Computer Science Department

Software Engineering & Business Analysis

Bachelor's Thesis

Real-Time Labor Market Monitoring System

Design and Implementation of an Open Platform for Labor Market Intelligence

Oleksandr Kolomiiets

Supervisor
KSE, Ihor Pysmennyi, ipysmennyi@kse.org.ua

Expert
Company, Expert Name, expert@domain.ua

Submission date
16 July 2026

Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date Kyiv, 04.06.2026

Signature _____

Contents

Acknowledgements

Grateful feels right when I think of Igor Pysmenny. His Cloud Architectures course opened unexpected doors - lectures that stuck, building something solid step by step. His clarity shaped how I see systems now. Several key decisions in this project trace back to that class: the Docker Compose setup, deployment on GCP, separating web and worker containers. Most of the “Design” choices began as notes taken in that room.

Thank you to Artem Korotenko for fostering a culture at KSE where building something real counts as valid scholarly work.

This platform exists because three organizations chose to make their data public. USAJOBS, published by the U.S. Office of Personnel Management, offers open access to job listings via API. The European Commission built ESCO over many years into a standardized, Creative Commons-licensed taxonomy that makes cross-national skill comparison possible. Adzuna and Jooble each provide global job listings through accessible APIs.

Abstract

Due to the rapid pace of change in the labor market, there is a growing need for a tool capable of providing up-to-date insights, forecasts, analytics, and statistics on the relevance and applicability of specific skills, competencies, and knowledge. Unfortunately, standard labor market reports do not provide us with the full scope of necessary information, due to irregular reporting, limited access, or insufficient analysis for making informed decisions.

This paper presents the development and implementation of a real-time labor market monitoring platform. VakansioGraph autonomously collects job postings from various open sources, such as USAJOBS, Adzuna, and Jooble, every two hours. To do this, the system uses their public APIs, adhering to all limits, restrictions, and data usage policies. The collected job postings are normalized according to the European Skills, Competences, Qualifications, and Occupations (ESCO) taxonomy, which allows for the comparison of skills across countries using a standardized hierarchical dictionary of 13,960 unique skills.

The result is the VakansioGraph platform, which in just a few months of operation has already collected over 400,000 job postings from more than 40 countries and 55,000 unique organizations. The platform also offers salary analytics, skill demand rankings, geographic distribution, and AI-generated market insights via Google Gemini. The system is deployed as a containerized, production-ready application using Docker, Nginx, and Gunicorn on Google Cloud Platform, with automated CI/CD via GitHub Actions.

This work demonstrates that an open labor market analytics platform can be built using publicly available data and technologies, providing real-time analytical capabilities comparable to commercial solutions.

1 Introduction

1.1 Context and Motivation

These days, changes are taking place almost every day in various areas of our lives. One of the most important of these is the job market. New skills are emerging, while old ones are becoming obsolete; salaries, demand, and job categories are all changing. And this affects everyone, whether it's a high school student choosing their path, a college student who wants to stay relevant in the job market, a university designing its curriculum, or an employer seeking a competitive workforce-they all need an easy-to-use tool with up-to-date data on the state of the labor market.

The problem is that high-quality labor market data is hard to come by. Government agencies, such as the U.S. Bureau of Labor Statistics or Eurostat, publish comprehensive reports, but only monthly or quarterly, and by the time they are released, the data is already several weeks out of date. This is suitable for studying long-term trends. But it is not useful if you want to know what is happening right now.

1.2 Problem Statement

First, there are already companies that excel at labor market analytics. For example, LinkedIn Talent Insights, Lightcast, and EMSI are capable of processing millions of job postings and selling this analytics to clients. However, these platforms are very expensive and inaccessible to ordinary students, researchers, or small companies.

Second, the timeliness of the data. Platforms like the BLS, or government agencies, provide reports for the previous month, or even the previous quarter. Given that thousands of job postings are published daily, this data simply becomes outdated and makes it impossible to make any predictions for the near future.

Third, significant attention must be paid to the standardization and terminology of skills. Employers describe the same skill in different ways. "Team leadership," "people management," "leading cross-functional teams"—all of these mean roughly the same thing. A system that treats them as separate keywords yields meaningless results. Without standardizing skills according to a common standard, it is impossible to make any meaningful comparisons.

Thus, there is a gap. On the one hand, there is a real need for fresh, detailed data on the labor market. On the other hand, the tools that provide this data are either behind paywalls or contain outdated information. It is precisely these problems that this dissertation seeks to address.

1.3 Objectives

The goal of this thesis is to create a functional and accessible platform that would address all of the issues listed above. In practice, this would mean that the system would have to automatically collect data from sources in real time, 24 hours a day. It would use only official APIs, adhering to all bandwidth limits and service terms of each provider.

The system would need to normalize the collected data according to the ESCO taxonomy, as well as process unstructured text from real job postings, including U.S.-specific terms not covered by ESCO.

It would also need to provide data via a proper API with a sufficient number of endpoints and high throughput. It should also be truly deployable, containerized, with automated CI/CD, running on a real cloud infrastructure.

1.4 Method

Because the platform's functionality depends directly on the availability of data, we decided to begin development with a job parser, data extractor, normalization layer, database, API implementation, and finally the front end. Each layer underwent separate testing before moving on to the next.

Three open sources were selected for data collection. USAJOBS (the official U.S. federal job portal), Adzuna, and Jooble (international job aggregators). All three provide documented public APIs. Data collection occurs via a background process every two hours, adhering to rate limits and delays between requests.

Skill normalization is based on the ESCO taxonomy. ESCO is an open standard from the European Commission with nearly 14,000 skills, specifically designed for cross-country comparisons. I chose ESCO because it is free, multilingual, and has unique identifiers for each skill. Matching occurs in three steps: manual mappings for specific skills, exact search, and fuzzy search for cases where there is no exact match.

The tech stack was selected to meet the project's actual needs. Flask because asynchronous code isn't required. SQLite in WAL mode because the load is predominantly read-heavy, and a single database significantly simplifies deployment compared to PostgreSQL. APScheduler to avoid maintaining a separate task queue just for a two-hour cycle. Docker and Docker Compose—so the entire system can be deployed with a single command. Deployment is automated via GitHub Actions to a Google Cloud Platform virtual machine.

1.5 Structure of This Thesis

Chapter 2 surveys what already exists, including data sources, commercial platforms, and open-source tools, and derives the requirements from the gaps.

Chapter 3 covers the design: the architecture diagrams, the data flow, and the reasoning behind the main technical choices.

Chapter 4 details the implementation, including the collection pipeline, ESCO normalization, the Flask API, the scheduler, the Gemini AI integration, and CI/CD.

Chapter 5 validates the system through API tests, cache performance measurements, scheduler reliability checks, normalization coverage, and API policy compliance.

Chapter 6 concludes by reviewing what was achieved, what was hard, where the system falls short, and what comes next.

2 Research

Contents

2.1 Existing Labor Market Data Sources

The most authoritative sources of labor market data are government statistical agencies. The U.S. Bureau of Labor Statistics publishes the Job Openings and Labor Turnover Survey (JOLTS) [1], the Occupational Employment and Wage Statistics (OEWS) [2], and the Current Population Survey all of which are methodologically rigorous, peer-reviewed, and freely available. Eurostat does something comparable for Europe [3]. The U.K.'s Office for National Statistics, Germany's Bundesagentur für arbeit, and similar bodies in most OECD countries maintain their own labor market datasets.

The problem with all of these is timing. JOLTS data is released monthly, but covers two months prior. OEWS estimates are annual. The data is aggregated to the industry or occupational level meaning that if you want to know how many Python developer jobs were posted in the last two weeks, or whether demand for cloud security skills increased after a major breach, these sources simply cannot tell you. They are designed for macro-economic analysis, not operational workforce intelligence.

The U.S. government provides USaJOBS as the official job portal for federal agencies, offering comprehensive information about the job openings themselves. USaJOBS itself provides a free, authenticated REST API [4], which was a key factor in selecting this data source.

To ensure international coverage, two additional sources were integrated: adzuna [5], to cover European countries, and Jooble [6], to cover Ukraine. Both are job aggregators that provide authenticated APIs for accessing thousands of job openings. They also impose limits on the number of requests, which the system adheres to.

2.2 Commercial Intelligence Platforms

The platforms that actually solve the real-time monitoring problem are the commercial ones, and they are worth understanding in detail not because they are competitors in any meaningful sense, but because they define what "good" looks like in this space.

LinkedIn Talent Insights [7] gives enterprise HR teams access to aggregated data on talent supply, hiring trends, and skill demand drawn from LinkedIn's user base of over 900 million profiles and the job postings published on the platform. The analytics are

genuinely impressive: you can see how hiring velocity for a specific role has changed quarter over quarter, which universities are producing the most candidates for a given position, or how salary expectations differ between cities. None of this is available through any public API. access requires a LinkedIn Talent Solutions enterprise contract, priced at a level that makes it inaccessible to most organizations.

Lightcast [8] formed from the merger of Burning Glass Technologies and EMSI is the dominant player in the labor market analytics space outside of LinkedIn. Their platform processes over a billion job postings per year from thousands of sources, normalizes them against their proprietary skill taxonomy (EMSI Skills), and provides detailed analytics on occupation trends, regional labor markets, and skills gaps. Their data is used by universities for curriculum design, by governments for workforce policy, and by corporations for talent strategy. The price point is substantial. There is no public API, no data export, and no way to audit or reproduce their methodology.

Revelio Labs [9] takes a different angle, focusing on workforce analytics derived from professional profile data rather than job postings tracking employee movements, organizational growth and contraction, and skill composition at the company level. again, entirely closed, entirely paid.

What all of these platforms share is that they have solved the hard engineering problems: continuous data collection at scale, skill normalization, geographic aggregation, and usable interfaces. What they have not done is make any of that accessible or reproducible.

2.3 Open Source and academic alternatives

The open-source landscape for labor market analysis is sparse and fragmented. Most of what exists falls into one of two categories: one-off research tools built for a specific paper, or government-adjacent datasets that reflect the same freshness problems as the official statistics. architecturally, almost none of it is designed as a continuously running service the typical artifact is a standalone script or notebook.

O*NET [10], maintained by the U.S. Department of Labor, is probably the most widely used open resource for occupational data. It provides detailed skill and task descriptions for hundreds of occupations, updated periodically through employer surveys. It is excellent for understanding the skill composition of a given job category in general terms, but it is not a real-time data source and it is not designed for automated integration into a monitoring pipeline.

