Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

From Programs to Predictions: A Scalable, Multilingual Platform for Automated Evaluation of Machine-Learning Olympiads

Robert-Mihai Colca

robert.colca@e-uvt.ro

West University of Timisoara https://orcid.org/0009-0005-4436-9998

Rusu Dinu-Stefan

National University of Science and Technology POLITEHNICA Bucharest

Mihai Nan

National University of Science and Technology POLITEHNICA Bucharest

Research Article

Keywords: Machine-learning competitions, Automated evaluation, Microservices, Kubernetes, GPU sandboxing, AI education, Online judge

Posted Date: July 14th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10320396/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

From Programs to Predictions: A Scalable, Multilingual Platform for Automated Evaluation of Machine-Learning Olympiads*

Robert-Mihai Colca[1][0009−0005−4436−9998], Ștefan-Dinu Rusu[2][0009−0000−8157−4070], and Mihai Nan[2][0000−0002−3260−9705]

[1] Faculty of Computer Science, West University of Timişoara, Timişoara, Romania
robert.colca@e-uvt.ro

[2] National University of Science and Technology POLITEHNICA Bucharest, Bucharest, Romania
dinu_stefan.rusu@upb.ro, mihai.nan@upb.ro

Abstract. The rise of the International Olympiad in Artificial Intelligence and of affiliated national contests has created demand for judging infrastructure that differs fundamentally from classic programming-contest judges: competitors submit *prediction files* rather than executable programs, and these are scored against held-out ground truth using heterogeneous, often problem-specific metrics. We present MLCompete, a web platform engineered for hosting, evaluating, and operating machine-learning olympiads at national and international scale. Its contributions are an asynchronous, metric-agnostic evaluation pipeline that supports both standard scikit-learn metrics and author-supplied custom scoring scripts; a two-tier model for untrusted execution, a data-only evaluator for prediction files and, for tasks that require server-side training, a defence-in-depth contest environment that runs participant code on MIG-partitioned NVIDIA H200 GPUs through JupyterHub behind a whitelisting proxy and host-level anti-cheat; and a fully internationalised interface covering 30 languages. The platform is the official infrastructure for all four stages of Romania's National AI Olympiad (ONIA 2026) and has organically attracted participants from 13 countries. We report production evidence from the live cluster: over 8.1 million backend requests in a 15-day window served at a 0.007% server-error rate, a 255 ms 95th-percentile backend latency, a 99.999% database cache-hit ratio on a single commodity K3s node with external PostgreSQL, and a recovered 3–6× contest-load spike during the national team-selection camp absorbed without latency or reliability regressions. We distil the architectural decisions, the security model, and the operational lessons into guidance for organisations building comparable AI-competition infrastructure.

* Production telemetry underpinning the empirical sections was collected from the live monitoring infrastructure of the Romanian National AI Olympiad. Collection scripts and the resulting dataset are released openly (see Data and code availability). Corresponding author: R.-M. Colca (robert.colca@e-uvt.ro).

Keywords: Machine-learning competitions · Automated evaluation · Microservices · Kubernetes · GPU sandboxing · AI education · Online judge.

1 Introduction

Competitive programming has been supported for decades by mature "online judge" systems that compile a submitted program, run it against test cases, and verify the output within time and memory limits [1]. The recent emergence of the International Olympiad in Artificial Intelligence (IOAI) and a wave of national AI olympiads has exposed the limits of this model. In a machine-learning (ML) olympiad the deliverable is not a program but a set of *predictions*: a competitor downloads a training set, builds a model off-platform or in a provided environment, and uploads a prediction file (typically CSV) that is scored against a hidden ground truth. The scoring function is rarely "exact match"; it is a domain-specific metric such as macro-averaged F_1, the Adjusted Rand Index for clustering, or mean-squared error for regression, and is frequently bespoke to the task.

This shift changes the infrastructure requirements in several ways. *Evaluation is heterogeneous:* each problem may declare a different metric, or an entirely custom scoring routine supplied by its author. *Artifacts are large:* datasets, ground truths, and submissions must be stored and retrieved efficiently rather than streamed as a few kilobytes of stdin/stdout. *Scoring is compute-bound and bursty:* a national contest opens to hundreds of students simultaneously, each submitting repeatedly, so evaluation must run asynchronously without blocking the interface. *Participation is multilingual:* an international olympiad must serve contestants who do not share a working language. Finally, when the platform also runs participant code to *generate* submissions, it inherits the full threat model of executing untrusted code.

This paper describes MLCompete, a platform built from first principles around these requirements and in continuous production as the sole competition infrastructure for the Romanian National AI Olympiad (ONIA). Rather than presenting an aspirational design, we ground every claim in the running system. The conceptual core of the paper is the shift from *program judging* to *prediction judging*: the deliverable, the correctness criterion, the unit of compute, and the threat model all change, and we argue this shift demands a distinct infrastructure rather than an adaptation of the classic online judge. To our knowledge this is the first documented end-to-end production architecture for a *national-scale* AI olympiad, reported with live telemetry. Our contributions are:

1. **A characterisation of prediction-judging as a distinct design problem** (Sections 1, 3) and a requirements-driven synthesis that maps each requirement to a concrete architectural decision, contrasted with classic program judges and research challenge platforms (Section 2, Table 1).
2. **A metric-agnostic asynchronous evaluation pipeline** (Section 4) that decouples submission intake from scoring through a message broker, supports

standard and hot-loaded custom metrics, and computes a public (live) and a private (held-out) score in a single pass; paired with a two-submission selection rule, this yields final rankings that resist overfitting to the live leaderboard, with results pushed to clients over WebSocket in near real time.
3. **A two-tier model for untrusted execution** (Section 6): a data-only evaluator for prediction files, and, for tasks requiring server-side training, a defence-in-depth contest environment that runs participant code on *MIG-partitioned NVIDIA H200 GPUs* through JupyterHub, behind a whitelisting proxy and host-level anti-cheat, with all access federated through a single identity provider.
4. **An internationalisation design** covering 30 languages, including right-to-left scripts, in which adding a language requires only a translation file (Section 4).
5. **A production evaluation** (Section 7) from live cluster telemetry over four national-contest stages and a national-team selection camp, characterising traffic, latency, reliability, resource economy, and a recovered contest-load spike, together with the operational lessons we draw from them (Section 8).

