FedSource — Complete Build Reference (job-search engine)

Paste this ENTIRE file into your AI chat as context. It contains the architecture, all 18 source connectors, the output tools, the apply-side browser extension, and the full source code. Your AI can then wire any piece into your tracker. Built by Matthew; pure Python standard library — no pip installs, no framework.

---

1. What it is

One engine that pulls jobs (and federal contracts) from **18 live sources**, normalizes every result to a single record shape, dedupes, fit-scores against a person's keywords, and emits filterable spreadsheets, an HTML dashboard, and a browser extension that fills application forms. The whole thing is stdlib-only and ~40 lines per connector.

2. The five moving parts

1. **Connectors** (`source.py`) — one `fetch_*` function per source. Every one returns the same `opp()` dict, so the rest of the system never cares where a row came from.

2. **Normalization** — the `opp()` factory. THE key design decision: one contract = mix any sources freely.

3. **Dedup + scoring** — SQLite upsert by `uid` ; fit = count of a profile's keywords found in the title.

4. **Profiles** (`config.json`) — per-person search settings (`profiles.<name>` = fit keywords; `sheet_profiles.<name>` = per-lane search overrides). Swap the profile, retarget the whole engine to a different career.

5. **Outputs** — `make_sheet.py` (filterable .xlsx), `make_radar.py` (self-contained HTML dashboard), and an MV3 Chrome extension for the apply side (fills forms in your own browser, never auto-submits).

3. The data contract — `opp()`

Every connector returns this dict. Map it straight onto your DB columns:

field	meaning
`uid`	stable dedup id, `source:nativeid`
`source`	which connector (ADZUNA, Greenhouse, …)
`title`	job title
`agency`	company / org
`otype`	job type / category
`naics`	federal NAICS (jobs: blank)
`posted` / `due`	dates
`set_aside`	federal set-aside (jobs: blank)
`url`	apply / posting link
`summary`	location · salary

4. The 18 connectors

Job aggregators (need a free key): `adzuna` (commercial + salary/employer intel), `findwork`, `jooble` (aggregator-of-aggregators, highest volume).

Federal jobs (key): `usajobs`.

Remote boards (keyless): `remoteok`, `remotive`, `himalayas`, `jobicy`, `themuse`.

Employer ATS (keyless, highest precision — you pick the companies): `greenhouse`, `lever`.

Hiring signals (keyless): `hn` (HN Who-is-Hiring), `yc`.

Federal contracts/funding (mostly keyless): `sam` (key), `grants`, `sbir`, `federalregister`, `usaspending`.

Keyed connectors skip cleanly if the key is absent — keyless ones work the second you import the file.

5. How to run

```
python3 source.py                # pull all enabled sources -> SQLite + a digest
python3 make_sheet.py <profile>  # -> <profile>-jobs.xlsx (filter by Source + Fit)
python3 make_radar.py <profile>  # -> HTML dashboard
```

Or call any connector directly: `import source; rows = source.fetch_adzuna(cfg)`.

6. The apply side (browser extension)

An MV3 Chrome extension whose content script fills Greenhouse/Lever/Ashby (and embedded Databricks/Stripe) forms **in your real browser session** — standard fields from a saved profile, factual screening answers (Yes/No on work-auth etc.) derived only from validated data (never invented), and grounded free-text drafts. It **never auto-submits** — you review and click. Code below (`content.js` + `manifest.json`).

---

7. Deploying beyond local — Cloudflare Pages & GitHub

The system has two halves that deploy differently. Don't try to host them the same way.

A) The sourcing engine (the Python connectors). This is a *data job*, not a web server — it pulls from the 18 sources and writes rows. Run it on a schedule:

- **Local cron** — a line in `crontab` calling `python3 source.py` every few hours.
- **GitHub Actions (free)** — a scheduled workflow (`.github/workflows/sync.yml` with a `schedule:` cron) that runs `source.py` and commits the output or pushes to your DB. Zero servers.
- **Always-on** — a $5 VPS, or Railway / Render / Fly.io, if you want the engine live with its database.

B) The tracker UI (your web app). This is what gets a public URL.

Cloudflare Pages — this is exactly how `nk-worklist.pages.dev` is hosted (free, instant, custom domains, no server):

```
# install once
npm install -g wrangler && wrangler login

# one-time: create the project
wrangler pages project create my-tracker --production-branch main

# deploy a folder of static files (your build output)
wrangler pages deploy ./dist --project-name my-tracker
# -> live at https://my-tracker.pages.dev in ~5 seconds
```

(If your account has more than one login, set `CLOUDFLARE_ACCOUNT_ID=<id>` before the deploy.)

GitHub (auto-deploy on every push) — the hands-off version:

1. Push the tracker to a GitHub repo.

2. Cloudflare dashboard -> **Pages -> Connect to Git** -> pick the repo -> set the build command + output dir.

3. Every `git push` now auto-builds and deploys. (Vercel and Netlify work the same way if you prefer them.)

4. Pure-static tracker with no build step? **GitHub Pages**: repo Settings -> Pages -> deploy from a branch — done, no Cloudflare needed.

Full-stack tracker (UI + Python backend + DB in one app). Cloudflare Pages is static-only, so if your app runs the connectors *server-side* on each request, host it where Python runs: **Railway / Render / Fly.io** (Python + Postgres/SQLite), or **Cloudflare Workers + D1** (connectors as a Worker, D1 as the DB, Pages as the UI). Simplest path, and what we use: keep the two halves separate — **engine on a schedule writes the data, static UI on Pages reads it.**

