KSE | Kyiv School of Economics

Computer Science Department

Software Engineering & Business Analysis

Bachelor's Thesis

Vakansio

A Personalized Job-Matching Service

Yan Khadnevich

Supervisor
KSE, Volodymyr Skochko, vskochko@kse.org.ua

Submission date
16 June 2026

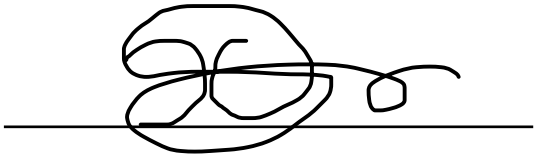
Academic Integrity Statement

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, 16.06.2026

Signature 

Contents

Acknowledgements

I would like to express my gratitude to my supervisor, **Volodymyr Skochko**, whose methodological guidance shaped both the architectural choices in this project and the way I framed the evaluation chapter. Your feedback turned several rough drafts into the structure the thesis stands on today, and the methodology pivot reported in Section 5 would not have landed cleanly without it. Many thanks for your time and your patience throughout the supervision process.

I also owe a great deal to several lecturers at the Kyiv School of Economics whose teaching shaped what I was able to build.

Artem Korotenko, the Academic Director, first showed me what programming actually is. Long before I could think about Clean Architecture or SOLID, his courses gave me the intuition for what a well-structured program even looks like. The discipline that the Domain layer of Vakansio enforces, with no external dependencies and every collaborator behind an interface, is the same discipline I first encountered in his classroom.

Ihor Pysmennyi taught me how production systems actually run. His DevOps course was the first place I saw containers, reverse proxies, and managed cloud services treated as concrete engineering choices rather than background magic. The Caddy-on-EC2 deployment that hosts Vakansio today is a direct descendant of the patterns he walked us through.

Dmytro Nomirovskiy ran the toughest mathematics courses of my degree. The classes were not easy, but the habit of working through a problem until the proof actually closes is something I have carried into every layer of this project, including the statistical inference that backs Section 5.

Finally, **Vadym Yaremenko** delivered the clearest lectures I had on programming paradigms, networking, and C++. The instinct to reach for the simpler abstraction when a problem feels tangled, and to treat the wire format as a first-class contract, came from his classes and shows up everywhere from the layer interfaces in Domain to the v6.7.8 contract drift fix described in Section 4.

Abstract

Looking for an IT job in Ukraine means visiting several websites. Each one has its own listings, format, and filters. For every vacancy a candidate has to decide alone whether the role fits. Every candidate repeats this work on every search, spending tens of hours of reading while fatigue degrades the choices made late in the session.

This thesis presents **Vakansio**, a deployed job-matching service for candidates. The system pulls vacancies from seven sources, turns them and the uploaded CV into structured data, and scores every (CV, vacancy) pair. The score comes from a deterministic seven-axis calculator, refined by a Composite LLM Judge, and is shown with bilingual English and Ukrainian explanations. The back-end runs on .NET 8 with Clean Architecture and SOLID; the front-end is React and TypeScript; the deployment is on AWS. The service is live on the open internet.

I propose a six-layer evaluation: CV normalisation, vacancy normalisation, score, reason text, ranking, and a held-out validation on a fresh-pair, fresh-vacancy gold set. Each layer is graded by the method that fits its data type, following the Holistic Evaluation of Language Models framework. The reason-text layer shows a significant win for the v6 prompt over the v4 baseline (overall $\Delta = +0.30$, 95% paired-bootstrap CI $[+0.19, +0.40]$, $N \approx 391$ pairs, judge: Claude Opus 4.7). The ranking layer is inconclusive on 14 CVs (Δ NDCG@10 $= -0.047$, 95% CI $[-0.168, +0.065]$, judge: Claude Sonnet 4.6). The held-out layer, on 398 pairs the prompt never saw, reaches Spearman $\rho = 0.648$ (95% CI $[0.580, 0.712]$) and NDCG@5 = 0.822; a controlled cap-on/off ablation justifies the production safety caps, and a post-hoc isotonic calibrator cuts Expected Calibration Error from 0.140 to 0.027. The methodology stands regardless of the empirical outcome, and the conditions that would resolve the open questions are stated explicitly.

Keywords:

Job Matching, Large Language Models, Clean Architecture, .NET 8, Recommender Systems, LLM-as-Judge

1 Introduction

1.1 Initial situation and motivation

The Ukrainian IT job market is split across half a dozen websites: Djinni, DOU, Robota.ua, Work.ua, LinkedIn, Jooble. Each has its own catalogue, account, and search interface. None of them ranks an externally uploaded CV against listings from multiple sites with a human-readable explanation. The default order is by date. The filters are coarse. Where a personalised signal exists, it works on the site's on-platform profile, not on the CV the candidate uploads. A candidate has to repeat the same query on every site and read every posting one by one.

I went through this myself. A careful read of one vacancy takes three to five minutes, plus a minute or two to decide whether to apply. One evening covers twenty to thirty postings. A full search takes thirty to fifty hours of reading before the first application. The hidden cost is cognitive: every vacancy switches the context — different company, stack, seniority, sometimes country. The resulting decision fatigue (the canonical evidence [1] is contested in subsequent work, but the direction of the effect on cognitive context-switching is the relevant observation here) makes the judgement worse late in the session. I missed good matches and pursued bad ones. The emotional cost compounds the first two: a long search without visible progress reduces motivation, and the absence of replies casts doubt on the candidate's own qualifications. A tool that reads each vacancy, compares it against the CV, and surfaces the most promising listings first would cut the largest of these costs. It turns the search from an endurance task into a tractable one. This is what Vakansio does for the candidate. The same scoring engine powers the recruiter-side workflow, where the held-out evaluation in Section 5.7 reports its strongest results.

1.2 Objectives

Vakansio is built around four explicit objectives, each restated as a measurable goal in a later chapter:

1. **A working public service.** The system is reachable on the open internet, supports user registration and login, accepts a CV as a PDF upload, searches for vacancies from several sources, and returns a personalised list with verdicts and bilingual explanations. The deployed instance is at dsus1dizgh006.cloudfront.net.
2. **A disciplined software architecture.** The code base follows the Clean Architecture pattern of Robert C. Martin [2] and the five SOLID principles, with layer separation visible in the project-reference graph and each principle traceable to a concrete artefact.
3. **Measurable match quality.** The evaluation methodology can be rerun by another engineer. It decomposes the pipeline into independent layers, each graded by the technique appropriate to its data type, and reports the empirical results honestly — including where they are inconclusive.

4. **Cost transparency for every LLM call.** Each call is logged with its pipeline stage, input and output token counts, wall-clock duration, and dollar cost. The data lives in a PostgreSQL table and is queryable by an operator.

Out of scope: employer-side vacancy posting and candidate search, multi-tenant organisation accounts, and an external user study with real candidates. The evaluation is internal, against the gold sets described in Section 5.

1.3 Method

The back-end is a .NET 8 solution split into four projects — Domain, Application, Infrastructure, API — using CQRS through MediatR. The Application layer declares its needs as interfaces. The Infrastructure layer provides the implementations: Entity Framework Core on PostgreSQL, Gemini 2.5 Flash, AWS S3 and SSM Parameter Store, and seven job-source adapters.

The front-end is a React + TypeScript single-page application bundled by Vite. Zustand holds cross-cutting UI state. TanStack Query holds server data.

The scoring approach is hybrid. A deterministic seven-axis sub-scorer gives the first numeric estimate of fit. A Gemini-based Composite Judge refines it against a per-domain rubric. A batched LLM call generates bilingual English and Ukrainian explanations for the top results.

The deployment is a single EC2 t3.micro instance behind Caddy in eu-central-1, with managed PostgreSQL on RDS, S3 for CV files, and CloudFront for the front-end bundle. I developed in versioned releases of the scoring pipeline rather than fixed-length sprints. The methodology and coding standards are in Section 4.

1.4 Structure of this thesis

- Section 2 surveys the Ukrainian job market, the legal and technical limits on data collection, the related work in recommender systems and LLM evaluation, and the functional and non-functional requirements.
- Section 3 covers the architectural drivers, the technology stack, the Clean Architecture decomposition with the SOLID mapping, the scoring pipeline, the cache hierarchy, the front-end design, the deployment topology, and the security and observability responses.
- Section 4 walks through the Domain, Application, Infrastructure, and front-end implementations, the deployment automation, and the engineering stories that shaped the architecture.
- Section 5 details the six-layer evaluation methodology, the held-out validation on a fresh-pair gold set, and the threats to validity.
- Section 6 revisits the objectives, lists the encountered difficulties, and outlines future work.

2 Research

2.1 The Ukrainian job-search landscape

Before I started designing Vakansio, I looked carefully at the Ukrainian IT market and the way candidates search for jobs in it. The market is large enough that a manual search is time-consuming. At any moment, tens of thousands of open positions are visible across Ukrainian job sites, with new vacancies posted daily. Candidates check several sites in parallel because no single source covers the whole market.

The landscape is fragmented. Different sites serve different audiences:

- Some focus on IT and product roles only.
- Some are general-purpose and cover every industry.
- Some are international networks with a Ukrainian segment.
- One is a meta-aggregator that re-publishes listings from other boards.

Each site stores its own catalogue, requires its own account, and presents listings in its own format. A candidate cannot search across all sources from one place. They have to repeat the same query on each site separately.

Inside each site, the ranking is rarely personalised to an externally uploaded CV. The default order is by posting date. The filters are coarse — location, seniority, salary range, sometimes a fixed set of technology tags. Where a personalised signal exists, it works on the candidate's on-platform profile, not on the CV they uploaded, and it comes without an explanation.

These two problems — **split sources** and **no CV-based ranking with explanation** — define the gap Vakansio fills. No mainstream Ukrainian platform today aggregates vacancies from multiple sources and ranks them against an uploaded CV with a human-readable explanation.

2.2 Existing job sites and what they offer

Table 1 profiles the six platforms a Ukrainian IT candidate is most likely to use, comparing their coverage, ranking signal, and explanation surface. Several smaller boards (Grc.ua, Happy Monday, Lobby.ua, Jobs.ua) follow the same general pattern and are not profiled separately.

Site	Coverage	Default order	Personalisation signal	Explanation
Djinni.co	IT only, ~50k monthly developers	By date	On-platform profile (stack, seniority, salary)	None
DOU.ua	IT-leaning, employer-written, public RSS feed	By date	None	None
Robota.ua	All industries, ~400k seekers, Pracuj.pl network	Date + filters	On-platform AI Job Search Agent (2024–2025)	Suggestion only, no rationale
Work.ua	All industries, ~3.5M CVs, largest UA reach	Keyword + filters	None for externally uploaded CV	None
LinkedIn	International, ~300 ranking signals	Profile-driven	LinkedIn profile + network engagement	None
Jooble	Meta-aggregator, 15k+ public sources, link-out	Inherited from source	None	None

Table 1: Major Ukrainian-relevant job sites. None ranks an externally uploaded CV against multi-source listings with a human-readable explanation — the gap Vakansio fills, restated as a requirement in Section 2.6.

The pattern is uniform across all six: keyword search, date ordering, coarse filters. The two specific gaps named in Section 2.1 — **split sources** and **no CV-based ranking with explanation** — show up on every surveyed platform.

2.3 Legal and technical constraints on data collection

Pulling from multiple job sources raises two questions before any commercial launch: is the data collection legally defensible, and is the channel sustainable in production?

The main civil risk for a Ukrainian aggregator is the sui generis database right. The 2022 Law on Copyright and Related Rights (No. 2811-IX) introduces this right, and the Court of Justice clarified it for job listings in CV-Online Latvia v. Melons [3]: re-utilising a job-board database is actionable only when it adversely affects the maker's investment. Recent injunctions against hiQ Labs and Proxycurl confirm that civil liability for scraping mature platforms is real in practice, even where criminal liability is not.

CV files trigger a separate basis — personal data under the Ukrainian Law on the Protection of Personal Data and, where applicable, the GDPR. Vakansio handles them through explicit consent at upload, scope limitation (CVs are used only for matching), and deletion on account closure.

The seven sources Vakansio currently uses have different statuses with respect to commercial use. Table 2 summarises each one, together with the channel that a commercial version would have to adopt.

Source	Status for commercial use	Channel for product
DOU.ua	Permitted via official RSS feed	Keep current RSS channel
Jooble	Permitted via official REST API	Keep current API channel
Djinni.co	Negotiable through partnership	Partner API via direct outreach
Robota.ua	Restricted; aggregator access disallowed	Business-development partnership
Work.ua	Restricted by terms of service	Partnership through partner contacts
LinkedIn	Prohibited; high legal risk	Paid licensed data provider
Manual URL	User-driven, no policy issue	Keep as is

Table 2: Per-source status for commercial use and the channel a market version would adopt.

A commercial version of Vakansio would:

- Keep the two free legitimate channels (DOU RSS, Jooble API).
- Add an OLX Partner API channel and an ATS-aggregation channel over Greenhouse, Lever, Workable, and SmartRecruiters.
- Replace direct scraping of Djinni, Robota.ua, and Work.ua with partnership-based access.
- Replace LinkedIn scraping with a licensed paid provider — Coresignal at ~$1,000 per month for the cleanest legal basis, or Apify / Bright Data at $100–300 per month for an MVP.

The capstone prototype in this thesis uses the public-page scraping channels described above. The full per-precedent legal analysis, the per-provider pricing, and the channel-mix rationale are in Section B.

2.4 Related work in recommender systems and language-model evaluation

Adomavicius and Tuzhilin [4] distinguish three families of recommender system:

- **Content-based** — match an item against facts the user has shared about themselves.
- **Collaborative filtering** — recommend what similar users picked.
- **Hybrid** — combine both.

Vakansio is content-based, for three reasons. A first-session candidate has no past behaviour, so a collaborative filter has nothing to learn from. CVs and vacancies both contain structured fields, so attribute matching is straightforward. The candidate expects an explanation that names attributes from their own CV, which content-based recommendation gives for free.

Ranking quality is measured by Normalised Discounted Cumulative Gain (NDCG@10) [5]. NDCG@10 weights relevance higher at the top of the list, lower at the bottom. I compute NDCG@10 in a separate offline analysis (Section 5), not in the production scoring pipeline.

Two lines of language-model evaluation work shape the methodology in Section 5.

HELM (Holistic Evaluation of Language Models, Liang et al. [6]) evaluates a model on many separate metrics — accuracy, robustness, calibration, fairness — instead of one single score. Strengths and weaknesses become visible per axis. Vakansio borrows this multi-axis idea: every pipeline stage gets its own metric.

LLM-as-Judge (Zheng et al. [7] on MT-Bench and Chatbot Arena) uses a strong language model to evaluate another model's output. The empirical argument: strong-model judges agree with human annotators more than 80% of the time, so they are a cheaper substitute when the evaluation set is too large for hand rating. Vakansio uses LLM-as-Judge in two places — a Composite Judge step inside the scoring pipeline (Section 3.4), and an Anthropic Claude judge against the Gemini system in the evaluation (Section 5).

2.5 Gaps and positioning

The product-side gap is specific: no Ukrainian job site ranks an externally uploaded CV against multi-source listings with a human-readable explanation. Where partial elements exist — Robota.ua's AI Job Search Agent over its on-platform profile, LinkedIn's profile-and-engagement ranking — they are tied to one source's internal representation of the candidate, and they come without an explanation.

Vakansio fills this gap by aggregating from seven sources, ranking against the uploaded CV, and surfacing bilingual reason text.

The methodological contribution is the combination of three ideas:

- Content-based recommendation.
- LLM-as-Judge refinement of the deterministic score.
- Multi-axis HELM-style evaluation.

Applied to the Ukrainian market, with the empirical results in Section 5. None of the three ingredients is novel on its own. The combination — together with the held-out validation that retires the selection-circularity threat (Section 5.7) — is the contribution.

2.6 Functional and non-functional requirements

The gap motivates a specific set of requirements: seven functional requirements (FR1–FR7) describing what the system must do as user-testable behaviour, and seven non-functional requirements (NFR1–NFR7) describing how well it must do it once deployed.

Functional requirements. The system must:

1. **FR1 — Account management.** Allow a candidate to register an account and log in.
2. **FR2 — CV upload and parsing.** Accept a CV as a PDF upload and parse it into a structured representation.
3. **FR3 — Multi-source search.** Search for vacancies from several external sources in response to a keyword query.
4. **FR4 — Personalised ranking.** Return a ranked list of vacancies, ordered by the system's estimate of fit between the uploaded CV and each vacancy.
5. **FR5 — Verdict and explanation.** Show, for each returned vacancy, a numerical match score between zero and one, a verdict label (Strong, Partial, Weak, or Mismatch),

the matched skills, the missing must-have skills, and a short bilingual explanation in English and Ukrainian.

6. **FR6 — Filtering.** Support filtering of returned results by source, seniority level, and work format.
7. **FR7 — Application tracker.** Allow a candidate to save vacancies of interest to a personal application tracker and to update their application status.

Non-functional requirements. The system must satisfy the following properties:

1. **NFR1 — Public deployment.** The system is reachable on the open Internet at a stable URL, served over HTTPS with a valid certificate, without manual intervention between user sessions.
2. **NFR2 — Architectural discipline.** The back-end follows the four-project Clean Architecture decomposition with the dependency rule enforced at build time, and each of the five SOLID principles maps to a concrete artefact in the code base.
3. **NFR3 — Per-stage cost transparency.** Every external model call is recorded with its pipeline stage, input and output token counts, wall-clock duration, and dollar cost, in a relational table that an operator can query.
4. **NFR4 — Graceful degradation.** No single external dependency — a job source, the language-model API, the database — may fail the entire user-facing request. The system returns a partial result with a typed indicator when a stage exceeds its budget.
5. **NFR5 — Data protection.** User accounts and uploaded CV files are processed under explicit consent, transmitted over enforced TLS, encrypted at rest, and deleted on account closure. Production secrets live outside the source tree.
6. **NFR6 — Reproducible evaluation.** The evaluation harness ships with the code base, runs offline against versioned gold sets, and shares the production interfaces, so a third party can rerun the layered methodology from Section 5 without access to author state.
7. **NFR7 — Test coverage and continuous integration.** The back-end has three xUnit test projects:

   - **`Domain.Tests`** — pure scoring rules.
   - **`Application.Tests`** — MediatR behaviours and use-case logic.
   - **`Infrastructure.Tests`** — sub-score calculators, prompt builders, role routing, anti-flag evaluation.

   Coverlet measures code coverage. Every push to **`main`** triggers the **`ci.yml`** workflow, which runs the full test suite. If any test fails, the API and the front-end do not get deployed — the broken version never reaches production. The CI/CD pipeline is described in Section 4.6.

