YAHR

Elvis Perlika

elvis.perlika@studio.unibo.it

Intelligent System Engineering University of Bologna 2026

Table of Contents

1 YAHR

Yet Another HR

1.1 Abstract

YAHR (Yet Another HR) is a command-line tool that automates the job search. It takes a resume in PDF form, converts it into a structured text profile, searches for matching open positions, scores each one against the candidate's background, and suggests concrete edits to the resume to better fit a chosen position. Everything happens in the terminal.

YAHR runs on the A2A (Agent-to-Agent) protocol, an open standard for communication between agents. Converting the resume is a separate command-line step; the rest of the work is handled by a network of four agents. An Orchestrator agent reads each request, picks the specialized agent that fits, and forwards the work. The domain logic is split across the remaining three: the Job Searcher finds openings, the Ranker scores them against the profile, and the CV Assistant compares the resume against a specific job and reports what to improve. This report explains why YAHR exists, how it is built, and how the A2A protocol keeps its agents loosely coupled.

1.2 Domain

YAHR works in the domain of the job search: the process a person goes through to find work that fits them and to present themselves well enough to be called for an interview. Done by hand, this is slow, repetitive work. A candidate has to find openings across several sites, read each posting, judge how well it matches their background, and

then decide whether and how to adjust their resume for the roles worth applying to. One resume rarely fits every posting equally well, so the tailoring has to be redone for each role.

YAHR treats this as a single-user problem. The only actor is the candidate, who runs the tool locally from the terminal and supplies one resume. There is no recruiter, no employer, and no shared account: every run is one person looking for work on their own machine. Keeping the domain this small lets the system treat the candidate's resume as the single source of truth about their background.

1.3 Core Concepts

- Resume (or CV): the candidate's background, and the input to everything else. YAHR holds it as structured Markdown (a `# Name` heading, `##` sections, and bullets) produced by converting the source PDF. It is the only evidence the system may reason from; nothing reads in outside facts or invents experience that is not written down.
- Job (open position): one posting found for the candidate. Each has a stable id (used to drop duplicates across searches), a title, the hiring company, a location, the posting body, a link to apply, and, when listed, the gross annual salary. The posting body is the main text the system matches against.
- Query: what the candidate is looking for, in plain language. It can name a role and constraints such as location, salary, or contract type, and it can ask for a set number of results, as in "find 3 java jobs in Milan".
- Fit (score): how well a job matches the resume, on a scale from 0 to 100. Fit is judged on overlap: shared skills first, then relevant experience and seniority, then the role and domain, and finally any constraints the candidate stated.

- Gap: for one chosen job, a requirement the posting asks for that the resume does not clearly show. A gap is either something the candidate already has but worded poorly (reword) or something genuinely missing that they would need to acquire. The gap analysis pairs these with concrete edits to strengthen the resume for that specific job.

1.4 Design

YAHR is built on A2A (Agent-to-Agent), an open protocol that lets independent agents talk to each other over HTTP. The work is split across one orchestrator and three specialized agents, each running as its own A2A server on its own port and doing a single job. The orchestrator sits in front of all three, routing each request to the agent that fits and streaming the result back to the terminal.

The agents know nothing about each other or about the orchestrator. An agent answers A2A requests and returns a result, and that is all. This is what keeps the system loosely coupled. A new agent comes online by starting its server and serving an agent card, and the orchestrator finds it the next time it looks; nothing else has to change. The job boards sit one layer further out, behind MCP (Model Context Protocol), which keeps the job providers swappable too.

Every agent follows two design rules. The first separates protocol from logic: each agent has a thin executor that speaks A2A, the agent-to-agent protocol, and hands the real work to a protocol-free core that does the actual searching, scoring, or analysis. Because no A2A type ever reaches the core, that core is plain Python that runs and can be tested offline, without starting a server. The second rule targets determinism: the two agents that reason over the resume, the Ranker and the CV Assistant, call the LLM with temperature set to zero and a fixed seed, so the model always picks its single most likely

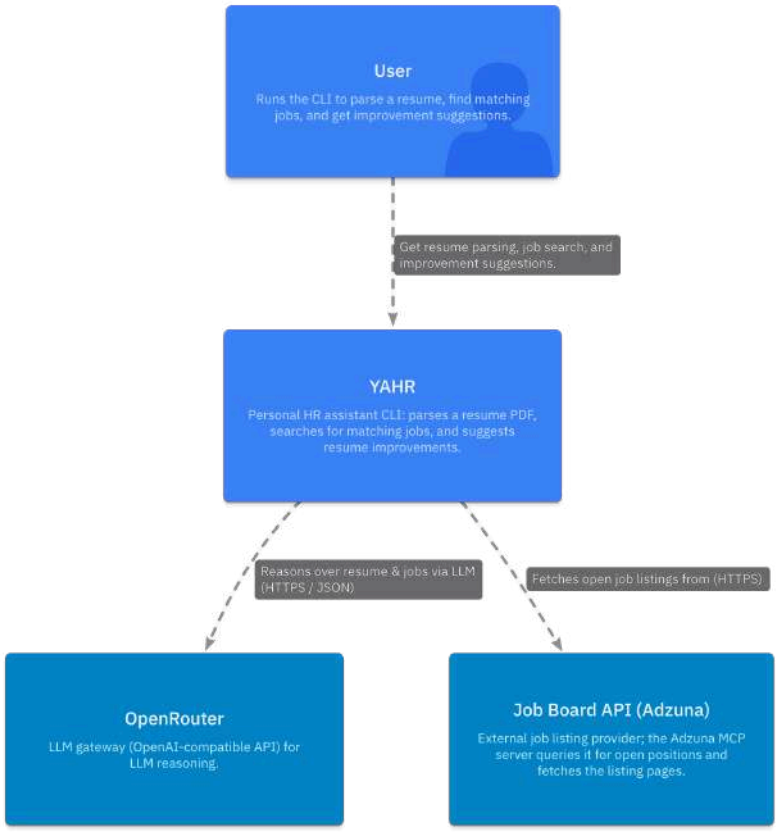
next token instead of sampling randomly, and the same resume and jobs produce the same ranking or advice every time. The orchestrator's routing call, which decides which agent handles a query, is pinned the same way, so a given query always reaches the same agent. This determinism is best effort, not a guarantee: it holds only as far as the LLM provider actually honors the seed, since even greedy decoding can drift slightly between runs on the provider's end.