FULL SOURCE CODE

`source.py`

```
#!/usr/bin/env python3
"""
source.py — Government / Defense / Grants contract-opportunity sourcing pipeline.

Polls free federal opportunity APIs (SAM.gov contracts, Grants.gov grants,
SBIR.gov defense innovation), normalizes them, scores them for relevance to
your profile, dedupes against a local SQLite store so you only see NEW matches,
and writes a ranked Markdown digest.

Stdlib only — no pip install. Python 3.9+.

Usage:
    python3 source.py                    # run once, show new matches, write digest
    python3 source.py --all              # show all matches (ignore dedup)
    python3 source.py --source sam,grants
    python3 source.py --min-score 3 --since-days 7
    python3 source.py --help / --version

Exit codes: 0 ok · 1 runtime error · 2 usage error.

SLED (state/local) aggregators and private RFP boards have no clean public API
and are intentionally out of scope here — watch those via their portals.
"""
from __future__ import annotations

import argparse
import html
import json
import os
import re
import sqlite3
import sys
import urllib.error
import urllib.parse
import urllib.request
from datetime import datetime, timedelta, timezone

__version__ = "1.0.0"

HERE = os.path.dirname(os.path.abspath(os.path.realpath(__file__)))
DEFAULT_CONFIG = os.path.join(HERE, "config.json")
DEFAULT_DB = os.path.join(HERE, "opportunities.db")
DEFAULT_DIGEST_DIR = os.path.join(HERE, "digests")
UA = "gov-sourcing-pipeline/1.0 (+contracting research)"
```

```
def _load_env(path: str | None = None) -> None:
    """Load KEY=VALUE lines from a gitignored .env. A real (non-empty) env var still wins,
    but an EMPTY env var does NOT shadow the .env value — some launchers export KEY='' which
    setdefault() would otherwise keep, hiding the real key."""
    path = path or os.path.join(HERE, ".env")
    try:
        with open(path) as f:
            for line in f:
                line = line.strip()
                if not line or line.startswith("#") or "=" not in line:
                    continue
                k, v = line.split("=", 1)
                key = k.strip()
                if not os.environ.get(key):  # absent OR empty
                    os.environ[key] = v.strip().strip('"').strip("'")
    except FileNotFoundError:
        pass


_load_env()  # intel.py and mcp_server.py import source, so this covers all three


# ---------------------------------------------------------------------------- helpers
def warn(msg: str) -> None:
    print(f"[warn] {msg}", file=sys.stderr)


def http_json(url: str, *, data: bytes | None = None, headers: dict | None = None, timeout: int = 30):
    """GET (or POST if data) returning parsed JSON, or None on any failure."""
    hdrs = {"User-Agent": UA, "Accept": "application/json"}
    if headers:
        hdrs.update(headers)
    req = urllib.request.Request(url, data=data, headers=hdrs, method="POST" if data else "GET")
    try:
        with urllib.request.urlopen(req, timeout=timeout) as resp:
            return json.loads(resp.read().decode("utf-8", "replace"))
    except urllib.error.HTTPError as e:
        warn(f"HTTP {e.code} from {url.split('?')[0]}")
    except (urllib.error.URLError, TimeoutError) as e:
        warn(f"network error for {url.split('?')[0]}: {e}")
    except json.JSONDecodeError:
        warn(f"non-JSON response from {url.split('?')[0]}")
    return None


def _clean(s):
    return html.unescape(str(s or "").strip())


def opp(uid, source, title, agency, otype, naics, posted, due, set_aside, url, summary=""):
    return {
        "uid": uid, "source": source, "title": _clean(title),
        "agency": _clean(agency), "otype": _clean(otype),
        "naics": _clean(naics), "posted": _clean(posted),
        "due": _clean(due), "set_aside": _clean(set_aside),
        "url": (url or "").strip(), "summary": _clean(summary),
    }


# ---------------------------------------------------------------------------- connectors
def fetch_sam(cfg: dict) -> list[dict]:
    """SAM.gov Contract Opportunities (needs a free api.data.gov key)."""
    key = cfg.get("sam_api_key") or os.environ.get("SAM_API_KEY", "")
    if not key:
        warn("SAM: no api key (set sam_api_key in config or SAM_API_KEY env) — skipping. "
             "Get a free key at https://open.gsa.gov/api/get-opportunities-public-api/")
        return []
    days = int(cfg.get("since_days", 14))
    pf = (datetime.now() - timedelta(days=days)).strftime("%m/%d/%Y")
    pt = datetime.now().strftime("%m/%d/%Y")
    out, seen = [], set()
    for ncode in cfg.get("naics", []):
        q = urllib.parse.urlencode({
            "api_key": key, "postedFrom": pf, "postedTo": pt,
            "ncode": ncode, "limit": 100,
```

```
        })
        data = http_json(f"https://api.sam.gov/opportunities/v2/search?{q}")
        if not data:
            continue
        for r in data.get("opportunitiesData", []) or []:
            nid = r.get("noticeId", "")
            if not nid or nid in seen:
                continue
            seen.add(nid)
            out.append(opp(
                f"sam:{nid}", "SAM", r.get("title"), r.get("fullParentPathName"),
                r.get("type"), r.get("naicsCode"), r.get("postedDate"),
                r.get("responseDeadLine"), r.get("typeOfSetAsideDescription"),
                r.get("uiLink") or f"https://sam.gov/opp/{nid}/view",
            ))
    return out


def fetch_grants(cfg: dict) -> list[dict]:
    """Grants.gov search2 (keyless). Filtered to for-profit / small-business eligibility,
    with an exclusion list that drops the academic/biomedical research noise a for-profit
    services shop can't win — so 'assistance' results are actually applicable (SBIR/STTR/BAA)."""
    out, seen = [], set()
    statuses = cfg.get("grants_statuses", "forecasted|posted")
    elig = cfg.get("grants_eligibilities", "06|21|25|99")  # for-profit, small biz, others, unrestricted
    exclude = [x.lower() for x in cfg.get("grants_exclude", [])]
    for kw in cfg.get("grants_keywords", ["artificial intelligence"]):
        body = json.dumps({"keyword": kw, "oppStatuses": statuses,
                           "eligibilities": elig, "rows": 50}).encode()
        data = http_json("https://api.grants.gov/v1/api/search2", data=body,
                         headers={"Content-Type": "application/json"})
        if not data or not isinstance(data.get("data"), dict):
            continue
        for r in data["data"].get("oppHits", []) or []:
            gid = str(r.get("id", ""))
            if not gid or gid in seen:
                continue
            if any(x in f"{r.get('title','')} {r.get('agency','')}".lower() for x in exclude):
                continue
            seen.add(gid)
            out.append(opp(
                f"grants:{gid}", "Grants", r.get("title"), r.get("agency"),
                r.get("docType"), "", r.get("openDate"), r.get("closeDate"),
                "", f"https://www.grants.gov/search-results-detail/{gid}",
            ))
    return out


def fetch_sbir(cfg: dict) -> list[dict]:
    """SBIR.gov open solicitations (keyless, needs UA, rate-limited — best effort)."""
    data = http_json("https://api.www.sbir.gov/public/api/solicitations?open=1",
                     headers={"User-Agent": "Mozilla/5.0"})
    if not isinstance(data, list):
        warn("SBIR: API unavailable/rate-limited right now — skipping (retry later).")
        return []
    out = []
    for r in data:
        if not isinstance(r, dict):
            continue
        sid = str(r.get("solicitation_id") or r.get("solicitation_number") or r.get("solicitation_title", ""))
        out.append(opp(
            f"sbir:{sid}", "SBIR", r.get("solicitation_title"),
            r.get("agency") or r.get("branch"), "SBIR/STTR", "",
            r.get("release_date") or r.get("open_date"), r.get("close_date"),
            "", r.get("sbir_topic_link") or r.get("solicitation_agency_url") or "https://www.sbir.gov/solicitations",
        ))
    return out


def fetch_usajobs(cfg: dict) -> list[dict]:
    """USAJOBS federal employment search (free key from developer.usajobs.gov +
    the registered email as the User-Agent header). This is the JOBS track — federal
    W-2 positions, distinct from the contract/grant opportunities above; toggle it in
    config 'sources'. Doubles as a working reference for a search-and-apply aggregator.

    Note: USAJOBS has a clean public SEARCH API but no public APPLY API — ApplyURI
```

```
    deep-links into each agency's own system. Aggregate + stage, don't auto-submit.
    """
    key = os.environ.get("USAJOBS_API_KEY", "")
    email = os.environ.get("USAJOBS_EMAIL", "")
    if not key or not email:
        warn("USAJOBS: set USAJOBS_API_KEY + USAJOBS_EMAIL in .env "
             "(free key at https://developer.usajobs.gov) — skipping.")
        return []
    hdrs = {"Host": "data.usajobs.gov", "User-Agent": email, "Authorization-Key": key}
    out, seen = [], set()
    for kw in cfg.get("usajobs_keywords", ["artificial intelligence"]):
        q = urllib.parse.urlencode({"Keyword": kw, "ResultsPerPage": 50})
        data = http_json(f"https://data.usajobs.gov/api/search?{q}", headers=hdrs)
        if not data:
            continue
        for it in (data.get("SearchResult") or {}).get("SearchResultItems", []) or []:
            d = it.get("MatchedObjectDescriptor") or {}
            pid = str(it.get("MatchedObjectId") or d.get("PositionID") or "")
            if not pid or pid in seen:
                continue
            seen.add(pid)
            apply_uri = d.get("ApplyURI") or []
            url = (apply_uri[0] if isinstance(apply_uri, list) and apply_uri
                   else d.get("PositionURI") or "")
            pay = d.get("PositionRemuneration") or []
            sal = ""
            if isinstance(pay, list) and pay:
                p0 = pay[0]
                sal = f"${p0.get('MinimumRange','?')}–{p0.get('MaximumRange','?')}/{p0.get('RateIntervalCode','')}"
            out.append(opp(
                f"usajobs:{pid}", "USAJOBS", d.get("PositionTitle"),
                d.get("OrganizationName"), "Federal Job", "",
                d.get("PublicationStartDate"), d.get("ApplicationCloseDate"),
                "", url, summary=f"{d.get('PositionLocationDisplay','')} · {sal}",
            ))
    return out


def fetch_adzuna(cfg: dict) -> list[dict]:
    """Adzuna job aggregator — commercial + public-sector postings from across the web
    (free app_id + app_key at developer.adzuna.com). The private-sector complement to
    USAJOBS on the JOBS track; toggle in config 'sources'. Trial plan is rate-limited.

    Like USAJOBS: aggregate + deep-link via redirect_url; there's no public apply API.
    """
    app_id = os.environ.get("ADZUNA_APP_ID", "")
    app_key = os.environ.get("ADZUNA_APP_KEY", "")
    if not app_id or not app_key:
        warn("ADZUNA: set ADZUNA_APP_ID + ADZUNA_APP_KEY in .env "
             "(free at https://developer.adzuna.com) — skipping.")
        return []
    country = cfg.get("adzuna_country", "us")
    out, seen = [], set()
    for kw in cfg.get("adzuna_keywords", ["artificial intelligence"]):
        q = urllib.parse.urlencode({
            "app_id": app_id, "app_key": app_key,
            "results_per_page": 50, "what": kw,
            "sort_by": "date", "max_days_old": int(cfg.get("since_days", 14)),
        })
        data = http_json(f"https://api.adzuna.com/v1/api/jobs/{country}/search/1?{q}")
        if not data:
            continue
        for r in data.get("results", []) or []:
            jid = str(r.get("id") or "")
            if not jid or jid in seen:
                continue
            seen.add(jid)
            company = (r.get("company") or {}).get("display_name", "")
            loc = (r.get("location") or {}).get("display_name", "")
            smin, smax = r.get("salary_min"), r.get("salary_max")
            sal = f"${int(smin):,}–${int(smax):,}" if smin and smax else ""
            out.append(opp(
                f"adzuna:{jid}", "ADZUNA", r.get("title"), company,
                (r.get("category") or {}).get("label", "Job"), "",
                r.get("created"), "", "", r.get("redirect_url", ""),
                summary=f"{loc} · {sal}".strip(" ·"),
```

```
            ))
    return out


def _job_match(title: str, cfg: dict) -> bool:
    """Relevance gate for firehose job sources — title must hit a job_filter term AND
    avoid every job_exclude term (cuts sales/recruiting/marketing noise)."""
    t = (title or "").lower()
    if any(x in t for x in cfg.get("job_exclude", [])):
        return False
    kws = cfg.get("job_filter", [])
    return (not kws) or any(k in t for k in kws)


def fetch_himalayas(cfg: dict) -> list[dict]:
    """Himalayas remote jobs (keyless, ~88k, structured salary)."""
    out, seen = [], set()
    for off in (0, 20, 40, 60):
        data = http_json(f"https://himalayas.app/jobs/api?limit=20&offset={off}")
        for r in (data or {}).get("jobs", []) or []:
            if not _job_match(r.get("title"), cfg):
                continue
            uid = f"himalayas:{r.get('title','')}-{r.get('companyName','')}"
            if uid in seen:
                continue
            seen.add(uid)
            lo, hi = r.get("minSalary"), r.get("maxSalary")
            sal = f"${lo:,}-${hi:,}" if isinstance(lo, int) and isinstance(hi, int) else ""
            out.append(opp(uid, "Himalayas", r.get("title"), r.get("companyName"),
                           "Remote Job", "", r.get("pubDate", ""), "", "",
                           r.get("applicationLink") or r.get("url", ""), summary=sal))
    return out


def fetch_jobicy(cfg: dict) -> list[dict]:
    """Jobicy remote jobs (keyless, salary)."""
    out, seen = [], set()
    data = http_json("https://jobicy.com/api/v2/remote-jobs?count=100&geo=usa")
    for r in (data or {}).get("jobs", []) or []:
        if not _job_match(r.get("jobTitle"), cfg):
            continue
        uid = f"jobicy:{r.get('id') or r.get('url', '')}"
        if uid in seen:
            continue
        seen.add(uid)
        lo, hi = r.get("salaryMin"), r.get("salaryMax")
        sal = f"${int(lo):,}-${int(hi):,}" if lo and hi else ""
        out.append(opp(uid, "Jobicy", r.get("jobTitle"), r.get("companyName"),
                       "Remote Job", "", r.get("pubDate", ""), "", "",
                       r.get("url", ""), summary=sal))
    return out


def fetch_themuse(cfg: dict) -> list[dict]:
    """The Muse jobs (keyless)."""
    out, seen = [], set()
    for cat in cfg.get("themuse_categories", ["Software Engineering", "Data Science"]):
        data = http_json(
            f"https://www.themuse.com/api/public/jobs?page=1&category={urllib.parse.quote(cat)}")
        for r in (data or {}).get("results", []) or []:
            if not _job_match(r.get("name"), cfg):
                continue
            uid = f"themuse:{r.get('id') or r.get('name', '')}"
            if uid in seen:
                continue
            seen.add(uid)
            locs = ", ".join(l.get("name", "") for l in (r.get("locations") or [])[:2])
            out.append(opp(uid, "TheMuse", r.get("name"),
                           (r.get("company") or {}).get("name"), "Job", "",
                           r.get("publication_date", ""), "", "",
                           (r.get("refs") or {}).get("landing_page", ""), summary=locs))
    return out


def fetch_greenhouse(cfg: dict) -> list[dict]:
    """Greenhouse ATS — jobs straight from target employers' boards (keyless).
```

```
    Curate `greenhouse_companies` in config to the employers you want to watch —
    this is the highest-precision job source because YOU choose the companies."""
    out = []
    for tok in cfg.get("greenhouse_companies", []):
        data = http_json(f"https://boards-api.greenhouse.io/v1/boards/{tok}/jobs?content=true")
        for r in (data or {}).get("jobs", []) or []:
            if not _job_match(r.get("title"), cfg):
                continue
            out.append(opp(f"gh:{tok}:{r.get('id')}", "Greenhouse", r.get("title"),
                           tok.replace("-", " ").title(), "Job", "", r.get("updated_at", ""),
                           "", "", r.get("absolute_url", ""),
                           summary=(r.get("location") or {}).get("name", "")))
    return out


def fetch_lever(cfg: dict) -> list[dict]:
    """Lever ATS — jobs straight from target employers (keyless). Curate `lever_companies`."""
    out = []
    for co in cfg.get("lever_companies", []):
        data = http_json(f"https://api.lever.co/v0/postings/{co}?mode=json")
        if not isinstance(data, list):
            continue
        for r in data:
            if not _job_match(r.get("text"), cfg):
                continue
            cats = r.get("categories") or {}
            out.append(opp(f"lever:{co}:{r.get('id')}", "Lever", r.get("text"),
                           co.replace("-", " ").title(), "Job", "", "", "", "",
                           r.get("hostedUrl") or r.get("applyUrl", ""),
                           summary=cats.get("location", "")))
    return out