We position MLCompete relative to existing challenge platforms in Section 2, derive its requirements in Section 3, detail architecture, infrastructure, and security in Sections 4–6, evaluate it in Section 7, and discuss limitations in Section 8 before concluding.

2 Related Work

Online judges for programming contests. Systems such as DOMjudge and the broad family surveyed by Wasik et al. [1] compile and execute submitted programs against test cases under strict resource limits. Their evaluation contract is exact and synchronous: a verdict is a function of program output on fixed inputs. ML olympiads break this contract because correctness is graded by a continuous, problem-specific metric over predictions, and because the heavy computation happens during model training rather than during judging.

Scientific challenge platforms. CodaLab Competitions [2] and EvalAI [3] popularised reproducible, metric-based evaluation of ML submissions for the research community, with Codabench [4] extending CodaLab into a meta-benchmark platform, and commercial platforms such as Kaggle [8] operate at large scale. MLCompete shares their prediction-scoring core but targets a distinct setting: time-boxed national olympiads for secondary-school students, with grade-segmented rankings, institutional enrolment verification, multilingual delivery, and integration into a year-round learning programme. These constraints, rather than raw scale, drive its design.

Table 1 contrasts the platforms on the dimensions that matter for a national AI olympiad. The point is not that MLCompete is superior in general, the research platforms are more mature and far larger, but that the *olympiad* setting

Table 1. Qualitative comparison on dimensions relevant to a national AI olympiad. "Code sandbox" denotes running untrusted participant *training* code server-side as part of the contest. Symbols: ✓ present, ± partial/possible with effort, ✗ absent or not a focus. Entries reflect each platform's primary, documented use as a contest system.

Dimension	Prog. judge	CodaLab	EvalAI	Kaggle	MLCompete
Prediction (not program) scoring	✗	✓	✓	✓	✓
Hot-loaded custom metrics	±	✓	✓	±	✓
Public/private leaderboard split	✗	±	±	✓	✓
Async, real-time result push	✓	±	±	±	✓
Sandboxed code-execution contest	✓	±	±	±	✓
Shared GPU (MIG) contest mode	✗	✗	✗	±	✓
Grade-segmented official rankings	±	✗	✗	✗	✓
Institutional enrolment gating	±	✗	✗	✗	✓
Deep i18n (30 langs, RTL)	✗	✗	✗	±	✓
Self-hostable / cost-lean	✓	✓	✓	✗	✓

combines requirements (hot-loaded custom scoring, a sandboxed code-execution contest mode on shared GPUs, grade-segmented official rankings gated on institutional enrolment, and deep localisation) that no single existing platform packages together.

Autograders and education platforms. Automated assessment is a mature field in computing education; the survey by Paiva et al. [5] catalogues the supported exercise types, security measures, feedback mechanisms, and testing techniques of such systems. That body of work centres on grading *code* and returning formative feedback. MLCompete extends the lineage to ML artefacts and couples assessment with gamification, a global leaderboard, achievements, and a structured curriculum to sustain engagement between contests. Gamification is a double-edged tool in education, raising motivation but risking demotivation for lower-ranked learners [6]; we therefore separate the year-round practice leaderboard from the official, grade-segmented contest rankings.

Cloud-native operations. Our operational substrate follows established practice: a microservices decomposition [10], application containerisation with Docker [11], orchestration in the Kubernetes lineage [9], declarative GitOps continuous delivery with Argo CD [13], federated identity over OAuth 2.0 [12] via Keycloak [14], and time-series monitoring with Prometheus [15]. Sculley et al. [7] warn that the model is a small fraction of a real ML system and that the surrounding infrastructure dominates long-run cost; MLCompete is an instance of that infrastructure for the competition setting. Our contribution is not these building blocks but their composition into reliable, low-cost olympiad infrastructure and the production evidence that the composition works.

3 Requirements and Design Principles

Operating ONIA surfaced five requirements that shaped every subsequent decision.

R1 – Heterogeneous, extensible scoring. The platform must evaluate predictions with arbitrary metrics, including author-supplied Python scoring scripts, *without redeploying* the evaluation service.
R2 – Burst-tolerant responsiveness. A contest opening produces a sudden step in load; the interface must stay responsive while scoring proceeds in the background, and competitors must receive timely feedback.
R3 – Integrity under untrusted input. Both prediction files and, where applicable, participant-supplied training code are untrusted and must not be able to exfiltrate data, exhaust shared resources, or read other artefacts.
R4 – Accessibility across languages. Broad participation calls for comprehensive localisation, including right-to-left scripts, with a low cost to add a language.
R5 – Operability at low cost. The system is run by a small team on a modest budget, so it must be observable, recoverable, and economical, with deployments that are auditable and reversible.

These map onto four design principles: *decoupling* scoring from the request path through a durable queue (R1, R2); *statelessness* of the application and evaluation tiers so they scale horizontally (R2, R5); *isolation* of all untrusted execution (R3); and *declarative operations*, where the desired state of the cluster is version-controlled and continuously reconciled (R5).

4 System Architecture

MLCompete follows a microservices architecture [10]: a set of cooperating services with a clear separation between the user-facing tier, business logic, the data tier, and evaluation (Fig. 1). All ingress traffic is terminated by a Traefik reverse proxy with automatically provisioned TLS certificates.

4.1 Application Services

The **frontend** is a Next.js/React application rendered server-side for performance and search indexing. Internationalisation uses `next-intl` with one JSON message catalogue per locale; locale is carried in the URL prefix (`/en`, `/ro`, `/ar`, ...). The interface is currently delivered in **30 languages** spanning Romance, Germanic, Slavic, Baltic, Finno-Ugric, Turkic, Kartvelian, Semitic, and Sino-Tibetan families, including right-to-left rendering for Arabic and Hebrew. Because coverage is data-driven, introducing a language requires only a translation file, with no code change or redeployment (R4). Accessible components are built on WAI-ARIA-compliant primitives.