1.4.1 System Context

At its outermost level YAHR is one system the candidate runs locally, and it depends on two outside services.

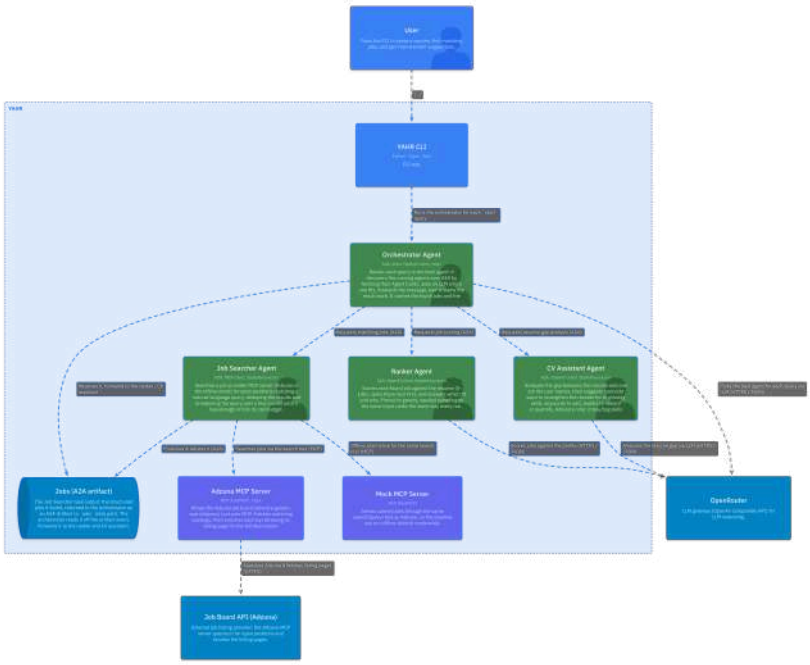
The only actor is the User. The candidate runs the CLI to parse a resume, search for matching jobs, and get suggestions for improving it, and every result comes back through that one program on their own machine.

YAHR itself reaches out to two external systems. It calls OpenRouter, an LLM gateway with an OpenAI-compatible API, whenever it has to reason over the resume or the jobs, over HTTPS as JSON. It calls the Adzuna job board API for the open listings themselves, also over HTTPS. OpenRouter covers all the model reasoning; Adzuna is where the actual postings come from. How that work is divided inside YAHR is what the rest of the Design section covers.



1.4.2 Container Diagram

Inside YAHR the work splits into a handful of containers. The candidate starts only the CLI; every other container is a server reached over the network, except the orchestrator, which runs in-process inside the CLI rather than on a port of its own yet.



The orchestrator is the hub of the system. It fans out to three A2A agents, each its own server on its own port: the Job Searcher, the Ranker, and the CV Assistant. It also calls OpenRouter to pick which agent should handle each query.

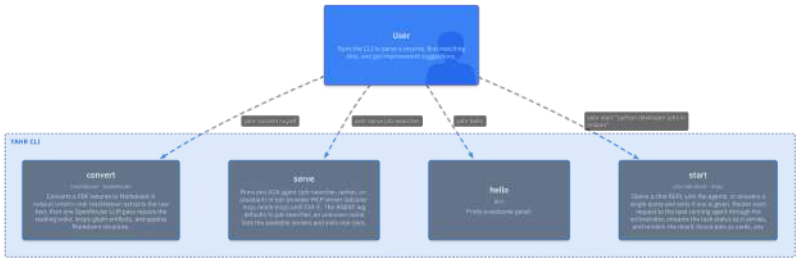
The three agents differ in how far they reach. The Ranker and the CV Assistant are self-contained: each answers the orchestrator and touches nothing else. The Job Searcher is the only agent that goes after job listings, and it does so as an MCP client rather than calling a board directly. Behind it sit two MCP job servers that expose the same search tool, the Adzuna server and the offline mock, and only the Adzuna server reaches the external job board API.

One container is data rather than a process: the jobs artifact. The Job Searcher produces it as the structured output of its A2A task, and the orchestrator receives it, passes it to the CLI to render, and keeps it so the Ranker and the CV Assistant can reuse it without a fresh search.

The two MCP servers are deliberately outside the agent roster. They are MCP servers, not A2A agents: the Job Searcher calls them as a tool it invokes to get job listings, not as a peer it routes work to, so the orchestrator never discovers them and never routes a query to them; the Job Searcher is the only thing that reaches them. This is the purpose the two protocols split between them: A2A divides the work itself across independent agents, while MCP divides the context and tools an agent draws on to do its own share of that work.

CLI Architecture

The CLI is a Typer application that renders with Rich, and it exposes four commands. `convert` turns a resume PDF into the Markdown profile: markitdown extracts the raw text, then one LLM pass repairs the reading order and applies Markdown structure. `serve` runs one server until interrupted, either an A2A agent (`job-searcher`, `ranker`, `cv-assistant`) or a job-provider MCP server (`adzuna-mcp`, `mock-mcp`). `start` is the main command: given a query it answers once and exits, and with no query it opens a chat REPL. `hello` just checks that the CLI is installed.