def fetch_federalregister(cfg: dict) -> list[dict]:
    """Federal Register notices (keyless) — forward-demand signal: agencies post
    sources-sought / RFIs / notices here, often before or alongside the SAM solicitation."""
    out, seen = [], set()
    for term in cfg.get("federalregister_terms", ["sources sought", "artificial intelligence"]):
        url = ("https://www.federalregister.gov/api/v1/documents.json?per_page=50&order=newest"
               f"&conditions[type][]=NOTICE&conditions[term]={urllib.parse.quote(term)}")
        data = http_json(url)
        for r in (data or {}).get("results", []) or []:
            uid = f"fedreg:{r.get('document_number')}"
            if uid in seen:
                continue
            seen.add(uid)
            ag = ", ".join(a.get("name", "") for a in (r.get("agencies") or [])[:2])
            out.append(opp(uid, "FedRegister", r.get("title"), ag,
                           r.get("type", "Notice"), "", r.get("publication_date", ""),
                           "", "", r.get("html_url", ""),
                           summary=(r.get("abstract") or "")[:160]))
    return out


def fetch_hn_hiring(cfg: dict) -> list[dict]:
    """Hacker News 'Who is Hiring' (keyless via Algolia) — the latest monthly thread's
    hiring comments, one self-declared hiring company each. Startup/tech-heavy, fresh."""
    s = http_json("https://hn.algolia.com/api/v1/search_by_date"
                  "?query=Ask%20HN%20Who%20is%20hiring&tags=story,author_whoishiring&hitsPerPage=1")
    hits = (s or {}).get("hits") or []
    if not hits:
        return []
    thread = http_json(f"https://hn.algolia.com/api/v1/items/{hits[0].get('objectID')}")
    out = []
    for ch in (thread or {}).get("children", []) or []:
        raw = ch.get("text") or ""
        if not raw:
            continue
        txt = html.unescape(re.sub(r"\s+", " ", re.sub("<[^>]+>", " ", raw))).strip()
        if not _job_match(txt, cfg):
            continue
        out.append(opp(f"hn:{ch.get('id')}", "HN-Hiring", txt[:110], "", "Hiring post",
                       "", "", "", "", f"https://news.ycombinator.com/item?id={ch.get('id')}",
                       summary=txt[:200]))
    return out
```

```
def fetch_yc(cfg: dict) -> list[dict]:
    """Y Combinator companies currently hiring (keyless community mirror). An employer-target
    feeder: relevant YC cos to add to your Greenhouse/Lever watchlists."""
    data = http_json("https://yc-oss.github.io/api/companies/hiring.json")
    if not isinstance(data, list):
        return []
    out = []
    for c in data:
        if not _job_match(f"{c.get('name','')} {c.get('one_liner','')}", cfg):
            continue
        out.append(opp(f"yc:{c.get('id') or c.get('name')}", "YC",
                       f"{c.get('name')} (hiring)", c.get("name"), "Startup", "",
                       "", "", "", c.get("website") or c.get("url", ""),
                       summary=(c.get("one_liner") or "")[:140]))
    return out


def fetch_remoteok(cfg: dict) -> list[dict]:
    """RemoteOK remote jobs (keyless; needs a browser UA). The first array element is a
    legal/metadata notice with no 'position' — skipped by the guard below."""
    data = http_json("https://remoteok.com/api", headers={"User-Agent": "Mozilla/5.0"})
    if not isinstance(data, list):
        warn("RemoteOK: API unavailable right now — skipping.")
        return []
    out, seen = [], set()
    for r in data:
        if not isinstance(r, dict) or not r.get("position"):
            continue
        if not _job_match(r.get("position"), cfg):
            continue
        rid = str(r.get("id") or r.get("slug") or r.get("url", ""))
        if not rid or rid in seen:
            continue
        seen.add(rid)
        lo, hi = r.get("salary_min"), r.get("salary_max")
        sal = f"${int(lo):,}–${int(hi):,}" if lo and hi else ""
        loc = r.get("location") or "Remote"
        out.append(opp(f"remoteok:{rid}", "RemoteOK", r.get("position"), r.get("company"),
                       "Remote Job", "", r.get("date", ""), "", "",
                       r.get("url") or f"https://remoteok.com/remote-jobs/{rid}",
                       summary=f"{loc} · {sal}".strip(" ·")))
    return out


def fetch_remotive(cfg: dict) -> list[dict]:
    """Remotive remote jobs (keyless). Pulls per search term (defaults to job_filter)."""
    out, seen = [], set()
    terms = cfg.get("remotive_search") or cfg.get("job_filter", []) or [""]
    for term in terms[:6]:
        url = "https://remotive.com/api/remote-jobs?limit=50"
        if term:
            url += "&search=" + urllib.parse.quote(term)
        data = http_json(url)
        for r in (data or {}).get("jobs", []) or []:
            if not _job_match(r.get("title"), cfg):
                continue
            rid = str(r.get("id") or r.get("url", ""))
            if not rid or rid in seen:
                continue
            seen.add(rid)
            sm = f"{r.get('candidate_required_location','')} · {r.get('salary','')}".strip(" ·")
            out.append(opp(f"remotive:{rid}", "Remotive", r.get("title"), r.get("company_name"),
                           "Remote Job", "", r.get("publication_date", ""), "", "",
                           r.get("url", ""), summary=sm))
    return out


def fetch_findwork(cfg: dict) -> list[dict]:
    """Findwork.dev jobs (free token: header 'Authorization: Token <key>'; set
    FINDWORK_API_KEY in .env, key at https://findwork.dev/developers/). Skips without a key."""
    key = os.environ.get("FINDWORK_API_KEY", "")
    if not key:
        warn("FINDWORK: set FINDWORK_API_KEY in .env (free at https://findwork.dev/developers/) — skipping.")
        return []
    hdrs = {"Authorization": f"Token {key}"}
```

```
    out, seen = [], set()
    for kw in (cfg.get("findwork_search") or cfg.get("job_filter") or ["engineer"])[:6]:
        q = urllib.parse.urlencode({"search": kw, "sort_by": "date"})
        data = http_json(f"https://findwork.dev/api/jobs/?{q}", headers=hdrs)
        for r in (data or {}).get("results", []) or []:
            if not _job_match(r.get("role"), cfg):
                continue
            rid = str(r.get("id", ""))
            if not rid or rid in seen:
                continue
            seen.add(rid)
            loc = r.get("location") or ("Remote" if r.get("remote") else "")
            out.append(opp(f"findwork:{rid}", "Findwork", r.get("role"), r.get("company_name"),
                           "Job", "", r.get("date_posted", ""), "", "",
                           r.get("url", ""), summary=loc))
    return out


def fetch_jooble(cfg: dict) -> list[dict]:
    """Jooble aggregator (free key: POST https://jooble.org/api/<key>; set JOOBLE_API_KEY in
    .env, request at https://jooble.org/api/about). Skips without a key."""
    key = os.environ.get("JOOBLE_API_KEY", "")
    if not key:
        warn("JOOBLE: set JOOBLE_API_KEY in .env (free at https://jooble.org/api/about) — skipping.")
        return []
    loc = cfg.get("jooble_location", "")
    out, seen = [], set()
    for kw in (cfg.get("jooble_keywords") or cfg.get("job_filter") or ["program manager"])[:6]:
        body = json.dumps({"keywords": kw, "location": loc}).encode()
        data = http_json(f"https://jooble.org/api/{key}", data=body,
                         headers={"Content-Type": "application/json"})
        for r in (data or {}).get("jobs", []) or []:
            if not _job_match(r.get("title"), cfg):
                continue
            rid = str(r.get("id") or r.get("link", ""))
            if not rid or rid in seen:
                continue
            seen.add(rid)
            sm = f"{r.get('location','')} · {r.get('salary','')}".strip(" ·")
            out.append(opp(f"jooble:{rid}", "Jooble", r.get("title"), r.get("company"),
                           "Job", "", r.get("updated", ""), "", "",
                           r.get("link", ""), summary=sm))
    return out


def fetch_usaspending(cfg: dict) -> list[dict]:
    """USAspending.gov awards (keyless, NO quota) — who already WON contracts in this lane,
    and for how much. Intel, not open opportunities: surfaces incumbents, award amounts and
    agencies for competitor/teaming targeting. Filtered by the profile's NAICS over a window
    (usaspending_lookback_days, default 730). Each opp carries _recipient/_amount extras for
    intel views like make_incumbents.py."""
    naics = [str(n) for n in cfg.get("naics", []) if n][:25]
    if not naics:
        return []
    end = datetime.now().strftime("%Y-%m-%d")
    start = (datetime.now() - timedelta(days=int(cfg.get("usaspending_lookback_days", 730)))).strftime("%Y-%m-%d")
    body = json.dumps({
        "filters": {
            "award_type_codes": ["A", "B", "C", "D"],
            "naics_codes": naics,
            "time_period": [{"start_date": start, "end_date": end}],
        },
        "fields": ["Award ID", "Recipient Name", "Award Amount", "Awarding Agency",
                   "Awarding Sub Agency", "Start Date", "End Date", "Description"],
        "sort": "Award Amount", "order": "desc", "limit": 100, "page": 1,
    }).encode()
    data = http_json("https://api.usaspending.gov/api/v2/search/spending_by_award/",
                     data=body, headers={"Content-Type": "application/json"})
    out, seen = [], set()
    for r in (data or {}).get("results", []) or []:
        gid = r.get("generated_internal_id") or ""
        aid = str(r.get("Award ID") or gid or "")
        if not aid or aid in seen:
            continue
        seen.add(aid)
        amt = r.get("Award Amount") or 0
```

```
        recip = r.get("Recipient Name") or "?"
        try:
            amt_s = f"${float(amt):,.0f}"
        except (TypeError, ValueError):
            amt_s = str(amt)
        o = opp(
            f"usaspending:{aid}", "USAspending",
            r.get("Description") or r.get("Award ID") or "(award)",
            r.get("Awarding Agency"), "Award (won)", "",
            r.get("Start Date", ""), r.get("End Date", ""), "",
            f"https://www.usaspending.gov/award/{gid}" if gid else "https://www.usaspending.gov",
            summary=f"Won by {recip} · {amt_s}",
        )
        o["_recipient"], o["_amount"], o["_subagency"] = recip, (amt or 0), r.get("Awarding Sub Agency", "")
        out.append(o)
    return out


CONNECTORS = {"sam": fetch_sam, "grants": fetch_grants, "sbir": fetch_sbir,
              "usajobs": fetch_usajobs, "adzuna": fetch_adzuna,
              "himalayas": fetch_himalayas, "jobicy": fetch_jobicy, "themuse": fetch_themuse,
              "greenhouse": fetch_greenhouse, "lever": fetch_lever,
              "federalregister": fetch_federalregister,
              "hn": fetch_hn_hiring, "yc": fetch_yc,
              "remoteok": fetch_remoteok, "remotive": fetch_remotive,
              "findwork": fetch_findwork, "jooble": fetch_jooble,
              "usaspending": fetch_usaspending}


# ---------------------------------------------------------------------------- scoring
def score_opportunity(o: dict, cfg: dict) -> float:
    """Relevance score. Higher = better fit.

    THIS IS THE TUNING LEVER. Adjust the `scoring` weights in config.json to
    change what surfaces (precision vs recall). See README.
    """
    w = cfg.get("scoring", {})
    text = f"{o['title']} {o['summary']} {o['agency']}".lower()
    s = 0.0
    # keyword hits
    hits = sum(1 for kw in cfg.get("keywords", []) if kw.lower() in text)
    s += hits * float(w.get("keyword_hit", 1.0))
    # NAICS match
    if o["naics"] and o["naics"] in set(cfg.get("naics", [])):
        s += float(w.get("naics_match", 2.0))
    # set-aside match (small-business friendly)
    if o["set_aside"] and any(sa.lower() in o["set_aside"].lower() for sa in cfg.get("set_asides", [])):
        s += float(w.get("setaside_match", 1.5))
    # recency bonus (posted within window)
    if _is_recent(o["posted"], int(cfg.get("since_days", 14))):
        s += float(w.get("recency", 0.5))
    return round(s, 2)


def _is_recent(posted: str, days: int) -> bool:
    for fmt in ("%Y-%m-%d", "%m/%d/%Y", "%Y-%m-%dT%H:%M:%S"):
        try:
            d = datetime.strptime(posted[:19] if "T" in posted else posted[:10], fmt)
            return (datetime.now() - d).days <= days
        except (ValueError, TypeError):
            continue
    return False


# ---------------------------------------------------------------------------- store
SCHEMA = """
CREATE TABLE IF NOT EXISTS opps (
  uid TEXT PRIMARY KEY, source TEXT, title TEXT, agency TEXT, otype TEXT,
  naics TEXT, posted TEXT, due TEXT, set_aside TEXT, url TEXT, summary TEXT,
  score REAL, first_seen TEXT, last_seen TEXT
);
"""


def store_and_diff(opps: list[dict], db_path: str) -> list[dict]:
    """Upsert all opps; return the ones that are NEW (first seen this run)."""
```

```
    con = sqlite3.connect(db_path)
    con.execute(SCHEMA)
    now = datetime.now(timezone.utc).isoformat(timespec="seconds")
    new = []
    for o in opps:
        row = con.execute("SELECT uid FROM opps WHERE uid=?", (o["uid"],)).fetchone()
        if row is None:
            con.execute(
                "INSERT INTO opps VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?)",
                (o["uid"], o["source"], o["title"], o["agency"], o["otype"], o["naics"],
                 o["posted"], o["due"], o["set_aside"], o["url"], o["summary"],
                 o["score"], now, now),
            )
            new.append(o)
        else:
            con.execute("UPDATE opps SET last_seen=?, score=? WHERE uid=?", (now, o["score"], o["uid"]))
    con.commit()
    con.close()
    return new


# ---------------------------------------------------------------------------- output
def render_digest(opps: list[dict], title: str) -> str:
    lines = [f"# {title}", "", f"_{len(opps)} opportunit{'y' if len(opps)==1 else 'ies'} · generated {datetime.now():%Y-
%m-%d %H:%M}_", ""]
    by_src: dict[str, list[dict]] = {}
    for o in sorted(opps, key=lambda x: -x["score"]):
        by_src.setdefault(o["source"], []).append(o)
    for src, items in by_src.items():
        lines.append(f"## {src} ({len(items)})")
        lines.append("")
        lines.append("| Score | Title | Agency | NAICS | Set-aside | Due | Link |")
        lines.append("|---|---|---|---|---|---|---|")
        for o in items:
            t = (o["title"][:70] + "…") if len(o["title"]) > 71 else o["title"]
            lines.append(f"| {o['score']} | {t} | {o['agency'][:30]} | {o['naics']} | "
                         f"{o['set_aside'][:20]} | {o['due'][:10]} | {o['url']} |")
        lines.append("")
    return "\n".join(lines)


# ---------------------------------------------------------------------------- main
def main(argv=None) -> int:
    p = argparse.ArgumentParser(description="Source government/defense/grant contract opportunities.")
    p.add_argument("--config", default=DEFAULT_CONFIG)
    p.add_argument("--db", default=DEFAULT_DB)
    p.add_argument("--digest-dir", default=DEFAULT_DIGEST_DIR)
    p.add_argument("--source", help="comma list: sam,grants,sbir (default: config)")
    p.add_argument("--min-score", type=float, default=None)
    p.add_argument("--since-days", type=int, default=None)
    p.add_argument("--all", action="store_true", help="show all matches, not just new")
    p.add_argument("--version", action="version", version=f"%(prog)s {__version__}")
    args = p.parse_args(argv)

    try:
        with open(args.config) as f:
            cfg = json.load(f)
    except (OSError, json.JSONDecodeError) as e:
        print(f"error: cannot read config {args.config}: {e}", file=sys.stderr)
        return 2
    if args.since_days is not None:
        cfg["since_days"] = args.since_days
    min_score = args.min_score if args.min_score is not None else float(cfg.get("min_score", 1.0))
    enabled = (args.source.split(",") if args.source
               else [s for s, on in cfg.get("sources", {}).items() if on])

    all_opps: list[dict] = []
    for name in enabled:
        fn = CONNECTORS.get(name.strip())
        if not fn:
            warn(f"unknown source '{name}'")
            continue
        got = fn(cfg)
        print(f"  {name}: {len(got)} fetched", file=sys.stderr)
        all_opps.extend(got)
```

```
    for o in all_opps:
        o["score"] = score_opportunity(o, cfg)
    matches = [o for o in all_opps if o["score"] >= min_score]

    new = store_and_diff(matches, args.db)
    show = matches if args.all else new

    os.makedirs(args.digest_dir, exist_ok=True)
    label = "All matches" if args.all else "New opportunities"
    digest = render_digest(show, f"{label} — {datetime.now():%Y-%m-%d}")
    out_path = os.path.join(args.digest_dir, f"digest-{datetime.now():%Y%m%d-%H%M}.md")
    with open(out_path, "w") as f:
        f.write(digest)

    print(f"\n{len(all_opps)} fetched · {len(matches)} above score {min_score} · "
          f"{len(new)} new · digest → {out_path}")
    for o in sorted(show, key=lambda x: -x["score"])[:10]:
        print(f"  [{o['score']:>4}] {o['source']:<6} {o['title'][:64]}")
    return 0


if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        sys.exit(1)
```