3 Design

3.1 Architectural drivers

Every major design decision traces back to one or more requirements from Section 2.6. Five drivers shape the architecture.

Driver 1 — multi-source aggregation, CV-driven ranking, human-readable explanation. The system needs pluggable source adapters so each job board can be added or replaced on its own. One centralised scoring pipeline handles every (CV, vacancy) pair. The output carries the score, the verdict, the matched skills, the missing must-haves, and the bilingual reason text. Covered in Section 3.2, Section 3.3, and Section 3.4.

Driver 2 — production deployment at predictable cost. The system has to stay reachable on the open Internet without manual intervention. This drives container-based packaging, managed services for the database and object storage, a reverse proxy that handles TLS automatically, and a static-hosting model for the front-end. See Section 3.7.

Driver 3 — measurable layer separation. SOLID compliance has to be visible in the code, not just claimed. The architecture is a strict four-project split (Domain, Application, Infrastructure, API). Interfaces live in the layer that uses them. Commands and queries are separated so every handler is single-purpose. See Section 3.3.

Driver 4 — per-stage evaluability and cost transparency. Every pipeline stage produces a measurement and a cost record. This drives:

- A modular pipeline whose stages can be evaluated on their own.
- A request-scoped accumulator that gathers per-stage cost without coupling the layers.
- A persistent cost-ledger table for offline querying.
- An offline evaluation harness separate from the production code path.

See Section 3.4 and Section 3.8.

Driver 5 — security and data protection. CV files and user accounts are subject to the Ukrainian Law on the Protection of Personal Data and, where applicable, the GDPR. The architectural responses are:

- Stateless authentication based on JSON Web Tokens.
- Secrets in AWS SSM Parameter Store rather than in the source tree.
- Startup guards that refuse to start without an enforced SSL connection string and a non-empty CORS allow-list.
- A cascade-delete pathway for full account removal.

See Section 3.8.

The sections below turn each driver into a concrete architectural choice.

3.2 Technology stack selection

Table 3 summarises the major technology choices behind Vakansio, the strongest alternative considered for each, and the dominant driver behind the decision. The per-alternative pro-and-con reasoning is in Section C.

Layer	Choice	Strongest alternative	Dominant driver
Back-end runtime	.NET 8 with C#	Node.js + TypeScript	DI + Options + middleware align with Clean Architecture; nullable types enforce layer contracts
Front-end framework	React + TypeScript + Vite	Vue 3	Largest companion-library ecosystem (Zustand, TanStack Query); typed contracts mirror C# back-end
Database	PostgreSQL on RDS	MongoDB	`jsonb` for normalised payloads alongside relational data; managed backups
Large language model	Gemini 2.5 Flash	Anthropic Claude	Lower cost per token at pipeline scale; native JSON output; reliable rubric following
Cloud platform	AWS	Google Cloud	Free tier covers a solo capstone; CloudFront edge in Ukraine
ORM	Entity Framework Core	Dapper	First-class migration tooling
Mediator	MediatR	Hand-rolled dispatcher	Pipeline-behaviour ecosystem for cross-cutting concerns
Password hashing	BCrypt.Net-Next	Argon2	Standard work-factor parameter; long audit track record

Table 3: Major technology decisions, the strongest alternative each was compared against, and the dominant driver. Full per-alternative analysis in Section C.

3.3 Clean Architecture layering and SOLID mapping

Section 2.6 requires that SOLID compliance be visible in the code, not just claimed. Clean Architecture [2] is the structural answer; SOLID is the class-level answer. This section maps both onto the Vakansio code base.

The four layers. The back-end is split into four .NET projects, one per layer:

- **Domain** — entities, value objects, enumerations, and pure scoring rules. Depends on no other project and on no third-party library. `Domain/Domain.csproj` declares zero package references.
- **Application** — use cases as MediatR handlers, options classes, and the abstractions through which use cases call out to the rest of the system. Depends only on Domain.
- **Infrastructure** — concrete implementations of those abstractions. EF Core, Gemini clients, AWS S3 adapter, SSM Parameter Store reader, seven job-source adapters. Depends on Application (Domain comes in transitively).

- **API** — the thin HTTP entry layer. ASP.NET Core controllers, JWT middleware, CORS, options binding, **`Program.cs`**. Depends on Application and Infrastructure.

The Dependency Rule. Clean Architecture has one strict rule: source-code dependencies only point inward, never outward. I enforce this through the project-reference graph. Domain references nothing. Application references only Domain. Infrastructure references Application. API references Application and Infrastructure. Nothing can make Domain depend on the Gemini SDK or EF Core. The compiler enforces the rule at build time. Because Domain has no infrastructure inside it, the layer is trivially unit-testable.

Dependency Inversion in practice. The Application layer declares its needs as interfaces in **`Application/Common/Interfaces/`** — **`ICvFileStorage`**, **`ICostLogService`**, **`IScoringService`**, **`IBatchedJudgeService`**, **`IBatchedReasonService`**, and others. Repository contracts owned by the entities themselves (for example **`IJobVacancyRepository`**) live in Domain instead. The Infrastructure layer provides the implementations. **`Program.cs`** wires the implementations to the interfaces through the built-in .NET DI container. Handler classes never see a concrete class — every collaborator comes in through an interface. This makes the dependency direction explicit at every call site. It also means I can swap whole infrastructure components in tests, or in production (swapping Gemini for OpenAI would touch only the Infrastructure layer).

CQRS via MediatR. Use cases follow the Command–Query Responsibility Segregation pattern. Read-only operations live under **`Application/*/Queries/`**, state-changing operations under **`Application/*/Commands/`**. Every handler implements **`IRequestHandler<TRequest, TResponse>`** with exactly one **`Handle`** method. MediatR dispatches incoming requests to the right handler. Cross-cutting concerns (validation, logging, retry) plug in as pipeline behaviours. The pattern keeps every handler single-purpose. The largest handler, **`GetAggregatedJobsV6Handler`**, runs the full search-and-score pipeline. It contains no Gemini call, no EF Core context, and no scraping logic. It composes the stages through interfaces injected into its constructor.

Versioning as cache-invalidation. Cache invalidation works through version strings. The Composite Judge prompt body has a **`BodyVersion`** constant. The scoring pipeline exposes a composite **`VersionWithJudge`** string — the scoring version + the judge body version + the model identifier. Each persisted **`ScoringCache`** row stores the composite version it was generated against. When any underlying constant changes (because a prompt was edited or a weight was rebalanced), the cache key changes, and every previously cached row becomes unreachable. No separate invalidation pass is needed. Cache correctness does not depend on a separate operational procedure.

SOLID mapping. Table 4 maps the five SOLID principles onto concrete artefacts in the code base.

Principle	Concrete artefact in Vakansio
Single Responsibility	`Domain/Scoring/Verdict.cs` and `Domain/Scoring/GenericGapFilter.cs` each contain a single rule and no orchestration; `GetAggregatedJobsV6Handler` delegates all stages to injected services rather than executing them inline.
Open/Closed	`Application/Common/Configuration/ScoringOptions.cs` with the `IOptions<T>` pattern makes tuning knobs such as `SyncNormalizeTimeoutSeconds` configurable through `appsettings.json` without recompilation.
Liskov Substitution	Scoring services depend on the interfaces `IScoringService`, `IBatchedJudgeService`, and `IBatchedReasonService` alternative LLM-provider implementations satisfy the same contracts with no consumer changes.
Interface Segregation	The three scoring interfaces are kept as separate single-purpose contracts; a class implementing `IBatchedReasonService` is not forced to also provide judging or scoring methods.
Dependency Inversion	Application handlers take only abstractions in their constructors; the composition root in `Program.cs` wires concrete implementations at startup, so the Application layer never imports an Infrastructure type directly.

Table 4: SOLID principles mapped onto concrete artefacts in the Vakansio code base.

Each row points to a file path that can be opened and read.

3.4 Scoring pipeline design

The scoring pipeline converts a (CV, vacancy) pair into a numerical score, a verdict label, and a bilingual explanation. It is the system's core technical contribution and what Section 5 evaluates.

Pipeline overview. Once the CV and the returned vacancies have been normalised, every (CV, vacancy) pair flows through five stages in order:

1. Deterministic sub-scoring.
2. Anti-flag penalty.
3. Optional Composite Judge refinement.
4. Safety cap.
5. Batched reason generation (produces the bilingual explanation).

The shape is fixed. The stages are wired together by `ScoringServiceV2.cs`.

The seven sub-score axes. The deterministic score is a weighted sum of seven sub-scores. Each sub-score is a pure C# calculator in `Infrastructure/RelevancePipeline/V2/Scoring/SubScoreCalculators/`. Table 5 shows the axes and weights (from `Domain/Scoring/ScoringConstants.cs:29–35`). I picked the weights by hand, not by automated search. The most recent rebalance (Skill 0.30 → 0.40, Language 0.10 → 0.05, Education 0.05 → 0.02) came from one observation: Language and Education are uniformly high across the candidate pool, so they add little to discrimination. Skill carries most of the signal.

Axis	Weight	Note
Skill match	0.40	Primary discriminator (rebalanced from 0.30)
Role-intent match	0.15	
Seniority match	0.15	
Experience match	0.15	
Domain alignment	0.08	Modest reduction from 0.10
Language match	0.05	Was 0.10 — near-uniform high
Education match	0.02	Was 0.05 — near-uniform high
Total	**1.00**	

Table 5: Composite weights of the seven scoring axes, as configured in **`Domain/Scoring/ScoringConstants.cs:29–35`**.

The deterministic linear score is the weighted sum

$$S_{\text{linear}} = \sum_{i=1}^{7} w_i \cdot s_i, \quad \text{where} \sum_{i=1}^{7} w_i = 1.00.$$

Anti-flag penalty. The linear score is then multiplied by an anti-flag penalty between zero and one. **`AntiFlagEvaluator.cs`** produces the penalty. It triggers when the candidate is structurally unsuited to the vacancy — for example, the vacancy requires on-site work in a city the candidate is unwilling to relocate to. I made the penalty multiplicative, not additive, so a single strong anti-flag can pull an otherwise strong score down decisively.

Role-family routing. Different job families weight skills differently. Before the Composite Judge step, **`KeywordRoleRouter.cs`** routes each vacancy to one of five families: Product, Engineering, Data, Design, or a Generic fallback. The router scores each candidate family by counting weighted keyword matches — weight 5 for the title, weight 1 for the description. The family with the highest score above 0.30 wins. Otherwise the routing falls through to Generic. The selected family determines which calibration examples the Composite Judge sees in the next step.

Multi-country source routing. A second routing layer runs upstream of scoring. The **`Country`** enum in **`Domain/Enums/`** has six members: Ukraine, UnitedStates, UnitedKingdom, Germany, Poland, and an All sentinel. Each **`IJobSourceService`** declares a **`SupportedCountries`** set. The aggregator filters adapters per request — a search for Germany only fans out to adapters that opt in for Germany. This keeps quota usage focused and avoids empty-result bugs where a Ukraine-only source receives an English-only query. The country travels through the V6 query DTO, lands in the cached snapshot, and shows up as a country selector on the front-end.

Composite Judge with skip-band. The Composite Judge is a Gemini 2.5 Flash call. It takes the (CV, vacancy, deterministic score, selected sub-scores) tuple and returns a refined score plus a brief justification. The prompt in **`JudgePromptCore.cs`** includes:

- A calibration rubric that anchors score bands to qualitative descriptions (0.85–0.95 EXCELLENT, down to 0.20–0.25 MISMATCH).

- Family-specific worked examples.

The Composite Judge is feature-flag gated through **`Ml:EnableCompositeJudge`** and includes a **skip-band optimisation**. A pair whose deterministic linear score is outside $[0.30, 0.85]$ and has no active anti-flags skips the Judge entirely — the deterministic signal is already extreme. The skip-band cuts Judge calls per query by 30–40%.

Sub-score caps as a safety layer. After the Composite Judge, **`IScoringCapService`** applies a floor of 0.20, a ceiling of 0.88, and asymmetric caps that limit how high a score can go on a clear seniority gap, language gap, or role-family mismatch. The caps act as a belt-and-braces safety layer. Even if the judge returns a misleadingly high score on a clear mismatch, the cap pulls it back into a sensible range.

Two-pass ranking. The full pipeline above is expensive — the Composite Judge and the reason generation are model calls. To keep the per-request cost manageable, I run a cheap pre-filter first. The deterministic linear score is computed for every returned vacancy. Only the top 40 (set by **`PreFilterTopN`**) pass to the Composite Judge. Of those, the top 30 (set by **`ReasonGenerationCap`**) pass to the batched reason-generation step. The cheap pre-filter handles the easy decisions. The model calls are reserved for the cases where refinement is most likely to change the ranking.

Reason generation. For the top thirty pairs, a batched step produces a structured JSON response with six text fields per pair — strengths, gaps, and recommendation, each in English and Ukrainian. The work is split into chunks of ten pairs per Gemini call. Up to three chunks dispatch in parallel. Each call uses **`temperature=0.2`** for predictability and validates the response against a strict schema. On failure it retries once, then falls back to a deterministic template. The bilingual JSON schema lets the front-end render the chosen language without a second round trip.

Skill expansion and canonicalisation. Before any sub-score is computed, raw skill tokens from both the CV and the vacancy are expanded and canonicalised against a global vocabulary. **`SkillCanonicalizer.cs`** maps synonyms — "React.js", "ReactJS", "react" — onto one canonical form. **`SkillVocabularyService`** handles rare or compound terms that the static vocabulary does not cover, through one batched Gemini call per request. I introduced the global-vocabulary path in version 6.3 to replace an earlier per-entity expander that produced many false-negative skill matches because of surface-form variation.

Evidence pipeline. The matched and missing skill chips the front-end shows are not the raw output of the sub-score calculators. They pass through **`EvidenceCleaner`** in four conceptual stages:

1. **Build** — collect candidate chips from sub-score outputs.
2. **Sanitise** — deduplicate, strip CEFR tokens, drop self-contradictions.
3. **Prioritise** — Tier 1 brand acronyms first, Tier 3 generic concepts last.
4. **Trim** — at most twelve matched skills and eight missing must-haves.

This pipeline is what makes the tier-based progressive disclosure on the front-end (Section 3.6) feasible.

3.5 Cache hierarchy

The scoring pipeline is expensive — each cold (CV, vacancy) pair requires several model calls. Caching is not an optimisation but a structural requirement. Vakansio uses four back-end cache layers plus a client-side cache. Each layer answers a different question about cost and freshness.

- **L1 — Response cache.** In-process **`IMemoryCache`**, 5-minute TTL. Key: user + CV version + search parameters. A repeated identical search by the same user is served from memory.
- **L2 — Scoring cache.** PostgreSQL **`ScoringCache`** table, key: (**`CvHash`**, **`VacancyId`**, **`ScoringVersion`**). **`CvHash`** is the SHA-256 of a canonical CV projection, so trivial edits (whitespace, field order) do not change the key. No time-based expiry — invalidation by version-string bump in the composite key (Section 3.3).
- **L3 — Vacancy normalisation.** The vacancy-normalisation Gemini output, stored as a **`jsonb`** column (**`VacancyAnalysisJson`**) on the **`JobVacancy`** row. The most expensive normalisation step. Valid until the vacancy is re-extracted.
- **L4 — Scrape cache.** In-process **`IMemoryCache`**, 30-minute TTL. Key: lowercased keywords + location. Lets a second identical search reuse the previous scrape rather than re-fetch every source.
- **Front-end cache.** TanStack Query persisted to IndexedDB. 72-hour max age, 10-minute stale time. Writes throttled to one per second. A cache-buster version string prevents stale data after an incompatible deploy. A multi-tab synchronisation channel keeps several open browser tabs consistent.

The four back-end layers share one invalidation philosophy. Where time-based expiry is appropriate (L1 response, L4 scrape) it is short because the data is volatile. Where it is not (L2 scoring, L3 normalisation) the key itself encodes the version of the procedure that produced the cached value. A procedure change drops the affected slice without any explicit operation. This is what makes the scoring pipeline safe to evolve without coordinated cache flushes.

Cache hardening for hot keys. Three additions guard the warm path against thundering-herd and pollution failures:

1. **Stampede coalescing. `JobAggregationService`** wraps each cache miss in a **`Lazy<Task<T>>`** keyed by the normalised search key. Concurrent identical requests coalesce into one scrape instead of fanning out to the sources in parallel.
2. **Negative caching.** Empty result sets are cached for three minutes under a dedicated key. An upstream outage cannot produce repeated full-cost requests for a query that just returned nothing.
3. **Background workers. `CacheRetentionWorker`** sweeps once per day. It prunes snapshots older than 30 days, scoring cache entries older than 90 days, cost-log entries older than 180 days, and audit entries older than 365 days, through EF Core's **`ExecuteDeleteAsync`**. **`DescriptionRetryWorker`** periodically re-fetches vacancies whose description capture failed with a 429 or 403. Transient source failures

self-heal without an operator. Both workers are feature-flag gated and default on in production.

A separate read-path guard handles snapshot DTO evolution. **`UserSearchSnapshot`** carries a **`SchemaVersion`** column. The read handler silently re-runs the v6 pipeline when the persisted version no longer matches the current DTO shape.

3.6 Frontend architecture and UX design

The front-end is a React + TypeScript single-page application bundled by Vite. It serves twelve pages across two user roles. The job-seeker side has six pages — Login, Register, Profile, Job Feed, Tracker, and About. The recruiter side adds five pages — Vacancies, Vacancy Results, Candidate Lists, Candidate List Detail, and a shared list view. The component tree is split into three concerns:

- **`components/ui/`** — design-system primitives. No business knowledge.
- **`components/jobs/`** — feature-level components for vacancy display.
- **`pages/`** — route-level components that React Router maps to URLs.

Cross-component contracts are pinned to single-source-of-truth maps (**`verdictMeta.ts`**, **`evidenceTier.ts`**). A label or colour change is a one-file edit.

State management. Three state systems, each with a clear role:

- Zustand — cross-cutting UI and session state.
- TanStack Query — server data with its loading and error state.
- Component-level **`useState`** — strictly local state with no external readers.

The split keeps the component tree free of imperative state code and avoids caching server data inside components.

Job feed and detail drawer. The Job Feed renders vacancies as cards in a scrollable list. Clicking a card opens a side drawer with the full description and the structured explanation. The drawer is a sibling component of the list, not a separate route. A route change would invalidate in-page list state (scroll position, filters, expanded chip groups) and force a re-render.