The orchestrator runs in-process inside `start`. For each request, `start` reads the resume file if it is there, passes the query and resume to the orchestrator, and renders what streams back: found jobs as boxed cards, any other agent reply as Markdown, and the intermediate status lines as the agent works. In the REPL those caches carry across

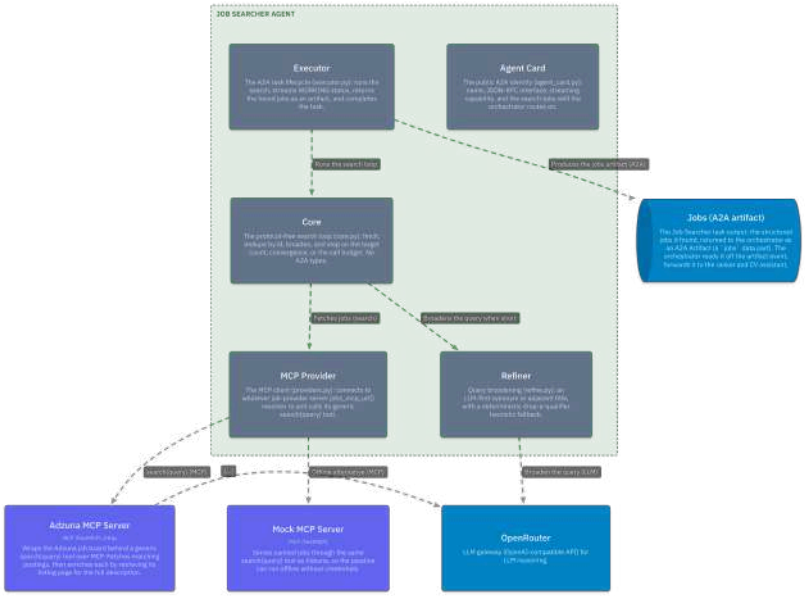
turns, so a search, a "which of these fits me?" question, and a "what should I fix for the Acme role?" question become one conversation instead of three from-scratch runs.

Agent names and addresses live in one place, the roster, and both the running servers and the orchestrator's discovery list derive from it. A name or a port cannot drift between the agent that advertises it and the orchestrator that looks for it.

Job Searcher Agent

The Job Searcher turns a natural-language query into a list of open positions. It never calls a job board directly. It connects to whatever MCP server its configured URL points at and calls that server's generic `search` tool, so Adzuna, the bundled mock, or any future source is just a different URL.

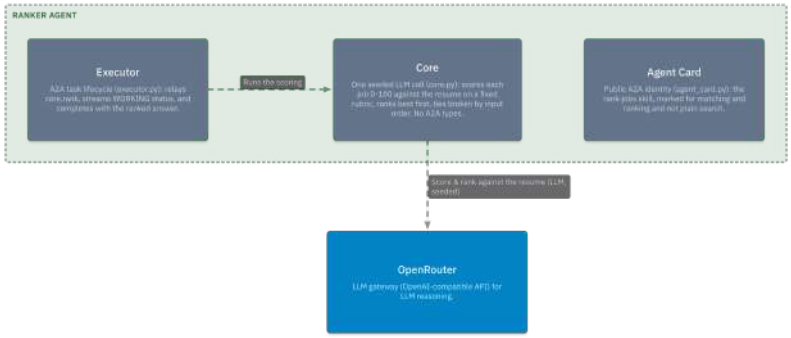
The search is a goal-seeking loop, not a single call. It fetches jobs for the query, dedupes them by id into a running set, and checks whether it has enough. If not, it broadens the query and searches again. Broadening is LLM-first: the model offers a synonym, an adjacent title, or a wider location within the same country, while keeping the constraints the candidate stated. If the model is unreachable or returns junk, a deterministic fallback drops one qualifier (such as "senior") or the trailing word, so the loop always moves forward. It stops on the first of three conditions: it has as many jobs as the query asked for (a "find 3 jobs" count, or a default of five), the query can no longer be broadened and has converged, or it hits its budget of search rounds and fetch calls. The budget exists because the real provider is a paid, rate-limited API.



The result leaves the agent in two forms. The structured jobs go out as the `jobs` artifact. The same jobs, rendered as Markdown, are the readable result the CLI shows and the Ranker later reads.

Ranker Agent

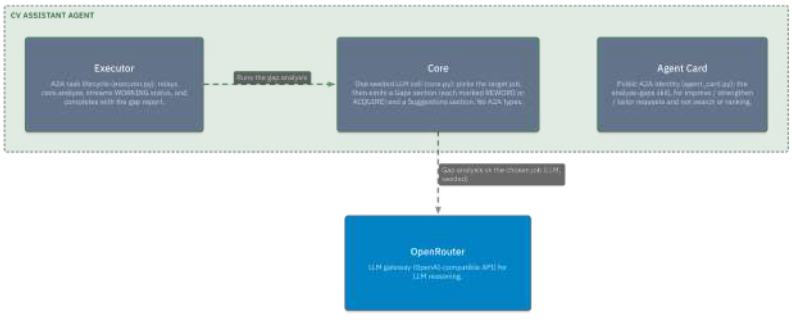
The Ranker scores the found jobs against the resume and answers the candidate's question in a single LLM call. The orchestrator bundles three things into that call: the question, the jobs, and the resume. The prompt fixes the Fit rubric so the scoring stays consistent: each job gets a score from 0 to 100, the list is ranked best first, and ties break by the order the jobs arrived in, never at random. The Ranker judges only from the text it is given, so a requirement the resume does not mention counts as not met rather than a guess.



CV Assistant Agent

The CV Assistant takes one job the candidate names and reports how to strengthen the resume for it. As with the Ranker, it works from a bundle of the request, the jobs, and the resume in a single LLM call. It first picks the single target job by matching the title or company in the request against the postings, and falls back to the first posting when the request names none. Then it writes two sections: a Gaps section listing what the posting asks for that the resume does not clearly show, and a Suggestions section of concrete edits.

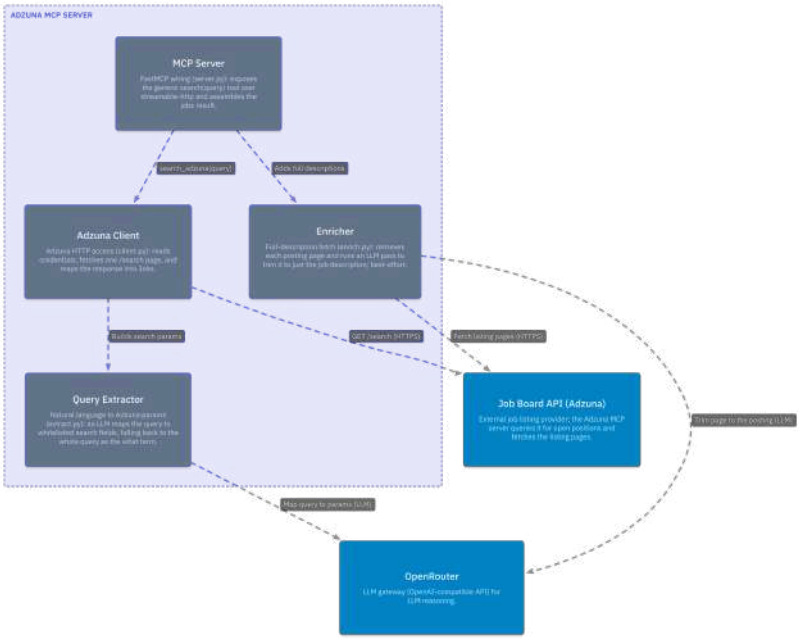
Each gap is marked REWORD, when the resume already states the fact and only needs better wording, or ACQUIRE, when it is genuinely missing. The agent stays advisory: it may flag skills the resume lacks, but it never rewrites the resume, and it credits the candidate with a strength only when that fact is actually written in the resume.