make_sheet.py

```
#!/usr/bin/env python3
"""make_sheet.py — generate a filterable .xlsx job spreadsheet for a profile.

Pulls every live job connector with the profile's search settings, fit-ranks the
results against the profile's keywords, dedupes, and writes <profile>-jobs.xlsx —
filterable by Source and Fit, every row a clickable Apply link. The spreadsheet
counterpart to make_radar.py (which renders the same data as an HTML dashboard).

Usage:
    python3 make_sheet.py            # default profile: matthew
    python3 make_sheet.py nicholas

Profiles live in config.json:
  - profiles.<name>        — fit keywords (matched against the job title)
  - sheet_profiles.<name>  — optional per-profile search OVERRIDES (job_filter,
                             per-connector keywords, job_exclude). With no override
                             the profile just uses the top-level config.

Requires openpyxl (the rest of the engine is stdlib-only; the .xlsx writer is not).
"""
from __future__ import annotations

import json
import os
import re
import sys
from collections import Counter

import source  # loads .env on import (connector API keys)

try:
    import openpyxl
    from openpyxl.styles import Border, Font, PatternFill, Side
except ImportError:
    sys.exit("make_sheet needs openpyxl:  python3 -m pip install openpyxl")

HERE = os.path.dirname(os.path.abspath(__file__))

# the job-board connectors (not the contract/grant ones) — these feed a job sheet
JOB_CONNECTORS = ("adzuna", "usajobs", "greenhouse", "lever", "jobicy", "himalayas",
                  "themuse", "remoteok", "remotive", "findwork", "jooble")

# config keys a profile may override to retarget the search to its lane
_OVERRIDE_KEYS = ("job_filter", "job_exclude", "adzuna_keywords", "usajobs_keywords",
```

```
                  "remotive_search", "findwork_search", "jooble_keywords",
                  "jooble_location", "themuse_categories", "greenhouse_companies",
                  "lever_companies")

DEFAULT_FIT = ["ai", "agent", "llm", "machine learning", "engineer", "developer"]


def main(profile: str = "matthew") -> int:
    cfg = json.load(open(os.path.join(HERE, "config.json")))
    fit_kws = (cfg.get("profiles") or {}).get(profile) or DEFAULT_FIT
    sp = (cfg.get("sheet_profiles") or {}).get(profile, {})
    run = {**cfg, **{k: v for k, v in sp.items() if k in _OVERRIDE_KEYS}}

    jobs: list[dict] = []
    for name in JOB_CONNECTORS:
        try:
            got = source.CONNECTORS[name](run)
        except Exception as e:  # noqa: BLE001 — one bad source must not kill the sheet
            source.warn(f"{name}: {e}")
            got = []
        print(f"  {name:11}: {len(got)} fetched", file=sys.stderr)
        jobs.extend(got)

    def norm(s: str) -> str:
        return re.sub(r"[^a-z0-9]", "", (s or "").lower())

    seen: set = set()
    ranked: list[tuple[int, dict]] = []
    for j in jobs:
        key = (norm(j["title"]), norm(j["agency"]))
        if not key[0] or key in seen:
            continue
        seen.add(key)
        fit = sum(1 for kw in fit_kws if kw in (j["title"] or "").lower())
        if fit > 0:
            ranked.append((fit, j))
    ranked.sort(key=lambda x: -x[0])

    wb = openpyxl.Workbook()
    ws = wb.active
    ws.title = "Job Matches"
    ws.append(["#", "Fit", "Source", "Title", "Company / Agency", "Location", "Apply"])
    navy = PatternFill("solid", start_color="1A3A5C")
    thin = Side(style="thin", color="D6DEE6")
    for c in range(1, 8):
        cell = ws.cell(row=1, column=c)
        cell.font = Font(bold=True, color="FFFFFF", size=11)
        cell.fill = navy
    for idx, (fit, j) in enumerate(ranked, 1):
        ws.append([idx, fit, j["source"], j["title"], j["agency"], (j["summary"] or "")[:60], ""])
        link = ws.cell(row=idx + 1, column=7)
        if j["url"]:
            link.hyperlink = j["url"]
            link.value = "Apply ▸"
            link.font = Font(color="1558B0", underline="single", bold=True)
        for c in range(1, 8):
            ws.cell(row=idx + 1, column=c).border = Border(bottom=thin)
    ws.freeze_panes = "A2"
    ws.auto_filter.ref = f"A1:G{len(ranked) + 1}"
    for col, width in zip("ABCDEFG", (5, 5, 11, 48, 28, 28, 11)):
        ws.column_dimensions[col].width = width

    out = os.path.join(os.path.dirname(HERE), f"{profile}-jobs.xlsx")
    wb.save(out)
    print(f"{len(jobs)} fetched · {len(ranked)} fit-ranked rows · "
          f"by source: {dict(Counter(j['source'] for _, j in ranked))}")
    print(f"wrote {out}")
    return 0


if __name__ == "__main__":
    raise SystemExit(main(sys.argv[1] if len(sys.argv) > 1 else "matthew"))
```