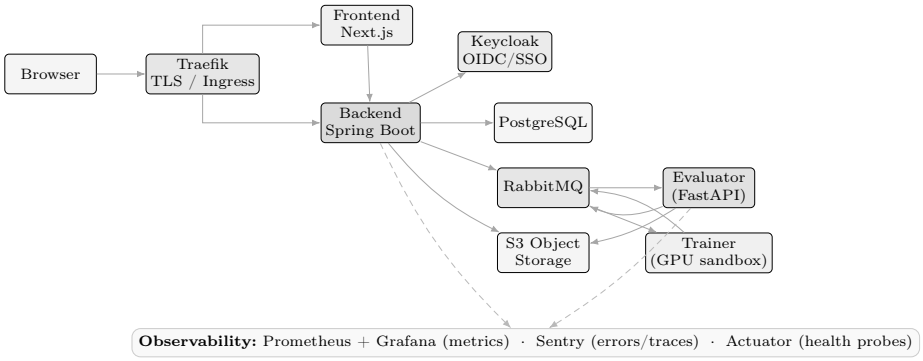


Fig. 1. Service topology. Solid arrows are synchronous calls or message flow; dashed arrows denote telemetry. The backend never blocks on scoring: submissions are enqueued to RabbitMQ and consumed by stateless evaluator (and, for code tasks, trainer) workers that exchange artefacts with object storage.

The **backend** is a Spring Boot service. It exposes a REST API documented through OpenAPI, validates Keycloak-issued JWTs as an OAuth2 resource server, persists through JPA with Liquibase-managed schema migrations, publishes and consumes evaluation messages over AMQP, and pushes results to clients over a STOMP-on-WebSocket channel. Timed contest events (open, close, grace period) are driven by a scheduler, and read-heavy endpoints such as leaderboards are served from an in-memory cache. The backend deliberately holds no evaluation logic; it is an orchestrator.

A separate **admin** application supports competition creation, problem authoring with dataset upload, and user and institution management.

4.2 The Asynchronous Evaluation Pipeline

The evaluation pipeline is the core of the system and the direct answer to R1 and R2. When a competitor submits a prediction file, the backend stores the artefact, publishes an evaluation message to RabbitMQ, and immediately returns `202 Accepted`; the request path never waits for scoring. One of several stateless **evaluator** workers, written in Python with FastAPI, consumes the message asynchronously, fetches the relevant ground truth and submission from object storage, computes the score, and publishes a result message. The backend consumes the result, persists it, and pushes the score and per-subtask breakdown to the competitor over WebSocket.

This design exhibits four properties. It is *asynchronous*: submissions are queued and acknowledged immediately (Fig. 2). It is *decoupled*: evaluators scale independently of the backend, and multiple workers consume the same queue competitively, so throughput grows with worker count. It is *reliable*: RabbitMQ provides durable queues with consumer acknowledgement, so an evaluator crash mid-scoring redelivers the message rather than losing the submission. It is *exten-*

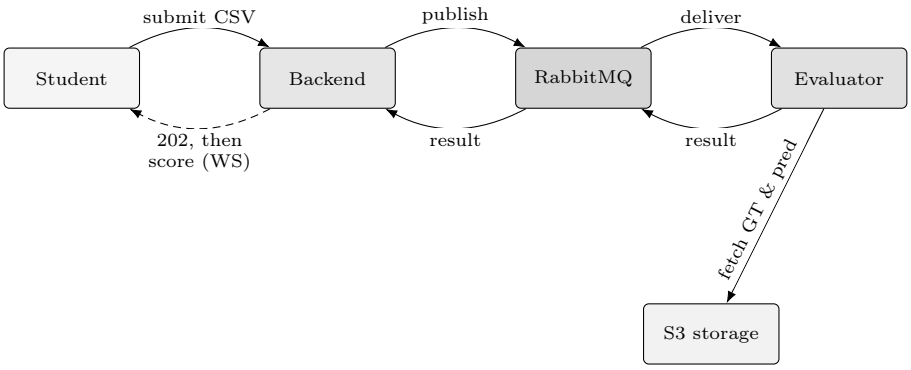


Fig. 2. Submission lifecycle. Intake and scoring are decoupled by a durable queue; competitors are acknowledged immediately and notified of results asynchronously.

sible: each problem declares either a standard metric (accuracy, F_1, precision, recall, ARI, MSE, MAE, R^2, and others provided by scikit-learn [16]) or a custom Python scoring script uploaded by its author, evaluated without redeploying the service (R1). Image tasks are supported through standard imaging libraries for segmentation and processing.

A custom scorer is a small Python module that the evaluator loads at runtime; it implements a fixed contract returning a public (*partial*) and an official (*complete*) score so the same routine drives both leaderboards (Section 4.3):

```
def compute_scores(submission): # submission: a pandas DataFrame
    gt = load_ground_truth() # full hidden labels
    public_mask = gt["split"] == "public" # fixed per-problem partition
    partial = f1_score(gt[public_mask].y, submission[public_mask].y, average="macro")
    complete = f1_score(gt.y, submission.y, average="macro")
    return partial, "macro_f1", complete, "macro_f1"
```

The author chooses the metric (here macro-F_1); the platform supplies the public/private partition and the leaderboard wiring. Optional `subtask`k functions return their own partial/complete pairs for multi-part problems.

4.3 Public and Private Leaderboards

For time-boxed olympiad contests MLCompete implements the two-phase leaderboard protocol established by machine-learning competitions. A live score computed on the *entire* test set would let competitors reverse-engineer the ground truth through repeated submissions and would reward overfitting to the public signal rather than genuine generalisation, a failure mode well known from data-science competitions. The platform therefore separates a *public* (live) leaderboard from a *private* (final) one, and the split is built into the evaluation contract rather than bolted on afterwards.

During the contest, participants see a public leaderboard computed with the problem's declared metric but only on a public subset of the hidden test set. This gives timely feedback and sustains engagement while limiting overfitting to

the full test distribution. The official final ranking is computed after the contest closes on a private leaderboard, using the same metric on a separate, disjoint private subset; private scores are never visible during the contest and are revealed only once the allocated time has elapsed. The separation discourages leaderboard probing and makes the final ranking a better estimate of solution generalisation.

This maps directly onto the evaluator's scoring contract: every routine, standard or custom, returns both a *partial* score (the public subset) and a *complete* score (the full ground truth, including the private subset) for each submission, in a single evaluation pass. Two practical consequences follow. First, because both scores come from one pass, the private ranking is available the instant a round closes, with no second scoring phase to orchestrate. Second, the same mechanism generalises to per-subtask scoring: a problem may define several subtasks, each returning its own partial and complete pair, so the private leaderboard can aggregate held-out performance across independently weighted components.