Tier-based progressive disclosure of evidence. Each vacancy carries up to 12 matched-skill chips and 8 missing-must-have chips. **`evidenceTier.ts`** classifies them into three tiers:

- **Tier 1** — brand names and tool acronyms. Shown by default.
- **Tier 2** — hyphenated or multi-word skills. Shown with slight visual hierarchy.
- **Tier 3** — generic concepts. Hidden behind a “+N similar” expansion link.

Exception lists rescue known false positives — lowercase brand names (“react”, “spring”) and product-manager metric phrases. The tier policy mirrors the back-end evidence-pipeline priorities (Section 3.4).

Verdict badges and bilingual UX. Each vacancy carries a coloured verdict badge — Strong, Partial, Weak, or Mismatch — read from **`verdictMeta.ts`**. The reason text is bilingual. The back-end returns both English and Ukrainian strings. The front-end renders

whichever language the user picks through **`LanguageContext`**. The i18n layer pairs locale tables (**`en.ts`**, **`uk.ts`**) with **`useT`** and **`usePlural`** hooks. A four-phase refactor consolidated the tables at 309 keys per language across sixteen user-facing components.

Persistence subsystem. A module under **`frontend/src/persistence/`** owns the lifecycle of the front-end cache:

- Schema migration step.
- Multi-tab synchronisation channel.
- Quota check.
- Eviction policy that drops the persisted blob beyond a 15 MB hard cap.

This is what makes the TanStack Query persisted cache safe across sessions.

3.7 Deployment topology

The deployment is a single-instance back-end on AWS, three managed components, and an edge cache for the front-end (Listing 1).

- **Compute host.** A single Amazon EC2 t3.micro instance in eu-central-1, running two Docker containers through Compose. Caddy at the public edge — terminates TLS via Let's Encrypt and applies security headers (HSTS, X-Content-Type-Options, X-Frame-Options DENY, Referrer-Policy). Behind it: the .NET 8 API container on a private bridge network.
- **Database.** PostgreSQL on RDS in the same region. **`SslMode=Require`** is enforced.
- **Object storage.** CV files in an S3 bucket with server-side AES-256 encryption and per-user prefixes. Exposed to the front-end through pre-signed URLs valid for five minutes.
- **Secrets.** SSM Parameter Store under **`/vacancies/prod/*`**. The API refreshes secrets in the background every five minutes.
- **IAM.** The EC2 instance carries an IAM role. No long-lived AWS access keys in the source tree or environment.
- **Front-end delivery.** The bundle is served from a separate S3 bucket through CloudFront. CloudFront provides edge caching for users across Ukraine.

The single-instance choice keeps the operational cost within the free tier and avoids a class of distributed-systems concerns (session affinity, distributed cache coherence). The migration path to ECS behind an Application Load Balancer is mechanical from this code base when scale demands it.

```
Browser
  │
  ├──► CloudFront ──► S3 bucket  (front-end bundle, static)
  │
  └──► Caddy (EC2 t3.micro, ports 80 / 443 / 443-UDP)
         │  reverse proxy, auto-TLS, security headers
         ▼
       .NET 8 API container  (Docker, private bridge network)
         │
         ├──► RDS PostgreSQL  (TLS-required connection)
         ├──► S3 bucket  (CV files, server-side AES-256)
         ├──► SSM Parameter Store  (/vacancies/prod/*)
         └──► Gemini 2.5 Flash  (HTTPS, external)
```

Listing 1: Vakansio production topology on Amazon Web Services.

3.8 Observability and security design

Per-stage cost transparency and data protection share a common pattern: cross-cutting infrastructure that has to be present at every stage of the pipeline without coupling the layers.

Cost ledger and tracing. Every Gemini call goes through an `AsyncLocal`-backed accumulator in `Application/Common/Diagnostics/CostBreakdown.cs`. The accumulator is request-scoped: parallel scoring tasks within one request share one record; concurrent requests do not interfere. At end-of-request the accumulator flushes to the PostgreSQL `GeminiCostLog` table — one row per stage, with call count, input/output tokens, wall-clock duration, and dollar cost. The endpoint `/api/admin/cost` returns the rows as CSV.

The accumulator lives in the Application layer on purpose. Both the orchestrating handler and the Infrastructure services write into it without the Application layer reaching downward into Infrastructure. The dependency direction stays clean. Logging failures are caught and swallowed by `CostLogService` — a missing log entry is preferable to a failed search.

Per-call detail lives behind the `ILlmTracer` abstraction and is forwarded to **LangSmith**. The tracer instruments every Gemini call made by the production pipeline. The offline evaluation harness uploads the Anthropic Claude judge ratings to LangSmith as dataset metadata through `LangSmithDatasetClient`, so every (CV, vacancy) pair sits under one published experiment number alongside the live Gemini traces. The tracer is swappable and optional through configuration.

Health endpoints. Two endpoints:

- `/health` — fixed liveness response, no dependencies touched. Consumed by Caddy and the Docker healthcheck.
- `/health/ready` — database round-trip. Used during deployment to confirm a freshly rolled container is ready to serve traffic.

Authentication, authorisation, and rate limiting. Stateless JWTs via `AddJwtBearer`. Passwords hashed with BCrypt.Net-Next. Rate limiting lives in ASP.NET Core middle-

ware inside the API project, not in Caddy. The policy has one source of truth and can apply per-route limits informed by application context. `[Authorize]` is declared at the class level on every controller that manages user-scoped resources (`JobsController`, `TrackerController`, and the recruiter and profile controllers). A new action inherits the policy by default. An unauthenticated route cannot be introduced by omission.

Startup guards. Two production-only guards refuse to start the application under misconfiguration:

- The first requires `SslMode=Require/VerifyCA/VerifyFull` on the database connection string.
- The second requires a non-empty `Cors:AllowedOrigins` list.

Both throw at startup and emit a clear log line. A misconfigured deployment cannot drift into a quietly insecure state.

GDPR cascade delete. Account deletion runs as a transactional cascade in `DeleteCurrentUserHandler`: tracker applications, relevance-explanation rows keyed on the CV version, the user-profile row. After the database transaction commits, the handler issues a best-effort S3 deletion of the CV file. The database is the authoritative artefact for the right-to-be-forgotten obligation. An orphan blob is recoverable via the bucket's lifecycle policy and does not violate the deletion semantics.

Resilience on model calls. Every Gemini call carries a short per-call timeout — 8 s in the inner scoring-loop reason call, 15–25 s in the Composite Judge, batched Judge, and batched reason services. On timeout or schema-validation failure, the call falls back to a deterministic template. On top of the timeouts, each of the 14 Gemini-bound `HttpClient` registrations is wrapped by a Polly policy through the `AddGeminiRetry()` extension in `Infrastructure/Resilience/GeminiRetryPolicy.cs`. The policy applies exponential backoff at 2, 4, and 8 seconds with jitter on HTTP 429, 5xx, and network failures. It recovers about 70% of the transient spikes observed in the production cost log without surfacing the failure to the request loop.

Prompt-injection mitigations. Four of the prompt-building stages accept text that comes from outside the trust boundary — the candidate's CV upload and scraped vacancy descriptions from third-party boards. The shared `PromptInputSanitizer` under `Infrastructure/RelevancePipeline/V2/Sanitization/` applies the OWASP LLM01 [8] mitigation pattern. Three defences on the input side:

1. **XML-style wrapping.** Every input is wrapped in `<untrusted_*>` tags.
2. **Anti-injection preamble.** The prompt body prepends a preamble that names the trust boundary and tells the model to treat content inside the tags as data, not as instructions.
3. **Length caps.** Vacancy text 30,000 chars, raw CV text 40,000 chars, normalised CV-summary JSON 20,000 chars. Delimiter echoes — an adversarial payload that prints the literal closing tag to confuse the parser — are neutralised by replacing angle brackets with square brackets before wrapping.

Three further architectural properties bound the blast radius of a successful jailbreak:

- **Schema enforcement.** The runtime parses the response and rejects schema mismatches.

- **Score clamping.** `ScoringCapService` clamps the final score to $[0.20, 0.88]$ regardless of what the model reports.
- **No agentic actions.** The pipeline produces only data DTOs — no LLM-triggered database writes, no LLM-triggered network calls.

On the front-end side, React's default text-node rendering auto-escapes every character in the reason text. An injected payload that survives the model is rendered as literal characters, not as executable HTML or JavaScript. The mitigations are best-effort and explicitly documented as such in Section 5.8. A model-vendor content moderation API and a regex pre-filter for known injection phrases are recorded as future work in Section 6.4.

3.9 Chapter summary

The design is a .NET 8 back-end and a React + TypeScript front-end on AWS. Four Clean Architecture projects, the dependency rule enforced by the compiler.

The scoring pipeline routes each (CV, vacancy) pair through deterministic sub-scores, an anti-flag penalty, a Gemini Composite Judge with a skip-band optimisation, asymmetric safety caps, and a batched bilingual reason-generation step.

A four-layer cache hierarchy with version-string invalidation, stampede coalescing, and retention workers keeps the warm-path cost low without coordinated flushes.

The front-end pairs Zustand with TanStack Query and a tier-based progressive disclosure of evidence chips.

Cross-cutting concerns — cost tracking, GDPR cascade deletion, startup guards, Polly retry policies, prompt-injection mitigations — live in the Application layer, not scattered across services. The boundaries stay intact.

4 Implementation

This chapter turns the design from Section 3 into a deployed system. After the development methodology I walk layer by layer through Domain, Application, and Infrastructure, document the front-end implementation, the deployment and operating costs, and the engineering narratives that shaped the architecture.

4.1 Development methodology

I developed Vakansio as versioned releases of the scoring pipeline rather than fixed-length sprints. Each version corresponds to an observable change in system behaviour — a rebalance of the scoring weights, the introduction of the global-vocabulary path in v6.3 that replaced the per-entity skill expander, or one of the contract-drift fixes documented in Section E. The version string is also the cache-invalidation key from Section 3.3. Every iteration is therefore both an engineering increment and a deployable artefact with a defined effect on the persisted caches.

The system grew outward from a minimal core. The first version had only the deterministic seven-axis sub-scorer and a thin API. I added the Composite Judge next, then the batched reason generator, then the tier-based evidence disclosure on the front-end, then the cost-tracking subsystem. Each layer was validated on the deployed instance against real CV–vacancy pairs before the next layer was added.

Conventions are enforced at the build level:

- The back-end runs on .NET 8 with nullable reference types enabled across every project.
- Each CQRS handler is a single class with one **`Handle`** method.
- Clean Architecture project references are enforced by the compiler. **`Domain/Domain.csproj`** declares zero packages. Application references only Domain.
- The front-end is built with **`noUnusedLocals`**, **`noUnusedParameters`**, and **`noFallthroughCasesInSwitch`** enabled. The build fails on dead code and missed switch cases.
- ESLint enforces the recommended TypeScript and React-Hooks rule sets.

Three quality measures run in parallel:

1. **Unit tests** — cover the pure Domain rules and sub-score calculators. 349 tests passing at submission time.
2. **Author acceptance tests** — for every observable version, I tested against the deployed instance on a fixed set of CVs and search queries before shipping.
3. **Offline evaluation harness** — separate from the production code path. Computes ranking-quality and reason-quality metrics on an LLM-judged gold set. Methodology and results in Section 5.

The continuous cost-tracking subsystem complements the test suite by surfacing performance regressions as cost changes in the **`GeminiCostLog`** table.

4.2 Domain layer walkthrough

`Domain/Domain.csproj` declares zero package references and no project references. The layer is unit-testable in isolation from EF Core, the Gemini SDK, and every other infrastructure concern.

Entities. **`Domain/Entities/`** holds thirteen entities. The seven that drive the core job-seeker matching pipeline:

- **`JobVacancy`** — a scraped vacancy plus its normalised analysis stored as **`jsonb`**. This is the L3 cache.
- **`UserProfile`** — the user record. Points to the current CV version and caches its normalisation.
- **`ApplicationTracker`** — one row per job the user is tracking.
- **`ScoringCacheEntry`** — the L2 cache row, keyed by (CV hash, vacancy ID, scoring version).
- **`SkillVocabularyEntry`** — backs the global-vocabulary path from v6.3.
- **`GeminiCostLogEntry`** — one row per Gemini call (cost ledger).
- **`SavedUrl`** — vacancies the user added manually by URL.

The remaining six entities support the recruiter-side workflow (**`RecruiterCandidate`**, **`CandidateList`**, **`CandidateListMembership`**, **`CandidateScore`**) and audit/snapshot persistence (**`AuditEntry`**, **`UserSearchSnapshot`**).

Value objects. Two, under **`Domain/ValueObjects/`**:

- **`RelevanceScore`** — wraps a score with the pipeline stage that produced it. Constructor rejects values outside $[0, 100]$.
- **`Salary`** — min, max, currency, and a raw-text fallback for unstructured source values.

Enumerations. Eight, under **`Domain/Enums/`**. The six used by the matching pipeline are **`ApplicationStatus`**, **`JobSource`**, **`ScoringStage`**, **`SeniorityLevel`**, **`WorkFormat`**, and the multi-country **`Country`** added in this iteration. Two more (**`UserRole`**, **`CandidateNormalizationStatus`**) support the recruiter workflow.

Pure scoring rules. The substantive core of Domain is **`Domain/Scoring/`**:

- **`CvHasher`** — SHA-256 hash of a canonical CV-summary projection (sorted skills, target roles, seniority, English level, experience in months). Non-substantive edits do not change the L2 cache key.
- **`GenericGapFilter`** — drops generic terms from the missing-skill list.
- **`LanguageGapDetector`** — ranks CEFR levels A1 to C2.
- **`RoleFamilyDetector`** — deterministic family routing from Section 3.4.
- **`SubScores`** — record carrying the seven per-axis scores. Weights live in **`Domain/Scoring/ScoringConstants.cs:29–35`** so a rebalance is one-file.
- **`ScoringResult`** — aggregates headline score, sub-scores, verdict, and evidence chips into the typed pipeline output.

Verdict and bilingual reason path. **`Domain/Scoring/Verdict.cs`** ties the verdict label and the bilingual output together. The enum has four members. **`VerdictExtensions`** exposes **`FromScore`**, **`ToEnglishText()`**, **`ToUkrainianText()`**, and **`ToShortName()`**. All three

vocabularies live in Domain, so a change to the verdict-label contract is a one-file edit (Listing 2).

```
public enum Verdict { Mismatch, WeakMatch, PartialMatch, StrongMatch }

public static class VerdictExtensions {
    public static Verdict FromScore(double s) => s switch {
        >= 0.75 => Verdict.StrongMatch,
        >= 0.50 => Verdict.PartialMatch,
        >= 0.25 => Verdict.WeakMatch,
        _       => Verdict.Mismatch,
    };

    public static string ToShortName(this Verdict v) => v switch {
        Verdict.StrongMatch  => "Strong",
        Verdict.PartialMatch => "Partial",
        Verdict.WeakMatch    => "Weak",
        Verdict.Mismatch     => "Mismatch",
    };
}
```

Listing 2: **`Domain/Scoring/Verdict.cs`** — the verdict enum and the short-name extension that the front-end keys on.

Interfaces. Two folders for Domain interfaces:

- **`Domain/Interfaces/Repositories/`** — twelve repository contracts. They live in Domain because the aggregates they serve are themselves Domain concepts.
- **`Domain/Interfaces/Services/`** — three contracts for domain operations whose implementation is technology-specific but whose intent is not: **`IDeduplicationService`**, **`IJobSourceService`**, **`IRelevancePipeline`**.

Application-level abstractions (**`ICvFileStorage`**, **`ICostLogService`**, **`IScoringService`**, the batched-judge and batched-reason services, **`ILlmTracer`**) live under **`Application/Common/Interfaces/`** — they are use-case-level, not domain-level.

4.3 Application layer walkthrough

`Application/Application.csproj` references only Domain and pulls in FluentValidation, MediatR, and the in-memory caching abstractions. No EF Core, no Gemini SDK, no AWS SDK.

Feature-folder CQRS organisation. Use cases live under four feature folders — **`Jobs/`**, **`Tracker/`**, **`User/`**, **`Eval/`**. Each is split into **`Commands/`** for state-changing operations and **`Queries/`** for read-only operations. Each use case has three files:

- A **`*Command.cs`** or **`*Query.cs`** with the request DTO. Implements **`IRequest<TResponse>`**.
- A **`*Handler.cs`** implementing **`IRequestHandler<TRequest, TResponse>`** with one **`Handle`** method.
- A **`*Validator.cs`** (when input validation is needed) extending **`AbstractValidator<TRequest>`**.

The folder layout makes the CQRS split visible at a glance.

The orchestrator handler. The largest handler is **`GetAggregatedJobsV6Handler`**, about one thousand lines. It is the entry point for the full search-and-score pipeline. Despite its size, it contains no direct Gemini call, no EF Core context, and no scraping logic. Every collaborator — the aggregator service, the scoring service, the Composite Judge, the batched reason generator, the skill-vocabulary service, the cost logger, the LangSmith tracer — comes in through an interface. The handler composes the stages in the right order, manages the per-request cost-tracking scope, and assembles the response DTO. The work of calling Gemini or the database lives in Infrastructure.

Options pattern for tuning knobs. Tuning knobs (**`ScoringOptions.SyncNormalizeTimeoutSeconds`**, the prefilter cap, the reason cap) bind through **`IOptions<T>`** from **`appsettings.json`**. Retuning the pipeline does not require a rebuild.

Cross-cutting MediatR validation. Input validation runs as a MediatR pipeline behaviour in **`Application/Common/Behaviors/ValidationBehavior.cs`**. For every dispatched request, the behaviour collects every registered **`IValidator<TRequest>`** from the DI container and runs them. If any rule fails, it throws a **`ValidationException`** with the accumulated failures. Only if every validator passes does control reach the handler.

Per-stage cost accumulator. A static class backed by an **`AsyncLocal<Accumulator?>`** field (Listing 3). The orchestrator handler opens a scope at request start. Each Gemini-calling service in Infrastructure reports its stage statistics through **`CostBreakdown.Track`**. At request end, the handler reads the snapshot and persists the rows.

```
public static class CostBreakdown {
    private const double InputPricePerMillion  = 0.30;
    private const double OutputPricePerMillion = 2.50;

    private static readonly AsyncLocal<Accumulator?> _current = new();

    public static IDisposable BeginScope() {
        _current.Value = new Accumulator();
        return new Scope(_current);
    }

    public static void Track(string stage, double ms, long inT, long outT) {
        _current.Value?.Add(stage, ms, inT, outT,
            (inT * InputPricePerMillion + outT * OutputPricePerMillion) /
1_000_000);
    }

    public static IReadOnlyList<StageStats> GetSnapshot()
        => _current.Value?.Snapshot() ?? Array.Empty<StageStats>();
}
```

Listing 3: **`Application/Common/Diagnostics/CostBreakdown.cs`** — request-scoped, parallel-safe per-stage cost accumulator. Pricing constants are private to this file, so a Gemini-tier price change is a one-line edit.