make_radar.py

```
#!/usr/bin/env python3
"""make_radar.py — generate a shareable 'FedSource Work Radar' HTML dashboard.

Pulls the live job sources (USAJOBS + Adzuna), dedupes, fit-ranks against a profile,
and layers in Adzuna market intel (top employers + salary histogram). Output is one
self-contained HTML file — open it, send it, or deploy it.

Usage: python3 make_radar.py
"""
from __future__ import annotations

import html
import json
import os
import re
import sqlite3
import sys
import urllib.parse
import urllib.request
from datetime import datetime

import source  # loads .env on import (needed for Adzuna intel keys)

JOB_SOURCES = ("ADZUNA", "USAJOBS", "Greenhouse", "Lever", "Jobicy", "Himalayas", "TheMuse",
               "RemoteOK", "Remotive", "Findwork", "Jooble")

HERE = os.path.dirname(os.path.abspath(__file__))

# Per-person fit profile — keywords matched against the job title (weighted).
PROFILE = [
    "ai", "agent", "mcp", "model context", "llm", "machine learning", "blockchain",
    "web3", "solidity", "rust", "developer relations", "devrel", "forward deployed",
    "solutions engineer", "full stack", "founding", "crypto", "autonomy", "applied",
]


def main(profile: str = "matthew") -> int:
    cfg = json.load(open(os.path.join(HERE, "config.json")))
    profile_kws = (cfg.get("profiles") or {}).get(profile) or PROFILE

    db = sqlite3.connect(os.path.join(HERE, "opportunities.db"))
    rows = db.execute(
        "SELECT title, agency, source, url, summary FROM opps "
        f"WHERE source IN ({','.join('?' * len(JOB_SOURCES))})", JOB_SOURCES).fetchall()
    jobs = [{"title": t, "agency": a, "source": s, "url": u, "summary": sm}
            for t, a, s, u, sm in rows]

    def norm(s: str) -> str:
        return re.sub(r"[^a-z0-9]", "", (s or "").lower())

    seen, deduped = set(), []
    for j in jobs:
        k = (norm(j["title"]), norm(j["agency"]))
        if k[0] and k not in seen:
            seen.add(k)
            deduped.append(j)

    for j in deduped:
        t = (j["title"] or "").lower()
        j["fit"] = sum(1 for kw in profile_kws if kw in t)
    ranked = sorted([j for j in deduped if j["fit"] > 0], key=lambda x: -x["fit"])
    top = ranked[:40]
    from collections import Counter
    src_counts = Counter(j["source"] for j in jobs)

    aid, akey = os.environ.get("ADZUNA_APP_ID", ""), os.environ.get("ADZUNA_APP_KEY", "")

    def adz(path: str, **p) -> dict:
        p.update(app_id=aid, app_key=akey)
        try:
            url = f"https://api.adzuna.com/v1/api/jobs/us/{path}?{urllib.parse.urlencode(p)}"
            return json.load(urllib.request.urlopen(url, timeout=30))
        except Exception:  # noqa: BLE001
            return {}

    companies = (adz("top_companies", what="artificial intelligence engineer").get("leaderboard") or [])[:10]
```

```
    hist = adz("histogram", what="machine learning engineer").get("histogram") or {}

    def esc(s) -> str:
        return html.escape(str(s or ""))

    now = datetime.now().strftime("%Y-%m-%d %H:%M")
    peak = max(hist, key=hist.get) if hist else None
    peak_lbl = f"${int(peak) // 1000}k+" if peak else "—"

    job_rows = "".join(
        f'<tr><td class="fit">{"●" * min(j["fit"], 4)}</td>'
        f'<td class="title"><a href="{esc(j["url"])}" target="_blank">{esc(j["title"])}</a></td>'
        f'<td>{esc(j["agency"])}</td>'
        f'<td class="src {"fed" if j["source"] == "USAJOBS" else "com"}">{esc(j["source"])}</td>'
        f'<td class="sal">{esc(j["summary"])}</td></tr>'
        for j in top
    )

    maxc = max((c.get("count", 0) for c in companies), default=1)
    comp_rows = "".join(
        f'<div class="bar"><span class="lbl">{esc(c.get("canonical_name"))}</span>'
        f'<span class="track"><span class="fill" style="width:{int(100 * c.get("count", 0) / maxc)}%"></span></span>'
        f'<span class="num">{c.get("count", 0):,}</span></div>'
        for c in companies
    )

    maxh = max(hist.values(), default=1)
    hist_rows = "".join(
        f'<div class="bar"><span class="lbl">${int(b):,}</span>'
        f'<span class="track"><span class="fill sal" style="width:{int(100 * hist[b] / maxh)}%"></span></span>'
        f'<span class="num">{hist[b]}</span></div>'
        for b in sorted(hist, key=lambda x: int(x))
    )

    maxs = max(src_counts.values(), default=1)
    src_rows = "".join(
        f'<div class="bar"><span class="lbl">{esc(s)}</span>'
        f'<span class="track"><span class="fill" style="width:{int(100 * src_counts[s] / maxs)}%"></span></span>'
        f'<span class="num">{src_counts[s]:,}</span></div>'
        for s in sorted(src_counts, key=lambda x: -src_counts[x])
    )

    css = (
        "*{box-sizing:border-box;margin:0;padding:0}"
        "body{background:#0b0e14;color:#dfe5ee;font:15px/1.5 -apple-system,Segoe UI,Roboto,sans-serif;padding:34px;max-
width:1000px;margin:0 auto}"
        "a{color:#7dd3fc;text-decoration:none}a:hover{text-decoration:underline}"
        ".head{border-bottom:1px solid #1e2636;padding-bottom:18px;margin-bottom:24px}"
        "h1{font-size:27px;font-weight:700;letter-spacing:-.5px}h1 .d{color:#4ade80}"
        ".sub{color:#8b97a8;font-size:13px;margin-top:6px}"
        ".stats{display:grid;grid-template-columns:repeat(4,1fr);gap:14px;margin-bottom:8px}"
        ".stat{background:#121826;border:1px solid #1e2636;border-radius:10px;padding:16px}"
        ".stat b{display:block;font-size:25px;color:#4ade80;font-weight:700}.stat span{font-size:12px;color:#8b97a8}"
        "h2{font-size:12px;text-transform:uppercase;letter-spacing:1.2px;color:#8b97a8;margin:30px 0 12px}"
        "table{width:100%;border-collapse:collapse;font-size:13.5px}"
        "td{padding:8px 10px;border-bottom:1px solid #161d2b;vertical-align:top}"
        ".fit{color:#4ade80;font-size:9px;white-space:nowrap;letter-spacing:-1px}"
        ".title{max-width:360px}.src{font-size:10px;font-weight:700;letter-spacing:.5px}"
        ".src.fed{color:#7dd3fc}.src.com{color:#fbbf24}.sal{color:#8b97a8;font-size:12px}"
        ".bar{display:flex;align-items:center;gap:10px;margin:5px 0;font-size:12.5px}"
        ".bar .lbl{width:160px;text-align:right;color:#bfc8d6;white-space:nowrap;overflow:hidden;text-overflow:ellipsis}"
        ".bar .track{flex:1;background:#161d2b;border-radius:4px;height:14px;overflow:hidden}"
        ".bar .fill{display:block;height:100%;background:#3b82f6;border-radius:4px}.bar .sal{background:#4ade80}"
        ".bar .num{width:58px;color:#8b97a8;font-size:11px}"
        ".foot{margin-top:38px;padding-top:18px;border-top:1px solid #1e2636;color:#8b97a8;font-size:12.5px;line-
height:1.8}"
        ".foot code{background:#161d2b;padding:1px 6px;border-radius:4px;color:#7dd3fc}"
    )

    page = (
        f"<!doctype html><html lang=en><head><meta charset=utf-8>"
        f'<meta name=viewport content="width=device-width,initial-scale=1">'
        f"<title>FedSource · Work Radar</title><style>{css}</style></head><body>"
        f'<div class=head><h1>FedSource <span class=d>·</span> Work Radar — {profile.title()}</h1>'
        f'<div class=sub>Live across {len(JOB_SOURCES)} job sources — federal, commercial &amp;amp; employer-direct.
Generated {now}.</div></div>'
```

```
        f'<div class=stats>'
        f'<div class=stat><b>{len(jobs)}</b><span>jobs scanned</span></div>'
        f'<div class=stat><b>{len(top)}</b><span>best-fit roles</span></div>'
        f'<div class=stat><b>{len(companies)}</b><span>top employers</span></div>'
        f'<div class=stat><b>{peak_lbl}</b><span>ML-eng salary peak</span></div></div>'
        f'<h2>Best-fit roles right now</h2><table><tbody>{job_rows}</tbody></table>'
        f"<h2>By source — what each connector pulls</h2>{src_rows}"
        f"<h2>Who's hiring — top AI employers</h2>{comp_rows}"
        f"<h2>Salary benchmark — Machine Learning Engineer</h2>{hist_rows}"
        f'<div class=foot>Built by Matthew Karsten with <b>FedSource</b> — a stdlib-only aggregator: one engine, 17
sources, ~40 lines per connector.<br>'
        f'The two job feeds are <code>fetch_usajobs</code> + <code>fetch_adzuna</code> — yours to fork.<br>'
        f'<b>Nicholas</b> — send me your stack and I&#39;ll spin up your column. Same engine, your feed. Let&#39;s build
the apply layer together.</div>'
        f"</body></html>"
    )

    fname = ("fedsource-work-radar.html" if profile == "matthew"
             else f"fedsource-work-radar-{profile}.html")
    out = os.path.join(os.path.dirname(HERE), fname)
    with open(out, "w") as f:
        f.write(page)
    print(f"jobs: {len(jobs)} fetched · {len(deduped)} deduped · {len(top)} best-fit shown")
    print(f"intel: {len(companies)} employers · {len(hist)} salary bands")
    print(f"wrote {out}")
    return 0


if __name__ == "__main__":
    raise SystemExit(main(sys.argv[1] if len(sys.argv) > 1 else "matthew"))
```