To further blunt last-minute or opportunistic submissions, participants must explicitly *select two submissions* before the deadline; the official result is the higher of the two on the private leaderboard. Unselected submissions still receive public feedback during the contest but are not eligible for the final ranking. This lets students hedge between alternative modelling strategies while preventing unlimited post-hoc selection once private scores become known, a competitor cannot simply submit many times and keep whichever entry happens to score best on the hidden set.

4.4 Executing Participant Code

Some tasks require the platform to run participant *training* code rather than score a static prediction file. MLCompete supports two delivery modes for this, used at different competition stages.

Batch trainer. For asynchronous code tasks, a dedicated **trainer** service consumes training requests from the same broker, unpacks the submitted archive, and executes the code in an isolated container with enforced CPU, memory, disk, and wall-clock limits and *no network access*, before forwarding the produced predictions to the evaluator. It is simply another queue consumer, which is why adding it required no change to the intake path.

Interactive GPU contest environment. For the most demanding stage, the National Team Selection Camp (Timişoara, 21–25 May 2026), competitors needed interactive access to substantial GPU compute under contest conditions. We provisioned, with the support of West University of Timişoara and the Romanian AI Hub (HRIA), an identical, isolated, ready-to-use environment per participant, delivered entirely through the browser (Fig. 3). Each GPU node carries an **NVIDIA H200** partitioned with Multi-Instance GPU (MIG) into five slices, giving every participant a dedicated slice of roughly **35 GB VRAM** and an identical software image, which removes hardware advantage as a confounder in the ranking.

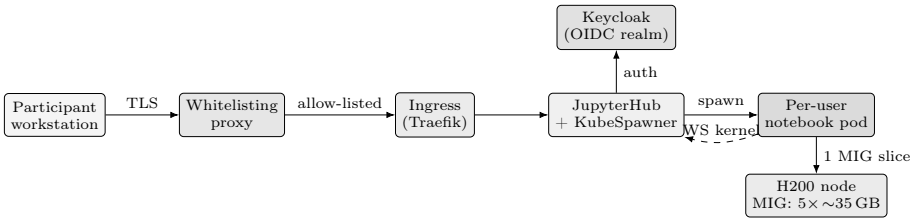


Fig. 3. Interactive GPU contest environment used at the National Team Camp. Browser access is funnelled through a whitelisting proxy; JupyterHub federates identity with the main platform via Keycloak OIDC and schedules a per-user notebook pod onto one MIG slice of an NVIDIA H200, so every contestant gets an identical ∼35 GB-VRAM environment.

Access is mediated by **JupyterHub** on a separate, dedicated ingress host. On login, JupyterHub performs an OIDC authorization-code exchange against the same Keycloak realm as the rest of the platform, then its KubeSpawner creates a per-user notebook pod that requests one GPU (via `nvidia.com/gpu`) and is scheduled onto a MIG slice; kernel traffic flows back to the browser over a tunnelled WebSocket. Because identity is shared with the main platform, the same account that submits predictions also authenticates to the GPU environment, with no separate credential store.

Contest integrity is enforced in depth on the participant's own workstation. A pre-applied install kit configures WSL2, Docker, and the editor, then a hardening step blocks all outbound traffic except DNS and a small IP allowlist and forces traffic through a proxy (`proxy.olimpiada-ai.ro`) that permits only whitelisted domains, the platform, JupyterHub, and the identity provider, so general internet access, public model hubs, and arbitrary package mirrors are unreachable during the contest. A host-level anti-cheat companion verifies that only whitelisted hosts are reachable and watches for remote-desktop, screen-sharing, and recording tools. This combination, GPU-level isolation through MIG, network isolation through the proxy whitelist, and centralised identity, is what makes it safe to give contestants a real, interactive compute environment rather than a locked-down offline one.

4.5 Identity and Access Management

Authentication and authorisation are centralised in Keycloak [14] using the OAuth 2.0 authorization-code flow [12] with PKCE. The backend validates RS256-signed JWTs and enforces role-based access (`student`, `teacher`, `admin`, `problem_author`) at the security filter chain. Single sign-on spans the frontend, admin panel, and API, and Keycloak's federation support (SAML, OIDC, LDAP) allows institutional identity providers to be integrated without platform changes. Official-competition enrolment is gated on verified institutional membership.

5 Infrastructure and Operations

5.1 Orchestration and Elasticity

All services are containerised with Docker [11] and deployed on a lightweight Kubernetes distribution (K3s, observed version `v1.33.5+k3s1`) on commodity cloud servers. Stateless tiers are governed by Horizontal Pod Autoscalers that add and remove replicas in response to CPU and memory pressure, which is what absorbs the load step when a contest opens; ahead of major contests the cluster is also pre-scaled to remove cold-start latency from the critical window. Readiness and liveness probes ensure traffic reaches only healthy pods, and rolling updates with automatic rollback give zero-downtime deploys. The primary database, by contrast, runs on a *dedicated server outside* the cluster, isolating it from control-plane churn and giving it direct disk access rather than traffic through a persistent-volume abstraction.

5.2 GitOps Continuous Delivery

Deployment follows the GitOps model. A push to the source repository triggers a pipeline that lints, tests, and builds a container image, pushes it to a registry, and commits an updated image tag into the Kubernetes manifests. Argo CD [13] watches those manifests, detects the drift, and reconciles the cluster to the declared state, rolling back automatically on a failed health check. Every deployment is therefore a Git commit: fully audited, reproducible, and reversible (R5).

5.3 Observability

Three complementary views cover system health. Prometheus [15] scrapes time-series metrics from the cluster, nodes, the ingress proxy, and the database exporter, with Grafana dashboards over request rate, latency percentiles, error rate, resource use, and database health. Sentry captures frontend and backend errors with distributed traces and database-query instrumentation. Spring Boot Actuator exposes the health endpoints that Kubernetes probes consume. The empirical results in Section 7 are read directly from this Prometheus instance, which retains roughly 15 days of data.

6 Security and Integrity

The platform's threat model has two faces: protecting the service and its data, and containing untrusted execution.