4.4 Infrastructure layer walkthrough

Infrastructure depends only on Application. The package references are:

- EF Core 8 with Npgsql and Pgvector.
- AWSSDK.S3 4.0.15.
- BCrypt.Net-Next 4.0.3.
- HtmlAgilityPack 1.12.0.
- System.ServiceModel.Syndication 8.0.0 (RSS).
- PdfPig 0.1.14 (PDF).
- Polly through the hosting and HTTP packages.

Persistence. `Infrastructure/Persistence/` holds the database integration:

- One `AppDbContext` with one `DbSet` per persisted entity.
- Twelve `IEntityTypeConfiguration<T>` classes for entities that need explicit column shaping.
- Twenty-seven sequential migrations from `20260314_InitialCreate` to `20260613_AddSnapshotSchemaVersion`. Applied idempotently at container startup.
- One repository implementation per Domain contract.

The GDPR cascade-delete is implemented through `OnDelete(DeleteBehavior.Cascade)` annotations in `UserProfileConfiguration` and reinforced transactionally by `DeleteCurrentUserHandler`.

Job-source adapters. `Infrastructure/JobSources/` holds the seven adapters that pull vacancies, split by access pattern:

- **REST API clients** (under `JobSources/Api/`) — `JoobleApiService`, `RobotaUaApiService`.
- **RSS feed parser** (under `JobSources/Rss/`) — `DouRssFeedService`.
- **HTML scrapers** (under `JobSources/Scraping/`) — `DjinniScraperService`, `LinkedInGuestService`, `WorkUaScraperService`, `ManualUrlScraperService`.

Every adapter implements the same `IJobSourceService` contract from Domain and declares its `SupportedCountries` set. `JobAggregationService` resolves the registered services per request, filters by the requested `Country`, fans out in parallel under a `Lazy<Task<T>>` that coalesces stampeding requests for the same key, and catches per-source exceptions (Listing 4). The combined result passes through `DeduplicationService` and is cached in `IMemoryCache` with a 30-minute TTL.

```
var tasks = _sources.Select(async source => {
    try {
        return await source.FetchJobsAsync(keywords, location, country, ct);
    } catch (OperationCanceledException) {
        throw;
    } catch (Exception ex) {
        _logger.LogWarning(ex, "Source {Source} failed; continuing",
source.SourceName);
        return Enumerable.Empty<ScrapedVacancyDto>();
    }
});
var perSource = await Task.WhenAll(tasks);
var all = perSource.SelectMany(x => x).ToList();

_cache.Set($"scrape:{country}:{keywords.ToLower()}:{location.ToLower()}", all,
    new MemoryCacheEntryOptions {
        AbsoluteExpirationRelativeToNow = ScrapeCacheTtl, // 30 minutes
        Size = 1,
    });
```

Listing 4: **`JobAggregationService`** — parallel fan-out with per-source isolation, separate cancellation handling, and the L4 scrape cache write keyed by country (excerpt).

Scoring pipeline. **`Infrastructure/RelevancePipeline/V2/Scoring/`** realises the stages from Section 3.4.

- **`ScoringServiceV2`** implements **`IScoringService`** and is the canonical source of the seven sub-score weights.
- Seven stateless calculators under **`SubScoreCalculators/`** compute the deterministic axes.
- **`AntiFlagEvaluator`** and **`ScoringCapService`** form the deterministic safety layer.
- **`GeminiCompositeJudgeService`**, **`GeminiBatchedJudgeService`**, **`GeminiBatchedReasonService`** share the prompt body and version string from **`JudgePromptCore`**. The version string participates in the cache key.

Normalisation modules. CV-normalisation and vacancy-normalisation follow the same five-component shape — keyword router, module resolver, per-domain modules, prompt builder, post-processor. The CV side ships two modules (Tech, Generic). The vacancy side ships six (Tech, HR, Healthcare, Marketing, Sales, Generic). Adding a new domain takes four steps: extend the domain enum, implement the module interface, register the module, extend the keyword router.

Adapters, ML API, workers, composition root. **`Infrastructure/Services/`** holds three small adapters:

- **`S3CvFileStorage`** — AES-256 encryption, 5-minute pre-signed URLs. Registered conditionally on the presence of **`S3:CvBucket`**. **`NoOpCvFileStorage`** is the boot fallback for development.
- **`CostLogService`** — best-effort writes to **`GeminiCostLog`**.
- **`DatabaseHealthService`** — the round-trip behind **`/health/ready`**.

Infrastructure/Observability/ implements **ILlmTracer** against the LangSmith management API and provides **LangSmithDatasetClient** for the evaluation harness. **Infrastructure/MlApi/** wraps an optionally deployed Python ML service (bi-encoder, multilingual-e5-base embedding, Qwen2.5-3B reasoning, MiniCheck factuality).

Six **BackgroundService** workers live under **Infrastructure/Workers/**. Four are flag-gated and default off in production (CV-summary, job-embedding, reasoning, vacancy-analysis). The two cache-retention and description-retry workers from Section 3.5 default on.

The composition root **Infrastructure/DependencyInjection.cs** wires every contract to its implementation through explicit registration and applies **AddGeminiRetry()** to all 14 Gemini **HttpClient** registrations.

4.5 Frontend implementation details

The front-end stack is React 19.2 with TypeScript on the Vite 8 bundler. Routing through react-router-dom 7.14 serves twelve pages across two user roles. Six pages support the job seeker (Login, Register, Profile, Job Feed, Tracker, About). Five more pages back the recruiter workflow (Vacancies, Vacancy Results, Candidate Lists, Candidate List Detail, plus a shared list view). The production build runs as **tsc -b && vite build**. A TypeScript compilation error fails the build before Vite emits a bundle.

API client. The shared axios client in **src/api/client.ts** injects the JWT into every request and redirects to **/login** on a 401 (Listing 5).

Vite dev-mode proxy. In development, Vite forwards **/api** requests server-side to **https://api.vakansio.online** with **changeOrigin: true**. The browser only ever issues same-origin requests to **localhost:5173/api/***. Two practical consequences:

- The production CORS allow-list does not need to include the developer's local host.
- The development loop runs against real production data and scoring caches.

```
const apiClient = axios.create({
    baseURL: import.meta.env.VITE_API_BASE_URL ?? "http://localhost:5180/api",
    timeout: 300_000, // 5 min: cold v6 search can take up to 2 min
});

apiClient.interceptors.request.use(cfg => {
    const token = localStorage.getItem("token");
    if (token) cfg.headers.Authorization = `Bearer ${token}`;
    return cfg;
});

apiClient.interceptors.response.use(r => r, (err) => {
    if (err.response?.status === 401 && location.pathname !== "/login") {
        ["token", "userId", "email"].forEach(k => localStorage.removeItem(k));
        window.location.href = "/login";
    }
    return Promise.reject(err);
});

export default apiClient;
```

Listing 5: **`src/api/client.ts`** — shared axios instance with JWT injection and unauthenticated-redirect handling.

State management. The state split designed in Section 3.6 maps onto two Zustand stores (**`authStore`**, **`jobStore`**) and the TanStack Query layer. Server data never leaks into the Zustand stores. Strictly local state (the identifier of the open vacancy drawer) stays in component-level **`useState`**.

Persistence subsystem. **`src/persistence/`** contains five files:

- **Async-storage persister** over IndexedDB.
- **One-shot migration** from the legacy **`localStorage`** blob to the new IndexedDB key.
- **Multi-tab `BroadcastChannel` synchronisation** — a write in one tab invalidates the matching query in every other open tab.
- **Diagnostic helpers** over **`navigator.storage.estimate()`**.
- **Eviction policy** with a 15 MB hard cap — drops the persisted blob entirely once the threshold is crossed.

Internationalisation. The i18n layer under **`src/i18n/`** pairs the EN/UK locale tables with a **`useT`** hook and a **`usePlural`** hook. The plural-form rules differ between English and Ukrainian. No user-visible string is hard-coded in a component file.

4.6 Deployment, operations, and operating costs

The production environment is a single Amazon EC2 t3.micro host. It runs two Docker containers — Caddy and the .NET 8 API — through a Compose file on a private bridge network. The container build, the Compose topology, the Caddyfile, the AWS resource identifiers, and the stop/start scripts are in Section D. This section covers the deployment automation, the operator-facing endpoints, and the operating cost.

CI/CD pipeline. Three GitHub Actions workflows run on every push to **`main`**:

1. **`ci.yml`** — runs the .NET test suite and the front-end build. On success it gates the two deploy workflows.
2. **`deploy-api.yml`** — builds the multi-architecture Docker image, copies it to the EC2 host over SSH, applies pending EF Core migrations idempotently at container startup, and rolls the Compose stack forward. The SSH key is base64-encoded in the repository secret and decoded inside the runner to avoid CRLF mangling.
3. **`deploy-frontend.yml`** — runs **`npm run build`**, uploads the bundle to the static-hosting bucket via **`aws s3 sync`**, and issues a CloudFront invalidation for the affected paths.

The health check that gates the rollout runs through SSH into the running container (**`docker exec vacancies-api curl 127.0.0.1:8080/health`**), not from the runner. This bypasses Caddy's TLS layer and avoids the SNI-mismatch failure that an IP-addressed curl would otherwise produce. End-to-end roll-out from push to live takes about ten minutes.

Operational endpoints. Two operator-facing endpoints supplement the standard health probes:

- **`POST /api/Admin/cache/prewarm-vacancies`** — accepts a keyword, scrapes and synchronously normalises up to a clamped maximum of vacancies without touching the CV or scoring pipeline. Warms the three CV-independent caches (scrape, vacancy normalisation, skill vocabulary). On the production cost log, a warm follow-up search on the same keyword cuts downstream Gemini spend by about 47%.
- **`/api/admin/cost`** — returns the aggregated **`GeminiCostLog`** rows as CSV, grouped by an optional dimension, for offline analysis.

Operating-cost analysis. The cost has two components.

The **fixed infrastructure** part. At publicly listed June 2026 prices in eu-central-1, the t3.micro EC2 instance, the db.t3.micro RDS instance with 20 GB of storage, and the S3 + CloudFront stack together cost approximately **$25–$28 per month** once any Free-Tier allowance is exhausted. The front-end bucket and the SSM Parameter Store stay within the always-free tier.

The **variable language-model** part. Gemini 2.5 Flash is priced at a flat $0.30 per million input tokens and $2.50 per million output tokens. These two rates appear verbatim as the private constants **`InputPricePerMillion`** and **`OutputPricePerMillion`** in **`CostBreakdown.cs:6–7`** (Listing 3), so the application's own cost-estimation arithmetic agrees with the published price list. Per-request cost depends on the pipeline path taken (cold vs warm, Judge skip-band hit rate, number of pairs that enter the batched reason generator). The **`GeminiCostLog`** table measures it continuously.

Three architectural decisions have the most direct effect on the variable cost:

- **Composite Judge skip-band** — avoids 30–40% of Judge invocations.
- **Global-vocabulary path** — replaces per-entity skill expansion with one batched call.
- **Disabled background workers** — idle Gemini spend stays at zero.

4.7 Selected engineering narratives

Two episodes from the implementation show how the architecture absorbed change without scattering the response across layers.

v6.7.8 contract drift fix. A single-line back-end serialisation change (adding `JsonStringEnumConverter` at `API/Program.cs:60`) broke the front-end source filter and the verdict-label vocabulary at the same time, in two independent files. The fix was a single Domain-layer extension method (`VerdictExtensions.ToShortName()`) plus a numeric-to-string update on the front-end `<select>`. The single-source-of-truth maps (`verdictMeta.ts`, `evidenceTier.ts`) localised the contract surface.

Graceful-degradation pattern. Every external call — per-source scraping, Gemini timeouts, cost-log writes — is wrapped in try-catch. The pipeline returns a typed degradation indicator (`SkippedNoAnalysis` on the V6 result) instead of failing the request. A single failing dependency reduces the result, not the response.

The full narratives, with file references and per-failure-mode handling, are in Section E.

4.8 Chapter summary

The implementation realises the four-project Clean Architecture with the dependency rule enforced by the compiler.

The Application layer hosts a request-scoped, parallel-safe cost accumulator. Both the orchestrating handler and the Infrastructure-bound services write into it without breaking the dependency direction.

Infrastructure wires the seven country-aware job-source adapters, the Gemini scoring pipeline with Polly retry on all 14 HTTP clients, the LangSmith tracer, the S3 and SSM adapters, and the cache-retention and description-retry workers.

The front-end pairs Zustand and TanStack Query, persists the query cache to IndexedDB under a 15 MB hard cap, and renders bilingually.

The deployment is automated through three GitHub Actions workflows behind Caddy on a single EC2 t3.micro. The fixed monthly cost is about $25–$28 once the Free-Tier window closes.

Two engineering narratives — the v6.7.8 contract drift fix and the graceful-degradation pattern — show how the architecture absorbed unexpected change without scattering the response across layers.

5 Validation

This chapter documents the evaluation methodology and its findings. The decomposition follows the Holistic Evaluation of Language Models framework [6] and uses an Anthropic Claude judge where no reference answer exists [7]. The primary result on ranking quality is directionally non-significant on the development gold set; the held-out evaluation tightens the picture on a fresh-pair gold set the prompt never saw.

5.1 Evaluation philosophy and methodology pivot

The original primary endpoint was score MAE against a deterministic ideal computed from gold inputs. Two independent audits found three structural problems with that comparison:

- **Cap asymmetry.** The pipeline applies safety caps and an anti-flag penalty. The ideal does not. Any pipeline that ships caps gets mechanically penalised on this metric.
- **Selection circularity.** The gold pair set was stratified on title overlap, seniority match, and skill overlap. These are the same inputs the Linear pipeline consumes. The test set was engineered to favour the Linear baseline.
- **Composite Judge over-penalisation.** Rule-precedence clauses let the Judge override its bounded-adjustment band on under-qualification, language gap, or family mismatch. The post-Judge clamp does not enforce the bound.

I revised the design. The new primary endpoint is top-ten NDCG against independent-vendor relevance ratings. The rater is in the Anthropic Claude family; the system under test runs on Google Gemini. Different vendor families remove the circular reasoning the original harness had.

The judging protocol follows Zheng et al. [7]:

- Anchor-only ordinal ratings.
- Randomised candidate order.
- Mandatory per-rating rationale.
- Stratified batches across verdict buckets.

The same protocol underpins Layers 4, 5, and 6.

5.2 Layered evaluation architecture

The pipeline produces four artefacts per (CV, vacancy) pair — a structured CV summary, a structured vacancy analysis, a numeric score, and a bilingual reason text. Each is evaluated by the method appropriate to its data type rather than collapsed into a single end-to-end number; a single metric blurs failures across stages and makes fixes harder. Five layers cover the development pipeline; a sixth held-out layer adds an evaluation on pairs the production prompt never saw:

- **Layer 1 — CV normalisation.** Per-field F1 = 0.896 on 25 CVs, frozen after nine prompt versions.

- **Layer 2 — Vacancy normalisation.** Per-field overall = 0.804 on 355 of 357 vacancies, frozen after twelve prompt versions.
- **Layer 3 — Score.** Linear MAE = 0.101, Composite MAE = 0.187 on 392 pairs; kept as a calibration diagnostic (Section 5.4).
- **Layer 4 — Reason text.** Bilingual natural language, judged by Claude Opus 4.7 on a six-dimension rubric. Prompt v6 wins $+0.30$ over the v4 baseline on 391 pairs, CI $[+0.19, +0.40]$.
- **Layer 5 — Ranking.** Top-ten ordered list, judged by Claude Sonnet 4.6 on a single `match_quality` score. Δ NDCG@10 $= -0.047$, CI $[-0.168, +0.065]$ on 14 CVs; inconclusive at $\alpha = 0.05$.
- **Layer 6 — Held-out validation.** 398 pairs the production prompt never saw, judged by Claude Opus 4.7. Spearman $\rho = 0.648$, NDCG@5 $= 0.822$, midrange-filter Spearman $\rho = 0.738$; the controlled cap-on/off ablation and the production-deployed isotonic calibrator (ECE $0.140 \rightarrow 0.027$) are reported in Section 5.7.

Each layer owns its own failures and uses the literature-standard method for its data type — hand-annotated references with field-level scoring for the structured layers, a computed ideal for the numeric layer, LLM-as-Judge on a rubric for the natural-language layer [7], and ordering-quality metrics against per-pair relevance ratings for the ranking layers [5]. The same separation Clean Architecture imposes on the production code (Section 3.3) organises the evaluation: the Domain types the scoring service consumes are exactly the inputs the eval tool feeds to each layer. The eval tool, a separate .NET project under `EvalTool/`, runs offline against snapshots of the gold data and shares the production interfaces, so it tests the same code paths that ship to users.

Reproducibility artefacts. Each layer ships a different artefact through which a third party can audit the headline numbers. For Layer 4 and Layer 5 the per-pair predictions, judge rationales, and metric reports are persisted as versioned JSON and Markdown under `results/` in the source repository; the directory naming encodes the prompt version and the gold-set size (for example `results/metrics_v1_6_n398/`). For Layer 6 the same artefacts plus a full per-call trace are available through LangSmith, where the headline run is registered as experiment `vakansio_v1_6_n398_v2` under dataset `vakansio_match_quality_heldout`. Each LangSmith trace records the input prompt, the judge response, the parsed score and rationale, the per-pair correlation identifier, and the wall-clock latency, so the headline numbers are reproducible at the level of the individual judgement rather than only at the level of the aggregate.

5.3 Layers 1 and 2 — frozen normalisation baselines

Layers 1 and 2 grade the two structured-output normalisation stages against author-curated reference JSON using per-field metrics declared in `EvalTool/Grading/`. Layer 1 (CV normalisation) reaches a macro-averaged 0.896 across nineteen field-level graders on a twenty-five-CV gold set after nine prompt versions; Layer 2 (vacancy normalisation) reaches a field-weighted 0.804 across seventeen graders on 355 of 357 stratified vacancies after twelve prompt versions. Both layers are frozen on a noise-threshold criterion — subsequent iterations change the headline by less than the variance observed across the last three runs. Because they are frozen, the later layers run against a known, reproducible

upstream. The two caveats are that the gold sets were curated by a single annotator (me) without an inter-annotator coefficient, and that the lowest-F1 fields (the skill lists, roughly 0.47–0.62) are the same fields the scoring formula leans on hardest. The per-grader breakdown and the iteration-history pointer live in Section F.