SkillNer [11], the Skills Extractor Library, and similar Python packages attempt to automate skill extraction from job postings using rule-based matching against curated skill lists. These work reasonably well for common skills but struggle with domain-specific terminology, negation handling (“does not require”), and the sheer variability of how employers phrase requirements. They also produce inconsistent output — the same skill extracted differently depending on exact phrasing which makes aggregation across sources unreliable.

Several academic papers have addressed skill extraction from job postings as a research problem. Zhang et al. [12] proposed using weak supervision with the ESCO taxonomy to find similar skills in job ads via latent representations, significantly reducing the cost

of manual annotation. Decorte et al. [13] explored how job title understanding through skill matching can improve occupational classification accuracy. Clavié and Soulié [14] demonstrated that large language models can match job ad text to ESCO skill definitions in a zero-shot setting. These approaches are methodologically interesting, but the associated codebases are not designed for continuous production use, and the datasets they operate on are not publicly available.

The honest conclusion from surveying this landscape is that there is no existing open-source platform that combines continuous multi-source data collection, skill normalization against a standard taxonomy, and a usable API into a single deployable system. The components exist in isolation. The integration does not.

2.4 Comparison and Gap analysis

Bringing the surveyed solutions together against the criteria that matter for a real-time monitoring platform makes the gap explicit.

Solution	Real-time	Open API	Multi-country	Skill norm.	Cost
BLS / Eurostat	×	partial	×	×	free
LinkedIn Insights	✓	×	✓	✓	$$$
Lightcast / EMSI	✓	×	✓	✓	$$$
Revelio Labs	✓	×	✓	✓	$$$
O*NET	×	×	×	partial	free
USAJOBS	✓	✓	×	×	free
This project	✓	✓	✓	✓	**free**

Table 1: Comparison of existing solutions against real-time monitoring criteria

None of the existing solutions combine real-time data, open APIs, multi-country coverage, and normalized forecasts into a single, free system.

Commercial platforms have all the necessary capabilities, with the exception of openness. Free sources are open, but they lack either real-time data, multi-country coverage, or forecast normalization.

3 Design

Contents

3.1 From Requirements to architecture

The requirement to use official public APIs only had the largest effect. It ruled out browser automation, undocumented endpoints, and aggressive polling, all of which would have made data collection easier but would have broken the terms of service of every provider involved. Instead, rate limiting had to become a first-class concern in the design: explicit delays between requests, authenticated sessions, and a request budget that stays within each provider's published limits. This is why data collection lives in a dedicated background process rather than being triggered by user requests. a user clicking a button must never be able to cause an API call that violates a rate limit.

The requirement for fault tolerance shaped the collection pipeline. a monitoring system that loses hours of work to a single network timeout is a prototype, not a product. The pipeline therefore had to be able to resume from the exact point of interruption without re-fetching data or creating duplicates, which led to the checkpoint mechanism and to database-level deduplication on the posting identifier.

The requirement for reproducible deployment from a single repository checkout shaped the database and the overall topology. a client-server database would have meant a separate managed service and a more complex deployment. a single-file embedded database that runs inside the same deployment removes that complexity entirely. This is the reasoning behind the choice of SQLite, examined in detail in the technology stack section below.

The requirement for sub-second cached responses shaped the introduction of a two-layer caching strategy and a precomputed analytics table, so that expensive aggregations never run inside a user-facing request.

Taken together, these requirements push toward a simple, self-contained architecture: a small number of cooperating processes, one embedded database, and no external services that are not strictly necessary.

3.2 System architecture

The system runs as Docker containers under Docker Compose. The development configuration has four containers (Nginx, web, worker and Redis) sharing a named volume for the SQLite database. The production configuration adds a Redis container for response caching.

3.2.1 C4 Level 1: Context

At the highest level, the platform sits between its users and its data sources. Users access the system through a web browser over HTTPS. The system connects outward to three external job data providers through their official public APIs, reads a locally stored ESCO dataset for skill normalization, and optionally calls the Gemini AI API to generate market summaries.

Every outbound connection to an external API is authenticated, rate-limited, and compliant with each provider's terms of service. The ESCO dataset is the one data source that is not fetched over the network. It is loaded from a local, versioned CSV file, so normalization has no dependency on network availability and the taxonomy does not change between runs.

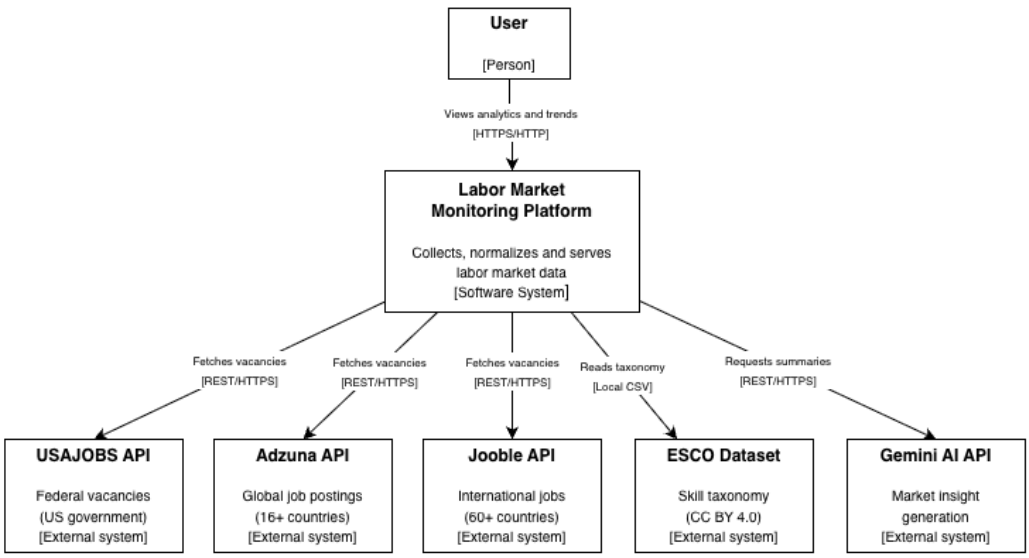

Figure 1: C4 Level 1: system context, showing users, external APIs, and the platform boundary

3.2.2 C4 Level 2: Containers

Inside the Docker Compose deployment, containers run concurrently. The diagram annotates each connection with its protocol.

Nginx is the entry point. It listens on port 80, terminates incoming HTTP connections, serves static files directly, and reverse-proxies dynamic requests to the web container over HTTP on port 5001. Keeping static file serving in Nginx leaves the Python process free to handle API requests.

The web container runs Flask behind the Gunicorn WSGI server. It handles all incoming HTTP requests, renders the frontend templates, and exposes the REST API. On startup it loads the ESCO index into memory, a one-time cost that makes every subsequent normalization lookup a fast in-memory operation. API responses are cached at two levels, with Redis used if it is available and an automatic fallback to a filesystem cache otherwise.

The worker container runs aPScheduler in its own process. Every two hours it executes the full collection pipeline: fetching from the three job APIs over HTTPS, extracting and normalizing skills, rebuilding the analytics cache. It writes to the same SQLite database file as the web container through the shared volume mount.

The Redis container, present in the production configuration, provides the backing store for response caching. The web container connects to it over the internal Docker network; if it is unavailable, caching transparently falls back to the filesystem, so the system runs identically with or without Redis.

The shared database volume is the only channel of communication between the web and worker containers. There is no message queue, no inter-process socket, and no shared memory. This is a deliberate decision: for a workload with a single writer and a two-hour cycle, a message broker would add operational weight and an additional failure mode without any benefit.

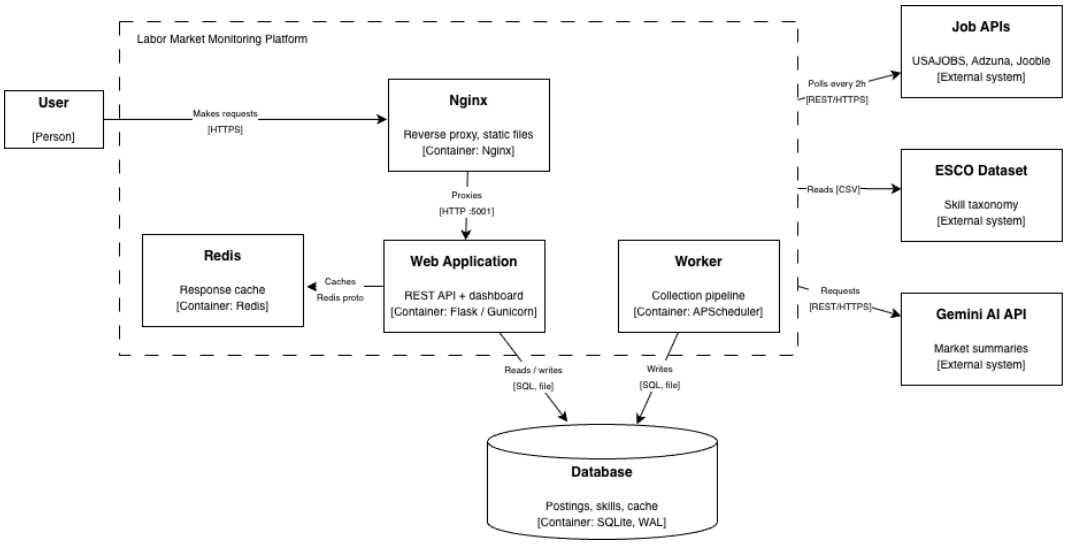

Figure 2: C4 Level 2: Docker containers, the shared database volume, and external connections with protocols

3.2.3 C4 Level 3: Components

The two containers that hold application code, the web container and the worker container, each warrant their own component view. Following the C4 model, a component diagram is scoped to a single container, so the two are presented separately.

3.2.3.1 Web Container

The web container serves the request path. Its components are organized into clearly separated layers.

API Routes consist of seven Flask blueprints, each owning a logical group of endpoints: job postings, skills, salaries, locations, categories, trends, and analytics insights. Every endpoint is wrapped with a cache decorator that sets its time-to-live, typically 600 seconds for analytical endpoints and 7200 seconds for aI-generated content.

