `{title, company, short_snippet, job_url, source, posted_date}`

Notes: store request metadata (query, timestamp) for auditing.

3.2 Job Fetch / Page Loader Node

Purpose: Fetch full job page HTML or API JSON for a job_url.

Techniques: - Prefer API endpoints (Greenhouse, Lever) - If not available, use Playwright (headful) or Selenium to render JS pages - Respect robots.txt, throttle requests

Outputs: raw_html, final_url, http_status, screenshots(optional)

3.3 Extractor Node (Job Description Parser)

Purpose: Convert raw_html -> structured JD fields

Techniques: - Use `unstructured` to extract main text - LLM-based extraction (prompt to extract sections) - Regex + heuristic fallback - Named-entity recognition (spaCy) for skills, locations

Output schema:

```
{
  "job_id": "",
  "title": "",
  "company": "",
  "location": "",
  "employment_type": "",
  "salary": "",
  "posted_date": "",
  "description": "",
  "responsibilities": [],
  "required_skills": [],
  "nice_to_have": [],
  "application_url": ""
}
```

LLM prompt pattern (short):

```
Extract and return JSON with fields: title, company, location, posted_date,
responsibilities (list), required_skills (list), nice_to_have (list),
description (cleaned text) from the following job text: <JOB_TEXT>
```

3.4 Matcher & Ranker Node

Purpose: Score and rank jobs for the user.

Techniques: - Embedding similarity (OpenAI/HF embeddings) between job description and user's resume/profile - Rule boosts: must-have skills, location match, remote preference - Heuristic scoring: recency, seniority match - Learning-to-rank (optional) using historical user clicks/applies

Output: Ranked job list with `score`, `matching_skills`, `explainability` (why ranked)

3.5 Resume Personalizer Node

Purpose: Generate tailored resume sections or full resume variants for a job

Techniques: - Input: base_resume JSON, extracted JD - Use LLM to rewrite bullets, reorder skills, generate tailored summary, and select projects - Ensure fidelity: do not invent facts — rely on user-provided achievements & projects - Output schema (sections): `summary`, `skills`, `experience_bullets`, `projects`, `achievements`, `keywords`

Sample prompt fragment:

```
Given this resume JSON and the job requirements, produce JSON containing:
personalized_summary (2-3 sentences), top_8_skills (ordered), 5 experience
bullets rewritten to match JD, 2 highlighted projects. Keep facts only from
resume.
```

Safety: Add a verification pass: every generated bullet must cite origin field (e.g., project id or experience id).

3.6 Cover Letter Generator Node

Purpose: Generate a personalized cover letter using JD + resume personalization outputs

Techniques: - LLM generation with structured template and tone settings - Include 2–3 bullets aligning achievements to job needs - Produce a short paragraph for why candidate fits & interest

Output: `cover_letter_text`, `cover_letter_html`, `highlights` (bullets used)

3.7 Form Filler / Auto-Apply Node

Purpose: Automate application submission via APIs or browser automation

Techniques & strategy: 1. **API-first:** Check if the job source exposes a submit API (Greenhouse/Lever). If yes, encode payload and POST. 2. **Selenium/Playwright:** If no API, spin a browser session (headful), locate fields (LLM-assisted), fill, upload resume, and submit. - HTML field detection approach: - Use

heuristics (name/id/labels) + LLM to map fields -> profile attributes - Example mapping JSON: `{ "email": "xpath://...", "resume": "xpath://..." }` 3. **Human-in-loop:** For captchas or multi-step forms, pause & notify user with screenshot and resume after manual solve.

File uploads: Use `input[type=file].send_keys(local_path)` in Selenium.

Safety & throttling: 1 apply/sec per account is too fast — use 1 apply / 1–5 minutes depending on source.

Audit log: store request/response, status, timestamp, screenshot.

3.8 Application Tracker & DB

Purpose: Record every job found and applied to, track status, follow-ups, interview pipeline.

Schema (Postgres): - jobs (job_id, user_id, title, company, url, posted_date, fetched_at) - applications (application_id, job_id, user_id, resume_variant_id, cover_letter_id, status, applied_at, response) - tasks (captcha_pending, manual_review)

3.9 Dashboard / UX Node

Purpose: Present matches, let users review, edit personalized sections, confirm apply, show application history.