5.4 Layer 3 — score as a calibration diagnostic

The score-error harness compares the pipeline's end-to-end composite against a deterministic ideal computed on gold inputs. On the 392-pair test set:

- Linear pipeline: MAE 0.101, verdict-bucket match 67.3%.
- Composite pipeline (Judge invoked on every pair): MAE 0.187, verdict-bucket match 31.9%.

The naive reading — that the Composite Judge is worse than the Linear baseline — is misleading. Two audits decomposed the 0.086 headline gap into three structural artefacts:

- **Cap asymmetry.** The pipeline applies safety caps and anti-flag penalties that the ideal does not. By per-pair decomposition, this accounts for the entire Linear MAE.
- **Selection circularity.** The test set was stratified on title overlap, seniority match, and skill overlap — the same primitives the Linear pipeline consumes. The audit attributes 0.02–0.04 MAE points of the gap to this asymmetry.
- **Composite Judge over-penalisation.** The Judge prompt's rule-zero ± 0.05 band is overridden by rules three to five, and the post-Judge clamp does not enforce the bound. This attributes the five largest negative divergences.

The remediation is a hard $\left|s_{\text{judge}} - s_{\text{linear}}\right| \leq 0.10$ clamp at the `GeminiCompositeJudgeService` boundary. The post-hoc sensitivity check in Section 5.6 confirms its effect. Because the metric measures the caps rather than the pipeline quality, Layer 3 stays as a per-sub-score bias diagnostic rather than the headline evidence. The full per-finding decomposition is in Section G.

5.5 Layer 4 — reason-text quality and the prompt iteration cycle

Layer 4 evaluates the bilingual reason text the pipeline generates for the top-thirty candidate vacancies in every result.

The judge. Claude Opus 4.7 — the strongest publicly available Anthropic model at the time of writing (June 2026). The system under test runs on Google Gemini 2.5 Flash, so judge and system come from different vendor families. The protocol follows Zheng et al. [7] and the hiring-rubric design of Hu et al. [9].

The rubric. Six dimensions per pair. Four ordinal on the anchor set $\{0, 2, 4, 6, 8, 10\}$:

- `factual_accuracy` — do the strengths and gaps match the actual CV and vacancy content?
- `completeness` — are the top factual signals surfaced?
- `specificity` — is the phrasing pair-unique, not boilerplate?
- `relevance_for_user` — is the explanation actionable for an apply-or-skip decision?

Two binary:

- **`verdict_match`** — does the verdict word agree with the score bucket?
- **`uk_term_preservation`** — are brand and tool tokens kept in Latin script in the Ukrainian half?

Bias mitigations. Following Zheng et al.: anonymised candidate identifiers, no intra-batch comparisons, anchor-only scoring, batches of 28 stratified across verdict buckets, per-rating rationale alongside every numeric score.

Judge self-consistency check. I ran Claude Opus 4.7 twice on the same 10 pilot pairs with randomised order and non-zero temperature, then computed Krippendorff's α across the rubric. Mean ordinal $\alpha = 0.695$. Binary raw agreement 93–97%. Both above my thresholds (0.6 ordinal, 0.7 binary). The lowest per-dimension α (**`factual_accuracy`** at 0.603) is 0.003 below the threshold and reported transparently. This is a self-consistency check, not a cross-vendor inter-rater check — the latter is the principal limitation in Section 5.8.

Statistical inference. Paired bootstrap with 10,000 resamples on the per-pair overall mean across the six dimensions. Per-dimension deltas are reported separately as exploratory signals.

The iteration cycle. Four prompt versions (v2 → v4 → v5 → v6), each developed in response to a regression that the previous iteration surfaced. Each detection was a confidence-interval calculation, not a guess. Each fix was a single component change with the rest of the pipeline held still (Table 6).

Comparison	Δ overall	95% CI	Sig.	Headline signal
v4 vs v2	$+0.55$	$[+0.42, +0.68]$	Yes	Context enrichment works; factual regression detected
v5 vs v4	-0.03	$[-0.14, +0.08]$	No	Targeted fix introduces a recall trade-off
v6 vs v4	$+0.30$	$[+0.19, +0.40]$	Yes	Templates resolve trade-off across all four ordinal dimensions

Table 6: Paired-bootstrap overall reason-quality deltas across the three head-to-head studies. Each study uses approximately 391 pairs, anonymised, with the protocol described in this section. "Sig." = significant at $\alpha = 0.05$.

Per-version summary:

- **v4** — added context-lead phrasing with a **`ReasonContext`** record. Gained $+0.55$ overall against v2 but regressed factual accuracy when the model inferred labels from its own judgement.
- **v5** — rewrote the rule to forbid inference and require strict **`ReasonContext`** referencing. Recovered factual accuracy but regressed completeness and relevance.
- **v6** — replaced free-form composition with four constrained templates (Role-Family Mismatch, Cross-Domain, Overqualification, Underqualification). Each template interpolates literal **`ReasonContext`** values into fixed sentence shapes. Significant gains across all four ordinal dimensions. This is the production version.

The full per-version regression story and the per-dimension deltas are in Section H.

5.6 Layer 5 — top-ten ranking quality on the development gold

Layer 5 measures NDCG@10 [5] of the Composite pipeline against the Linear baseline. The measurement is per-(CV, candidate-set), with paired-bootstrap confidence intervals.

The judge. Claude Sonnet 4.6, distinct from the Opus 4.7 judge in Section 5.5. The reason for the smaller tier: each per-pair rating is one ordinal value (no rubric), and I need the ratings at scale (391 pairs across 14 CVs). The cross-vendor configuration is preserved — Anthropic judge, Google Gemini system under test. The judge rates each pair on **`match_quality`** from "no signal of fit" at 0 to "exceptional match" at 10. The judge sees the full CV summary and vacancy analysis but neither pipeline's score, so the rating is a relevance signal independent of the systems under test.

The endpoint and test. For each of the 14 CVs, the candidate vacancy set (28 vacancies, stratified across verdict buckets) is ranked by the Composite and Linear pipelines independently. The top ten of each is scored against the judge's ratings using NDCG@10 with $\log_2$ discounting. The aggregate is the mean of the per-CV differences with a 95% paired-bootstrap CI over the 14 CVs (10,000 resamples).

The headline result. Δ NDCG@10 $= -0.047$, CI $[-0.168, +0.065]$. Cohen's $d = -0.199$. Five Composite wins, nine Linear wins, no ties. The interval crosses zero: the thesis claim that the Composite Judge improves ranking quality is neither supported nor refuted on this gold set.

Both pipelines rank well on the full ordering. Per-CV Spearman correlation is $\rho \approx +0.69$ for each pipeline; Kendall's $\tau \approx +0.49$. The underlying scoring methodology works regardless of whether the Composite Judge stage is enabled.

The aggregate hides a big spread. The Composite pipeline wins where the Judge's qualitative call changes the ordering for the better:

- Senior QA automation CV — $\Delta = +0.331$.
- Frontend CV with React vs Angular candidates — $\Delta = +0.158$.
- Career-switching PM CV — $\Delta = +0.199$.

And loses where the deterministic skill match is already strong and the Judge adds noise:

- Broad full-stack CV — $\Delta = -0.643$.
- Engineering-manager CV — $\Delta = -0.263$.
- Senior PM CV — $\Delta = -0.172$.

The single broad-full-stack outlier (**`17_realformat_swe_fullstack`**) drives most of the negative aggregate. Remove it and the mean drops to -0.001. I keep the outlier in the headline because cutting inconvenient data is itself risky, but I flag it explicitly so the reader can weigh it.

Sensitivity check under the proposed calibration clamp. I apply the ± 0.10 Composite/Linear clamp from Section 5.4 arithmetically to the saved Composite scores. No re-run of the Gemini judge is needed. Recomputing NDCG@10 against the unchanged Sonnet ratings reproduces the published headline within rounding ($\Delta = -0.040$). Under the clamp the mean moves to $\Delta = +0.018$, CI $[-0.054, +0.091]$. The sign flips from negative

to positive but the verdict — directionally non-significant at $\alpha = 0.05$ — does not change. The largest outlier drops from ≈ -0.54 to ≈ 0, which confirms the audit's attribution to the rule-precedence over-penalisation. Production deployment of the clamp is recorded as future work in Section 6.

The layer does not claim that the Composite Judge improves ranking quality on the development gold. It claims that the methodology is sound — independent-vendor judging, anchor-only relevance scale, paired-bootstrap intervals, per-CV variance reported in full, and the largest negative case attributed to a fixable defect. Layer 6 widens the picture on a held-out gold the prompt never saw.

5.7 Layer 6 — held-out validation against a fresh-pair, fresh-vacancy gold set

Layer 6 is the strongest evidence the thesis has on the recruiter-side scoring pipeline. The evaluation is against 398 (CV, vacancy) pairs the production prompt never saw during iteration, judged by Claude Opus 4.7 (a different vendor and a different model tier than the Sonnet 4.6 judge of Layer 5).

Headline numbers:

- Spearman $\rho = 0.648$ (95% CI $[0.580, 0.712]$).
- Quadratic Weighted Kappa $\kappa_Q = 0.638$.
- MAE 1.638 on the native 0–10 scale.
- Per-CV averaged NDCG@5 = 0.822 across the 25 curricula vitae used as queries. I use NDCG@5 here rather than the NDCG@10 of Layer 5 because the held-out gold averages ≈ 16 candidates per CV ($\frac{398}{25}$); a cut at 5 captures the practically actionable top of the list without diluting the metric across most of the candidate pool.

On the **midrange filter** — pairs where the gold rating is non-trivial ($\{4, 6, 8, 10\}$, $n = 110$) — Spearman rises to $\rho = 0.738$ and MAE falls to 1.13.

The layer retires the **selection-circularity** and **vacancy-pool partial leakage** threats named in earlier drafts (the second only partially) and provides the first controlled ablation of the production `ScoringCapService` against the bare language-model composite.

5.7.1 Methodology

Three subsets, each addressing a distinct failure mode.

- **Safety subset** ($n = 189$). Eleven CVs from professional families that never appeared in development — junior product designer, junior DevOps, HR, junior nurse, senior cardiologist, corporate lawyer, senior English teacher, senior accountant, growth marketer, academic humanities, mid security engineer. Each is paired with 17 vacancies sampled across the 657-vacancy pool and biased toward technology roles. The subset tests catastrophic-mismatch detection.
- **Coverage and strong-fit subset** ($n = 209$). The 14 development CVs plus the 11 fresh CVs, paired with vacancies stratified across role-family relationships (same family, adjacent family, cross-family, same-seniority-same-family strong fit). Each pair is novel.

- **Midrange filter** ($n = 110$). A cross-cutting filter over the above two subsets restricted to pairs where the gold rating is in $\{4, 6, 8, 10\}$ — the regime where the recruiter's decision actually happens.

Two-stage gold construction.

1. **Stage 1 — intra-pool held-out.** 131 → 272 pairs drawn from the existing 357-vacancy production pool. This retires **selection circularity** — the prompt did not see these pairings — but not **vacancy-pool partial leakage** because the vacancy text was seen during normalisation prompt iteration.
2. **Stage 2 — fresh-vacancy extension.** 1,543 new vacancies scraped from work.ua and djinni.co using non-tech queries (legal, healthcare, HR, marketing, finance, sales, design, education, security). The first 300 normalised through the same `IVacancyExtractionService` the production API uses, then paired with the 25-CV pool. The result is 126 fresh-vacancy pairs the prompt provably never saw. Vacancy-pool leakage is retired for $\frac{126}{398} = 31.7\%$ of the held-out gold.

Rater protocol. Matches Layer 5 — anchor-only ordinal scale $\{0, 2, 4, 6, 8, 10\}$, mandatory rationale, randomised order. The difference: Opus 4.7 instead of Sonnet 4.6, and the held-out dataset instead of the development gold.

Test-retest reliability. A 30-pair stratified subsample is re-rated by the same Opus checkpoint with a rephrased rubric (anchor descriptions reworded without changing ordinal semantics). Results:

- Spearman $\rho = 0.988$.
- Exact-match agreement 89.7%.
- Within-$\pm$1-anchor agreement 100%.
- MAE 0.21 on the native scale.

The 0.988 self-consistency bounds the upper limit of any system's correlation against this gold: no scoring system can exceed the rater's own self-agreement.

Production-system comparison. The production scoring service ran at prompt version `scoring_monolithic_recruiter_v1_6_source_weighting`. The full set of 398 predictions took 200 seconds at concurrency four and cost about $1.00 in Gemini calls (2.4M input tokens, 109K output).

Baselines. Two classical-IR baselines on the same 398 pairs, in pure .NET inside EvalTool:

- **TF-IDF cosine** over character-level word-boundary n-grams (3–5).
- **BM25 Okapi** (k1=1.5, b=0.75) with per-CV min-max normalisation.

5.7.2 Headline results

Table 7 reports the overall metrics; the per-bin reliability diagram is in Table 9.

Metric	Value	95% CI
Spearman ρ	0.648	[0.580, 0.712]
Kendall τ	0.524	—
Quadratic Weighted Kappa	0.638	—
Mean absolute error (0–10 scale)	1.638	[1.533, 1.745]
NDCG@3 (per-curriculum-vitae avg)	0.813	—
NDCG@5 (per-curriculum-vitae avg)	0.822	—
Expected Calibration Error	0.140	—

Table 7: Held-out evaluation overall metrics. $N = 398$. The confidence intervals are non-parametric 95-per-cent bootstrap intervals over 1000 resamples; the dataset was expanded in two stages (intra-pool 131 ⟶ 272, then fresh-vacancy 272 ⟶ 398) specifically to tighten these intervals and to populate the upper end of the score distribution.

On the aggregate, the lexical baselines are essentially tied with the production system:

- TF-IDF cosine — Spearman $\rho = 0.666$ (within 0.02 of the Gemini 0.648). MAE 1.65.
- BM25 — $\rho = 0.589$. MAE 1.43.
- Gemini production system — $\rho = 0.648$. MAE 1.64.

Taken at face value, the overall numbers do not support the thesis claim that the language-model architecture improves scoring over a classical IR baseline.

But the aggregate hides a structural artefact. Just under half of the gold ($\frac{189}{398}$) is the safety subset where the CVs are professionally distant from the technology vacancies they are paired with. Any scoring system that emits a low number on no-keyword-overlap pairs ranks these pairs correctly, no matter whether it understands the underlying match. The per-subset breakdown (Table 8) separates this regime.

Subset	n	Gemini ρ	Gemini QWK	Gemini MAE
Overall	398	0.648	0.638	1.64
Safety (catastrophic mismatch)	189	0.514	0.652	1.66
Coverage + strong-fit	209	0.694	0.603	1.62
Midrange (gold $\in \{4, 6, 8, 10\}$)	110	**0.738**	**0.651**	**1.13**

Table 8: Per-subset metrics for the Gemini scoring system on $N = 398$ pairs. The midrange filter — the regime where the recruiter's decision between candidates actually happens — is where the Gemini system performs best: Spearman $\rho = 0.738$ and mean absolute error 1.13 on a 0–10 scale. Bold = best column on this subset.

On the midrange filter — the regime where the recruiter's decision actually happens — the Gemini system reaches Spearman $\rho = 0.738$ (+0.090 over the aggregate) with MAE 1.13 (−0.51 relative to the aggregate). On the coverage-and-strong-fit subset, Spearman is 0.694.

The lexical baselines appear competitive only because the safety subset dominates the aggregate. On the regimes where the system is actually used, Gemini extends a clear

advantage. The honest framing: the language-model architecture is not a general upgrade over lexical baselines but a specific upgrade on the recruiter workload, where catastrophic mismatches are already filtered upstream by role-family selection and the remaining pairs are the ones the recruiter is genuinely choosing between.

5.7.3 Calibration and the production-grade mitigation

A scoring system used as a percentage must be calibrated, not just rank-correct. A predicted 65% that corresponds to a true match-quality of 50% is misleading even when the ordering is correct. The held-out evaluation computes the ten-bin reliability diagram and the Expected Calibration Error (Table 9).

Predicted bin	n	Mean predicted	Mean gold	$\lvert\Delta\rvert$
$[0.0, 0.1)$	2	0.043	0.000	0.043
$[0.1, 0.2)$	95	0.163	0.069	0.094
$[0.2, 0.3)$	119	0.253	0.094	0.159
$[0.3, 0.4)$	71	0.344	0.189	0.155
$[0.4, 0.5)$	36	0.445	0.306	0.139
$[0.5, 0.6)$	24	0.540	0.383	0.157
$[0.6, 0.7)$	16	0.656	0.463	0.193
$[0.7, 0.8)$	16	0.747	0.562	0.185
$[0.8, 0.9)$	9	0.843	0.756	0.087
$[0.9, 1.0]$	10	0.932	0.800	0.132

Table 9: Reliability diagram for the raw production scoring on the 398-pair held-out gold. All ten populated bins show the predicted mean exceeding the gold mean; the system is systematically over-confident across the full distribution, with bin-level gaps ranging from 0.043 at the bottom to 0.193 in the upper-mid range. Expected Calibration Error is 0.140 before the post-hoc calibration described below.

The raw composite is systematically over-confident across the full distribution. Every one of the ten populated bins shows the predicted mean exceeding the gold mean, by 4 to 19 percentage points.

The over-confidence is corrected in production by a post-hoc isotonic-regression calibrator. The calibrator is fitted offline on the held-out gold:

1. Pool-Adjacent-Violators isotonic regression and Platt scaling are both evaluated under 5-fold cross-validation.
2. The better of the two is persisted as a portable JSON artefact with fitted knots.
3. The artefact is loaded once at API startup by **`CalibratorLoader`** and applied to every raw composite via **`IScoreCalibrator.Calibrate`** before the recruiter UI displays the percentage.

The chosen calibrator reduces 5-fold cross-validated ECE from 0.140 to 0.027 — an 80.6% relative reduction. Runtime cost is sub-microsecond per call. Calibration as a thesis-named limitation is therefore **mitigated in production**, not merely documented as future work.

5.7.4 Caps on/off ablation

`ScoringCapService` applies rule-based caps for seniority gap, experience gap, language gap, a combined experience+seniority cap, and a domain-alignment subtractor (Section 3.4). The held-out gold gives the first controlled ablation of this architectural decision. The ablation applies the caps offline to the recorded raw composite scores (using each pair's recorded sub-scores and re-derived language-gap signal), then recomputes every metric (Table 10).