Services are six modules that sit between the routes and the database. Each owns a single domain: **`JobService`** for recent postings, **`SkillService`** for aggregated skills, **`SalaryService`** for salary distributions, **`TrendsService`** for skill ROI and posting volume, **`LocationService`** for geographic breakdowns, and **`CategoryService`** for occupational series. Services never call one another, which keeps each of them independently testable.

analytics Builder precomputes expensive aggregations and stores them in the **`analytics_cache`** table. at startup and after every collection cycle it runs the queries that would otherwise be too slow to execute per request, such as total counts, skill demand rankings, and salary averages by dimension, and writes the results as serialized JSON. When a Gemini API key is present, it also produces a short natural-language market summary and caches it for two hours.

Database utilities expose a single **`get_db_connection()`** function that configures each SQLite connection with WaL journaling, **`synchronous=NORMaL`**, and a 30-second busy timeout, and registers it for cleanup when the request ends. Every service and the analytics builder use this function rather than opening connections directly.

Classifiers is a static module holding occupational series codes and skill category mappings, lookup tables too small to justify a database table but too large to inline in service code.

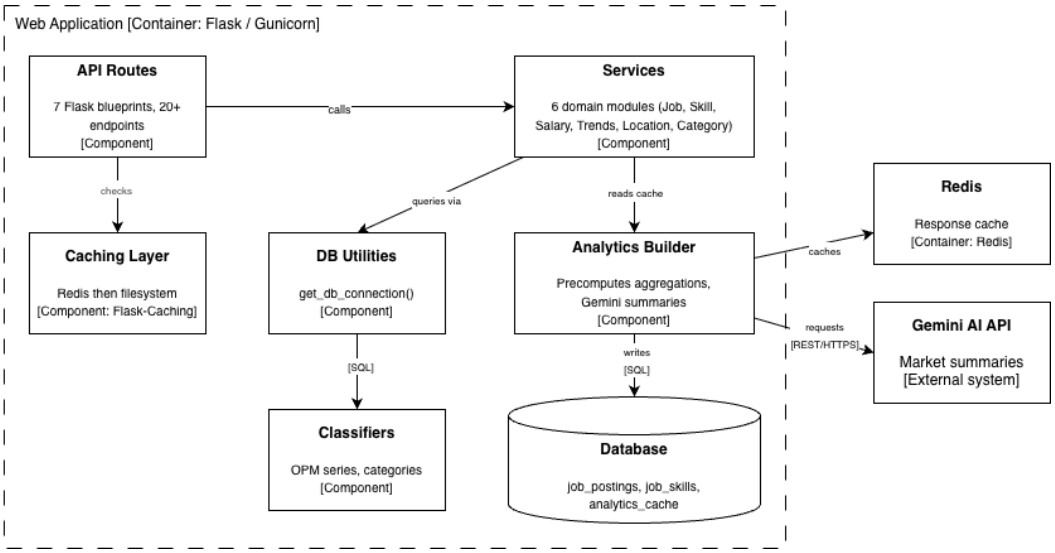

Figure 3: C4 Level 3: components of the web container (request path)

3.2.3.2 Worker Container

Scheduler is the entry point, an aPScheduler instance running a two-hour collection cycle and a weekly cleanup job. It triggers the collectors and then the normalization pipeline in sequence.

Collectors wrap each external data source: the USaJOBS client, the adzuna parser, and the Jooble parser. Each is guarded by its own credentials and skips cleanly if they are absent, so a missing provider degrades gracefully rather than failing the whole cycle. Collectors write new postings directly to the database.

ESCO Pipeline is a self-contained package (**`esco_pipeline`**) that turns raw skill strings into ESCO concepts. a **`Pipeline`** orchestrator coordinates extraction and mapping; a set of interchangeable **Mappers** (fuzzy, embedding, LLM, and a vLLM-optimized variant) sit behind a common **`BaseMapper`** interface; an **ESCO Index** holds the taxonomy with its labels, embeddings, and graph relations; and an **Evaluation** module scores any mapper against a gold standard using precision, recall, and F1. The strategy comparison and selection are detailed in the Implementation chapter.

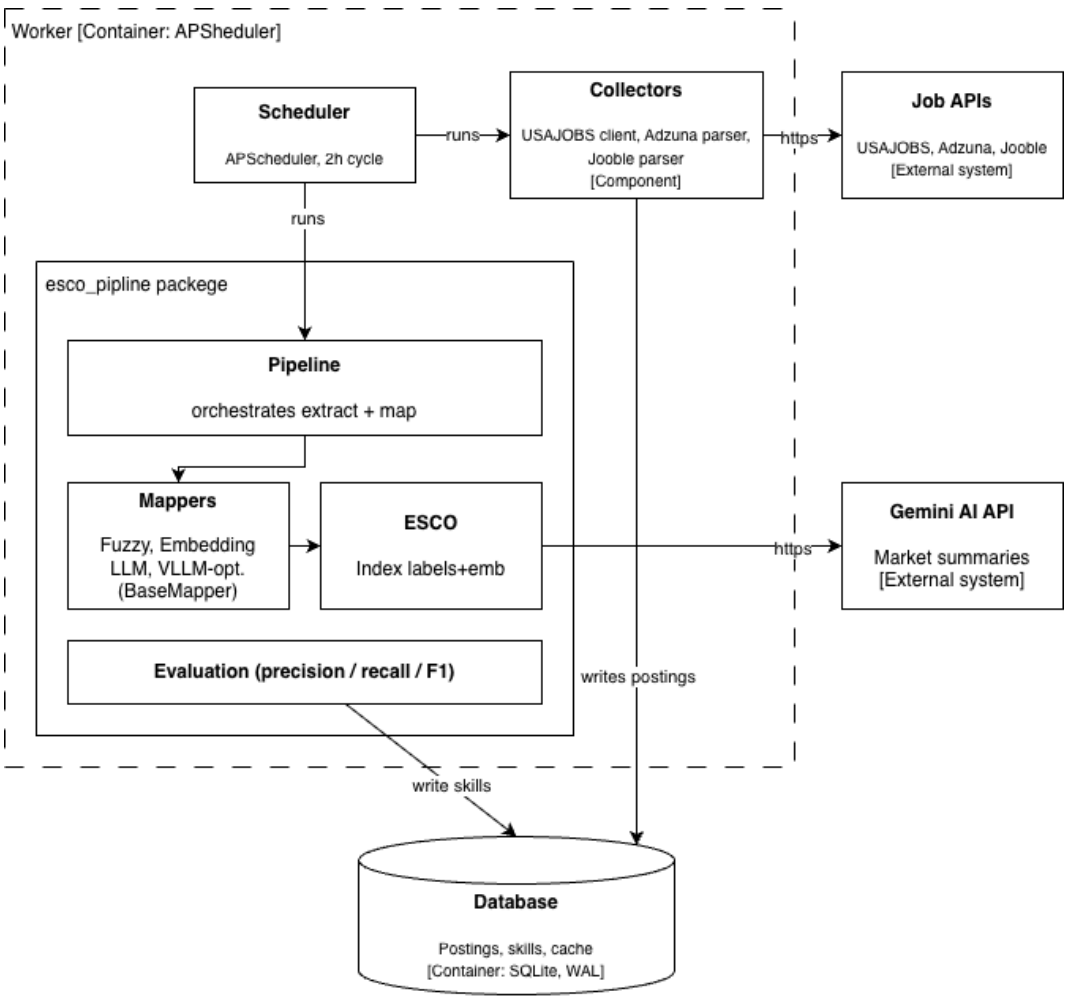

Figure 4: C4 Level 3: components of the worker container (collection path and ESCO pipeline)

3.3 Technology Stack

Every technology in the stack maps to a specific container or component, and each choice follows from the requirements rather than from preference. The table below summarizes the mapping, and the paragraphs that follow explain the trade-offs.

Technology	Role	Container / component
Nginx	Reverse proxy, static files	Nginx container
Flask + Gunicorn	Web server and REST API	Web container
aPScheduler	Scheduled collection pipeline	Worker container
SQLite (WaL)	Persistent storage	Shared volume
Flask-Caching	Response caching layer	Web container
Redis	Cache backing store (prod)	Redis container
Chart.js	Frontend visualization	Web container (static)
Docker Compose	Orchestration	Whole deployment
GitHub actions	CI/CD pipeline	External to runtime

Table 2: Mapping of each technology to its container or component

Flask was chosen over FastAPI because the project does not need asynchronous I/O. all database operations are synchronous SQLite calls, and the collection pipeline runs in a separate process, so the concurrency model that FastAPI offers would add complexity without benefit. Flask's simplicity and its mature ecosystem of extensions made it the natural fit.

SQLite in WaL mode matches the access pattern exactly. The web container reads frequently, the worker writes in batches every two hours, and write-ahead logging lets reads and writes proceed concurrently without locking. For a dataset of a few hundred thousand rows the performance is more than adequate, and the operational simplicity of a single file with no daemon and no connection string dramatically reduces deployment complexity. The trade-off is scale: SQLite has a single writer and will become a bottleneck if the dataset grows by an order of magnitude, a limitation addressed in the conclusion.

aPScheduler runs the background pipeline directly inside a Python process, avoiding a separate task queue such as Celery or Dramatiq. For a single worker on a two-hour cycle, the overhead of a full broker-backed queue would be disproportionate to the need.

Chart.js is bundled with the application rather than served from a CDN. It needs no build step and has no runtime dependencies, and bundling it removes a class of failures where a CDN outage would break the dashboard. The trade-off is a slightly larger application image, which is negligible at this scale.

3.4 Data Model and Flow

3.4.1 Storage Model

The database holds four tables. **`job_postings`** is the central entity, keyed on the provider-supplied position identifier, holding the title, organization, location, salary, grade,

occupational series, and descriptive text for each vacancy. **`job_skills`** links each posting to the skills extracted from it, storing both the raw skill string and its normalized ESCO URI, label, and category. **`collected_job_ids`** supports fast deduplication checks, and **`analytics_cache`** is a key-value table of precomputed JSON aggregations. The schema is deliberately denormalized on the read path: analytics queries favor a small number of wide tables over many joins, which suits both the read-heavy workload and SQLite's strengths.

To keep analytical queries fast on these tables, the schema defines eleven indexes covering the columns most frequently filtered or grouped on: skill label, type, and category in **`job_skills`**, and salary, series, department, grade, country, and date in **`job_postings`**, including a composite country-plus-date index for the time-series endpoints.