Tech stack: React + Tailwind (or create-agent-chat-app UI) + FastAPI backend

Features: - Job feed with inline highlights (matching_skills) - Edit preview of personalized resume bullets & cover letter - One-click apply (with confirm modal) - Notifications for captcha/manual steps

---

4. Data Schemas (JSON) — Base Resume Example

```
{
  "user_id": "u_123",
  "name": "Jugal Lachhwani",
  "email": "jugal@example.com",
  "phone": "",
  "summary": "...",
  "skills": ["Python","TensorFlow","FastAPI","LangChain","Selenium"],
  "experiences": [
    {"id":"e1","title":"ML
Engineer","company":"X","start":"2024-01","end":"2024-12","bullets":
["Built...","Deployed..."]}
  ],
  "projects": [{"id":"p1","title":"Email Agent","desc":"...","achievements":
```

```
["reduced time by 40%"]}]
}
```

---

5. Prompt Templates (Practical)

5.1 Extractor prompt (clean JSON):

```
You are a JSON extractor. Parse the job text and return only JSON with
fields: title, company, location, posted_date, responsibilities(list),
required_skills(list), nice_to_have(list), description. Do not add extra
commentary.
<JOB_TEXT>
```

5.2 Resume Personalizer prompt:

```
Input: base_resume_json, job_json
Task: Produce JSON: {personalized_summary, top_skills, experience_bullets,
projects_highlighted}
Rules: Use only facts from base_resume_json. Map each generated bullet to a
source id in the resume. Keep bullets <= 20 words.
```

5.3 Field mapping prompt (for Form Filler):

```
Given the following HTML snippet, identify which input fields correspond to
name, email, phone, resume upload, linkedin, github. Reply with JSON mapping
{field_label: css_selector_or_xpath, type}. Only output JSON.
<HTML_SNIPPET>
```

---

6. Example LangGraph / LangChain Node Map (Pseudo)

- job_search_node (python) -> paginated
- fetch_node (async) -> concurrency limit
- extractor_node (llm + unstructured)
- embeddings_node (generate embeddings for JD & resume)
- ranker_node (scoring function)
- personalize_node (LLM)
- apply_node (api or selenium)
- logger_node (db)

---

7. Code Snippets (Short)

7.1 SerpAPI search (python)

```
import requests
params = { 'engine': 'linkedin_jobs', 'q': 'machine learning engineer',
'location': 'Bangalore', 'api_key': SERPAPI_KEY }
r = requests.get('https://serpapi.com/search', params=params)
jobs = r.json()['jobs_results']
```

7.2 Playwright fetch (python)

```
from playwright.sync_api import sync_playwright
with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    page = browser.new_page()
    page.goto(job_url)
    html = page.content()
```

7.3 Selenium upload

```
upload = driver.find_element(By.XPATH, "//input[@type='file']")
upload.send_keys(r"C:\path\resume.pdf")
```

---

8. Testing & Metrics

E2E tests: mock job feeds, verify extractor outputs, verify mapping accuracy, test apply flow in staging accounts.

KPIs: - Match precision @ top5 - Apply success rate - Time per application (human-in-loop) - False-positive resumes generated (factual mismatch rate)

---

9. Deployment & Scaling

- Containerize nodes (Docker). Use kubernetes for scaling fetch/extractor.
- Use Redis queue for concurrency control.
- Use vector DB for embeddings; keep cold-start index warm for common skills.
- Rotate API keys and respect rate limits.

---

10. Legal & Ethical Checklist

- Respect TOS of sources. Prefer API partners.

- Never circumvent CAPTCHA or bot protections.
- Transparently inform users when human intervention is required.
- Provide an audit trail of generated text and sources.

---

11. Next Steps & Templates

- ✅Choose source (SerpAPI vs official API)
- ✅Build minimal proof-of-concept: Job Search -> Extractor -> Ranker -> Personalize resume -> Preview UI
- ✅Add apply automation for one source (e.g., Greenhouse) in API mode

---

12. Files to generate (when implementing)

- `resume_schema.json`
- `job_schema.json`
- `langgraph_workflow.json`
- `fastapi_app/` endpoints
- `frontend/` React components: JobFeed, ResumeEditor, ApplyModal

---

13. Quick Checklist for Implementation

1. Get SerpAPI key or Indeed API credentials
2. Build base resume JSON
3. Implement Job Search node (save to DB)
4. Implement Job Fetch & Extractor
5. Implement Embeddings and Ranker
6. Implement Resume Personalizer & Cover Letter generator
7. Implement Apply node (API first; Selenium fallback)
8. Build Dashboard & human-in-loop flows

End of workbook.

If you want, I can now: - Generate the LangGraph workflow JSON for this workbook - Scaffold the FastAPI + Selenium repo with code for 3 nodes - Create the React dashboard skeleton

Which one should I generate next?