Metric	Caps OFF	Caps ON	Δ
Spearman ρ	0.648	0.592	−0.056 (−8.6%)
Kendall τ	0.524	0.495	−0.029
Quadratic Weighted Kappa	0.638	0.632	−0.006 (essentially tied)
Mean absolute error (0–10)	1.638	**1.584**	−0.054 (−3.3%)
NDCG@3	0.813	0.806	−0.007 (preserved)
NDCG@5	0.822	0.799	−0.023 (−2.8%)
Expected Calibration Error	0.138	**0.118**	−0.020 (−14.5%)

Table 10: Side-by-side metrics for the held-out evaluation with **`ScoringCapService`** disabled (Caps OFF, raw language-model composite) and enabled (Caps ON, production behaviour). Caps fired on $\frac{304}{398}$ pairs (76.4%). Bold = better cell per metric. The caps substantially improve calibration (−14.5% Expected Calibration Error, −3.3% Mean Absolute Error) at essentially zero cost to top-of-list ranking (NDCG@3 moves by 0.007, ordinal-rater agreement (QWK) moves by 0.006). The pairwise rank correlation cost (−0.056 Spearman) is the trade-off the design accepts.

The trade-off is interpretable. When the cap fires, distinct raw composite values map onto identical capped values. The mapping destroys pairwise rank information between similarly-bad pairs, so Spearman drops. In return, the capped value is closer to the gold rating on average.

The numbers:

- MAE drops 3.3%.
- ECE drops 14.5%.
- Ordinal agreement (QWK) is preserved within noise.
- Top-of-list ranking (NDCG@3) moves by 0.007 — within the noise floor of the metric.
- The cost is an 8.6% reduction in pairwise rank correlation. The recruiter workflow does not directly expose pairwise correlation.

The architectural decision to apply caps is empirically justified on the held-out evidence. Combined with the isotonic calibrator from Section 5.7.3, total ECE in production is reduced from the raw 0.140 to under 0.03.

5.7.5 Limitations of the held-out evaluation

The held-out evaluation resolves several threats but introduces others:

- **Single-rater family.** All 398 ratings come from Claude Opus 4.7. The cross-vendor configuration is stronger than Layer 5′s intra-vendor one, but it remains intra-family on the rater side. A multi-rater ensemble adding an OpenAI GPT model and a Mistral or DeepSeek model with inter-rater Krippendorff's α [10] would yield a published-grade inter-vendor methodology.
- **Distribution skew of the safety subset.** About half of the gold ($\frac{189}{398} = 47.5\%$) is in the safety subset where ratings concentrate at 0 or 2. The aggregate metrics partly measure catastrophic-mismatch detection — arguably easier than the recruiter's actual workload. The midrange and coverage subsets isolate that workload.
- **Prompt-version ablation.** The pre-**`v1_6`** prompt history was not preserved in version control. The held-out evaluation reports the production prompt only. It does not include a controlled **`v1_5`** vs **`v1_6`** comparison.

5.8 Threats to validity

Table 11 summarises threats that have been retired or mitigated; Table 12 lists the threats that remain active, with the smallest resolution path for each.

Threat	Status
Selection circularity (Layer 3, 5)	Retired by Layer 6 — held-out gold the prompt never saw
Vacancy-pool partial leakage (Layer 6 Stage 1)	Retired for 31.7% of pairs by Stage 2 fresh-vacancy extension
Calibration of displayed percentage	Mitigated in production — isotonic calibrator reduces ECE from 0.140 to 0.027
Composite Judge over-penalisation	Sensitivity-checked under the proposed ± 0.10 clamp at **`GeminiCompositeJudgeService`**
Prompt injection through untrusted CV / vacancy text	Partially mitigated — XML-tagged input wrapping, anti-injection preamble, length caps; schema validation + cap service + no agentic actions bound the blast radius

Table 11: Threats retired or mitigated by Layer 6 or by production-deployed code. The selection-circularity and vacancy-pool-leakage threats are retired by the held-out gold (Section 5.7); the calibration threat is mitigated by the production isotonic calibrator; the Composite Judge over-penalisation has a one-line fix that the post-hoc sensitivity check in Section 5.6 quantifies; prompt-injection mitigations follow OWASP LLM01.

Threat	Current state	Resolution path
CV sample size on Layer 5 (n=14)	Interval crosses zero	Extend gold to ≥ 30 CVs per role family
Outlier on Layer 5 (broad full-stack CV)	Drives most of the negative aggregate	Reported openly; resolved by the clamp + a larger gold
Single-vendor judge family (Anthropic)	Layer 4 Opus, Layer 5 Sonnet, Layer 6 Opus	Add OpenAI GPT and Mistral judges; report inter-vendor Krippendorff's α
Single-annotator Layer 1, 2 gold	No inter-annotator coefficient	Independent annotator on stratified subset
Judge-in-the-loop Layer 4 prompt iteration	Partially defended by structural-not-stylistic v6 gains	Human or different-vendor judge on v6 vs v4
Surface-form sub-score calculators	Composite Judge is the partial fix	Embedding-based semantic similarity on the skill axis
No production A/B on v6	Reason-quality predicts satisfaction by literature precedent only	A/B against v4 with real-user click-through

Table 12: Active threats, the current state of each, and the smallest resolution path. Each row points to discrete future work recorded in Section 6.4.

Three changes in combination would resolve the central thesis claim on Layer 5 definitively:

1. Extend the CV gold set to at least thirty per role family, to support per-family inference.
2. Deploy the Composite Judge calibration clamp, to remove the largest identifiable confound.
3. Add a multi-vendor judge ensemble, to establish that the result is not an artefact of the single judge family.

None of the three is conceptually difficult. All three are recorded as future work in Section 6.

5.9 Chapter summary

The six-layer methodology is the result of a mid-thesis pivot away from a structurally biased score-error endpoint.

Layers 1 and 2 are frozen at per-field 0.896 (25 CVs) and 0.804 (355 of 357 vacancies).

Layer 3 stays as a calibration diagnostic. The 0.086 Linear-vs-Composite gap is explained by cap asymmetry, selection circularity, and a Composite Judge over-penalisation with a one-line fix.

Layer 4 shows a significant improvement of v6 over v4 across all four ordinal rubric dimensions: overall $\Delta = +0.30$, CI $[+0.19, +0.40]$, $N \approx 391$, judge Claude Opus 4.7.

Layer 5 is inconclusive on 14 CVs: Δ NDCG@10 $= -0.047$, CI $[-0.168, +0.065]$, judge Claude Sonnet 4.6. The post-hoc clamp shifts the headline to $+0.018$ but does not change the verdict.

Layer 6 reports the production system against a 398-pair held-out gold the prompt never saw. Spearman $\rho = 0.648$, NDCG@5 = 0.822, midrange-filter Spearman $\rho = 0.738$. The controlled cap-on/off ablation and the production-deployed isotonic calibrator (ECE $0.140 \rightarrow 0.027$) jointly justify the cap-and-calibrate architecture and retire the selection-circularity and vacancy-pool-leakage threats.

6 Conclusion

6.1 Project summary

Vakansio is a deployed job-matching service for the Ukrainian IT market.

The system pulls vacancies from seven sources, turns them and the candidate's CV into structured JSON, and scores every (CV, vacancy) pair. The score comes from a deterministic seven-axis sub-score calculator refined by a Gemini Composite Judge. The result is shown with bilingual English and Ukrainian explanations.

The deployed instance runs on a single EC2 t3.micro behind Caddy with managed PostgreSQL, S3, and CloudFront. It is reachable at dsus1dizgh006.cloudfront.net. The implementation follows the Clean Architecture decomposition of Martin [2] with the SOLID principles mapped to concrete code artefacts.

The evaluation decomposes the pipeline into six layers, following the multi-axis principle of HELM [6]:

- **Layers 1 and 2** — frozen at per-field 0.896 (25 CVs) and 0.804 (355 of 357 vacancies).
- **Layer 3** — kept as a calibration diagnostic after two audits identified structural artefacts.
- **Layer 4** — a significant improvement of v6 over the v4 baseline across all four ordinal rubric dimensions ($\Delta = +0.30$, CI $[+0.19, +0.40]$, $N \approx 391$, Claude Opus 4.7).
- **Layer 5** — directionally non-significant on 14 CVs (Δ NDCG@10 $= -0.047$, CI $[-0.168, +0.065]$, Claude Sonnet 4.6).
- **Layer 6** — 398 pairs the prompt never saw, Claude Opus 4.7. Spearman $\rho = 0.648$, NDCG@5 = 0.822, midrange-filter Spearman $\rho = 0.738$ on the recruiter-workload regime. A controlled cap-on/off ablation empirically justifies the production `ScoringCapService`. A post-hoc isotonic-regression calibrator reduces ECE from 0.140 to 0.027 under 5-fold cross-validation and ships with the API, so the displayed match percentage in production is calibrated.

6.2 Comparison with the initial objectives

I revisit the four objectives stated in Section 1.2 below.

Objective one — a working public service. The system runs on the open internet, supports user registration and login, accepts a CV upload as a PDF, fetches vacancies from seven external sources, and returns a personalised list with verdicts and bilingual explanations. The deployment is documented in Section 3.7 and Section 4.6. **Met.**

Objective two — a disciplined software architecture. The code base is organised as four .NET projects whose project-reference graph enforces the Clean Architecture dependency rule at build time. `Domain/Domain.csproj` declares zero package references; Application references only Domain; Infrastructure references Application (and Domain transitively); API references Application and Infrastructure. The SOLID principles are mapped to concrete artefacts in Section 3.3 and elaborated in Section 4.2, Section 4.3, and Section 4.4. **Met.**

Objective three — measurable match quality. The six-layer methodology in Section 5.2 is the architectural answer. The empirical answer is mixed and honestly so. Layer 4 is a clear v6-over-v4 win across all four ordinal dimensions. Layer 5 is inconclusive on 14 CVs at the standard significance level; both pipelines reach Spearman $\rho \approx 0.69$ against the cross-vendor ratings. Layer 6 fully retires the selection-circularity threat and partially retires the vacancy-pool-leakage threat (31.7% of pairs) with Spearman $\rho = 0.648$ on 398 held-out pairs and a controlled `ScoringCapService` ablation; the production-deployed isotonic calibrator mitigates the displayed-percentage calibration threat. **Methodology met; empirical answer partial and transparently reported.**

Objective four — transparency of every LLM call. Every Gemini call writes a row to the `GeminiCostLog` table through the per-stage cost accumulator in `Application/Common/Diagnostics/CostBreakdown.cs`. The accumulator records the stage, the call count, the input and output token counts, the wall-clock duration, and the dollar cost computed from the current Gemini 2.5 Flash published prices. An admin endpoint exposes the data for offline querying. **Met.**

6.3 Encountered difficulties

Four difficulties shaped the chapters above.

Mid-thesis methodology pivot. The original primary endpoint was score MAE. Two audits found structural problems with it — cap asymmetry, selection circularity, and Composite Judge over-penalisation in the harness (Section 5.4). I moved score MAE to a calibration diagnostic and adopted top-ten NDCG against independent-vendor relevance ratings as the new primary endpoint. I report the pivot as a methodological contribution, not as a hidden failure.

v6.7.8 contract drift. A back-end change to string-enum serialisation broke the front-end source filter and the verdict-label vocabulary at the same time. The failure was silent: the UI showed an empty list with no error. Both regressions were fixed through a single Domain-layer extension method (Section E). The episode is the strongest argument for the single-source-of-truth maps on the front-end.

Composite Judge calibration defect — identified but not deployed. A rule-precedence override in the Judge prompt lets the Composite Judge step outside its bounded-adjustment band. The post-hoc sensitivity check in Section 5.6 quantifies the effect. The methodology's verdict on the ranking layer holds whether the fix ships or not. Shipping the fix is the first item in the next section.

Single-vendor judge at submission time. All three judge runs are from Anthropic — Claude Opus 4.7 (Layer 4 and Layer 6), Claude Sonnet 4.6 (Layer 5). This was a scope choice forced by API access at submission time. Adding a second and third vendor is the second item in the next section.

6.4 Future perspectives

Seven extensions would tighten the empirical findings. Each is a discrete piece of work with a defined cost and a known location in the code or gold set.

1. Production deployment of the Composite Judge bounded-adjustment clamp. This is a separate mechanism from the isotonic calibrator already shipped in Section 5.7.3. The clamp is a single-file change at the `GeminiCompositeJudgeService` boundary that enforces the ± 0.10 band between the Linear and Composite scores. Requires re-running the ranking pipeline on the existing 14 CVs to produce clamped results, instead of the post-hoc sensitivity check the chapter currently reports.

2. Multi-vendor judge ensemble. Add an OpenAI model at its strongest tier and a third vendor (for example Mistral Large). Report inter-judge Cohen's κ across vendor families. Confirm that the reason-quality and ranking-quality results are not artefacts of the Anthropic family.

3. Gold-set extension for per-family inference. Extend the Layer 5 CV gold to at least 30 per role family (Engineering, Data, Product, Design). This would tighten the aggregate interval enough for per-family verdicts.

4. Independent inter-annotator agreement. An external annotator on a stratified subset of the match-quality ratings. This would establish external validity and complement the judge self-consistency check.

5. A/B production validation of v6. Compare v6 against v4 under real-user click-through and apply rates. This closes the gap between reason-quality and direct user satisfaction.

6. Reference-free factuality evaluation. Activate the MiniCheck integration [11] (already present in Infrastructure as `MlApiFactualityService`) as a third reason-quality method alongside rubric-based judging and programmatic checks.

7. Prompt-injection hardening beyond OWASP LLM01. The current XML-tagged input wrapping, anti-injection preamble, and per-stream length caps (Section 3.8) are best-effort against adversarial CVs and vacancy descriptions. The next step is:

- Integrate a model-vendor content-moderation API on every untrusted input.
- Add a regex pre-filter for known injection phrases ("ignore previous instructions", "you are now", "disregard the above").
- Commission an external red-team pass against the production pipeline.

6.5 Final reflection

The thesis delivers five things:

- A deployed system.
- A defensible layered evaluation methodology.
- A significant positive result on reason-text quality.
- An inconclusive result on the development-set ranking endpoint, with the conditions for resolution stated explicitly.
- A held-out evaluation that fully retires the selection-circularity threat and partially retires (31.7% of pairs) the vacancy-pool-leakage threat, alongside the production-deployed isotonic calibrator that mitigates the displayed-percentage threat. Against the same gold the lexical baselines tie on aggregate, but the language-model architecture extends a clear advantage on the midrange recruiter regime.

The methodology stands regardless of the empirical verdict on ranking. The open question is what an inter-vendor ensemble and a larger gold set would settle.

The methodology pivot and the calibration audits in Section 5 were the hardest parts of the work. They shaped the system more than any single architectural decision.

Glossary

Architecture

CQRS – **Command-Query Responsibility Segregation**: A pattern that separates request handlers into command handlers (which change state) and query handlers (which only read state). Vakansio uses MediatR to implement the pattern; every use case is a separate handler class with one Handle method.

SOLID – **Single Responsibility, Open–Closed, Liskov Substitution, Interface Segregation, Dependency Inversion**: Five class-design principles articulated by Martin (2017). The thesis maps each principle onto a concrete code artefact in the Vakansio source tree and treats the mapping as a non-negotiable evaluation criterion.

Domain

CEFR – **Common European Framework of Reference for Languages**: A six-level language-proficiency ladder (A1, A2, B1, B2, C1, C2) used to score the language-match axis in the Vakansio sub-score calculator.

Evaluation

HELM – **Holistic Evaluation of Language Models**: A multi-axis decomposed evaluation framework for language-model systems that grades distinct artefacts on methodology appropriate to each artefact's data type; the framework the thesis adopts for its five-layer evaluation decomposition.

LLM-as-Judge – **Large-Language-Model-as-Judge**: A reference-free evaluation methodology that uses a language model to score another model's output on a rubric, established by Zheng et al. (2023). The thesis uses an Anthropic Claude judge on the output of the Google Gemini production pipeline to avoid same-vendor bias.

MAE – **Mean Absolute Error**: The mean of the absolute differences between predicted and reference values. Used in the Layer 3 score-tracking diagnostic of the evaluation chapter to compare pipeline scores against a deterministic ideal.

MiniCheck – **MiniCheck factuality model**: A reference-free claim-level factuality checker proposed by Tang et al. (2024). Integrated into the Vakansio Infrastructure layer as MlApiFactualityService but not evaluated in this thesis; recorded as future work.

NDCG@10 – **Normalised Discounted Cumulative Gain at rank 10**: A ranking-quality metric that compares a system's top-ten ordering against an ideal ordering produced by sorting the same items by ground-truth relevance, with positions discounted by a logarithm. Reported as the primary endpoint of the ranking-quality layer in the evaluation chapter.

Infrastructure

AWS – **Amazon Web Services**: The cloud platform on which Vakansio is deployed: an EC2 t3.micro for the API, a managed RDS PostgreSQL for the database, an S3 bucket

for curriculum-vitae files, a second S3 bucket and CloudFront for the front-end, and the Systems Manager Parameter Store for secrets.

EC2 – **Elastic Compute Cloud**: Amazon's virtual-machine service. Vakansio runs the API on a single t3.micro instance in the eu-central-1 region.

RDS – **Relational Database Service**: Amazon's managed relational-database offering. Vakansio uses RDS for PostgreSQL with TLS-required connections.

SSM – **Systems Manager Parameter Store**: Amazon's managed key-value store used by Vakansio for production secrets (Gemini API key, JSON Web Token signing key, database connection string).

Machine Learning

LLM – **Large Language Model**: A neural network trained on large text corpora to predict the next token; in this thesis used for vacancy normalisation, candidate-vitae normalisation, the Composite Judge stage of the scoring pipeline, the bilingual reason text, and the Layer 4 and Layer 5 judging in the evaluation chapter.

Pipeline

Composite Judge – **Composite Large-Language-Model Judge**: The Gemini 2.5 Flash call in the Vakansio scoring pipeline that refines the deterministic linear score against a per-family calibration rubric. Distinct from the Composite Judge calibration buckets used to label verdicts.

skip-band – **Composite Judge skip-band**: The optimisation in the Vakansio scoring pipeline that bypasses the Composite Judge for pairs whose deterministic linear score falls outside the band 0.30 to 0.85 and that carry no active anti-flag, on the grounds that the deterministic signal is already decisive. Observed to reduce judge invocations by approximately thirty to forty per cent.

Security

JWT – **JSON Web Token**: A signed, URL-safe token format used for stateless authentication in the Vakansio API; the token is issued at login and presented on each subsequent request through the Authorization header.

Statistics

paired bootstrap – **Paired bootstrap confidence interval**: A non-parametric confidence-interval procedure that resamples the per-pair differences with replacement and takes the empirical quantiles of the resampled means. Used throughout the Layer 4 and Layer 5 statistical inference at ten thousand resamples.