`config.json` (no secrets — keys live in a gitignored .env)

```
{
  "sam_api_key": "",
  "since_days": 14,
  "sources": {
    "sam": true,
    "grants": true,
    "sbir": true,
    "usajobs": true,
    "adzuna": true,
    "himalayas": true,
    "jobicy": true,
    "themuse": true,
    "greenhouse": true,
    "lever": true,
    "federalregister": true,
    "hn": true,
    "yc": true,
    "remoteok": true,
    "remotive": true,
    "findwork": true,
    "jooble": true,
    "usaspending": true
  },
  "naics": [
    "541511",
    "541512",
    "541519",
    "518210",
    "541715",
    "512110"
  ],
  "keywords": [
    "agentic",
    "ai agent",
    "mcp",
    "model context protocol",
    "llm",
    "rag",
    "artificial intelligence",
    "machine learning",
```

```
    "autonomy",
    "data pipeline",
    "indexer",
    "dashboard",
    "analytics",
    "automation",
    "edge",
    "serverless",
    "cybersecurity",
    "zero trust",
    "blockchain",
    "website",
    "web development",
    "web application",
    "accessibility",
    "section 508",
    "wcag",
    "livestream",
    "streaming",
    "webcasting",
    "video production"
  ],
  "grants_keywords": [
    "small business innovation research",
    "SBIR",
    "STTR",
    "broad agency announcement",
    "artificial intelligence",
    "software",
    "cybersecurity",
    "open source security",
    "data analytics"
  ],
  "grants_statuses": "forecasted|posted",
  "grants_eligibilities": "06|21|25|99",
  "grants_exclude": [
    "national institutes of health",
    "clinical",
    "biomedical",
    "neuro",
    "cancer",
    "genetic",
    "fellowship",
    "scholarship",
    "nursing",
    "disease",
    "dental",
    "mental health",
    "bioengineering",
    "oceanographic",
    "wildlife",
    "habitat",
    "r01",
    "r21",
    "r03"
  ],
  "usajobs_keywords": [
    "artificial intelligence",
    "machine learning",
    "data scientist",
    "data engineer",
    "software engineer",
    "full stack",
    "cybersecurity",
    "cloud engineer",
    "devops",
    "computer scientist",
    "information technology specialist",
    "AI"
  ],
  "adzuna_country": "us",
  "adzuna_keywords": [
    "artificial intelligence",
    "machine learning engineer",
    "ai engineer",
    "llm engineer",
```

```
    "generative ai",
    "data scientist",
    "data engineer",
    "software engineer",
    "full stack developer",
    "developer relations",
    "solutions engineer",
    "forward deployed engineer",
    "blockchain",
    "smart contract",
    "rust developer",
    "cybersecurity"
  ],
  "job_filter": [
    "engineer",
    "developer",
    " ai",
    "a.i",
    "machine learning",
    " ml",
    "llm",
    "data scien",
    "data engineer",
    "software",
    "devrel",
    "developer relations",
    "developer advocate",
    "solutions engineer",
    "forward deployed",
    "blockchain",
    "web3",
    "smart contract",
    "solidity",
    "rust",
    "platform",
    "backend",
    "full stack",
    "fullstack",
    "cloud",
    "security",
    "scientist",
    "applied",
    "founding"
  ],
  "themuse_categories": [
    "Software Engineering",
    "Data Science",
    "Data and Analytics",
    "Engineering"
  ],
  "greenhouse_companies": [
    "stripe",
    "databricks",
    "coinbase",
    "brex",
    "ramp",
    "robinhood",
    "airbnb",
    "gitlab",
    "cloudflare",
    "figma",
    "discord",
    "anthropic",
    "clickhouse",
    "scaleai",
    "cresta",
    "grafanalabs",
    "vercel",
    "fireblocks",
    "temporaltechnologies",
    "fal",
    "blockchain",
    "gemini",
    "runpod",
    "labelbox",
    "planetscale",
```

```
    "consensys",
    "warp",
    "imbue",
    "paradigm",
    "parloa",
    "observeai",
    "polyai",
    "instabase",
    "assemblyai"
  ],
  "lever_companies": [
    "leverdemo",
    "plaid",
    "kraken",
    "attentive",
    "sourcegraph",
    "mistral"
  ],
  "federalregister_terms": [
    "sources sought",
    "request for information",
    "industry day",
    "presolicitation"
  ],
  "job_exclude": [
    "sales",
    "account executive",
    "account manager",
    "recruiter",
    "marketing",
    "customer success",
    "business development",
    "support specialist",
    "office manager",
    "administrative assistant"
  ],
  "profiles": {
    "matthew": [
      "ai",
      "agent",
      "mcp",
      "model context",
      "llm",
      "machine learning",
      "blockchain",
      "web3",
      "solidity",
      "rust",
      "developer relations",
      "devrel",
      "forward deployed",
      "solutions engineer",
      "full stack",
      "founding",
      "crypto",
      "autonomy",
      "applied"
    ],
    "nicholas": [
      "program manager",
      "product manager",
      "technical program",
      "engineering manager",
      "director",
      "cloud",
      "platform",
      "portfolio",
      "devops",
      "agile",
      "scrum",
      "azure",
      "telecom",
      "transformation",
      "head",
      "principal",
      "delivery"
```

```
    ]
  },
  "set_asides": [
    "8(a)",
    "HUBZone",
    "Total Small Business",
    "SDVOSB",
    "Service-Disabled",
    "WOSB"
  ],
  "min_score": 1.0,
  "scoring": {
    "keyword_hit": 1.0,
    "naics_match": 2.0,
    "setaside_match": 1.5,
    "recency": 0.5
  },
  "remotive_search": [
    "AI engineer",
    "forward deployed engineer",
    "machine learning engineer",
    "developer relations",
    "blockchain engineer",
    "full stack engineer",
    "solutions engineer"
  ],
  "findwork_search": [
    "AI engineer",
    "forward deployed engineer",
    "machine learning engineer",
    "developer relations",
    "blockchain engineer",
    "full stack engineer",
    "solutions engineer"
  ],
  "jooble_keywords": [
    "AI engineer",
    "forward deployed engineer",
    "machine learning engineer",
    "developer relations",
    "blockchain engineer",
    "full stack engineer",
    "solutions engineer"
  ],
  "jooble_location": "",
  "sheet_profiles": {
    "nicholas": {
      "job_filter": [
        "program manager",
        "product manager",
        "technical program",
        "engineering manager",
        "director",
        "cloud",
        "platform",
        "portfolio",
        "devops",
        "scrum",
        "agile",
        "delivery",
        "head of",
        "principal"
      ],
      "job_exclude": [
        "sales",
        "account executive",
        "account manager",
        "recruiter",
        "marketing",
        "customer success",
        "business development",
        "support specialist",
        "office manager",
        "administrative assistant",
        "therapist",
        "nurse",
```

```
        "clinical",
        "physician",
        "psycholog",
        "dental",
        "pharmac",
        "hbpc",
        "counsel"
      ],
      "adzuna_keywords": [
        "technical program manager",
        "senior product manager",
        "director of engineering",
        "engineering program manager",
        "cloud platform manager",
        "senior technical program manager",
        "director cloud"
      ],
      "usajobs_keywords": [
        "information technology program manager",
        "IT project manager",
        "program manager",
        "supervisory IT specialist"
      ],
      "remotive_search": [
        "technical program manager",
        "program manager",
        "engineering manager",
        "director",
        "product manager"
      ],
      "findwork_search": [
        "technical program manager",
        "program manager",
        "engineering manager",
        "director",
        "product manager"
      ],
      "jooble_keywords": [
        "technical program manager",
        "program manager",
        "IT program manager",
        "director of engineering",
        "senior product manager"
      ]
    }
  },
  "contract_profiles": {
    "thrive": {
      "naics": [
        "541810",
        "541613",
        "519130",
        "541820",
        "541430",
        "541890",
        "561920",
        "512110",
        "541511",
        "541512",
        "541519",
        "518210",
        "541715"
      ],
      "keywords": [
        "search engine optimization",
        "seo",
        "digital marketing",
        "marketing",
        "public affairs",
        "outreach",
        "communications",
        "advertising",
        "public relations",
        "campaign",
        "brand",
        "content",
```

```
        "social media",
        "creative services",
        "public awareness",
        "generative engine optimization",
        "answer engine",
        "agentic",
        "ai agent",
        "mcp",
        "model context protocol",
        "llm",
        "rag",
        "artificial intelligence",
        "machine learning",
        "autonomy",
        "data pipeline",
        "indexer",
        "dashboard",
        "analytics",
        "automation",
        "edge",
        "serverless",
        "cybersecurity",
        "zero trust",
        "blockchain",
        "website",
        "web development",
        "web application",
        "accessibility",
        "section 508",
        "wcag",
        "livestream",
        "streaming",
        "webcasting",
        "video production"
      ],
      "grants_keywords": [
        "outreach",
        "public awareness campaign",
        "communications",
        "digital services",
        "marketing"
      ],
      "grants_exclude": [
        "national institutes of health",
        "clinical",
        "biomedical",
        "neuro",
        "cancer",
        "genetic",
        "fellowship",
        "scholarship",
        "nursing",
        "disease",
        "dental",
        "mental health",
        "bioengineering",
        "oceanographic",
        "wildlife",
        "habitat",
        "r01",
        "r21",
        "r03"
      ],
      "federalregister_terms": [
        "public affairs",
        "marketing",
        "outreach",
        "creative services",
        "sources sought",
        "request for information",
        "industry day",
        "presolicitation"
      ],
      "set_asides": [
        "8(a)",
        "HUBZone",
```

```
        "Total Small Business",
        "SDVOSB",
        "Service-Disabled",
        "WOSB",
        "EDWOSB"
      ],
      "incumbent_naics": [
        "541810",
        "541613",
        "541820",
        "541430",
        "541890",
        "541850",
        "561920",
        "512110"
      ]
    }
  },
  "usaspending_lookback_days": 730
}
```