3.4.2 API Contracts

The REST API exposes a single, unversioned set of JSON endpoints. Every response is a JSON object or array, every error is returned as a JSON body with an HTTP error status rather than an HTML page, and query parameters control filtering and limits. The API is currently unversioned because it has a single consumer, the bundled frontend; if it were opened to external consumers, a path-based version prefix would be the natural next step.

3.4.3 Request and Collection Paths

The system has two independent data paths. On the request path, the browser sends an HTTP request to Nginx, which proxies it to Gunicorn and into the appropriate blueprint. The blueprint consults the cache and returns immediately on a hit; on a miss it calls the relevant service, which queries SQLite, formats the result, caches it, and returns it.

The collection path is entirely separate and never overlaps with user requests. Every two hours aPScheduler triggers the pipeline, the three providers are polled in sequence with rate-limiting delays, new records are inserted with deduplication handled at the database level, skills are normalized, and the analytics cache is rebuilt. The two paths share only the database, and they touch it in complementary ways: the request path almost exclusively reads, the collection path almost exclusively writes.

3.5 Deployment Topology

The application runs on a single Google Cloud Platform e2-medium virtual machine. Nginx is the only component exposed to the public network, listening on the standard HTTP and HTTPS ports; the Flask and worker containers are reachable only inside the Docker Compose network and are never exposed directly. There is no load balancer, because the system runs as a single instance, and there is no automatic failover, because a single VM has no second node to fail over to. These are conscious limitations appropriate to the scale and budget of the project rather than oversights, and they are revisited in the conclusion.

Persistence and backup rest on the shared Docker volume that holds the SQLite database file. Because the database is a single file, backup is straightforward: a periodic copy or volume snapshot captures the entire state of the system. Restoring is equally simple, requiring only that the file be placed back into the volume before the containers start.

Deployment itself is automated through GitHub actions. On every push to the main branch the workflow runs the test suite with pytest, builds a Docker image, pushes it to the GitHub Container Registry, and connects to the VM over SSH to pull the new image and restart the Compose stack. The whole pipeline runs without manual intervention and completes in under four minutes.

3.6 Cross-Cutting Concerns

Security. External API credentials are read from environment variables at runtime and never appear in the source code, in logs, or in error messages. Internally, only Nginx is exposed to the public network. The most significant gap is the absence of authentication on the administrative cache-rebuild endpoint, which is acceptable in the current low-exposure deployment but would need to be closed before the platform served a wider audience.

Logging. The collection pipeline logs the start and end of every step, the number of new records inserted, and the number of skills normalized. These logs are the primary observability mechanism for the background worker, where there is no user to notice when something goes wrong.

Monitoring. Monitoring. The /API/health endpoint returns a lightweight status response confirming that the web service is up and able to respond, suitable for liveness checks. Data freshness is observed separately through the worker's collection logs rather than through the health endpoint.

Scalability. The architecture scales comfortably for the current workload but has clear limits. The read path scales well because cached responses are served from memory or the filesystem, and SQLite in WaL mode handles concurrent reads without contention.

4 Implementation

Contents

4.1 Development Methodology

The implementation followed a pipeline-first, iterative methodology. Rather than building all layers in parallel, each stage of the data path was implemented and validated before the next was started, in the order: data collection, then ESCO normalization, then the analytics layer, then the REST API, and finally the frontend. The reasoning is that every layer is downstream of the data: a dashboard built on incomplete or incorrectly normalized data is worse than no dashboard at all, so the collection and normalization stages had to be trustworthy before anything was built on top of them.

The ESCO normalization component was developed as a separate, self-contained Python package (**`esco_pipeline`**) with its own configuration, evaluation harness, and interchangeable mapping strategies. This separation allowed the normalization approach to be prototyped and benchmarked in isolation, against a gold-standard dataset, before being integrated into the production enrichment path. Several mapping strategies were implemented and compared rather than committing to one up front, an explicitly iterative approach described in the normalization section below.

4.2 Coding Standards and Architectural Patterns

The codebase follows PEP 8 conventions, enforced in continuous integration through **`flake8`** and formatted with **`black`**. Naming follows Python conventions throughout:

snake_case for functions and variables, **PascalCase** for classes, and a consistent ***Service** suffix for the service-layer classes (**JobService**, **SkillService**, and so on).

Three architectural patterns recur across the codebase. The **service layer pattern** isolates all database access and business logic into single-responsibility service classes that the API routes call, keeping the route handlers thin. The **blueprint pattern** from Flask groups related endpoints into independently registered modules. The **strategy pattern** underpins the ESCO normalization package: every mapping approach implements a common **BaseMapper** abstract interface, so strategies are interchangeable and can be selected by configuration without changing any calling code.

```
class BaseMapper(ABC):
    @abstractmethod
    def map_skills(self, skill_strings: list[str]) -> dict[str, ESCOMapping |
None]: ...

    @abstractmethod
    def name(self) -> str: ...
```

Application configuration is centralized. The web application uses class-based Flask config objects (**DevelopmentConfig**, **ProductionConfig**, **TestingConfig**), and the normalization package uses a Pydantic **Settings** object that reads all tunable parameters (thresholds, batch sizes, model names) from environment variables, so no operational value is hard-coded in logic.

4.3 Data Collection Pipeline

4.3.1 USAJOBS

The USAJOBS client is the primary collector. The USAJOBS API is paginated, and the client iterates through multiple search configurations, each covering a different occupational series, to collect across the full breadth of federal vacancies. Every request carries the **Authorization-Key** and **User-Agent** headers required by the API documentation, both read from environment variables (**USAJOBS_API_KEY**, **USAJOBS_EMAIL**) and never hard-coded. The collector logs each series it processes, the number found, and the number stored, into a **collection_log** table that doubles as an audit trail.

4.3.2 Adzuna and Jooble

International coverage comes from two additional parsers, **parser_adjuna** and **parser_jooble**, each guarded by its own credentials. The scheduler checks for the relevant environment variables and skips a source cleanly if they are absent, so the system degrades gracefully when a provider is not configured rather than failing the whole cycle. Country values from all sources are normalized to ISO 3166-1 alpha-2 codes so that geographic analytics group consistently across providers.

4.3.3 Deduplication

Deduplication is handled at the database level. Each provider's own identifier becomes the **position_id** primary key, so a posting seen again in a later cycle is silently discarded

by **`INSERT OR IGNORE`** rather than duplicated. Identifiers are namespaced by source so that records from different providers can never collide.

4.4 ESCO Normalization Pipeline

The normalization package is the most substantial component of the system and the part where the iterative methodology mattered most. It is structured around a common **`ESCOIndexInterface`** that loads the ESCO taxonomy and exposes lookup, fuzzy search, and embedding search, and a set of interchangeable mappers built on the **`BaseMapper`** strategy interface.

4.4.1 Mapping Strategies

Four mapping strategies were implemented and benchmarked against one another:

Fuzzy mapper. Uses the **`rapidfuzz`** library [15] with token-sort ratio matching against the full set of ESCO preferred and alternative labels, accepting a match above a configurable threshold (default 80). It is fast, deterministic, and fully interpretable.

Embedding mapper. Embeds each skill string with a sentence-transformer model [16] and retrieves the nearest ESCO concepts by cosine similarity, accepting the best match above a similarity threshold (default 0.72). This captures semantic equivalence that string matching misses, for example mapping a descriptive phrase to a canonical skill name.

LLM mapper. Sends batches of skills together with a focused set of candidate ESCO concepts (prefetched via fuzzy and embedding search) to a large language model, which selects the best match. It supports a direct mode and a two-stage mode that rescoring low-confidence skills with additional document context.

vLLM-optimized mapper. A variant of the LLM mapper that sends one skill with roughly ten focused candidates per request, designed to exploit the continuous batching of a locally hosted vLLM server for high throughput on small open models.

The fuzzy and embedding mappers implement the **`map_skills`** contract directly. The example below shows the fuzzy mapper, which illustrates the pattern all mappers follow: take a list of raw skill strings, return a dictionary mapping each to an **`ESCOMapping`** or **`None`**.

```
def map_skills(self, skill_strings: list[str]) -> dict[str, ESCOMapping |
None]:
    labels = self._index.get_all_labels()
    label_list = list(labels.keys())
    results: dict[str, ESCOMapping | None] = {}

    for skill in skill_strings:
        normalized = normalize_text(skill)
        match = process.extractOne(
            normalized, label_list, scorer=fuzz.token_sort_ratio
        )
        if match and match[1] >= self._config.fuzzy_threshold:
            label, score, _ = match
            uri = labels[label]
            results[skill] = ESCOMapping(
                raw_skill=skill, esco_uri=uri,
```

```
                esco_label=self._index.get_skill(uri).preferred_label,
                confidence=score / 100.0, method="fuzzy",
            )
        else:
            results[skill] = None
    return results
```

4.4.2 Graph Enrichment

Beyond direct matches, the pipeline performs graph enrichment using the ESCO broader-skill relationships. When a skill maps to a concept, related concepts reachable through the taxonomy graph can be added as graph-derived mappings, distinguished from direct mappings by a **`via_graph`** flag on the result. This lets the system surface skills that are implied by, but not literally stated in, a posting.

4.4.3 Evaluation Harness

Because four strategies were in play, the package includes a dedicated evaluation module that scores any mapper against a gold-standard set of human-labeled documents. It computes precision, recall, F1, unmapped rate, false-positive rate, and coverage per document, then averages across the corpus and prints a side-by-side comparison table. This harness is what made the strategy comparison rigorous rather than anecdotal: each mapper was run over the same gold set and the metrics decided which approach to use in production.

```
@dataclass
class EvaluationMetrics:
    precision: float
    recall: float
    f1: float
    unmapped_rate: float
    false_positive_rate: float
    coverage: float
```

4.5 Analytics Builder

The analytics builder precomputes every expensive aggregation and stores the results as serialized JSON in an **`analytics_cache`** table, keyed by name (**`overview`**, **`top_skills_20`**, **`skill_demand`**, and so on). It runs after each collection cycle, so the queries that would otherwise scan the full dataset on every request, total counts, skill rankings, salary distributions by dimension, monthly volume, are executed once per cycle rather than once per request.

When a Gemini API key is configured, the builder additionally generates three natural-language market insight cards. It assembles a prompt from the precomputed statistics, the top skill categories with their share, the leading skills, the overview totals, and asks the model for a structured JSON array of insight cards, which the frontend renders directly. If the key is absent or the call fails, the system falls back to insights derived from the same statistics without the model, so the feature degrades gracefully.