Adzuna MCP Server

The Adzuna MCP server is the one place that talks to the real job board. It exposes the same generic `search(query) -> {"jobs": [...]}` tool the mock does, so the Job Searcher cannot tell the two apart; everything specific to Adzuna is sealed inside this server. It runs over streamable HTTP and binds its port from the roster, the same source the Job Searcher reads its URL from, so the address written once is the address both ends use.

Four modules split the work, and only two of them reach off the machine. The MCP Server (`server.py`) is thin wiring: it advertises the `search` tool and assembles the result, delegating everything else. The Adzuna Client (`client.py`) is the only module that calls the job board, fetching one `/search` page over HTTPS with credentials read from the environment and mapping the response into Jobs, skipping any entry missing the id it needs to dedupe or the title it needs to show. The Query Extractor (`extract.py`) turns the candidate's natural-language query into Adzuna's whitelisted search fields with an LLM pass, falling back to the whole query as the plain `what` term when the model is unreachable or returns nothing usable. The Enricher (`enrich.py`) exists because Adzuna's search results carry only a truncated description: it fetches each posting's own page and runs an LLM pass to trim it down to just the job description.



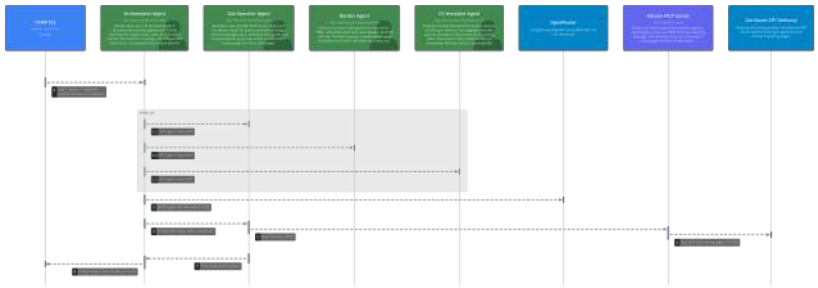
Both LLM steps lean on OpenRouter, the same gateway the agents use, and both are best effort by design. If the extractor's model call fails the search still runs on the raw query, and if a posting page cannot be fetched or trimmed the job keeps its truncated description rather than dropping out. This keeps the credentials and the paid API entirely behind the MCP boundary: the Job Searcher, and everything above it, only ever see the provider-agnostic `search` tool.

1.4.3 Sequence Diagram

The three commands the candidate runs all follow the same skeleton: the CLI hands the query and resume to the in-process orchestrator, the orchestrator discovers the running agents and asks the LLM which one fits, forwards the work, and streams the result back to render. What changes between them is which agent the router picks and how much work is fresh versus reused from the jobs cache. The flows below trace each one end to end.

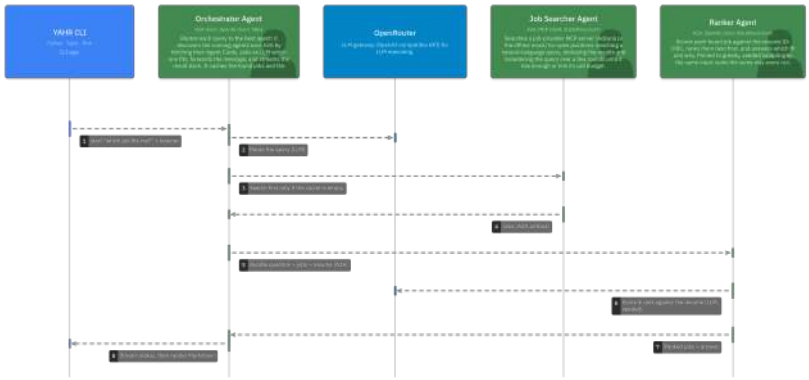
Job Searcher Flow

A plain search starts fresh. The CLI calls `start` with the query, attaching `output/resume.md` if it is there. The orchestrator first fetches every agent's card in parallel to learn who is up, then asks OpenRouter which agent fits and, for a search, gets the Job Searcher. It forwards the query over a streaming A2A call; the Job Searcher runs its `search(query)` MCP tool against the Adzuna server, which searches and fetches listing pages over HTTPS. The found jobs come back as an A2A artifact, and the orchestrator streams the status lines and then renders the job cards in the terminal. This run is what fills the jobs cache the next two flows reuse.



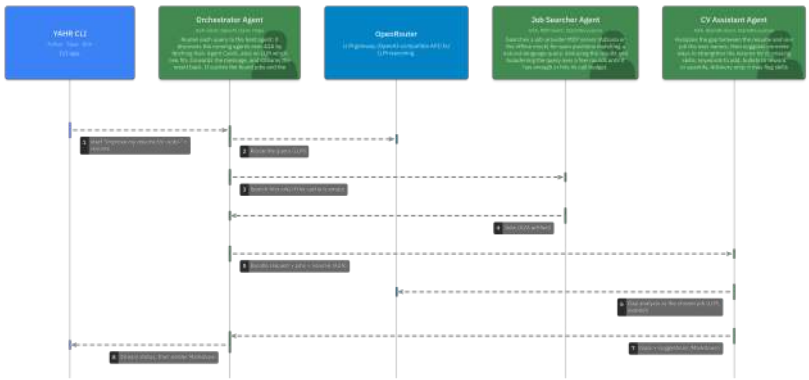
Ranking Flow

A ranking query reuses what the search already found. The CLI sends "which job fits me?" with the resume, and the orchestrator routes it to the Ranker. Rather than search again, it reads the jobs it cached on the earlier `start` run, calling the Job Searcher here only when the cache is empty. It then bundles the question, those jobs, and the resume into one A2A message to the Ranker, which scores and ranks them against the resume in a single seeded LLM call. The ranked jobs and the written answer stream back and render as Markdown.