content.js

```
// FedSource ATS Autofill — content script (P3: standard fields + screening layer).
// Greenhouse + Lever + Ashby. Fills standard fields, factual screening answers (from your
// VALIDATED profile — never fabricated), and free-text answers (LLM draft grounded in your
// résumé, or a reused saved answer). Runs in your real browser. NEVER submits — you review.

// ---- standard contact/identity fields ----
const MAP = {
  first_name:     { auto: ["given-name"],  label: [/first name/i],              nm: [/first[_-]?name/i] },
  last_name:      { auto: ["family-name"], label: [/last name|surname/i],       nm: [/last[_-]?name|surname/i] },
  preferred_name: {                        label: [/preferred (first )?name/i], nm: [/preferred[_-]?name/i] },
  email:          { auto: ["email"],       label: [/e-?mail/i],                 nm: [/email/i], type: ["email"] },
  phone:          { auto: ["tel"],         label: [/phone|mobile|cell/i],       nm: [/phone/i], type: ["tel"] },
  linkedin:       {                        label: [/linkedin/i],                nm: [/linkedin/i] },
  website:        {                        label: [/github|portfolio|personal (web)?site|website|^\s*url/i], nm:
[/github|portfolio|website|other.?url/i] },
  full_name:      {                        label: [/full name|legal name|^\s*name\s*\*?\s*$/i], nm: [/^name$/i,
/full[_-]?name|candidate[_-]?name/i, /_systemfield_name/i] },
};

// ---- factual screening: question pattern -> derive answer from the validated profile ----
const FACTS = [
  { id: "work_auth", q: [/authoriz(ed|ation) to work|legally authorized|eligible to work|work authorization/i], val: (p)
=> (p.work_authorized ? "Yes" : "No") },
  { id: "sponsor",   q: [/require.*sponsorship|need.*sponsorship|visa sponsorship|sponsorship now or in the future/i],
val: (p) => (p.requires_sponsorship ? "Yes" : "No") },
  { id: "worked",    q: [/worked (for|at).*(before|previously)|current or former (employee|contractor)|previously (been
)?employed/i], val: (p) => (p.worked_here_before ? "Yes" : "No") },
  { id: "relocate",  q: [/willing to relocate|open to relocat/i], val: (p) => (p.willing_to_relocate ? "Yes" : "No") },
  { id: "yoe",       q: [/years of (relevant |professional )?experience|how many years/i], val: (p) => p.years_experience
},
  { id: "salary",    q: [/salary expectation|compensation expectation|desired (salary|compensation)|expected
(salary|pay)/i], val: (p) => p.desired_salary },
  { id: "gender",    q: [/gender/i],           val: (p) => p.eeo_gender || "Decline to self-identify" },
  { id: "race",      q: [/race|ethnicit/i],    val: (p) => p.eeo_race || "Decline to self-identify" },
  { id: "veteran",   q: [/veteran/i],          val: (p) => p.eeo_veteran || "I don't wish to answer" },
  { id: "disability",q: [/disabilit/i],        val: (p) => p.eeo_disability || "I do not want to answer" },
];
const DECLINE_RE = /decline|prefer not|don.?t wish|not to (answer|disclose|identify)|do not (want|wish)|wish to answer/i;

const ADAPTERS = [
  { name: "Greenhouse", host: /(?:job-)?boards\.greenhouse\.io/, formSel: "input[type=email], input[name*='email' i],
input[id*='email' i]" },
  { name: "Lever",      host: /jobs\.lever\.co/,                 formSel: "input[name=email], input[name=name]" },
  { name: "Ashby",      host: /jobs\.ashbyhq\.com/,              formSel: "input[type=email], input[aria-label*='email'
i], input[name*='email' i]" },
  { name: "Databricks", host: /(?:www\.)?databricks\.com/,       formSel: "input[type=email], input[name*='email' i],
input[id*='email' i]" },
  { name: "Stripe",     host: /stripe\.com/,                     formSel: "input[type=email], input[name*='email' i],
input[id*='email' i]" },
];
```

```
const currentAdapter = () => ADAPTERS.find((a) => a.host.test(location.hostname));

function setNativeValue(el, value) {
  const proto = el.tagName === "TEXTAREA" ? HTMLTextAreaElement.prototype : HTMLInputElement.prototype;
  Object.getOwnPropertyDescriptor(proto, "value").set.call(el, value);
  el.dispatchEvent(new Event("input", { bubbles: true }));
  el.dispatchEvent(new Event("change", { bubbles: true }));
}

function labelText(el) {
  let t = (el.getAttribute("aria-label") || "") + " " + (el.placeholder || "");
  if (el.id) { const l = document.querySelector(`label[for="${CSS.escape(el.id)}"]`); if (l) t += " " + l.textContent; }
  const lbby = el.getAttribute("aria-labelledby");
  if (lbby) { const e = document.getElementById(lbby); if (e) t += " " + e.textContent; }
  const wrap = el.closest("label"); if (wrap) t += " " + wrap.textContent;
  return t.replace(/\s+/g, " ").trim();
}

// fallback question text for selects/textareas without a direct label
function nearbyQuestion(el) {
  let node = el;
  for (let i = 0; i < 4 && node; i++) {
    node = node.parentElement; if (!node) break;
    const lab = node.querySelector("label, legend");
    if (lab && lab.textContent.trim()) return lab.textContent.replace(/\s+/g, " ").trim();
  }
  return "";
}
// ATS question cards store the question text away from the field: Lever in `.application-label`
// (a div, not a <label>), Ashby in a <fieldset> heading. Read it from the enclosing card so radio/
// select questions resolve to the QUESTION, not an option label ("Female"/"Yes").
function cardQuestion(el) {
  const card = el.closest("li.application-question, .application-question, fieldset, [class*='fieldEntry' i],
[class*='field-entry' i]");
  if (!card) return "";
  const q = card.querySelector("legend, .application-label, [class*='heading' i], [class*='question-label' i]");
  return q && q.textContent.trim() ? q.textContent.replace(/\s+/g, " ").trim() : "";
}
// visible text of a single radio/checkbox OPTION (Lever wraps in <label>; Ashby uses a <div class*=option>)
function radioOptionText(r) {
  const lab = r.closest("label");
  if (lab && lab.textContent.trim()) return lab.textContent.replace(/\s+/g, " ").trim();
  const opt = r.closest("[class*='option' i]");
  if (opt && opt.textContent.trim()) return opt.textContent.replace(/\s+/g, " ").trim();
  return r.parentElement ? (r.parentElement.textContent || "").replace(/\s+/g, " ").trim() : "";
}
const qText = (el) => labelText(el) || cardQuestion(el) || nearbyQuestion(el);
const shorten = (s) => (s || "").replace(/\s+/g, " ").trim().slice(0, 44);
const normQ = (q) => q.toLowerCase().replace(/[^a-z0-9]+/g, " ").trim().slice(0, 120);

function matchKey(el) {
  const auto = (el.getAttribute("autocomplete") || "").toLowerCase();
  const nm = `${el.getAttribute("name") || ""} ${el.id || ""}`;
  const lbl = labelText(el);
  const type = (el.getAttribute("type") || "text").toLowerCase();
  for (const [key, r] of Object.entries(MAP)) if (r.auto && r.auto.includes(auto)) return key;
  for (const [key, r] of Object.entries(MAP)) {
    if (r.label && r.label.some((re) => re.test(lbl))) return key;
    if (r.nm && r.nm.some((re) => re.test(nm))) return key;
    if (r.type && r.type.includes(type)) return key;
  }
  return null;
}
const valueFor = (key, p) => (key === "full_name" ? `${p.first_name || ""} ${p.last_name || ""}`.trim() : p[key]);

// ---- standard inputs ----
function fillStandard(profile) {
  let n = 0;
  document.querySelectorAll(

"input:not([type=hidden]):not([type=file]):not([type=checkbox]):not([type=radio]):not([type=submit]):not([type=button])"
  ).forEach((el) => {
    if (el.value && el.value.trim()) return;
    const key = matchKey(el);
    const val = key && valueFor(key, profile);
    if (val) { setNativeValue(el, val); el.style.outline = "2px solid #4ade80"; n++; }
```

```
  });
  return n;
}

// ---- factual screening (selects + yes/no radios) from validated profile; flag when unsure ----
function matchOption(sel, want, factId) {
  const opts = [...sel.options];
  if (/^(gender|race|veteran|disability)$/.test(factId) && DECLINE_RE.test(String(want))) {
    return opts.find((o) => DECLINE_RE.test(o.text));
  }
  const w = String(want).toLowerCase().trim();
  return opts.find((o) => o.text.toLowerCase().trim() === w)
      || (["yes", "no"].includes(w) && opts.find((o) => new RegExp(`^\\s*${w}\\b`, "i").test(o.text)))
      || opts.find((o) => o.value && o.text.toLowerCase().includes(w))
      || null;
}

function fillFacts(profile) {
  let n = 0; const flagged = [];
  document.querySelectorAll("select").forEach((sel) => {
    const cur = sel.options[sel.selectedIndex];
    if (cur && cur.value && !/^(select|choose|--|please|^$)/i.test(cur.text.trim())) return; // already answered
    const q = qText(sel); if (!q) return;
    const fact = FACTS.find((f) => f.q.some((re) => re.test(q)));
    if (!fact) { if (sel.required) flagged.push(shorten(q)); return; }
    const want = fact.val(profile);
    if (want == null || want === "") { if (sel.required) flagged.push(shorten(q)); return; }
    const opt = matchOption(sel, want, fact.id);
    if (opt) { sel.value = opt.value; sel.dispatchEvent(new Event("change", { bubbles: true })); sel.style.outline = "2px
solid #4ade80"; n++; }
    else flagged.push(shorten(q));
  });
  // radio groups (Lever EEO, Ashby Yes/No screening). Question lives in the card (cardQuestion),
  // option text in each radio's wrapper (radioOptionText). Handles "decline" answers like the select path.
  const groups = {};
  document.querySelectorAll("input[type=radio]").forEach((r) => { if (r.name) (groups[r.name] = groups[r.name] ||
[]).push(r); });
  Object.values(groups).forEach((g) => {
    if (g.some((r) => r.checked)) return;
    const q = cardQuestion(g[0]) || nearbyQuestion(g[0]);
    const fact = FACTS.find((f) => f.q.some((re) => re.test(q)));
    if (!fact) { if (/[**]/.test(q)) flagged.push(shorten(q)); return; }
    const want = fact.val(profile);
    if (want == null || want === "") { flagged.push(shorten(q)); return; }
    const decline = /^(gender|race|veteran|disability)$/.test(fact.id) && DECLINE_RE.test(String(want));
    const w = String(want).trim();
    const pick = decline
      ? g.find((r) => DECLINE_RE.test(radioOptionText(r)))
      : g.find((r) => new RegExp(`^\\s*${escRe(w)}\\b`, "i").test(radioOptionText(r)) || radioOptionText(r).toLowerCase()
=== w.toLowerCase());
    if (pick) { pick.click(); const lab = pick.closest("label") || pick.closest("[class*='option' i]"); if (lab)
lab.style.outline = "2px solid #4ade80"; n++; }
    else flagged.push(shorten(q));
  });
  return { n, flagged };
}

// ---- factual screening for react-select / ARIA comboboxes (Greenhouse renders NO native <select>) ----
// Recipe verified live against a Greenhouse application form: react-select IGNORES synthetic keyboard
// events, so we (1) OPEN the menu with a full pointer+mouse gesture on the control, (2) FILTER by
// setting the search input's value + an 'input' event, (3) SELECT with mousedown+mouseup+click on the
// option (a bare .click() is not enough). Readback: the value lands in `.select__single-value` inside
// the control — NOT under the input's own *-input-container.
const sleep = (ms) => new Promise((r) => setTimeout(r, ms));
const escRe = (s) => String(s).replace(/[.*+?^${}()|[\]\\]/g, "\\$&");
const comboValue = (control) => { const sv = control.querySelector("[class*='single-value']"); return sv ?
sv.textContent.trim() : ""; };

async function pickComboOption(input, control, matchOpt, filterText) {
  for (const [Ctor, type] of [[PointerEvent, "pointerdown"], [MouseEvent, "mousedown"], [PointerEvent, "pointerup"],
[MouseEvent, "mouseup"], [MouseEvent, "click"]]) {
    try { control.dispatchEvent(new Ctor(type, { bubbles: true, cancelable: true, button: 0 })); } catch { /*
PointerEvent ctor unsupported — mouse events suffice */ }
  }
  input.focus();
  await sleep(160);
```

```
  if (filterText != null) {
    Object.getOwnPropertyDescriptor(HTMLInputElement.prototype, "value").set.call(input, filterText);
    input.dispatchEvent(new Event("input", { bubbles: true }));
    await sleep(160);
  }
  let opts = [...document.querySelectorAll(".select__option")];
  if (!opts.length) opts = [...document.querySelectorAll("[role=option], [class*='option']")];
  const opt = opts.find(matchOpt);
  if (!opt) { input.blur(); return; }             // close the menu, leave unanswered -> flagged by caller
  for (const type of ["mousedown", "mouseup", "click"]) opt.dispatchEvent(new MouseEvent(type, { bubbles: true,
cancelable: true, button: 0 }));
  await sleep(160);
}

async function fillComboboxes(profile) {
  let n = 0; const flagged = [];
  for (const input of document.querySelectorAll("input[role=combobox]")) {
    const control = input.closest("[class*='control']"); if (!control) continue;
    if (comboValue(control)) continue;                                    // already answered
    const q = qText(input); if (!q || /^search$/i.test(q)) continue;     // skip react-select's transient search box
    const required = input.required || input.getAttribute("aria-required") === "true" || /\*/.test(q);

    const fact = FACTS.find((f) => f.q.some((re) => re.test(q)));
    if (!fact) { if (required) flagged.push(shorten(q)); continue; }     // e.g. Country / City — needs you (P-next)
    const want = fact.val(profile);
    if (want == null || want === "") { flagged.push(shorten(q)); continue; }

    const decline = /^(gender|race|veteran|disability)$/.test(fact.id) && DECLINE_RE.test(String(want));
    const w = String(want).trim();
    const matchOpt = decline
      ? (o) => DECLINE_RE.test((o.textContent || "").trim())
      : (o) => { const t = (o.textContent || "").trim(); return new RegExp("^\\s*" + escRe(w) + "\\b", "i").test(t) ||
t.toLowerCase() === w.toLowerCase(); };
    await pickComboOption(input, control, matchOpt, decline ? null : w);

    const after = comboValue(control);
    const good = after && (decline ? DECLINE_RE.test(after) : new RegExp("^" + escRe(w), "i").test(after));
    if (good) { control.style.outline = "2px solid #4ade80"; n++; } else flagged.push(shorten(q));
  }
  return { n, flagged };
}

// ---- free-text screening: saved-answer cache -> LLM draft (grounded) -> flag ----
async function saveAnswer(key, text) {
  const { answers } = await chrome.storage.local.get("answers");
  const a = answers || {}; a[key] = text; await chrome.storage.local.set({ answers: a });
}
function jobContext() {
  const main = document.querySelector("main, [role=main], .posting, .job, #content, #main") || document.body;
  return `${document.title}\n${(main.innerText || "").slice(0, 2500)}`.slice(0, 3000);
}
function requestDraft(question, jd) {
  return new Promise((res) => {
    try { chrome.runtime.sendMessage({ type: "draft", question, jd }, (r) => res(r && r.answer ? r.answer : null)); }
    catch { res(null); }
  });
}
async function fillFreeText(profile) {
  const { answers } = await chrome.storage.local.get("answers");
  const cache = answers || {}; const jd = jobContext();
  let cached = 0, drafted = 0; const flagged = [];
  // visible = rendered & non-zero-size; excludes hidden fields like reCAPTCHA's display:none g-recaptcha-response
  // textarea, which would otherwise burn an LLM draft and corrupt the token. offsetParent is null for display:none.
  const visible = (t) => t.offsetParent !== null && t.getBoundingClientRect().width > 0 &&
t.getBoundingClientRect().height > 0;
  for (const ta of [...document.querySelectorAll("textarea")].filter((t) => !t.value.trim() && visible(t))) {
    const q = qText(ta); if (!q) continue;
    if (/r[eé]sum[eé]|\bcv\b|cover letter/i.test(q) || /resume|cover_letter|g-
recaptcha|captcha/i.test(ta.getAttribute("name") || "")) continue; // résumé/cover paste-fields + captcha tokens — never
draft
    const key = normQ(q);
    ta.addEventListener("blur", () => { if (ta.value.trim()) saveAnswer(key, ta.value.trim()); });
    if (cache[key]) { setNativeValue(ta, cache[key]); ta.style.outline = "2px solid #4ade80"; cached++; continue; }
    const draft = await requestDraft(q, jd);
    if (draft && !/^\[NEEDS YOU/i.test(draft)) { setNativeValue(ta, draft); ta.style.outline = "2px solid #fbbf24";
drafted++; saveAnswer(key, draft); }
```

```
    else flagged.push(shorten(q));
  }
  return { cached, drafted, flagged };
}

function toast(msg, ok = true) {
  let t = document.getElementById("fedsource-toast");
  if (!t) { t = document.createElement("div"); t.id = "fedsource-toast"; document.body.appendChild(t); }
  t.textContent = msg;
  t.style.cssText = `position:fixed;bottom:84px;right:20px;z-index:2147483647;background:${ok ? "#0b3d2e" :
"#5a1a1a"};color:#fff;padding:12px 16px;border-radius:10px;font:13px/1.45 -apple-system,Segoe UI,sans-serif;max-
width:360px;box-shadow:0 6px 20px rgba(0,0,0,.35)`;
  clearTimeout(t._h); t._h = setTimeout(() => t.remove(), 11000);
}

async function fillForm() {
  const { profile } = await chrome.storage.local.get("profile");
  if (!profile) { toast("No profile saved. Click the FedSource icon → Edit profile.", false); return; }
  const ats = currentAdapter();
  const std = fillStandard(profile);
  const facts = fillFacts(profile);
  const combos = await fillComboboxes(profile);
  const ft = await fillFreeText(profile);

  const parts = [`${std} fields`, `${facts.n + combos.n} screening`];
  if (ft.cached) parts.push(`${ft.cached} reused`);
  if (ft.drafted) parts.push(`${ft.drafted} AI-drafted (review the amber ones)`);
  let msg = `${ats ? ats.name + ": " : ""}${parts.join(", ")}. `;
  if (document.querySelector("input[type=file]")) msg += "Attach your résumé. ";
  const flags = [...facts.flagged, ...combos.flagged, ...ft.flagged];
  if (flags.length) msg += `⚠ ${flags.length} need you: ${flags.slice(0, 3).join("; ").slice(0, 100)}. `;
  msg += "Review everything, then submit. Nothing was submitted.";
  toast(msg, true);
}

function injectButton() {
  if (document.getElementById("fedsource-fill-btn")) return;
  const ats = currentAdapter();
  if (!ats || !document.querySelector(ats.formSel)) return;
  const b = document.createElement("button");
  b.id = "fedsource-fill-btn"; b.type = "button"; b.textContent = `⚡ FedSource Autofill (${ats.name})`;
  b.style.cssText = "position:fixed;bottom:20px;right:20px;z-
index:2147483647;background:#1f4e79;color:#fff;border:0;padding:12px 18px;border-radius:24px;font:600 14px -apple-
system,Segoe UI,sans-serif;cursor:pointer;box-shadow:0 6px 20px rgba(0,0,0,.35)";
  b.addEventListener("click", (e) => { e.preventDefault(); fillForm(); });
  document.body.appendChild(b);
}

new MutationObserver(() => injectButton()).observe(document.documentElement, { childList: true, subtree: true });
setTimeout(injectButton, 1200);
chrome.runtime.onMessage.addListener((m) => { if (m === "fill") fillForm(); });
```