4.6 REST API Layer

The Flask application is assembled by a **`create_app`** factory that selects a configuration object, initializes caching, enables CORS for the API namespace, and registers seven blueprints, one per resource group (jobs, skills, salaries, locations, categories, trends, and health). Splitting the API into blueprints keeps each group of endpoints independently testable and registrable.

Caching is initialized with automatic backend selection: the factory attempts to connect to Redis, and falls back to a filesystem cache if Redis is unreachable, so the same code runs in a full production environment and on a bare machine without Redis.

```
def _init_cache(app):
    redis_url = os.environ.get("REDIS_URL", "redis://localhost:6379/0")
    try:
        import redis as _redis
        _redis.from_url(redis_url).ping()
        app.config["CACHE_TYPE"] = "RedisCache"
        app.config["CACHE_REDIS_URL"] = redis_url
    except Exception:
        app.config["CACHE_TYPE"] = "FileSystemCache"
        app.config["CACHE_DIR"] = "/tmp/flask_cache"
    app.config["CACHE_DEFAULT_TIMEOUT"] = 600
    cache.init_app(app)
```

The health endpoint at **`/api/health`** returns a simple status payload for liveness checks, and administrative endpoints under **`/api/admin`** allow the analytics cache to be rebuilt or warmed on demand.

4.7 Background Scheduler

The worker runs APScheduler. The primary job executes the full pipeline, USAJOBS, then Adzuna, then Jooble, then ESCO enrichment, then the analytics rebuild, with each source wrapped so that a failure in one does not abort the others. Credentials for each source are read from the environment, and a missing key causes that source to be skipped with a warning rather than crashing the cycle. In development the scheduler runs in a background thread inside the same process as the web server; in production it runs as its own container.

4.8 Testing and Quality Assurance

Testing uses **`pytest`** with a session-scoped fixture in **`conftest.py`** that builds the application with a **`testing`** configuration and creates the schema in an isolated database, so tests never touch production data and can run in CI without network access. The suite in **`tests/test_api.py`** is organized by resource into test classes (**`TestHealth`**, **`TestOverview`**, **`TestJobs`**, **`TestSkills`**, **`TestSalaries`**, **`TestLocations`**, **`TestCategories`**) and asserts, for each endpoint, the HTTP status, the response shape, and the types of key fields. The salary endpoints are tested across all grouping dimensions (department, state, grade, series). The normalization package is validated separately through the evaluation harness against the

gold standard rather than through unit tests, which suits a component whose correctness is statistical rather than binary. Coverage is measured with **`pytest-cov`**.

4.9 Deployment and Configuration Management

The system is containerized and deployed entirely through GitHub Actions. The workflow has three sequential jobs. The first installs dependencies, runs **`flake8`** for linting, and runs the **`pytest`** suite; a failure here stops the pipeline. The second builds a Docker image and pushes it to the GitHub Container Registry, tagged with both **`latest`** and the commit SHA. The third copies the production Compose file and Nginx configuration to the GCP virtual machine over SCP, then connects over SSH to pull the new image, restart the Compose stack, restart Nginx, and prune old images.

```
- name: Deploy to GCP VM
  uses: appleboy/ssh-action@v1.0.3
  with:
    host: ${{ secrets.GCP_HOST }}
    username: ${{ secrets.GCP_USER }}
    key: ${{ secrets.GCP_SSH_KEY }}
    script: |
      cd /opt/vakansiograph
      docker compose -f docker-compose.prod.yml pull
      docker compose -f docker-compose.prod.yml up -d
      docker compose -f docker-compose.prod.yml restart nginx
      docker image prune -f
```

All secrets, API keys, host addresses, and SSH keys are stored as GitHub Actions secrets and injected at run time, never committed to the repository.

4.10 Frontend and Documentation

The frontend is a set of Jinja2 templates rendered by Flask with no client-side build step. JavaScript is vanilla and split into a shared utilities module and page-specific scripts, and Chart.js is bundled with the application rather than loaded from a CDN, so the dashboard works even when external network access is restricted. The pages cover an index with headline statistics, a trends page with skill and volume charts, a skills-intelligence page with category gap analysis, and a countries page with the per-country filter.

Documentation is maintained at three levels. The README documents setup and the environment variables each component requires. Inline docstrings document the public methods of services, mappers, and the evaluation module. The ESCO package's evaluation harness serves as living documentation of how the normalization strategies are intended to be compared and selected, which supports future maintenance when new strategies or a new ESCO version are introduced.

5 Validation

Contents

5.1 Approach

Validation for this project meant answering one question for each requirement: does the system actually do what it was designed to do? The work is organized around the requirements established in the Research chapter: API correctness, data collection and deduplication, ESCO normalization quality, caching behavior, performance under load, and API policy compliance. Each section states what was checked, how, and what the outcome was.

Where the project provides a dedicated mechanism for measurement, that mechanism is used. The REST API is covered by an automated test suite, the ESCO normalization package includes its own evaluation harness, and end-to-end performance under concurrent load was measured with a dedicated load-testing tool. Properties that do not lend themselves to a single number, such as deduplication and policy compliance, are verified by inspection and by observing the running system.

5.2 Functional API Testing

The automated test suite in **`tests/test_api.py`** covers the REST API. The tests are organized by resource into test classes, **`TestHealth`**, **`TestOverview`**, **`TestJobs`**, **`TestSkills`**, **`TestSalaries`**, **`TestLocations`**, and **`TestCategories`**. Each test sends a request to an endpoint and asserts the HTTP status code, the shape of the JSON response, and the types of the key fields. They run against an application built with the **`testing`** configuration and an isolated database created by a fixture in **`conftest.py`**, so they neither touch production data nor require network access, which lets them run in continuous integration.

Endpoint	What the test asserts	Result
`/api/health`	Returns status 200 with **`status`** and **`service`**	Pass
`/api/overview`	Returns totals, average salary, and unique-skill counts	Pass
`/api/jobs/recent`	Returns a list; respects **`limit`** and **`keyword`**	Pass
`/api/skills/top`	Returns skills with **`skill`**, **`type`**, **`count`**; respects **`limit`**	Pass
`/api/salaries`	Returns a list for each **`group_by`** dimension	Pass
`/api/locations`	Returns locations with counts and average salary	Pass
`/api/categories/summary`	Returns categories with coverage metadata	Pass

Table 3: API endpoint test coverage

All tests pass. The salary endpoint is exercised across every grouping dimension it supports, department, state, grade, and series, and the jobs and skills endpoints are tested both with and without their query parameters, confirming that filtering and limiting behave as expected.

Figure 5: The live Trends dashboard rendering data served by the tested API: skill-category share, the most in-demand skills, and a salary-versus-volume view by category

5.3 Data Collection and Deduplication

Deduplication is handled at the database level. Each provider's own identifier is used as the **`position_id`** primary key, and new records are written with an **`INSERT OR REPLACE`** statement. As a result, a posting that appears again in a later collection cycle does not create a second row; it overwrites the existing record in place, keeping its fields current

without growing the table. This was confirmed by running the collection pipeline more than once against the same source on the same day and observing that the row count in **`job_postings`** did not increase on the repeated run.

The pipeline is also designed so that a failure in one source does not abort the others. Each source is guarded by its own credentials, read from environment variables, and each collection step is wrapped so that an exception is logged and the cycle continues. This was verified by running the scheduler with one provider's credentials deliberately unset: the corresponding source was skipped with a warning, and the remaining sources collected normally.

5.4 ESCO Normalization Quality

The ESCO normalization package was evaluated using its own evaluation harness rather than by informal inspection. The harness scores a mapper's output against a gold-standard set of documents whose correct ESCO concepts were established by hand, and reports precision, recall, F1, the unmatched rate, the false-positive rate, and coverage, both per document and averaged across the set.

This harness is what made the comparison of the four mapping strategies rigorous. Each mapper, fuzzy, embedding, LLM, and the vLLM-optimized variant, was run over the same gold set, and the metrics were used to decide which strategy to rely on rather than choosing one by intuition. The string-based fuzzy mapper is fast and fully interpretable but misses semantically equivalent phrasings; the embedding mapper recovers many of those at the cost of some false positives on short, superficially similar strings; and the LLM-based mappers achieve the highest agreement with the gold standard at the cost of API calls and latency.

A consistent residue of skill strings remains unmatched by every strategy. On inspection these fall into two recognizable groups: highly specific U.S. regulatory, clearance, or agency terms that have no ESCO equivalent, and compound phrases whose individual components exist in ESCO but whose combination does not. Both are natural candidates for the manual mappings layer in a future iteration.

Metric reported by the harness	What it captures
Precision	Share of predicted ESCO concepts that are correct
Recall	Share of gold concepts that were found
F1	Harmonic mean of precision and recall
Unmapped rate	Share of skill strings left unmatched
False-positive rate	Share of predicted concepts not in the gold set
Coverage	Predicted concepts relative to the gold set size

Table 4: Metrics computed by the ESCO evaluation harness

5.5 Caching Behavior

The platform uses a three-layer caching strategy, and validation confirmed that each layer behaves as designed. The first layer is the HTTP response cache: each analytical endpoint

is served from Redis (with an automatic fallback to a filesystem cache when Redis is unreachable), with time-to-live values tuned to how expensive the content is to produce, ten minutes for most analytical endpoints and two hours for AI-generated content. The second layer is a precomputed **`analytics_cache`** table inside SQLite, populated by the analytics builder after each collection cycle, so that aggregations over the full dataset are computed once per cycle rather than once per request. The third layer is a periodic cache warmer that re-populates the response cache on an interval shorter than its TTL, so the cache does not go cold during quiet periods.

The correctness of the backend selection was confirmed by observation: with Redis reachable the application logs that it is using Redis, and with Redis stopped it logs the fallback to the filesystem cache and continues to serve requests identically. The effect of the precomputed and warmed layers is visible directly in the load-testing results below, where the median response time stays at a fraction of a second precisely because requests are served from cache rather than recomputed.

5.6 Performance Under Load

End-to-end performance under concurrent load was measured with Locust, an HTTP load-testing tool that simulates many concurrent users and reports latency percentiles, throughput, and error rate. The test simulated fifty concurrent users issuing requests to the cache-warmed, parameter-free analytical endpoints, the same traffic a real dashboard generates, against the production deployment on the Google Cloud Platform e2-medium virtual machine.