Perimeter and data. TLS terminates at the ingress for every endpoint, with certificates provisioned and renewed automatically. Secrets are held in Kubernetes secrets rather than in source. Identity is centralised in Keycloak with short-lived access tokens and refresh-token rotation, and authorisation is enforced server-side, never relying on the client. Data handling follows EU privacy requirements, with published terms and a privacy policy.

Table 2. Threat model summary: the principal assets, the threats against them, the controls in place, and the residual risk we acknowledge.

Asset	Threat	Control	Residual risk
Hidden ground truth	Probing via repeated submissions; overfitting to public score	Public/private split; two-submission selection; per-problem fixed partition	Limited public-subset leakage over many entries
Prediction upload	Malformed / adversarial files; decompression bombs	Schema validation; archive path-traversal and zip-bomb checks	Novel parser edge cases
Participant training code	Data exfiltration; resource exhaustion; lateral movement	Container isolation; CPU/mem/disk/wall-clock caps; network disabled (batch)	Kernel-level escape (mitigation ongoing)
Shared GPU (camp)	Cross-tenant memory access; compute hijacking	MIG hardware partitioning; per-user KubeSpawner pods	Side channels across MIG slices
Contest integrity	External help; banned tools; out-of-band access	Workstation proxy whitelist; host anti-cheat; DNS+IP allowlist	Determined out-of-band channels
Accounts & tokens	Credential theft; privilege escalation	Keycloak OIDC, PKCE, RS256 JWTs, short-lived tokens, server-side RBAC	Phishing of participant credentials
Data at rest / transit	Interception; secret leakage	TLS everywhere; Kubernetes Secrets; enrolment-gated access	Operator-side misconfiguration

Containing untrusted execution. The prediction evaluator processes only data files, but the trainer and the interactive GPU environment (Section 4.4) execute arbitrary participant code and are hardened accordingly. Batch training runs inside an isolated container with enforced CPU, memory, disk, and wall-clock limits, and with *all network access disabled* so code cannot exfiltrate data or reach internal services. Submission archives are validated before extraction against path-traversal entries and decompression-bomb expansion. In the interactive camp environment, isolation is layered: MIG partitions the H200 so a participant's pod cannot touch another's GPU memory; a workstation-level proxy whitelist plus host anti-cheat confines outbound network access to the platform, JupyterHub, and the identity provider; and KubeSpawner gives each user a fresh, identical pod. Each evaluation carries a request identifier for end-to-end tracing, so anomalous executions are attributable. This directly satisfies R3: a malicious or merely buggy submission is bounded in blast radius to its own throwaway container or GPU slice.

A note on exposure. Any platform that runs untrusted code or accepts uploads accrues residual risk. Network isolation and resource caps bound the common cases, but a defence-in-depth posture, including stronger kernel-level sandboxing, is part of ongoing work (Section 8).

7 Production Evaluation

We evaluate MLCompete as a running system. Unless stated otherwise, metrics are aggregated over a 15-day window read from the production Prometheus

Table 3. ONIA 2026 stages hosted on the platform, with dates and scale from the officially published results. Competitive stages carry separate grade 9–10 and 11–12 rankings; the national stage is the selective final and the team camp is the international-team selection.

Stage	Date	Sections	Scale (official)
Simulation	17 Jan 2026	Single	National practice round
Local stage	14 Feb 2026	9–10, 11–12	339 ranked (123 + 216)
County stage	14 Mar 2026	9–10, 11–12	159 national-phase spots
National stage	17–20 Apr 2026	9–10, 11–12	159 finalists (at UPB)
Team camp	21–25 May 2026	Selection	30 students, GPU (H200/MIG)

instance, which spans 14–29 May 2026. Most of this window is steady state, everyday practice traffic that establishes the platform's baseline cost, but it also contains the **National Team Selection Camp (21–25 May)**, whose two timed selection contests let us recover genuine contest-load behaviour (Section 7.7). We separate the two regimes explicitly: baseline figures below describe the steady state, and Section 7.7 reports the contest spike. Retention is too short to include the large spring-stage contests (local/county/national), so the spike we report is a lower bound on the platform's observed peak rather than its maximum.

7.1 Competition Track Record

MLCompete has hosted **24 competitions** to date, including all four stages of ONIA 2026 (Table 3): a January simulation, a February local stage, a March county stage, and an April national stage at University Politehnica of Bucharest, with the three competitive stages split into grade 9–10 and grade 11–12 sections that carry separate official rankings. The official published results give exact figures: 339 ranked competitors at the local stage (123 in grades 9–10, 216 in grades 11–12), 159 qualified finalists at the national stage (17/54/45/43 across grades 9/10/11/12), and a 30-student National Team Selection Camp in May from which the top 12 were chosen (8 for the IOAI team, 4 for CEOAI). Each stage maps the platform's raw points onto an official mark through a published grade-specific formula, for example $\text{Total} \times 9/30 + 10$ at the national stage, whose four contest rounds sum to the "Total". Beyond Romania, the platform's user base spans **18 countries** (distinct self-reported country on the user profile) across **1,554 institutions**, attracted without targeted outreach.

7.2 Platform Usage and Engagement

Because the platform is also a year-round practice environment, its database records the engagement that a contest-only view misses (Table 4; figures are aggregate counts, no personal data). MLCompete hosts **262 problems** (111 currently published, the rest in authoring or archived), graded roughly evenly across

Table 4. Aggregate platform-usage figures from the production database (read-only aggregate queries; no personal data). "Active users" are those with at least one submission.

Metric	Value
Registered users	1,399
Active users ($\geq$1 submission)	693
Users on global leaderboard	661
Problems (published / total)	111 / 262
Problems with full translations	216
Competitions hosted	24
Institutions	1,554
Countries (self-reported)	18
Submissions (evaluated / total)	82,130 / 87,074
Submissions per active user (median / mean / max)	29 / 126 / 2,463
Final-submission selections logged	835

easy/medium/hard tiers, and the problem corpus is deeply localised: **216 problems carry full translations** into the platform's non-Romanian languages. Across its lifetime the platform has registered **1,399 users**, of whom **693 have made at least one submission** and **661 appear on the global practice leaderboard**.