Improve Flow

The resume-improvement query has the same shape, with the CV Assistant in the Ranker's place. The CLI sends "improve my resume for <job>" with the resume, and the orchestrator routes to the CV Assistant, again reusing the cached jobs and searching first only if the cache is empty. It bundles the request, the jobs, and the resume, and the CV Assistant makes one seeded LLM call that picks the single named job, compares it against the resume, and emits a Gaps section (each gap marked REWORD or ACQUIRE) and a Suggestions section. There is no automatic rewrite loop: the advice streams back as Markdown and the candidate makes the edits.



1.5 Tech Stack

YAHR is written in Python and needs version 3.14 or newer. The command-line front end uses two libraries. Typer turns the command functions into the `yahr` program, and Rich draws the status spinner, the job cards, and the Markdown replies in the terminal. Hatchling builds the package, so `pip install -e .` puts `yahr` on the path.

The two protocols each come from a reference SDK. A2A runs on `a2a-sdk`, with its `http-server` extra: the orchestrator talks to agents through the SDK's client, and each agent is an ASGI app that uvicorn serves over HTTP. MCP comes from the `mcp` package: the job-provider servers are built on its FastMCP helper, and the Job Searcher reaches them with the same package's client over streamable HTTP.

Every LLM call goes through OpenRouter, which exposes an OpenAI-compatible API, so YAHR uses the official `openai` client with its base URL pointed at OpenRouter (its config loader automatically strips trailing `/chat/completions` paths). `httpx` is the async HTTP client under the A2A traffic. `markitdown` does the PDF text extraction for `convert`, pinned to `0.0.2` because the 0.1 line needs onnxruntime and there is still no Python 3.14 wheel for it. The job listings come from Adzuna's public API (falling back silently to a deterministic mock when credentials are not set), which sits behind the MCP server.

The Python is formatted and linted with ruff, and everything else (these docs included) with Prettier. A Husky pre-commit hook runs the formatters (`ruff format` and `prettier`), while linting (`ruff check`) is handled dynamically by editor hooks. Commitlint holds commit messages to the conventional-commits format.

1.5.1 A2A Protocol

A2A saves YAHR from building three things itself: discovering an agent and reading its capabilities, handing work to any agent and reading the result through one uniform shape, and following that work over a streaming channel while it runs. YAHR uses a subset of the protocol through the Python `a2a-sdk`.

Every agent publishes an Agent Card. It is a small JSON document that states the agent's identity (`name`, `description`, `version`), its service `url`, the `capabilities` it supports (streaming among them), the media types it accepts, and a list of `skills`. A client reads the card before it sends anything, so the card is where discovery and capability negotiation happen. A2A allows three ways to find one: a well-known URI (`https://{host}/.well-known/agent-card.json`, following RFC 8615), a curated registry, or direct configuration. YAHR combines the last two. The roster is the direct configuration, a fixed list of host and port pairs, and the orchestrator resolves the card at each one's well-known URI. Nothing about an agent is hard-coded into the orchestrator: the description and skills it routes on come off the live card.

A2A defines three transport bindings: JSON-RPC 2.0, gRPC, and HTTP with JSON (REST). An agent may offer any of them, and its card's `url` says where to reach it. YAHR's agents speak JSON-RPC 2.0 over HTTP, and the orchestrator's client speaks the same binding.

A request is a Message: a `role` (`user` or `agent`), a `messageId`, an optional `taskId` and `contextId`, and a list of Parts. A Part holds exactly one kind of content; YAHR uses two, a text part and a data part that carries arbitrary JSON. When an agent takes on real work it answers with a Task rather than a lone message. A Task has its own `id`, a `status`, a `history` of messages, and a list of Artifacts. An Artifact is the task's output, again a bundle of Parts. A2A uses `contextId` to

group several tasks into one conversation; YAHR leaves that continuity to the orchestrator's cache, so every request is a fresh task.

A Task moves through a fixed set of states. It begins `submitted` (accepted and queued), turns to `working` while the agent runs, and finishes in one of the terminal states `completed`, `failed`, `canceled`, or `rejected`. Two more states, `input-required` and `auth-required`, let an agent pause for something it needs before going on. YAHR's tasks are short and stay on the happy path: `submitted`, then `working`, then `completed`. On the server each agent's executor drives these transitions through a `TaskUpdater`, and the orchestrator turns the protocol's state names back into the plain lowercase labels it streams to the terminal.

Because the agents' cards advertise streaming, the orchestrator uses the streaming method (`message/stream`) and receives Server-Sent Events instead of a single reply. Every event is one of four kinds: the opening Task, a plain Message, a status-update event (a state change with an optional message), or an artifact-update event (a new or extended Artifact). The orchestrator handles all four: it lifts the structured jobs out of an artifact-update event, turns status-update events into the progress lines the CLI prints, and treats the status-update that reaches `completed`, or a final Message, as the finished result.

A2A defines more of a surface than YAHR uses. There are methods to fetch a task's current state, cancel a running task, resubscribe to a task's event stream after a dropped connection, and register webhooks for push notifications. YAHR needs none of them: its tasks finish in seconds with the CLI watching the stream live, so the agents implement cancel as a no-op and never register for push notifications.