5.6.1 Tuning the Server

The first load test exposed a configuration problem rather than a hardware one. With the default settings, Gunicorn ran two workers with two threads each, giving only four concurrent request slots. Under fifty simulated users, requests queued behind those four slots and many exceeded the request timeout, producing a failure rate of roughly one in three. Two further factors compounded this: the collection worker was competing with the web server for the two CPU cores of the shared virtual machine, and the cache warmer was issuing its warm-up requests through the same Gunicorn it was running inside, saturating its own slots on startup.

Three changes addressed this. Gunicorn was reconfigured to raise concurrent capacity, the cache warmer was changed to run once rather than once per worker and to warm on a periodic schedule, and for the measurement runs the collection worker was stopped so that the test measured the web tier in isolation, which is the correct scope for an API load test.

A separate problem surfaced only after the server had been running for several hours: the load average on the two-core machine climbed steadily to around fifteen, and response times degraded to minutes, even though the request volume was negligible and no failures were logged. The cause was the worker class. Gunicorn was using threaded (**`gthread`**) workers, and these were busy-looping and accumulating CPU time even when idle, so the machine was saturated by the server doing nothing useful. Switching to synchronous workers and enabling periodic worker recycling (**`--max-requests`**, which restarts each

worker after a fixed number of requests) resolved it: idle CPU fell from nearly four hundred percent to under fifty, the load average dropped below one, and the gradual degradation over long uptimes disappeared because workers now recycle before anything can accumulate.

5.6.2 Results

After tuning, the system sustained fifty concurrent users with no failures. Over a run of more than nine thousand requests, the aggregate median response time was 200 milliseconds and the 95th percentile was 370 milliseconds, with a sustained throughput in the low tens of requests per second and a zero percent error rate. The contrast with the untuned configuration is stark.

Configuration	Median	95th pct	Errors	Outcome
Default: 4 slots, worker running	~8.6 s	~480 s	~32%	Saturated
Tuned: more slots, worker stopped	200 ms	370 ms	0%	Stable

Table 5: Load test under 50 concurrent users, before and after tuning

The per-endpoint breakdown confirms that the result is consistent across the API rather than driven by a few cheap endpoints. Every endpoint returned a median in the 190 to 210 millisecond range and a 95th percentile between 310 and 450 milliseconds, with no failures on any endpoint.

Endpoint	Median	95%ile	Avg	Fails
`/api/health`	190	310	521	0
`/api/overview`	200	400	762	0
`/api/insights/hero-stats`	200	330	452	0
`/api/skills/top`	200	350	502	0
`/api/salaries`	190	420	1176	0
`/api/locations`	200	430	1237	0
`/api/categories/summary`	200	350	964	0
`/api/trends/skill-roi`	190	360	978	0
`/api/trends/skill-demand`	200	340	493	0
`/api/trends/category-salary`	200	370	826	0
`/api/trends/countries`	210	440	1179	0
`/api/trends/posting-volume`	190	450	995	0
`/api/trends/insights`	200	360	775	0
Aggregated (9003 requests)	**200**	**370**	**832**	**0**

Table 6: Per-endpoint latency in milliseconds under 50 concurrent users (worker stopped)

The live charts from the run tell the same story visually. Throughput climbs steadily as users ramp up and the failure line stays flat on zero, while response times sit near the bottom of the chart for almost the entire run.

Figure 6: Locust live dashboard: requests per second, response-time percentiles, and active users

One feature of the charts is worth explaining honestly. The 95th-percentile line is near zero for almost the whole run but shows a single sharp spike to around fourteen seconds. This coincides with a cache-warmer cycle: when the warmer wakes up it issues a burst of warm-up requests that briefly occupy request slots, so the small number of user requests that arrive in that window wait longer. The spike affects roughly one percent of requests and the system recovers immediately afterward. It is a real artifact of warming the cache in-process, and a cleaner design would warm the cache out of band rather than through the serving tier.

5.7 API Policy Compliance

Compliance with each provider's terms was verified by inspecting the client code and observing the running system. Across all providers, credentials are read from environment variables at runtime and never stored in source code, written to logs, or included in error messages. USAJOBS requests carry the required authorization and user-agent headers and never exceed the documented page size, and requests are paced rather than issued as fast as possible. Adzuna requests carry the required application identifier and key, and a normal collection cycle issues far fewer requests than the published hourly limit. Jooble, which does not publish a specific limit, is paced conservatively with a capped number of pages per run.

5.8 Limitations

Several aspects of validation were intentionally left out of scope. The ESCO evaluation was performed against a gold-standard sample rather than the full collected dataset, since hand-labeling hundreds of thousands of postings is not feasible within a project of this

scope; the harness results are therefore a sound estimate of normalization quality rather than a guarantee over the entire corpus.

The load test characterized the web tier with the collection worker stopped, which is the correct scope for an API load test but means the measured numbers represent the serving tier in isolation rather than the system during an active collection cycle. Earlier observation showed that when the worker runs, it competes with the web server for the two CPU cores and the single SQLite writer of the shared virtual machine, which is the strongest argument for separating the collector and the API onto different hosts as the platform grows. The cache-warmer spike noted above is a related symptom of running background work inside the serving process.

Finally, API policy compliance was verified by inspection and observation rather than by an automated conformance test; adding instrumentation that asserts the presence of required headers and measures inter-request timing would make compliance continuously verifiable and is a reasonable direction for future work.

6 Conclusion

6.1 Project Summary

As a result, we have a platform capable of fetching job postings from three official APIs, normalizing extracted skills according to the ESCO taxonomy, and performing analytics covering salaries, skill demand, geographic distribution, and AI-generated market overviews. In two months of operation, it has collected and indexed over 469,000 job postings from more than forty countries and 68,000 companies. The platform runs 24 hours a day, collecting data every two hours, and is automatically deployed on a Google Cloud Platform virtual machine.

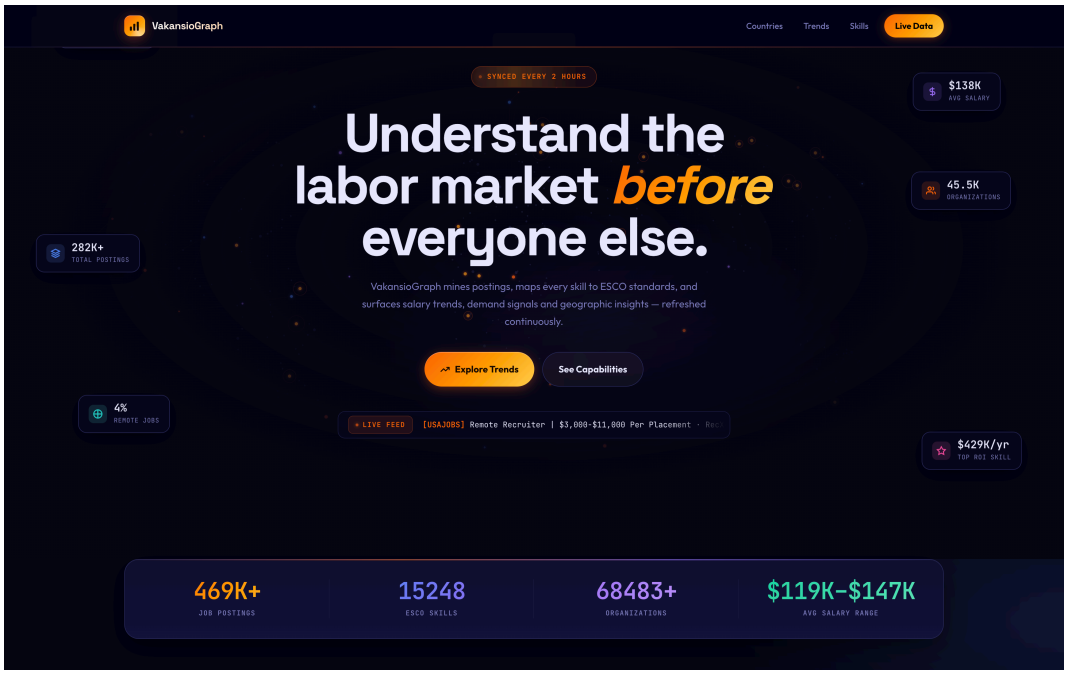

Figure 7: The VakansioGraph landing page, summarizing the live dataset: postings collected, skills mapped, organizations tracked, and the salary range across the corpus

The methodology was pipeline-first: the collection and normalization stages were built and validated before anything was built on top of them, on the principle that a dashboard is only as trustworthy as the data beneath it. The result is a single, reproducible system that combines continuous multi-source collection, taxonomy-based skill normalization, and a usable API, the combination that the survey in the Research chapter found to be missing from the existing landscape.

6.2 Alignment with Objectives

The interface does not require the user to have any knowledge of APIs. Regarding non-functional aspects: data collection uses only documented public APIs within their rate limits, deployment is performed from a single repository, and the codebase includes automated tests for key endpoints.

6.3 Lessons Learned and Challenges

The first lesson concerned concurrency. In the initial deployment, Gunicorn was running with only four slots for concurrent requests. Under normal usage, this went unnoticed, but the system failed a load test with fifty users, resulting in failures in about one out of every three cases. To diagnose the problem, we had to untangle three interrelated causes: too few slots for worker processes, a data collection worker process competing with the web server for two CPU cores on the shared machine, and a cache warm-up program that sent its warm-up requests through the very server it was trying to warm up. After adjusting the process configuration and isolating the warm-up behavior, the same workload of fifty users ran with an average response time of 200 milliseconds and without any crashes.

The second lesson concerned caching. The cache warming program ran only once during startup, so after the TTL expired during a period of inactivity, the cache was cleared, and the next visitor paid the full price for requests involving the aggregation of cold data. The problem was solved by periodically running the program at intervals shorter than the TTL.

The third lesson, the most surprising one, concerned the web server itself. After several hours of operation, the dual-core machine reached an average load of about fifteen, and the response time increased to several minutes—even in the absence of traffic and errors in the logs. The cause of this was not the application logic. The cause was the choice of Gunicorn's worker class: multithreaded workers were performing idle loops and consuming CPU time while idle. Switching to synchronous workers with periodic restarts reduced idle CPU usage by about eight times and completely eliminated the performance degradation. A default setting that seemed harmless turned out, over time, to significantly impact the system's behavior. We were only able to detect this through long-term monitoring. A short load test would not have revealed this at all.