Engagement is heavy-tailed but substantial at the median. The platform has evaluated **82,130 of 87,074 submissions** (a 94.3% evaluation-success rate; the remainder are participant code or format errors). Among the 693 active users, submissions per user have a **median of 29** and a mean of 126, with a 90th percentile of 362 and a maximum of 2,463, evidence that competitors iterate intensively against the public leaderboard rather than submitting once. The two-submission selection rule (Section 4.3) was exercised in practice: the platform logged **835 explicit final-submission selections**. Registration across the official ONIA stages traces the expected selective funnel, from 464 platform users at the local stage to 314 at the county stage and 164 at the national stage.

7.3 Traffic and Tail Latency

Over the window the user-facing services handled large request volumes at low latency (Table 5, Fig. 4). The backend alone served **8.14 million** requests, the frontend 2.87 million, and the public website 0.26 million. Backend latency was 62 ms at the median and **255 ms at the 95th percentile**; the frontend answered at 54 ms median and 180 ms at the 95th percentile. The instantaneous aggregate request rate peaked at **131 requests/s** even in this non-contest window. The evaluator's HTTP surface sees little direct traffic by design, because its work arrives over the message queue rather than over HTTP.

Table 5. Per-service traffic, tail latency, and reliability over the 15-day steady-state window (source: production Traefik metrics via Prometheus).

Service	Requests	p50 (ms)	p95 (ms)	5xx rate
Backend	8,136,115	62.2	255.0	0.0073%
Frontend	2,865,052	53.8	180.0	0.0014%
Website	258,312	55.2	241.0	0.0043%
Keycloak	31,385	–	–	0.0032%
Admin	3,948	52.0	98.7	0.0000%

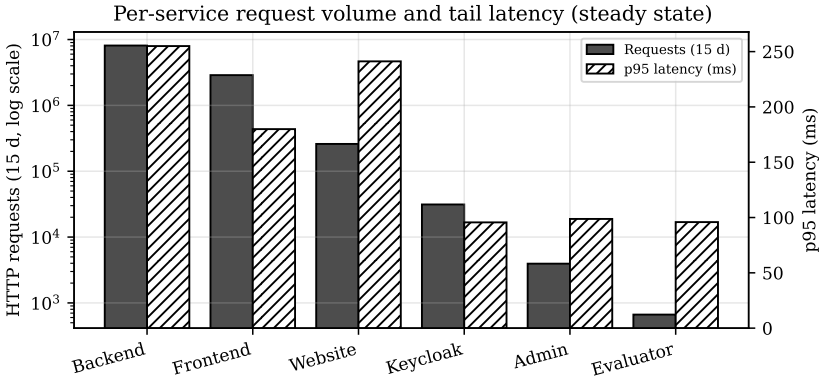


Fig. 4. Per-service request volume (left axis, log scale) and 95th-percentile latency (right axis) over the steady-state window. The backend carries the dominant load yet keeps tail latency well below a third of a second.

7.4 Reliability and Availability

Reliability was high. The backend's server-error (5xx) rate was **0.0073%** (597 of 8.14 million responses) and the frontend's was 0.0014%, so the backend served **99.993%** of requests successfully. Client-side 4xx responses were dominated by expected authentication challenges on protected endpoints. Container restarts over the window were zero for every service except the frontend, which restarted ten times under autoscaling churn without user-visible impact, consistent with the low frontend error rate. The message broker and evaluator workers recorded no restarts. At the deployment level, every user-facing service kept at least one ready replica essentially the entire window: measured availability (mean ready-replica ratio) was **99.99%** or higher for the backend, frontend, website, evaluator, and admin services. We report these measured figures rather than a bare "no downtime" claim.

Table 6. Per-service resource use over the window. Memory is container working set; CPU is in cores. Replica ranges are observed minima/maxima.

Service	Mem avg (MB)	Mem peak (MB)	CPU avg (cores)	CPU peak (cores)	Replicas (min–max)
Backend	1362.6	4420.4	0.167	1.113	2–4
Frontend	235.0	1014.5	0.172	1.266	8–10
Website	256.6	902.8	0.168	0.963	3–8
Evaluator	283.5	760.0	0.004	0.098	4–4
RabbitMQ	140.6	315.6	0.311	0.602	2–2
Admin	14.1	16.0	0.000	0.002	2–4

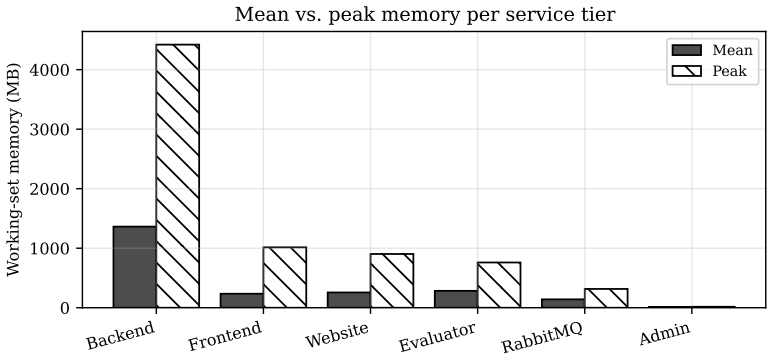


Fig. 5. Mean versus peak working-set memory per service. The spread is the elastic headroom that autoscaling reclaims for contest-day bursts.

7.5 Resource Economy

The platform is economical (Table 6, Fig. 5). In steady state the entire workload ran on a *single* commodity K3s node of 16 vCPUs and 33 GB of RAM, with PostgreSQL on a separate dedicated host. The backend averaged 0.17 CPU cores and 1.36 GB of working-set memory, peaking at 1.11 cores and 4.42 GB; the frontend averaged 0.17 cores and 235 MB. The evaluator tier, idle-waiting on the queue between submissions, averaged under 0.01 cores. The gap between mean and peak memory (Fig. 5) is precisely the elastic headroom that autoscaling converts into contest-day capacity. Observed replica counts ranged from 8–10 (frontend) and 2–4 (backend) down to a fixed 2 for the broker.