`manifest.json`

```
{
  "manifest_version": 3,
  "name": "FedSource ATS Autofill",
  "version": "0.3.1",
  "description": "Fills Greenhouse, Lever & Ashby applications — standard fields, factual screening from your validated
profile, and grounded free-text drafts. You review and submit; it never auto-submits or fabricates facts.",
  "permissions": ["storage", "activeTab", "tabs"],
  "host_permissions": [
    "https://boards.greenhouse.io/*",
    "https://job-boards.greenhouse.io/*",
    "https://jobs.lever.co/*",
    "https://jobs.ashbyhq.com/*",
    "https://*.databricks.com/*",
    "https://*.stripe.com/*",
    "http://localhost:8765/*"
  ],
  "background": { "service_worker": "background.js" },
  "content_scripts": [
    {
```

```
      "matches": [
        "https://boards.greenhouse.io/*",
        "https://job-boards.greenhouse.io/*",
        "https://jobs.lever.co/*",
        "https://jobs.ashbyhq.com/*",
        "https://*.databricks.com/*",
        "https://*.stripe.com/*"
      ],
      "js": ["content.js"],
      "all_frames": true,
      "run_at": "document_idle"
    }
  ],
  "action": {
    "default_popup": "popup.html",
    "default_title": "FedSource Autofill"
  },
  "options_page": "options.html"
}
```

Connector quick-reference

FedSource Connectors — for your tracker

18 job/contract sources, ~40 lines each, **pure Python stdlib** (no pip installs).

Drop `source.py` next to your tracker and call any connector.

Call one

```
import source
cfg = {
    "job_filter":  ["program manager","director","cloud","product manager"],  # title must contain one
    "job_exclude": ["sales","recruiter"],                                      # and none of these
    "adzuna_keywords":  ["technical program manager","director of engineering"],
    "usajobs_keywords": ["IT program manager"],
    "since_days": 14,
}
jobs  = source.fetch_adzuna(cfg)     # -> list of dicts (shape below)
jobs += source.fetch_remoteok(cfg)   # mix and match
```

Pull everything:

```
for name, fn in source.CONNECTORS.items():
    rows = fn(cfg)
```

Record shape — drops straight into your DB

`uid · source · title · agency(company) · otype · naics · posted · due · set_aside · url · summary`

Keyless (work immediately)

greenhouse · lever · remoteok · remotive · jobicy · himalayas · themuse · hn · yc · grants · sbir · federalregister · usaspending

Need a free key (put in a `.env` next to source.py, KEY=value per line)

- **adzuna** → developer.adzuna.com → `ADZUNA_APP_ID` + `ADZUNA_APP_KEY`

- **usajobs** → developer.usajobs.gov → `USAJOBS_API_KEY` + `USAJOBS_EMAIL`
- **findwork** → findwork.dev/developers → `FINDWORK_API_KEY`
- **jooble** → jooble.org/api/about → `JOOBLE_API_KEY`
- **sam** (federal contracts) → sam.gov → `SAM_API_KEY`

Highest precision: watch specific employers

Set `greenhouse_companies` / `lever_companies` in cfg to a list of company tokens —

pulls jobs straight from each company's ATS board (no key needed).