Cohen's d – **Cohen's d**: A standardised effect-size measure equal to the mean of the paired differences divided by the standard deviation of the differences. Used in the Layer 5 paired-bootstrap analysis of the evaluation chapter alongside the headline confidence interval.

***Krippendorff α* – Krippendorff's alpha**: An inter-rater reliability coefficient that handles ordinal, interval, and ratio data and corrects for chance agreement. Used in the Layer 4 pilot of the evaluation chapter as a judge self-consistency check.

User Interface

***tier-based disclosure* – Tier-based progressive evidence disclosure**: The front-end pattern that classifies evidence chips into three tiers — brand-and-tool names (Tier 1), hyphenated and multi-word skills (Tier 2), and generic concepts (Tier 3) — and renders Tier 3 behind an expansion link to keep the chip list scannable.

Bibliography

[1] S. Danziger, J. Levav, and L. Avnaim-Pesso, "Extraneous factors in judicial decisions," *Proceedings of the National Academy of Sciences*, vol. 108, no. 17, pp. 6889–6892, 2011, doi: 10.1073/pnas.1018033108.

[2] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[3] Court of Justice of the European Union, "CV-Online Latvia SIA v. Melons SIA." [Online]. Available: https://curia.europa.eu/juris/document/document.jsf?docid=242047&doclang=EN

[4] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 734–749, 2005, doi: 10.1109/TKDE.2005.99.

[5] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Transactions on Information Systems*, vol. 20, no. 4, pp. 422–446, 2002, doi: 10.1145/582415.582418.

[6] P. Liang *et al.*, "Holistic Evaluation of Language Models," *Transactions on Machine Learning Research*, 2023, Accessed: June 03, 2026. [Online]. Available: https://arxiv.org/abs/2211.09110

[7] L. Zheng *et al.*, "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," in *Advances in Neural Information Processing Systems*, 2023. Accessed: June 03, 2026. [Online]. Available: https://arxiv.org/abs/2306.05685

[8] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications 2025 — LLM01:2025 Prompt Injection." [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/

[9] F. P.-W. Lo *et al.*, "AI Hiring with LLMs: A Context-Aware and Explainable Multi-Agent Framework." Accessed: June 04, 2026. [Online]. Available: https://arxiv.org/abs/2504.02870

[10] K. Krippendorff, "Computing Krippendorff's Alpha-Reliability," *Departmental Papers (ASC). 43, University of Pennsylvania*, 2011, Accessed: June 10, 2026. [Online]. Available: https://repository.upenn.edu/asc_papers/43

[11] L. Tang, P. Laban, and G. Durrett, "MiniCheck: Efficient Fact-Checking of LLMs on Grounding Documents," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 2024. [Online]. Available: https://arxiv.org/abs/2404.10774

A End-to-end worked example

In this appendix I take one CV and one vacancy, and run them through all six layers of the evaluation. The point is to show how the numbers actually work on a real pair. I picked this pair because the Composite pipeline wins on it clearly — exactly the kind of win the Composite Judge is meant to add. The aggregate result across all 14 CVs is still inconclusive (Section 5.6), but the layered methodology lets us see per-CV wins like this one instead of hiding them inside an average.

The pair I picked. The CV is **`15_qa_senior_automation`** — a senior QA automation engineer with almost nine years of experience, English at B2, and a stack of Python, Selenium, Playwright, Cypress, and JMeter. The vacancy is **`e66e80c0`** — a senior API and back-end performance/load testing role asking for exactly that Python + JMeter stack. I read the pair myself and put it at the top of the candidate set. The Sonnet judge agrees: it rates this pair 10/10 on **`match_quality`**.

Layer 1 — what the pipeline understood about the CV. From the gold normalisation:

- Seniority: senior
- English: B2
- Experience: about 107 months (8 years, 11 months)
- Skills: Python, Selenium, Playwright, Cypress, pytest, JMeter

The production pipeline reproduces these fields with roughly the same accuracy as the Layer 1 average reported in Section 5.3.

Layer 2 — what the pipeline understood about the vacancy. From the gold normalisation:

- Required seniority: senior
- Required English: B2
- Must-have skills: Python, API testing, JMeter, Gatling

The production normalisation reproduces this shape with roughly the same accuracy as the Layer 2 average in Section 5.3.

Layer 3 — the score. The Linear pipeline returns **0.642** (just the weighted sum of the seven sub-scores). The Composite pipeline returns **0.680** — the Composite Judge nudges the score up a little because the qualitative match is strong. The deterministic ideal from the gold inputs (Section 5.4) is in the same area. The small Linear → Composite step on this pair is exactly the kind of refinement the Judge is meant to make.

Layer 4 — the explanation the pipeline writes. The v6 prompt produces a bilingual reason text. It opens with **"Strong match: senior automation experience with the requested stack"**. Then three short sections:

- **Strengths:** Python, JMeter, API testing, Selenium (cross-context familiarity).
- **Gaps:** Gatling — but it sits close to JMeter, so it's a minor gap.
- **Recommendation:** apply.

The Ukrainian version keeps brand names (Python, JMeter, Gatling, Selenium) in Latin script, as the **`uk_term_preservation`** rule asks. The Opus judge rates this text at or near the maximum on every rubric dimension.

Layer 5 — where the pipeline ranks the pair. I report two NDCG@10 numbers. **NDCG@10** is a ranking-quality metric that gives more weight to relevance at the top of the list than at the bottom. **Δ** is the difference between Composite NDCG@10 and Linear NDCG@10 — a positive Δ means Composite ranks better on this CV.

Like-for-like comparison. Only the 19 candidate pairs where both pipelines produced a clean score. (Composite had 11 scoring failures on the original 30-pair set.) Composite puts the chosen vacancy at **rank 5**; Linear puts it at **rank 8**. Composite NDCG@10 = **0.564**. Linear NDCG@10 = **0.442**. Per-CV $\Delta = +0.122$.

Full coverage. Use every candidate each pipeline produced. Linear ranks the gold-ten vacancy at **rank 11** — outside the top ten — so its NDCG@10 drops to **0.230**. Composite NDCG@10 stays at **0.561**. Per-CV $\Delta = +0.331$.

The two numbers measure different things: how the two pipelines compare on the same set, and how each pipeline handles its full set. The $+0.331$ figure is the one that feeds the aggregate reported in Section 5.6.

Composite rank	Composite score	Gold match_quality	Note
1	0.808	4	Composite ranks too high
2	0.780	8	
3	0.730	8	
4	0.730	8	
5	0.680	10	The pair we traced above
6	0.660	4	
7	0.630	4	
8	0.600	4	
9	0.588	4	
10	0.588	2	

Table 13: Composite top-ten ranking for the CV **`15_qa_senior_automation`** on the like-for-like subset. Our traced pair is at rank 5. The Linear pipeline puts the same pair at rank 8 on the same subset.

Layer 6 — where the pair sits in the held-out gold. The same vacancy is also one of the 398 held-out pairs scored against the Claude Opus 4.7 gold (Section 5.7). The held-out gold rates this pair at 10/10; the production system's calibrated score after the isotonic regressor (Section 5.7.3) is 0.81, against the gold mean of 0.80 in the corresponding bin — a per-pair absolute error of 0.01, well below the headline MAE on the native scale.

What this example shows. On this one pair, the Composite Judge does its job: it pushes a strongly relevant candidate higher than the deterministic calculator alone would. Layer 5 shows a clear positive Δ. The aggregate across all 14 CVs is still inconclusive — but

an inconclusive aggregate does not mean every CV behaves the same way. The layered methodology shows us the per-CV picture instead of hiding it inside an average.

B Legal and commercial deployment

This appendix expands on the legal landscape and commercial deployment options sketched in Section 2.3. The summary table of per-source status and the paragraph that motivates the channel mix remain in the main text; the per-precedent reasoning and the per-provider pricing live here.

Civil database right under Ukrainian law. The 2022 Law of Ukraine on Copyright and Related Rights (No. 2811-IX), in force since 1 January 2023, introduced a sui generis database right that protects, for fifteen years, databases the maker has invested substantially in. Sites such as Work.ua and Robota.ua are likely to qualify, although no Ukrainian court has yet ruled on this in the context of job-board aggregation. The biggest risk for a Ukrainian company aggregating job listings is therefore not criminal liability under Article 361 of the Criminal Code, nor GDPR applied to vacancy text — that text, without personal identifiers, is not personal data — but civil action by a database maker whose investment is adversely affected.

Controlling European Union precedent. The Court of Justice ruling in CV-Online Latvia v. Melons [3] addressed exactly this fact pattern. The operator of a Latvian meta-search engine indexing job listings was found to "extract and re-utilise" the database of CV-Online; the Court held that such acts may be prohibited by the database maker only where they adversely affect the investment in obtaining, verifying, or presenting the database contents. The ruling is directly applicable to the Vakansio situation and is the strongest external reference point for the partnership-channel approach proposed below.

CV files and personal data. CV files contain personal data and trigger the full set of obligations under the Ukrainian Law on the Protection of Personal Data and, where the candidate is in the European Union, the General Data Protection Regulation. These obligations include an explicit lawful basis for the processing, retention limits, and the candidate's right to access and deletion. Vakansio honours them through explicit consent at upload time, scope limitation (CVs are used solely for matching), and deletion on account closure with a transactional cascade through the database and a best-effort S3 object deletion.

Recent scraper litigation as context. In December 2022 the hiQ Labs v. LinkedIn proceedings ended with a stipulated judgment under which hiQ paid USD 500,000 and accepted a permanent injunction. This outcome stood despite an earlier Ninth Circuit preliminary-injunction affirmance holding that scraping publicly accessible data likely does not violate the United States Computer Fraud and Abuse Act. In January 2025 LinkedIn filed suit against the LinkedIn data provider Proxycurl; Proxycurl ceased operations on 4 July 2025 under a consent-based permanent injunction. These outcomes indicate that, while scraping public data is generally not a criminal matter, civil liability arising from breach of platform terms of service is real and has materialised against well-funded specialised providers.

Paid replacement options for LinkedIn data. Direct scraping of LinkedIn is not a sustainable commercial channel; replacement is therefore required. Apify, Bright Data,

and JSearch make the data available through scraping-as-a-service offerings at approximately USD 100–300 per month for the volume needed by a small commercial deployment; these providers outsource the technical collection but do not assume contractual liability for breaches of LinkedIn terms of service. Coresignal positions itself as a licensed data vendor and provides equivalent coverage at approximately USD 1,000 per month under a negotiated commercial agreement, which is the option offering the most clearly delineated legal basis for a commercial launch. The LinkedIn Talent Solutions partner programme is, in practice, not accessible to a pre-revenue startup.

Path to commercial deployment. The mix I propose for a market version of Vakansio is therefore as follows. The free legitimate channels — the DOU RSS feed and the Jooble API — remain unchanged. A new channel is added through the OLX Partner API, which the prototype does not currently use. A parallel applicant-tracking-system aggregation channel is built on top of the free public boards exposed by Greenhouse, Lever, Workable, and SmartRecruiters; this channel covers Ukrainian-relevant employers such as Grammarly, GitLab, MacPaw, Kyivstar, and Ajax Systems, and brings several thousand additional openings without per-listing cost. Direct scraping of Djinni, Robota.ua, and Work.ua is replaced by partnership-based access through the data-licensing routes described above. Direct scraping of LinkedIn is replaced by a licensed paid provider — Coresignal for a production launch on a clearly delineated legal basis, or Apify or Bright Data for a minimum-viable product.

The Vakansio implementation evaluated in this thesis is a capstone prototype. The scraping channels currently in use are appropriate for that purpose: they access only publicly visible listings, do not bypass authentication, respect robots.txt where it expresses an explicit position with respect to the public listing pages, and operate at human-level rates. Commercial deployment would replace those channels with the official, partnership, and paid options described above.

C | Technology stack decision log

This appendix expands the per-decision rationale summarised in Section 3.2. The main text presents a compact table; the per-alternative reasoning lives here.

Back-end runtime: .NET 8 over Node.js, Go, and Python. ASP.NET Core has a mature dependency-injection container, a configuration-binding system, and a middleware pipeline that align well with the Clean Architecture and Options patterns required by Section 2.6. C# offers a strong static type system with nullable reference types, which makes the interface contracts at layer boundaries enforceable at compile time. Node.js with TypeScript was the closest second on type safety, but its DI ecosystem is fragmented and the configuration-binding story is weaker. Go has a lean runtime but lacks ergonomic generics for the CQRS handlers that MediatR provides on .NET. Python with FastAPI handles HTTP fluently but lacks compile-time enforcement of cross-layer interface contracts. Practical experience with the .NET ecosystem also kept the development pace realistic for a solo project. The project pinned to .NET 8 (LTS, support window through November 2026) rather than the newer .NET 10 LTS (November 2025) because the pipeline was already a year into development by the time .NET 10 shipped; the upgrade carries no functional benefit for the libraries the project uses and would have introduced migration risk for no production value.

Front-end framework: React with TypeScript on Vite over Vue 3 and Next.js. React has the largest npm download share among SPA frameworks and the deepest set of companion libraries — Zustand, TanStack Query, React Router — each of which Vakansio uses. TypeScript gives the same compile-time contract enforcement on the front-end that C# gives on the back-end. Vite provides fast hot-module reloading during development and a clean production build with no `webpack.config.js` to maintain. Vue 3 is a credible alternative on the same axes, but the candidate-facing ecosystem for state management and server-data caching is narrower. Next.js was rejected because the user interface is stateful and client-driven; server-side rendering would add operational cost without a search-engine-optimisation benefit on an authenticated application.

Database: PostgreSQL via Amazon RDS over MySQL and MongoDB. First-class `jsonb` columns let normalised vacancy and CV representations live alongside relational data without a separate document database. Window functions, full-text search, and advisory locks cover the analytical queries that the cost-ledger and evaluation paths require. Amazon RDS removes the operational overhead of running and backing up the database, which matters for a solo project. MySQL was rejected because it lacks native `jsonb` support, which the normalisation cache relies on. MongoDB was rejected because the data model does not require schema-less storage, and PostgreSQL's strong-consistency guarantees simplify the cache-invalidation reasoning described in Section 3.5. The deployment pins the major version at PostgreSQL 16 — one major behind the current 17/18 — because all features the project uses (`jsonb`, Pgvector, the migration set, advisory locks) are stable on 16 and a major-version upgrade carries no functional benefit for this workload while it would force a coordinated EF Core provider revalidation.

Large language model: Gemini 2.5 Flash over GPT-4, Claude, Groq, and self-hosted open weights. Cost per token is materially lower than GPT-4 and Claude at the scale of the pipeline. Native JSON output simplifies parsing. Latency is acceptable at the volumes seen by Vakansio. The Composite Judge benefits from a model that can follow a long calibration rubric reliably and return scores within a strict numerical range. A locally hosted open-weight model was rejected because the t3.micro deployment target does not have enough memory or compute to serve a model with comparable instruction-following quality, and a separate inference host would absorb the cost saving the cheaper model provides.

Cloud platform: Amazon Web Services over Google Cloud, Azure, and DigitalOcean. My prior working experience with the AWS console and command-line tooling kept the deployment timeline realistic. The twelve-month free-tier programme covers one t3.micro EC2 instance, twenty gigabytes of RDS storage, and modest S3 traffic — together this covers the full production cost for a solo capstone project. CloudFront ensures low front-end latency for users across Ukraine without operating a separate content-delivery network.

Supporting libraries. I chose Entity Framework Core over Dapper for its first-class migration tooling and its tight integration with the .NET DI container. I chose MediatR over a hand-rolled dispatcher for the ecosystem of pipeline behaviours (validation, logging, retry) available through its abstraction. I chose BCrypt.Net-Next for password hashing because it implements the standard work-factor parameter and has a long track record of independent audits.

D Operations runbook

This appendix expands on the deployment, container-build, and operations details summarised in Section 4.6.

Container build stages. `Dockerfile` is a three-stage build that produces a slim multi-architecture runtime image (`linux/amd64` and `linux/arm64`):

1. `sdk-restore` — copies only the four `.csproj` files into the build context and runs `dotnet restore`. The layer cache invalidates only when the package graph changes.
2. `sdk-build` — copies the rest of the source and runs `dotnet publish -c Release --no-restore /p:UseAppHost=false` into `/app/publish`.
3. **Runtime** — starts from `mcr.microsoft.com/dotnet/aspnet:8.0`. Installs `curl` for the health probe. Creates a non-root `app` user. Copies the published output. Exposes port 8080. Declares a container-level `HEALTHCHECK` that polls `/health` every 30 seconds.

Compose topology. `docker-compose.production.yml` defines two services on a private bridge network `vacancies-net`:

- `caddy` — the Caddy 2.8-alpine image. Ports 80, 443, and 443 UDP exposed to the host. Two persistent volumes carry the Let's Encrypt account material and configuration.
- `api` — the `vacancies-api:latest` image. Port 8080 exposed inside the network only.

A 700 MB memory cap protects the t3.micro host (which has ~900 MB of usable memory) from a runaway .NET allocation. The environment variable `BackgroundWorkers__EnableVacancyAnalysis=false` explicitly disables the vacancy-analysis worker. An always-on background worker would burn $3–$6 per day in Gemini calls with no user traffic. Caddy's `depends_on: api { condition: service_healthy }` makes sure the reverse proxy does not start serving traffic before the API container reports healthy.

Production resource identifiers.

- AWS account 621120073581, region eu-central-1.
- EC2 instance i-037e7739d050849bf (t3.micro, Amazon Linux 2023).
- Elastic IP 18.199.45.202 (allocation eipalloc-0587fc74a54069b7a).
- RDS instance vacancies-db (PostgreSQL 16, db.t3.micro). Endpoint: vacancies-db.cluk4u60w0a2.eu-central-1.rds.amazonaws.com.
- Front-end bucket vacancies-frontend-prod, served through CloudFront at dsus1dizgh006.cloudfront.net.
- SSM secrets under `/vacancies/prod/*`.
- Domain vakansio.online via Namecheap, expiry 1 June 2027. Caddy obtains the Let's Encrypt certificate for `api.vakansio.online` automatically.

Stop and restore scripts. Three PowerShell scripts manage the AWS lifecycle between demos:

- **`scripts/stop-aws.ps1`** — stops EC2 and RDS. Keeps the Elastic IP, S3 buckets, SSM parameters, security groups, IAM artefacts, and snapshots. Stopped-state cost is about ten cents per day.
- **`scripts/start-aws.ps1`** — restarts both. Re-authorises the SSH ingress rule for the current operator IP.
- **`scripts/deploy-restore.ps1`** — an idempotent full restore from EC2 AMI snapshot ami-012d676c63c268a30 and RDS snapshot vacancies-db-defense-snapshot-001. Allocates a fresh Elastic IP and updates the SSM ConnectionString. State persists to **`scripts/deploy-state.json`** so subsequent invocations reuse the same identifiers.