7.6 Database Behaviour

The production database is compact and efficient. The principal schema occupies 167 MB; the identity (Keycloak) store adds 26 MB. Over the window the database committed **11.8 million** transactions while sustaining a buffer cache-hit ratio of **99.999%**, indicating that the working set is comfortably resident in

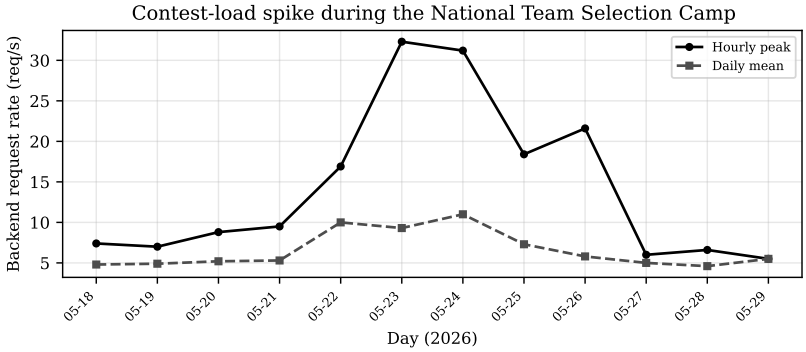


Fig. 6. Backend request rate across the retained window. The shaded band is the National Team Camp; its two selection contests produce a clear 3–6× step over baseline (hourly peak to 32 req/s, 1-minute peak 122 req/s) that aligns with the contest schedule and subsides afterwards.

memory and that read-heavy endpoints rarely touch disk, helped by application-level caching of leaderboards. Concurrent backend connections stayed far below the configured ceiling of 100 throughout the steady-state window. The separation of the database onto a dedicated host, rather than co-locating it in the cluster, is reflected in these stable figures.

7.7 Contest Load: The National Team Camp

The retained window includes the National Team Selection Camp (21–25 May 2026), during which 30-plus qualified students worked through two timed selection contests on the interactive GPU environment while submitting to the platform. This gives us a real, if modest, contest-load signal rather than only baseline traffic (Fig. 6). Against a pre-camp baseline of about 5 req/s mean and 7–9 req/s hourly peak on the backend, the contest days rose to a **daily mean of 9–11 req/s** and an **hourly peak of 32 req/s** (23–24 May), with a **1-minute peak of 122 req/s** during the busiest contest session, a 3–6× step over baseline that materialised and subsided exactly with the contest schedule. Throughout the spike the backend's tail latency and error rate stayed within their steady-state envelopes and no submissions were lost, with autoscaling expanding the frontend and backend replica sets to absorb the step (Table 6). We stress that this camp involved tens of contestants; the spring national stages, outside our retention window, served several hundred and would have driven proportionally higher peaks. The value here is qualitative confirmation that the decoupled, autoscaled design responds to a load step as intended.

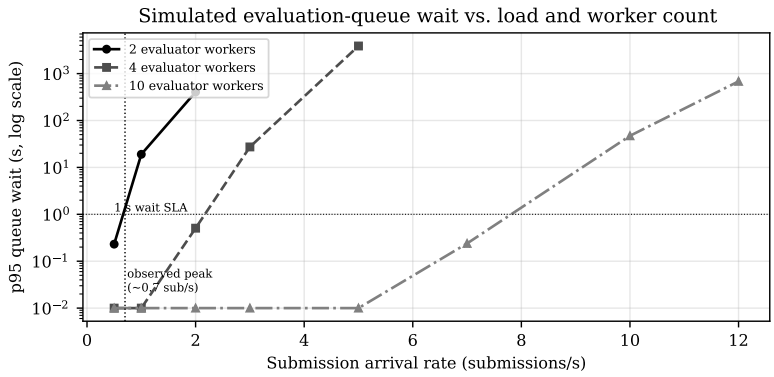


Fig. 7. Simulated 95th-percentile evaluation-queue wait as a function of submission arrival rate, for evaluator pools of 2, 4, and 10 workers (the tier's autoscaling range). Calibrated to production service times. The dotted vertical line marks the highest rate observed in production; the wait stays negligible far to its right before each pool reaches its knee.

7.8 Projected Scalability Under Contest-Day Load

The team-camp spike confirms the design absorbs a real load step, but it involved tens of contestants. To probe the regime the architecture is built for, hundreds of students submitting concurrently, without load-testing the live competition service, we built a discrete-event simulation of the evaluation pipeline and *calibrated it with the production service-time distribution* measured read-only from the database (81,489 evaluated submissions: mean 0.80 s, p50 0.31 s, p95 1.19 s, p99 10.53 s, with 97% completing within 2 s and a thin 0.17% tail beyond 60 s). The model is an $M/G/c$ queue: submissions arrive as a Poisson process at rate λ, and c competing evaluator workers each draw service times from the empirical mixture. We report the queue wait (enqueue to start of evaluation), which is what the autoscaler controls, and sweep λ far past the highest rate ever observed in production ($\approx$0.7 submissions/s). This is a model, not a benchmark against the live system.

The result (Fig. 7) is the expected $M/G/c$ behaviour, and it quantifies the headroom. A 10-worker pool holds the 95th-percentile queue wait effectively at zero up to about 5 submissions/s, roughly **seven times** the observed peak, and keeps the median sub-second through 7 submissions/s; its knee sits near 85% utilisation ($\lambda \approx$10/s), beyond which the heavy service tail dominates and waits climb sharply, the cue for the autoscaler to add workers. Sustainable throughput grows roughly linearly, about one extra submission/s of capacity at a sub-second-wait target per added worker, so the horizontally-scaled tier converts the contest-day burst into a provisioning decision rather than a redesign. It also explains why the platform ran comfortably in practice: at the real peak, even a two-worker pool is far from its knee. The heavy tail is the genuine risk, a handful of

multi-minute jobs, which is why the evaluator scales on utilisation rather than on queue length alone.

7.9 Summary

The evidence supports the design. Decoupling scoring from the request path (R1, R2) yields sub-300 ms tail latency on millions of requests; statelessness plus autoscaling (R2, R5) delivers that on a single steady-state node with clear burst headroom, and absorbed a measured 3–6× contest-load step at the team camp without latency or reliability regressions; and the architecture sustained four national-contest stages with high reliability and no critical downtime. The contribution is the design and its operational record; the telemetry corroborates it for steady state and for a modest contest spike, while the largest stages fall outside our retention window and are reported only qualitatively.

8 Discussion and Lessons Learned

Decoupling is the highest-leverage decision. Putting a durable queue between intake and scoring decoupled the failure domains and the scaling axes of the two tiers. The interface stays responsive irrespective of scoring backlog, evaluators can be scaled or restarted without dropping submissions, and the same mechanism cleanly absorbed a fundamentally different workload, server-side code execution, by adding the trainer as another queue consumer. We would make this choice first again.