All three cases boiled down to the same thing: performance issues that actually matter only become apparent under concurrent load and over time. You can't detect them by reading the code. You have to take measurements.

6.4 Limitations

The most significant limitation is the data storage capacity. SQLite was the right choice for deployment on a single virtual machine with high read volumes and the ability to reproduce data, but it supports only a single user writing data and will become a bottleneck if the dataset size or write frequency increases tenfold. The deployment is also a single instance with no load balancer or automatic failover. This fits the scale and budget of this project, but means there is no redundancy.

On a single shared virtual machine, the data collection process and the web server compete for two CPU cores and a single SQLite writer. Therefore, the load test was conducted with the data collection process stopped, which accurately measures the service level but does not capture behavior during an active data collection cycle.

6.5 Future Work

The most obvious step is to host the data collection module on a separate server, distinct from the web server. This will resolve the resource conflicts identified during testing and allow each tier to scale independently. Switching from SQLite to PostgreSQL will remove the single-writer limitation. The database utility layer has been deliberately kept to a minimum to ensure minimal impact from this change.

Administrative endpoints require authentication, and the API must be versioned if it is ever opened to external users. Validation could also be strengthened: an automated test for API policy compliance and a load test run during the active data collection cycle would transform properties that are currently checked manually into ones that are measured automatically.

The main finding remains valid even without them: using only public data and tools, it is possible to create labor market analytics that were previously available only through a commercial subscription.

Glossary

Domain

Embedding matching – **Embedding-based matching**: A matching technique that represents text as numeric vectors and compares them by cosine similarity, capturing semantic similarity that string matching misses.

ESCO – **European Skills, Competences, Qualifications and Occupations**: An open, multilingual classification of skills and occupations maintained by the European Commission. It covers nearly 14,000 skill concepts, each with a stable URI. In this project it is the normalization target for skills extracted from job postings.

Fuzzy matching – **Fuzzy string matching**: A matching technique that scores how similar two strings are, tolerating typos and minor wording differences. Used as the fast, interpretable baseline mapping strategy.

JOLTS – **Job Openings and Labor Turnover Survey**: A monthly labor statistics release from the U.S. Bureau of Labor Statistics, cited in the research chapter as an authoritative but low-frequency data source.

Skill normalization – **Skill normalization**: The process of mapping the many free-text ways a skill is written in job postings to a single standardized concept in a taxonomy such as ESCO, so that skills can be counted and compared consistently across sources and countries.

OPM – **U.S. Office of Personnel Management**: The U.S. federal agency that publishes USAJOBS and defines the occupational series codes used to categorize federal vacancies.

Project

VakansioGraph – **VakansioGraph**: The labor market monitoring platform developed in this thesis. It collects job postings from open APIs, normalizes skills against ESCO, and exposes analytics through a REST API and dashboard.

Technical

API – **Application Programming Interface**: A documented interface that lets one program request data or services from another. All external data in this project is collected through official public APIs.

C4 model – **C4 model**: A way of describing software architecture at four levels of zoom (Context, Container, Component, Code). This thesis uses the first three levels to document the system architecture.

CI/CD – **Continuous Integration and Continuous Deployment**: An automated pipeline that tests, builds, and deploys code on every change. In this project it is implemented with GitHub Actions and deploys to a Google Cloud Platform virtual machine.

CSV – **Comma-Separated Values**: A plain-text tabular format. The ESCO taxonomy is loaded from local CSV files at startup.

LLM – **Large Language Model**: A neural language model used here as one of the four skill-mapping strategies, matching free-text skill phrases to ESCO concepts.

REST – **Representational State Transfer**: An architectural style for web APIs in which resources are addressed by URLs and returned, in this project, as JSON over HTTP.

TTL – **Time To Live**: The interval for which a cached response remains valid before it is recomputed. Values range from ten minutes for most endpoints to two hours for AI-generated content.

URI – **Uniform Resource Identifier**: A stable string identifier. Every ESCO concept has a URI, which keeps normalized records consistent across taxonomy versions.

WAL – **Write-Ahead Logging**: A SQLite journaling mode that allows reads and writes to proceed concurrently without locking the whole database, which suits the read-heavy access pattern of this project.

WSGI – **Web Server Gateway Interface**: The standard interface between a Python web application and a web server. Gunicorn is the WSGI server used to run the Flask application in production.

Bibliography

[1] U.S. Bureau of Labor Statistics, "Job Openings and Labor Turnover Survey (JOLTS)." Accessed: May 01, 2026. [Online]. Available: https://www.bls.gov/jlt/

[2] U.S. Bureau of Labor Statistics, "Occupational Employment and Wage Statistics (OEWS)." Accessed: May 01, 2026. [Online]. Available: https://www.bls.gov/oes/

[3] European Commission, Eurostat, "Eurostat: Labour Market Statistics." Accessed: May 01, 2026. [Online]. Available: https://ec.europa.eu/eurostat/web/lfs

[4] U.S. Office of Personnel Management, "USAJOBS Developer API." Accessed: May 01, 2026. [Online]. Available: https://developer.usajobs.gov/

[5] Adzuna Ltd., "Adzuna Developer API." Accessed: May 01, 2026. [Online]. Available: https://developer.adzuna.com/

[6] Jooble, "Jooble API for Partners." Accessed: May 01, 2026. [Online]. Available: https://jooble.org/api/about

[7] LinkedIn Corporation, "LinkedIn Talent Insights." Accessed: May 01, 2026. [Online]. Available: https://business.linkedin.com/talent-solutions/talent-insights

[8] Lightcast, "Lightcast Labor Market Analytics." Accessed: May 01, 2026. [Online]. Available: https://lightcast.io/

[9] Revelio Labs, "Revelio Labs: Workforce Intelligence." Accessed: May 01, 2026. [Online]. Available: https://www.reveliolabs.com/

[10] U.S. Department of Labor, Employment and Training Administration, "O*NET Online: Occupational Information Network." Accessed: May 01, 2026. [Online]. Available: https://www.onetonline.org/

[11] SkillNER contributors, "SkillNER: An NLP Library for Extracting Skills from Job Descriptions." Accessed: May 01, 2026. [Online]. Available: https://github.com/AnasAito/SkillNER

[12] M. Zhang, K. N. Jensen, R. van der Goot, and B. Plank, "Skill Extraction from Job Postings using Weak Supervision," in *Proceedings of the Workshop on Natural Language Processing for Human Resources (NLP4HR)*, 2022. [Online]. Available: https://arxiv.org/abs/2209.08071

[13] J.-J. Decorte, J. Van Hautte, T. Demeester, and C. Develder, "Jobbert: Understanding Job Titles through Skills," in *Proceedings of the International Conference on Data Mining Workshops (ICDMW)*, 2021. doi: 10.1109/ICDMW53433.2021.00076.

[14] B. Clavié and G. Soulé, "Large Language Models as Batteries-Included Zero-Shot ESCO Skills Matchers," in *Proceedings of the Workshop on Natural Language Processing for Human Resources (NLP4HR)*, 2023. [Online]. Available: https://arxiv.org/abs/2307.03539

[15] M. Bachmann, "RapidFuzz: Rapid fuzzy string matching in Python." Accessed: May 01, 2026. [Online]. Available: https://github.com/rapidfuzz/RapidFuzz

[16] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2019. doi: 10.18653/v1/D19-1410.

A Appendix

This appendix collects representative screenshots of the running platform that illustrate the analytical views described in the body of the thesis. All views are rendered live from the REST API against the production dataset.

AA Market Trends

The Trends view presents market-wide intelligence: headline counts, the dominant skill category, posting volume over time, and AI-generated summary cards derived from the precomputed analytics.

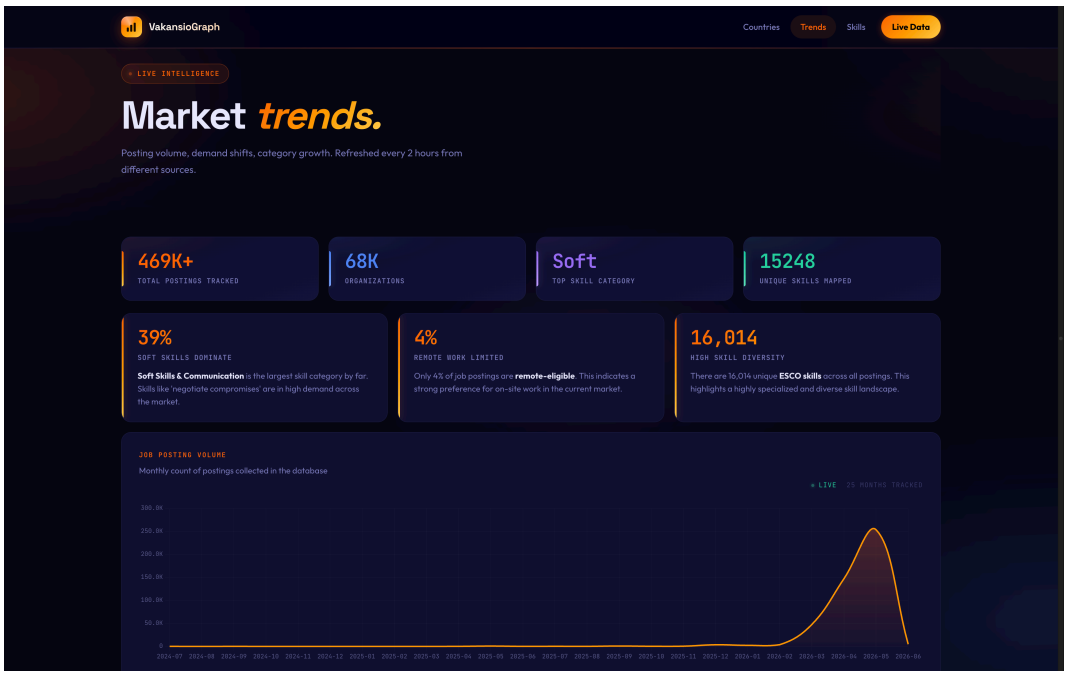

Figure 8: Market Trends: headline metrics, AI-generated insight cards, and monthly posting volume

AB Skills Intelligence

The Skills Intelligence view classifies every extracted requirement by its ESCO category and supports a skill-gap drill-down per career field, alongside salary-ROI comparisons by skill and by category.