Health probes and rollout safety. The Docker **`HEALTHCHECK`** polls **`/health`** (liveness only, no database touch) with **`curl`** every 30 seconds, with a 5-second timeout, 3 retries, and a 30-second start period. Caddy refuses to start until that probe reports healthy through **`depends_on: service_healthy`**. The deeper **`/health/ready`** endpoint performs the database round-trip — use it when manually confirming a freshly rolled API container is ready to serve traffic. The CI/CD workflow runs **`/health`** through SSH into the container, not through Caddy's TLS layer. This bypasses the SNI-mismatch failure that an IP-addressed curl would otherwise produce.

E Engineering narratives

This appendix preserves the two engineering narratives summarised in Section 4.7.

Frontend–back-end contract drift fix (v6.7.8).

The most consequential bug I encountered during the second half of the project was a silent empty-result failure on the front-end search page after a routine back-end change. The change itself was a single line at `API/Program.cs:60` — adding `new JsonStringEnumConverter()` to the JSON serialiser's converter collection. After the change, the API emitted enum members by name (`"Djinni"`, `"StrongMatch"`) instead of by their underlying ordinal values. The motivation: Swagger readability and the front-end's preference for typed string unions over ordinal magic numbers.

The change broke the front-end in two related but independent places. Both fixed under v6.7.8.

Regression 1 — source filter. In `frontend/src/pages/JobFeed/JobFeedPage.tsx`, around the source-filter `useState` and the `<select>` it drives. The filter used numeric option values (`"0"`, `"1"`) and read `Number(sourceFilter)` to compare against the response payload. After the back-end switch the payload carried strings. `Number("Djinni")` returned `NaN`. Every `job.source !== sourceFilter` comparison was vacuously truthy. The filter rejected every job. The user saw an empty list with no error. The fix: change option values to the back-end string names, remove the `Number()` coercion.

Regression 2 — verdict-label vocabulary. The Domain enum `Verdict` defined four members whose CLR names are `StrongMatch`, `PartialMatch`, `WeakMatch`, and `Mismatch`. The string-enum converter emitted these long names on the wire. But `verdictMeta.ts` keyed display metadata on the short forms (`Strong`, `Partial`, `Weak`, `Mismatch`). I resolved this in Domain rather than at the boundary by adding `VerdictExtensions.ToShortName()` at `Verdict.cs:38–44`. The DTO uses `ToShortName()` when it serialises a verdict. The reasoning prompt continues to use `ToString()` because it expects the long names. Two callers, two vocabularies, both routed through a single file.

The lesson. Contract-changing edits need a cross-stack audit, not a unit test on one side. Single-source-of-truth maps (`verdictMeta.ts`, `evidenceTier.ts`) and exhaustive extension methods (`VerdictExtensions`) are what made the second fix a one-file edit rather than a scattered touch-up.

Graceful degradation as an architectural pattern.

The second narrative is not a single bug but a recurring pattern. I built the v6 pipeline on the assumption that any single external call can fail or run out of time. The right response is rarely to fail the entire user-facing request.

The clearest example is the `SkippedNoAnalysis` field on `GetAggregatedJobsV6Result`. A cold v6 search can return up to about 150 freshly scraped vacancies. Each needs a Gemini vacancy-normalisation call before scoring can run. The synchronous fan-out is bounded by `ScoringOptions.SyncNormalizeTimeoutSeconds` (currently 300 seconds). When the

budget blows, the remaining un-normalised vacancies are not scored. They are counted into **`SkippedNoAnalysis`** and returned alongside the partial result set. On a subsequent request from the same user, the still-unanalysed vacancies are picked up and normalised inside the next budget window. The XML documentation on **`ScoringOptions`** records this explicitly as a "self-healing" property of the pipeline.

The same pattern reappears in three other places:

- **`JobAggregationService`** wraps every individual **`IJobSourceService`** call in a try-catch. A single failing source — most often LinkedIn returning a rate-limit response — does not break the aggregated search.
- Every Gemini call carries a short per-call timeout (8 s for the inner scoring-loop reason call, 15 s for the Composite Judge, 18 s for the batched Judge, 25 s for the batched reason service). On timeout or schema-validation failure, the call falls back to a deterministic template.
- **`CostLogService`** catches every database exception. A missing log entry is preferable to a failed user response.

The lesson. At design time, ask not whether each external call will fail, but how the system will behave when it does. Recording the chosen answer in a typed surface such as **`SkippedNoAnalysis`** makes the degradation observable rather than silent.

F | Layers 1 and 2 — normalisation gold sets

This appendix expands on the two frozen normalisation layers summarised in Section 5.3.

Layer 1 — curriculum-vitae normalisation. The gold set has 25 CVs, normalised by me into the same JSON shape the production pipeline persists as **`UserProfile.CvSummary`**. Nineteen field-level scored metrics are computed by graders in **`EvalTool/Grading/EvaluationEngine.cs`**:

- F1 for skill list comparisons (**`technical_skills`**, **`domain_skills`**, **`unverified_skills`**).
- Jaccard overlap for target-role lists.
- Exact-match for categorical fields (seniority, work format).
- Ordinal-distance grader on the CEFR ladder (A1–C2) for English level.

The headline number is the macro-averaged score across these metrics. The current production CV-normalisation service reaches 0.896 after nine prompt versions. Iteration history in **`eval_history/runs.csv`**. I consider the layer frozen because subsequent runs change the headline by less than the per-iteration noise observed across the last three iterations.

Layer 2 — vacancy normalisation. The gold set has 357 vacancies, sampled across the seven source adapters and stratified across four verdict-bucket strata (Strong, Partial, Weak, Mismatch). Each vacancy is normalised into the same JSON shape the production pipeline persists as **`JobVacancy.VacancyAnalysisJson`**. Seventeen field-level scored metrics are computed by graders in **`EvalTool/Grading/VacancyEvaluationEngine.cs`**, adapted to the vacancy schema (must-have and nice-to-have skill lists, domain, seniority, work format, language requirement).

The current production vacancy-normalisation service reaches 0.804 across 355 of the 357 gold vacancies (two excluded by the loader because of malformed source records) after twelve prompt versions. Iteration history in **`eval_history/vacancy/runs.csv`**. The layer is frozen on the same noise-threshold criterion as Layer 1.

What the frozen baselines fix, and what they do not measure. Because Layers 1 and 2 are frozen, the later layers run against a known, reproducible upstream. Layer 3 and Layer 5 results are reported both against the production normalisation services and (in parallel) against the gold normalisations directly. The latter gives an upper bound on what the pipeline could reach with perfect upstream. Layer 4 is reported only against the production path because reason text is generated after scoring.

Two caveats:

- Per-field F1 measures how well the pipeline matches my gold annotation, not how well the annotation matches reality. Both gold sets were curated by a single annotator (me) without an inter-annotator coefficient.
- The fields with the lowest F1 (the skill lists, roughly 0.47–0.62) are the same fields the scoring formula leans on hardest.

G | Layer 3 — score-error audit decomposition

This appendix expands the three audit findings on the score-error harness summarised in Section 5.4. The full audit decomposition is documented in **`Capstone Project/EVAL_VALIDITY_APPENDIX.md`** in the source repository.

The harness. For every (CV, vacancy) pair in a fixed 392-pair test set, the harness:

1. Loads the gold CV summary and the gold vacancy analysis.
2. Runs the seven sub-score calculators on those gold inputs.
3. Clamps the weighted sum to $[0, 1]$.

The resulting **`ideal_score`** is the raw weighted sum with no anti-flag penalty, no missing-must-have soft penalty, no language-gap cap, and no family-mismatch cap. The harness then reads the pipeline's end-to-end output for the same pair and reports the absolute difference. Implementation in **`EvalTool/Grading/ScoringEvaluationEngine.cs`**, run through **`ScoringEvalOrchestrator.cs`**.

First-look numbers on the 392-pair test set:

- Linear pipeline (weighted sum of the seven sub-scores, no Composite Judge): overall 0.899, MAE 0.101, verdict-bucket match 67.3%.
- Composite pipeline (Judge invoked on every pair): overall 0.813, MAE 0.187, verdict-bucket match 31.9%.

The naive reading says the Composite Judge is worse than the Linear baseline. The two audits show why that reading is misleading.

Cap asymmetry. The pipeline applies the safety caps and anti-flag penalty from Section 3.4. The ideal does not. A per-pair decomposition:

- 14% of pairs carry an active anti-flag. They drift by an average of -0.379 against the ideal.
- 86% of pairs have no anti-flag. They drift by an average of -0.056.

Together these account for almost all of the Linear pipeline's headline error: $0.14 \times 0.379 + 0.86 \times 0.056 = 0.101$, matching the Linear MAE exactly. The metric measures the caps, not pipeline quality.

Selection circularity. I stratified the 392-pair test set on title overlap, seniority match, and skill overlap. These are the same primitives the Linear pipeline's **`RoleIntentMatchCalculator`** and **`SkillMatchCalculator`** consume. The Linear pipeline is graded on its home turf. The audit estimates this asymmetry inflates the Linear advantage by 0.02–0.04 MAE points (40–50% of the visible gap).

Composite Judge over-penalisation. The Judge prompt's rule zero requires output within ± 0.05 of the input linear score. But three other rules override rule zero when triggered:

- Rule 3 — under-qualification.
- Rule 4 — language cap.
- Rule 5 — different-family cap.

The post-Judge clamp in **`GeminiCompositeJudgeService.cs`** enforces $[0, 1]$ but not the ± 0.05 band. The five largest Linear-to-Composite divergences are $-0.39, -0.38, -0.26, -0.23, -0.21$ — all on pairs where one of the three override rules fired. The fix is straightforward: enforce a hard $\left|s_{\text{judge}} - s_{\text{linear}}\right| \leq 0.10$ clamp at the boundary.

What remains after the artefacts are subtracted. Of the visible 0.086 gap between the Linear and Composite headlines:

- 0.04 is attributed to eval-engine artefacts (cap asymmetry + selection circularity).
- 0.05 is attributed to Composite over-penalisation.

Subtract both. The Composite Judge would still underperform on score-tracking, but the gap would be about half what the headline suggests. The post-hoc sensitivity check in Section 5.6 confirms this empirically: running the saved Composite scores through a post-hoc ± 0.10 clamp collapses the largest negative outlier from about -0.54 to essentially zero.

H | Layer 4 — prompt iteration history

This appendix expands on the prompt iteration cycle summarised in Section 5.5. The headline comparison table and the v6 design choice remain in the main text; the per-version regression story lives here.

v2 baseline. The v2 prompt produced verdict-anchored reason text but lacked an explicit context-lead. Free-form composition allowed the model to invent context phrases that the rating gold then flagged as factually unsupported.

v4 — context lead with `ReasonContext`. v4 added a context-lead phrasing rule. It introduced the **`ReasonContext`** record built by **`ScoringServiceV2.BuildReasonContext()`**. A **`ReasonValidator`** safety net rejects reason texts whose verdict word disagrees with the score bucket. The v4 prompt gained $+0.55$ overall against v2 (95% CI $[+0.42, +0.68]$) — a strong, significant improvement. The same comparison surfaced a regression on **`factual_accuracy`** of -0.43 when the model inferred labels from its own judgement instead of from the structured **`ReasonContext`** fields.

v5 — strict context referencing. v5 rewrote the context-lead rule to forbid inference and require strict **`ReasonContext`** referencing. The v4 regression on **`factual_accuracy`** recovered by $+0.71$. But two other dimensions regressed:

- **`completeness`** — -0.62.
- **`relevance_for_user`** — -0.37.

The cause: the model became too cautious. It omitted strong signals that were unambiguously in the **`ReasonContext`** because the literal field name did not match. The aggregate delta against v4 was -0.03, CI $[-0.14, +0.08]$, not significant. A recall trade-off, not a regression in quality.

v6 — four constrained templates. v6 replaced free-form context generation with four constrained templates:

- Role-Family Mismatch.
- Cross-Domain.
- Overqualification.
- Underqualification.

Each template interpolates literal **`ReasonContext`** values into fixed sentence shapes. The templates resolved the v5 trade-off because the model no longer has to choose between inference and silence: the template picks the salient signals and the model fills in the literal values. Against the v4 baseline, v6 produced significant gains on all four ordinal dimensions:

- **`factual_accuracy`** $+0.50$.
- **`completeness`** $+0.23$.
- **`specificity`** $+0.77$.
- **`relevance_for_user`** $+0.31$.
- Overall $+0.30$, CI $[+0.19, +0.40]$.

v6 is the production version at the time of submission.

What this iteration cycle demonstrates. Each detection was a confidence-interval calculation, not a guess, and each fix was a single component change with the rest of the pipeline held still. The same gold set, the same judge model, the same rubric — only the prompt body changed between runs. The per-dimension decomposition is what distinguished v5 as a recall regression rather than a quality regression; without it, v5 would have looked like an outright loss against v4.

I LangSmith experiment index

This appendix lists the LangSmith dataset and experiment identifiers behind the held-out evaluation reported in Section 5.7. A reader with workspace access can navigate to smith.langchain.com, open the named dataset, and inspect every per-pair trace that contributes to a published number in this thesis.

Artefact	Identifier	Source
Dataset	`vakansio_match_quality_heldout`	LangSmith
Dataset UUID	`6273f7bd-a2d7-46c2-b144-c6bab935a010`	LangSmith
Headline experiment (398 pairs)	`vakansio_v1_6_n398_v2`	LangSmith
Intermediate (272 pairs)	`vakansio_v1_6_n398_full`	LangSmith
Stage-1 source weighting (131 pairs)	`vakansio_v1_6_source_weighting`	LangSmith
Per-pair predictions	`results/heldout_v1_6_n398.json`	Repository
Metric report + reliability bins	`results/metrics_v1_6_n398/report.md`	Repository
Caps on/off ablation	`results/ablation_caps_v1_6_n398/report.md`	Repository
Calibrator artefact	`results/calibration_v1_6_n398/calibrator.json`	Repository

Table 14: Reproducibility index for the held-out evaluation. Each LangSmith row corresponds to a Session that groups per-pair Runs against the dataset; the JSON artefacts in the repository carry the raw predictions and the post-hoc metrics in machine-readable form.

Layer 4 and Layer 5 reproducibility artefacts. The reason-quality (v6-vs-v4) and ranking (n=14 CVs) experiments were run through the same harness that produced the held-out runs above, with per-pair judge ratings and aggregated metrics archived under LangSmith experiment metadata. Re-running either experiment requires only the saved prompt versions (v4 and v6 of the Layer 4 prompt; v1.6 of the ranking pipeline) and the corresponding gold subsets (`gold_set_v2/reason_quality/` and `gold_set_v2/match_quality/`) — the harness and judge prompts are version-pinned in source.

J | Reliability diagram

This appendix contains the reliability diagram referenced from Section 5.7.3. The figure is laid out as two side-by-side reliability diagrams against the same 398-pair held-out gold. Both panels share the same Y-axis (mean gold match-quality per bin) and the dashed diagonal of perfect calibration. The **left panel** plots the raw composite score (orange) — points fall well below the diagonal across the full distribution, and the shaded orange area visualises the over-confidence (Expected Calibration Error 0.140). The **right panel** plots the same gold against the score after the production isotonic-regression calibrator (green) — points now hug the diagonal closely, and the green shaded area collapses to a thin strip (ECE 0.027, an 80.6 per cent relative reduction under 5-fold cross-validation). The bin-count histogram across the bottom records how many of the 398 pairs fall into each predicted-score bin; bins concentrate in the lower half because most candidates score below 0.5 against the recruiter-side rubric.

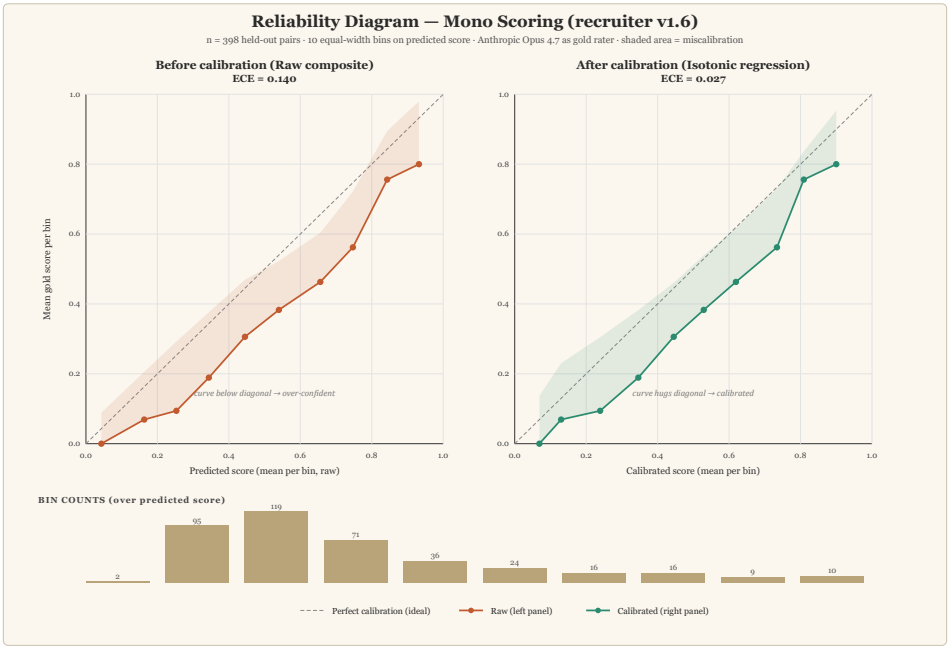


Figure 1: Reliability diagram for the production scoring system on the 398-pair held-out gold. The raw curve reports Expected Calibration Error 0.140; the curve after 5-fold cross-validated isotonic regression reports ECE = 0.027, an 80.6 per cent relative reduction. The bin-count histogram on the lower axis records the population of each predicted bin.

The fitter applies Pool-Adjacent-Violators isotonic regression and, independently, Platt scaling; both are evaluated under 5-fold cross-validation, and the better of the two is persisted as a portable JSON artefact at **`results/calibration_v1_6_n398/calibrator.json`**. The chosen calibrator is loaded once at API startup by **`CalibratorLoader`** and applied to every raw composite via **`IScoreCalibrator.Calibrate`** before the recruiter UI displays the percentage. Runtime cost is one binary search over knots plus linear interpolation — sub-microsecond per call.