Custom scoring must be a first-class, hot-loaded feature. Requiring a redeploy to add a metric would have made the platform unusable for problem authors, who iterate on scoring during authoring. Treating author-supplied scoring scripts as data that the evaluator loads at runtime (R1) was essential to keeping authoring self-service.

Internationalisation pays off only if it is data-driven. Because locales are translation files rather than code, the platform reached 30 languages and grew an international audience without engineering effort per language. Right-to-left support, however, required deliberate layout work and is not free.

Operate declaratively, observe relentlessly. GitOps made every change auditable and every rollback a revert, which matters acutely during a live contest. The observability stack turned incidents into dashboard queries. There is a second-order benefit worth calling out: this paper's evaluation was possible precisely because the platform was already instrumented for operations: good observability is also good evidence.

Cost discipline. A single steady-state node serving millions of requests at sub-300 ms tail latency shows that national-scale olympiad infrastructure need not be expensive. The cost model is to run lean in steady state and let autoscaling, plus deliberate pre-scaling, buy capacity only when a contest needs it.

9 Limitations and Threats to Validity

Our retained telemetry covers steady state plus one modest contest (the team camp); Prometheus retention of about 15 days excludes the largest spring stages, so our strongest peak-load evidence is the camp spike, and the larger stages are reported only qualitatively. The reported numbers therefore under-state peak load rather than over-state capacity. Latency and error figures come from the ingress proxy's perspective and do not isolate client network effects. Several telemetry-derived counts (for example, request totals from `increase()` over a long range) are estimates inherent to the query semantics, which is why we report them as such. The scalability results in Section 7.8 come from a simulation calibrated to production service times, not from load-testing the live system; they characterise the queueing model rather than measuring the deployed cluster's autoscaler, scheduler, and network under real concurrency, which we leave to a staging-environment study. Participant counts are drawn from official result exports and are approximate. The qualitative comparison in Table 1 reflects each platform's primary documented use, not an exhaustive feature audit. Finally, this is a single-platform experience report from one national olympiad; the architectural lessons should generalise to comparable settings, but we do not claim a controlled comparison against alternative designs.

10 Conclusion

ML olympiads need judging infrastructure built around predictions and heterogeneous metrics rather than programs and exact output checks. MLCompete embodies that shift: a metric-agnostic asynchronous evaluation pipeline, a hardened sandbox for untrusted participant code, and a data-driven multilingual interface, all operated declaratively at low cost. As the live infrastructure for all four stages of Romania's National AI Olympiad and a growing international audience, it offers both a reference architecture and production evidence that the architecture works. Future work targets stronger kernel-level sandboxing for code execution, richer per-task analytics for educators, and a staging-environment load test that validates the scalability projected here (Section 7.8) against the live autoscaler under a full contest-day profile.

Data and code availability. The telemetry-collection scripts, the collected measurements, and the verbatim Prometheus API responses behind every reported figure and table are openly available at https://github.com/RO-OAI/ro-oai-platform-artifacts (code under the MIT License, data under CC BY 4.0). Because the production Prometheus retains roughly 15 days of data, the repository also archives frozen snapshots so the results remain verifiable after the live series expire; the figures can be regenerated from the committed data without access to the cluster. The platform service code is not released, as it is operational infrastructure for a live national competition.

Acknowledgments. The authors thank the organising teams, problem authors, and volunteers of the Romanian National AI Olympiad, and the participating students and institutions.

Disclosure of Interests. The first author is the technical lead of the platform described and serves on the organising commission of the Romanian National AI Olympiad. The authors declare no other competing interests.

References

1. Waśik, S., Antczak, M., Badura, J., Laskowski, A., Sternal, T.: A survey on online judge systems and their applications. ACM Computing Surveys **51**(1), Article 3, 1–34 (2018). https://doi.org/10.1145/3143560
2. Pavao, A., Guyon, I., Letournel, A.-C., Tran, D.-T., Baro, X., Escalante, H.J., Escalera, S., Thomas, T., Xu, Z.: CodaLab Competitions: an open source platform to organize scientific challenges. Journal of Machine Learning Research **24**(198), 1–6 (2023)
3. Yadav, D., Jain, R., Agrawal, H., Chattopadhyay, P., Singh, T., Jain, A., Singh, S.B., Lee, S., Batra, D.: EvalAI: towards better evaluation systems for AI agents. arXiv:1902.03570 (2019). https://doi.org/10.48550/arXiv.1902.03570
4. Xu, Z., Zhao, H., Tu, W.-W., Richard, M., Escalera, S., Guyon, I.: Codabench: flexible, easy-to-use, and reproducible meta-benchmark platform. Patterns **3**(7), 100543 (2022). https://doi.org/10.1016/j.patter.2022.100543
5. Paiva, J.C., Leal, J.P., Figueira, Á.: Automated assessment in computer science education: a state-of-the-art review. ACM Transactions on Computing Education **22**(3), Article 34, 1–40 (2022). https://doi.org/10.1145/3513140
6. Dicheva, D., Dichev, C., Agre, G., Angelova, G.: Gamification in education: a systematic mapping study. Journal of Educational Technology & Society **18**(3), 75–88 (2015)
7. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., Dennison, D.: Hidden technical debt in machine learning systems. In: Advances in Neural Information Processing Systems 28 (NeurIPS), pp. 2503–2511 (2015)
8. Kaggle: your machine learning and data science community. https://www.kaggle.com, last accessed 2026/05/29
9. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. Communications of the ACM **59**(5), 50–57 (2016). https://doi.org/10.1145/2890784
10. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
11. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux Journal **2014**(239), Article 2 (2014)
12. Hardt, D. (ed.): The OAuth 2.0 authorization framework. RFC 6749, Internet Engineering Task Force (2012). https://doi.org/10.17487/RFC6749
13. Argo CD: declarative GitOps continuous delivery for Kubernetes. https://argo-cd.readthedocs.io, last accessed 2026/05/29

14. Keycloak: open source identity and access management. https://www.keycloak.org, last accessed 2026/05/29
15. Prometheus: from metrics to insight. https://prometheus.io, last accessed 2026/05/29
16. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al.: Scikit-learn: machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)