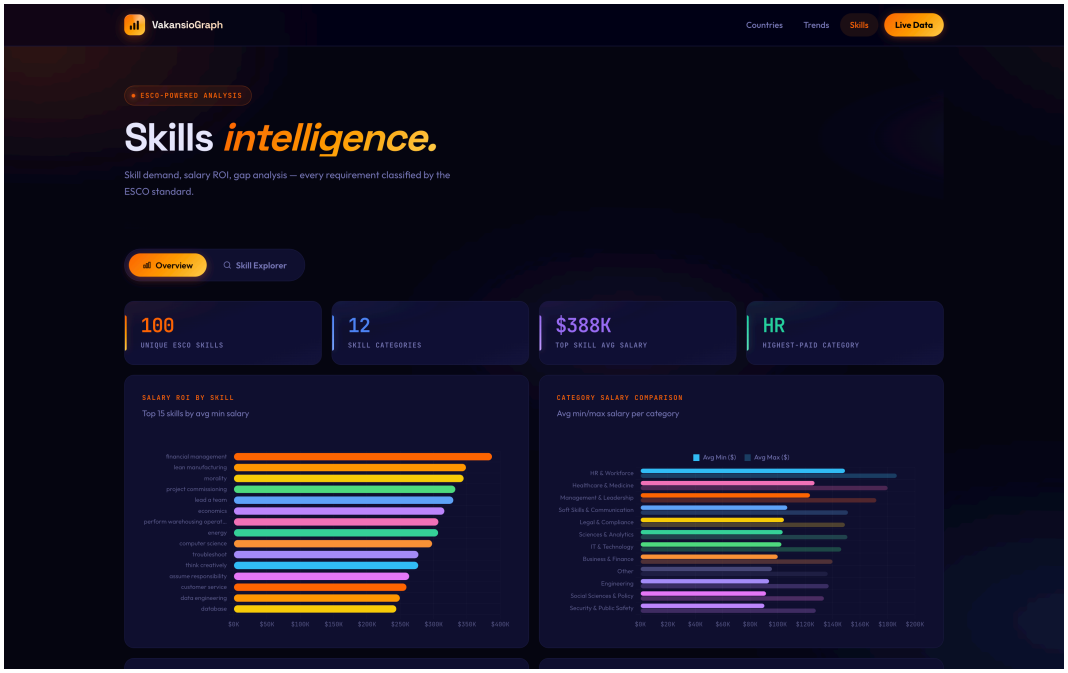

Figure 9: Skills Intelligence overview: salary ROI by skill and average salary comparison by category

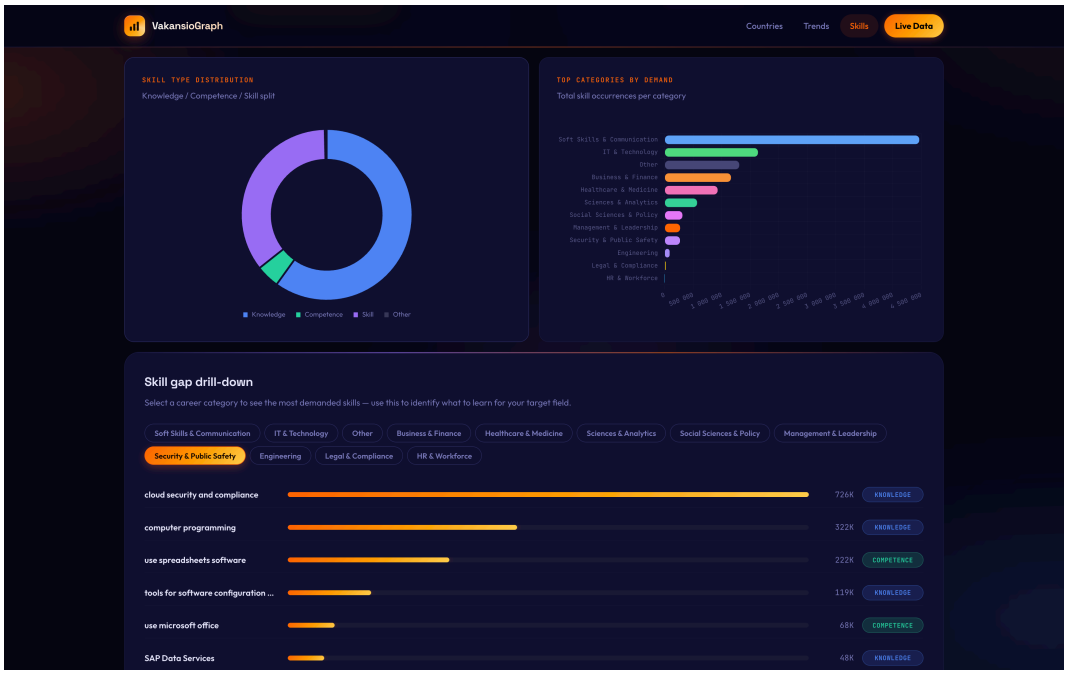

Figure 10: Skill-type distribution and a per-category skill-gap drill-down showing the most demanded skills in a chosen field

AC Per-Skill Detail

Searching for an individual skill resolves it to its ESCO concept and returns a focused view: demand over time, the leading hiring countries, salary range, and related skills in the same ESCO category.

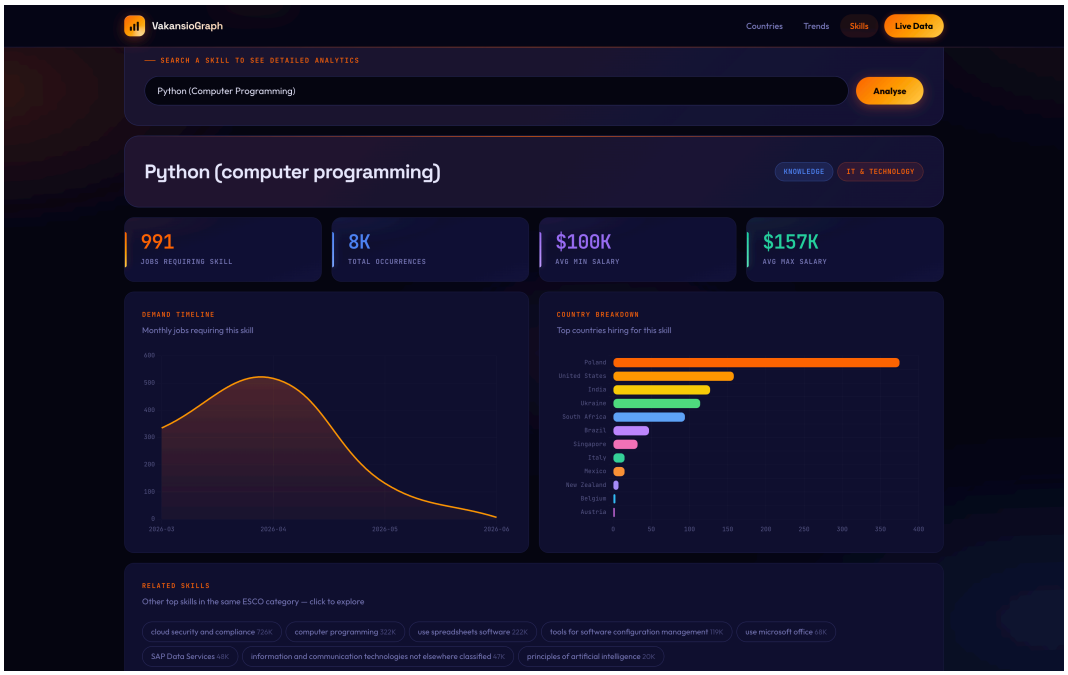

Figure 11: Per-skill analytics for “Python (computer programming)”: demand timeline, country breakdown, salary range, and related ESCO skills

AD Country View

The Countries view filters the entire dataset to a single country, here Ukraine, and recomputes posting volume, the most in-demand skills, and the ESCO category distribution for that region.

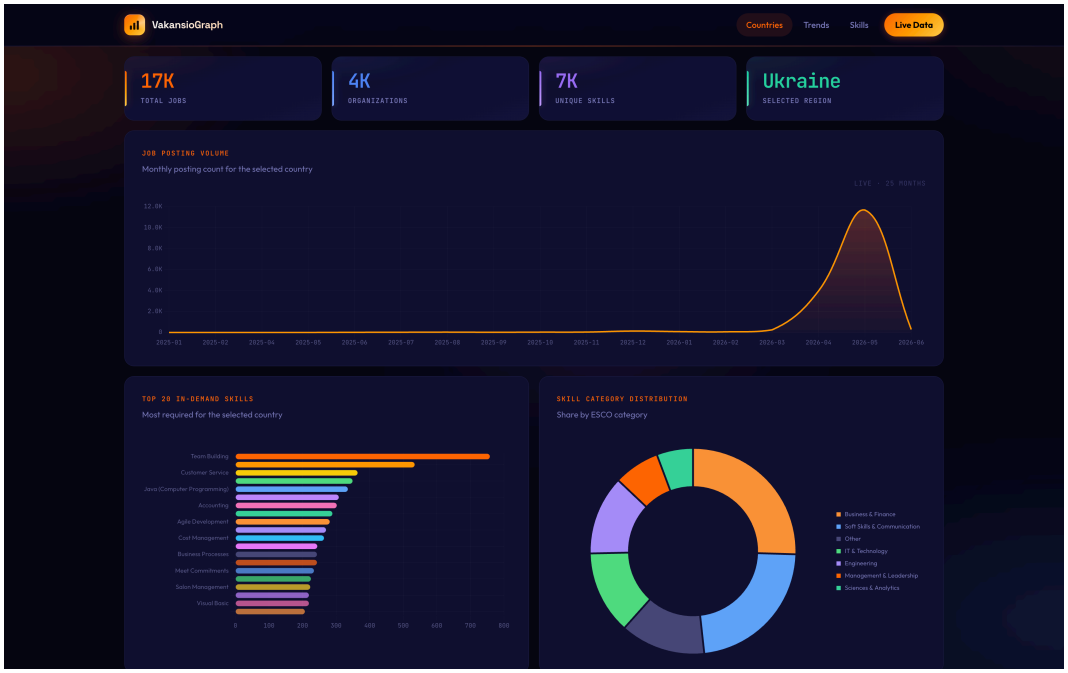

Figure 12: Country view for Ukraine: monthly posting volume, top in-demand skills, and skill-category distribution