1.6 Code

The package lives under `src/yahr` and splits three ways: the command-line interface, the agents, and the MCP job providers.

```
src/yahr/
  cli/commands/      convert · serve · start · hello
  agents/
    orchestrator.py  LLM router over A2A discovery
    roster.py        agent names and addresses (one source of truth)
    models/job.py    the Job dataclass
    job_searcher/    agent_card · core · executor · providers · refine
    ranker/          agent_card · core · executor
    cv_assistant/    agent_card · core · executor
  mcp/
    adzuna/          real job board behind the search tool
    mock/            canned jobs for offline runs
  config.py          OpenRouter client and .env loading
```

Package layout

Each agent is the same small set of files: `agent_card.py` for its public identity, `core.py` for the work, `executor.py` for the A2A task lifecycle, and an `__init__.py` whose `serve()` assembles the card and executor into the running server. The Job Searcher adds two files of its own, `providers.py` (the MCP client) and `refine.py` (query broadening).

The core/executor split is easiest to read in the Ranker, the smallest agent. The core is an async generator that takes a string and yields `(text, is_final)` steps, and it imports nothing from `a2a`:

```
# agents/ranker/core.py
```

```
    async def rank(message: str) -> AsyncIterator[Step]:
        yield "Scoring jobs against your resume", False
        client, model = openrouter_client()
        reply = client.chat.completions.create(
            model=model,
            messages=[{"role": "system", "content": _SYSTEM},
                      {"role": "user", "content": message}],
            # greedy + fixed seed
            temperature=_TEMPERATURE, seed=_SEED,
        )
        yield (reply.choices[0].message.content or "(no answer
```

The executor is the only place A2A types appear. It runs the core and turns each step into a task update, completing the task on the final one:

```
    # agents/ranker/executor.py
    await updater.start_work()
    async for text, is_final in core.rank(message):
        if is_final:
            await updater.complete(
                updater.new_agent_message([new_text_part(text)
            )
        else:
            await updater.update_status(
                TaskState.TASK_STATE_WORKING,
                updater.new_agent_message(
                    [new_text_part(text + "…")]
                ),
            )
```

The Job Searcher's core is the one piece with real control flow. Its three `break` statements are the three places the search stops, and jobs

```python
# agents/job_searcher/core.py
async def _run(goal: Goal,
               fetch: Fetch = mock_jobs,
               refiner: Refiner = refine,
           ) -> AsyncIterator[Step]:
    cache: dict[str, Job] = {}
    query, calls = goal.query, 0
    for _ in range(goal.max_rounds):
        if calls >= goal.max_calls:
            # spent the call budget
            break
        for job in await fetch(query):
            # dedupe by stable id
            cache.setdefault(job.id, job)
        calls += 1
        if len(cache) >= goal.target:
            # found enough
            break
        next_query = refiner(
            query, len(cache), goal.target
        )
        if next_query == query:
            # cannot broaden, converged
            break
        query = next_query
    selected = dict(
        list(cache.items())[: goal.target]
    )
    yield _render(selected), True, list(selected.values())
```

The roster is the single source of truth for which agents exist and where, so a name or a port is written once:

```python
# agents/roster.py
```

```
4   CV_ASSISTANT_NAME = "CV Assistant Agent"
5
6   JOB_SEARCHER_ADDRESS = AgentAddress("127.0.0.1", 8002)
7   RANKER_ADDRESS = AgentAddress("127.0.0.1", 8003)
8   CV_ASSISTANT_ADDRESS = AgentAddress("127.0.0.1", 8007)
9
10  # The orchestrator probes exactly these for agent cards.
11  AGENT_URLS = [
12      JOB_SEARCHER_ADDRESS.url,
13      RANKER_ADDRESS.url,
14      CV_ASSISTANT_ADDRESS.url
15  ]
```

The agent cards import these names and the `serve()` functions bind these addresses, while the orchestrator's `discover()` walks `AGENT_URLS` fetching each card. Throughout, the code leans on plain dataclasses (`Job`, `Goal`, `AgentAddress`), type hints, and `asyncio`, and every core or logic module ends with an assert-based `__main__` self-check (see Testing).

The orchestrator ties the agents together without hard-coding which one handles what. It fetches the cards of the agents that are actually running, asks the LLM to name the best fit, and forwards the message over A2A. Two seeded LLM calls do the routing — `choose_agent` picks the agent and `needs_resume` decides whether to attach the user's CV — so a given query routes the same way every run (status yields and cache writes are elided here):

```
1   # agents/orchestrator.py
2
3   # cards of the agents actually running
4   cards = await discover(http)
5
```

```
    if name is None:
        yield "error", "No discovered agent fits that request.
        return
    _url, card = cards[name]

    if name in (RANKER_NAME, CV_ASSISTANT_NAME):
        jobs = _load_jobs() or await _collect(
            http,
            cards[JOB_SEARCHER_NAME][1],
            query
        )
        message = _rank_message(query, resume, jobs)   # scori
    else:
        message = _attach(query, resume)
            if needs_resume(query)
            else query

    async for update in _send(http, card, message):
        yield update
```

The Ranker and CV Assistant score against artifacts the query never carries, so the orchestrator supplies them: the jobs the Job Searcher last cached (searching first if there are none) plus the resume. A plain search forwards the raw query and attaches the CV only when the LLM judges it relevant. Either way `_send` streams the chosen agent's task status back as `(state, text, data)` triples for the CLI to render — the same `data` slot the Job Searcher uses to hand back its structured jobs.

1.7 Testing

Testing follows from the same split that shapes the code: because every agent keeps a protocol-free core apart from its A2A executor,

Each core and logic module ends with an assert-based `__main__` self-check that does exactly that. The check imports nothing from `a2a`, makes no HTTP call, and reaches no LLM, so running the module runs its tests offline and in a fraction of a second. There is no test framework and no separate test tree; the checks live at the bottom of the module they guard, next to the code they describe.

The self-checks target the pure logic, the parts that must be right regardless of what the model or the job board returns. `refine.py` checks the deterministic `_broaden` fallback (dropping a seniority qualifier, then the trailing token, then converging on a single-token query) and the `_clean` validation that decides when an LLM reply is unusable and the fallback takes over. `roster.py` asserts the invariants that would silently break a run if they drifted: every server binds a distinct port, 8004 stays reserved, and the orchestrator's discovery list holds exactly the three A2A agents and neither MCP server. `config.py` checks that a pasted OpenRouter URL ending in `/chat/completions` is trimmed back to the base the SDK expects, and that the `.env` loader skips comments and malformed lines, strips quotes, and never overrides a value already in the environment. The Job Searcher's core checks its three stop conditions and the id-based dedupe, and the Adzuna modules check their query mapping and description trimming.

Running one check is `python -m yahr.<module>`, which prints a single `... self-check ok` line on success or raises an `AssertionError` at the first broken invariant. The whole suite is the same command over each module:

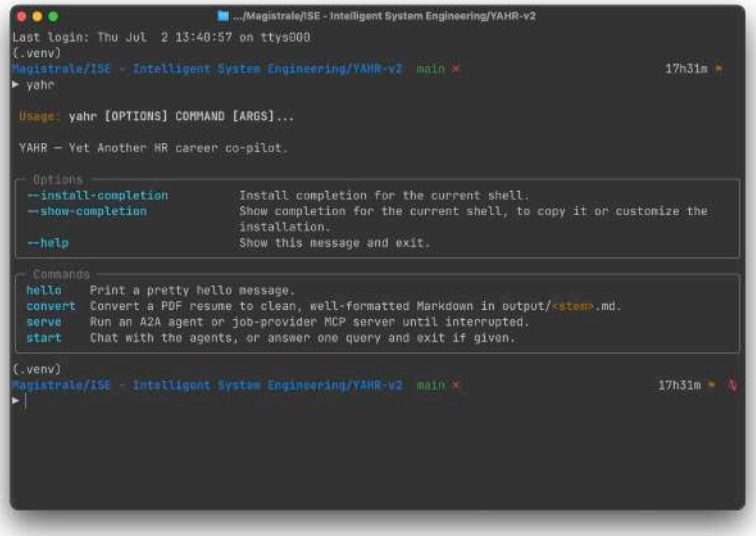
```
for m in agents.roster config \
         agents.job_searcher.refine agents.job_searcher.cor
         agents.ranker.core agents.cv_assistant.core \
         mcp.mock.jobs mcp.adzuna.extract mcp.adzuna.enrich
```

```
6     python -m "yahr.$m" || break
7   done
```

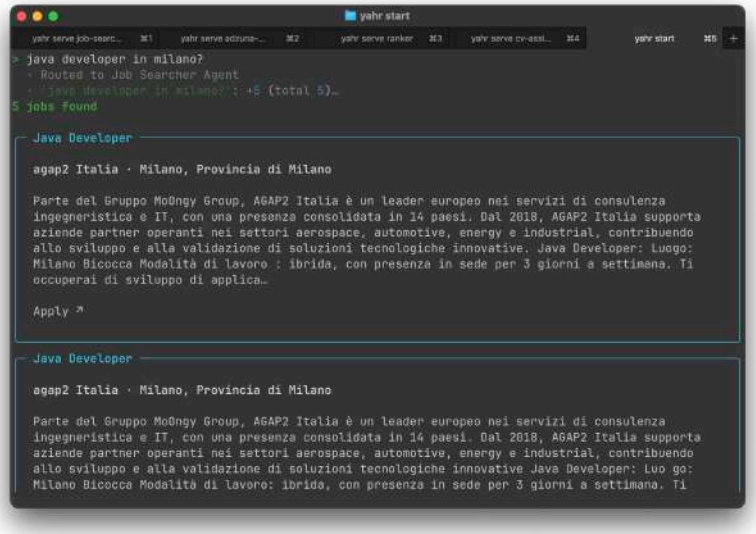
The boundary is deliberate. The self-checks cover the deterministic logic, not the best-effort paths that depend on an outside service: an actual OpenRouter completion, a live Adzuna page, or a running A2A agent answering over HTTP. Those are the seams the design already treats as fallible, where a failed model call drops to a heuristic and an unreachable page keeps its truncated text, so they are validated by their fallbacks rather than by asserting on a network round-trip. Testing the agent-to-agent and MCP flows end to end would need fixtures that start servers and mock the model, which is where a `pytest` suite would earn its place; the current checks stop where a server or a network would have to start.

1.7.1 End-to-end, by hand

Where the self-checks stop, a manual run takes over. The screenshots below drive the whole live system — the four A2A agents and the Adzuna MCP server each running under `yahr serve` (one pane per process), with `yahr start` as the client — exercising exactly the paths the offline checks leave out: LLM routing, a real Adzuna page, and each agent answering over HTTP.

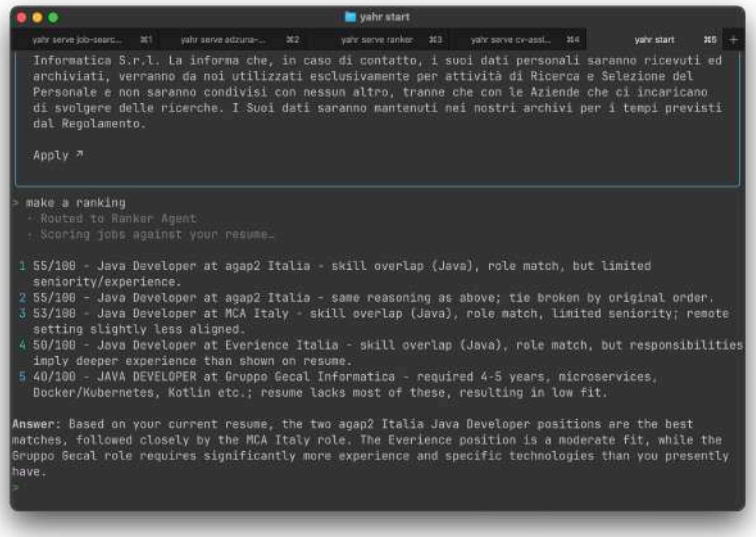


`yahr` with no command prints the top-level help: the entry points the rest of the walkthrough leans on — `serve` to run one agent or the MCP server, `start` to chat with them.

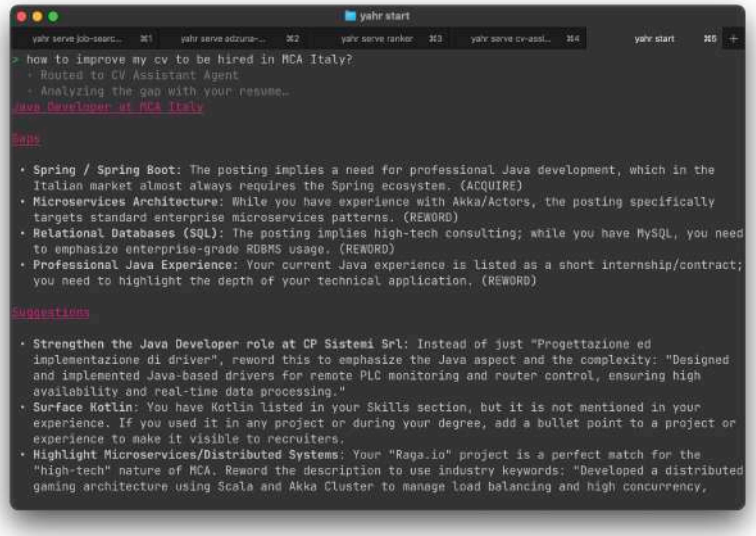


`java developer in milano?` is routed to the Job Searcher, which queries live Adzuna and streams the postings back as cards; the `+5`

`(total 5)` line is the goal-seeking loop reporting its cache against the target.



`make a ranking`, a follow-up turn in the same REPL, routes to the Ranker; it reuses the jobs and resume cached from the search and scores each posting `<score>/100 - <title> at <company> - <reason>`, breaking ties by original order, then answers in prose.



`how to improve my cv to be hired in MCA Italy?` routes to the CV Assistant, which diffs the resume against the chosen posting into Gaps — each tagged ACQUIRE or REWORD — and concrete rewording Suggestions.

Together these cover the seam the assert-based checks cannot reach: the routing decision, the network round-trips, and the model's own output — the part of the system only a running instance can show is working.

1.8 Deployment

YAHR deploys as a set of local processes on one machine. There is no cloud, no container, no orchestration platform. The candidate installs the package once and runs `yahr` from a terminal, and every part of the system is a process that starts and stops on that same host. This follows from the domain: one person looking for work on their own machine, so the deployment never has to deal with multi-tenancy, remote access, or shared state.

Installation is a single editable install. YAHR needs Python 3.14 or newer; from a virtual environment, `pip install -e .` builds the package with Hatchling and puts the `yahr` command on the path. Every process starts through that one command.

At runtime there are two layers of process. The A2A agents and the job-provider MCP servers are long-running servers, each started with `yahr serve <name>` and each binding its own TCP port on `127.0.0.1`:

Server	**`serve` name**	**Port**
Job Searcher (A2A)	`job-searcher`	8002
Ranker (A2A)	`ranker`	8003
CV Assistant (A2A)	`cv-assistant`	8007
Adzuna (MCP)	`adzuna-mcp`	8005
Mock jobs (MCP)	`mock-mcp`	8006

Table 1.1

They bind the loopback interface only, so nothing is reachable from outside the machine. The addresses come from the roster, the single source of truth described in the Code section, so the port a server binds and the port the orchestrator probes for it are the same value written once.

The two kinds of server run on different stacks. Each A2A agent is a Starlette ASGI app that uvicorn serves over HTTP, with its agent card at the well-known URI and its JSON-RPC endpoint at the root. Each MCP server is a FastMCP app served over streamable HTTP, mounted at `/mcp`. The candidate starts only the servers a run needs: a ranking run wants a job-provider server, the Job Searcher, and the

Ranker, each in its own terminal, while an offline run swaps the Adzuna server for the mock and points `YAHR_JOBS_MCP_URL` at it.

The orchestrator is not a server. It runs in-process inside `yahr start`, so the CLI itself is the second layer: a short-lived process that reads the query and resume, routes to whichever agents are up, streams the result, and exits, or stays open as the REPL. Port 8004 is reserved for a standalone orchestrator that is planned but not yet built. Because `start` only routes to agents that are actually running and serving a card, an agent is part of a run exactly when its server is up. A new agent joins the same way: start its server, list it in the roster, and nothing else changes.

The deployment keeps almost no state, and what little it keeps is plain files. Each agent holds its A2A task state in an in-memory store, so those tasks vanish on restart and there is no database. What does survive is two files the orchestrator writes under `output/`: `jobs.md`, the jobs the Job Searcher last found, and `resume.md`, the converted profile that `convert` writes and `start` reads back. They persist on disk on purpose, so a later ranking or resume query can reuse the jobs and the resume without a fresh search or re-attaching the file; a restart picks them up rather than starting clean. Configuration is external, read at startup from the environment or a local `.env`: the OpenRouter key, base URL, and model for every reasoning step, the Adzuna credentials plus the optional `ADZUNA_COUNTRY` (defaults to `it`) for real listings, and `YAHR_JOBS_MCP_URL` to choose the job provider. The only things the deployment reaches off the machine are those two HTTPS services: OpenRouter for the model calls and Adzuna for the listings behind its MCP server. Everything else is loopback traffic between local processes.

1.9 Conclusion

YAHR set out to automate the parts of a job search that are mechanical and repetitive: finding open positions, judging how well each one fits, and working out what a resume is missing for a given role. It does this from the terminal, over a resume the candidate converts once, and it keeps every judgment anchored to what the resume actually says rather than to anything the model might invent.

Splitting the work across an orchestrator and three single-purpose agents, each an A2A server that knows nothing about the others, kept the pieces loosely coupled: a new agent joins by starting its server and listing itself in the roster, and nothing else changes. Pushing the job boards one layer further out behind MCP made the providers swappable too, so the same Job Searcher runs against Adzuna or an offline mock by changing one URL. Keeping each agent's protocol layer apart from a protocol-free core meant the searching, scoring, and analysis could be exercised without a server at all. These choices cost more moving parts than a single script would, and for one person on one machine that overhead is real; what it buys is that each part can change, be tested, or be replaced on its own.

The system is honest about its limits. Determinism is best effort, holding only as far as the model provider honors the seed. The orchestrator still runs in-process inside the CLI rather than as its own server, with a port reserved but the standalone version not yet built. The search loop is bounded by a budget because the real job API is paid and rate-limited, so "enough jobs" is a compromise between coverage and cost rather than an exhaustive sweep. And the CV Assistant stays deliberately advisory: it names gaps and suggests edits, but it never rewrites the resume for the candidate.