

<https://doi.org/10.31891/csit-2026-3-20>

UDC 004.415.2:004.8

Dmytro NOVAK

Candidate of Technical Sciences, Associate Professor of Information and Computer Technologies Department, Kyiv National University of Technologies and Design, Kyiv, Ukraine

<https://orcid.org/0000-0002-1796-8857>

e-mail: novak.knutd@gmail.com

Oleksandr PANASIUK

Master of Computer Science, PhD student, Kyiv National University of Technologies and Design, Kyiv, Ukraine

<https://orcid.org/0000-0003-3338-4744>

e-mail: vohigi@gmail.com

Received: 23/08/2026

Accepted: 22/09/2026

Published: 30/09/2026

© Copyright
2026 by the author(s)



This is an Open Access article distributed under the terms of the [Creative Commons CC-BY 4.0](https://creativecommons.org/licenses/by/4.0/)

ARCHITECTURAL PATTERNS FOR SOFTWARE DESIGN OF INTELLIGENT SYSTEMS

The design of software for intelligent systems increasingly extends beyond model training to encompass the full lifecycle of a machine learning (ML) component within a production software system: data ingestion, feature preparation, model serving, monitoring, and retraining. Conventional software architecture patterns, developed for deterministic systems, do not directly address the specific engineering challenges posed by ML components, namely non-determinism in outputs, dependence on data distributions that change over time, and coupling between code artifacts and data or model artifacts. This study addresses the problem of formulating a coherent architectural methodology for designing software for intelligent systems that treats the ML model as a first-class, versioned software component rather than as an external black box invoked by conventional application code. Recent research on MLOps, continuous training (CT) pipelines, microservice decomposition of inference services, and data-drift monitoring was analyzed, and the previously insufficiently addressed problem of integrating these separate practices into a single, reproducible architectural pattern was identified. A layered reference architecture for intelligent systems that separates the data layer, the model-training and versioning layer, the inference/serving layer, and the observability layer was proposed and justified. The supporting continuous integration, continuous delivery, and continuous training (CI/CD/CT) pipeline that binds these layers together was described. The main research material presents the architecture in detail and validates it with three reproducible Python experiments: a regression case study that predicts the volume electrical resistivity of carbon-filled polyethylene composites from experimental data used as a concrete intelligent-system component situated in the training and versioning layer; a discrete simulation of the observability layer's Kolmogorov-Smirnov drift monitor, which detected an injected gradual distribution shift with a measured delay of 100 requests; a discrete-event simulation of the serving layer comparing a synchronous and an asynchronous, task-queue-based design under identical offered load, showing a 136-fold reduction in 95th-percentile latency for the asynchronous design. The results indicate that separating concerns across the proposed four layers reduces the coupling between data-science experimentation and production software engineering, improves deployment reproducibility, and provides an explicit, empirically bounded mechanism for continuous training that keeps deployed models aligned with the current data distribution. These advantages, together with the architecture's current limitations, point to prospects for further research, including the extension of the architecture to support multi-model orchestration and automated rollback strategies driven by online quality metrics.

Keywords: intelligent systems; software architecture; continuous training; model versioning; data drift monitoring; microservice architecture.

Introduction

The engineering of intelligent systems, such as software systems in which one or more components implement behavior learned from data rather than behavior explicitly programmed by a developer, has become a mainstream concern in software engineering rather than a specialized subfield of data science [1]. As intelligent components move from research prototypes to production systems serving real users, the software design decisions surrounding a machine learning (ML) model become at least as consequential as the choice of the modeling algorithm itself [2]. A model with excellent offline accuracy has limited practical value if it cannot be deployed, versioned, monitored, and safely updated within the software system that hosts it [3].

The general problem addressed in this article is that conventional software architecture, built on the assumption that a program's behavior is fully determined by its source code, does

not, by itself, provide adequate guidance for the design of intelligent systems [1, 3]. Three properties distinguish an ML-enabled component: First, the ML component's behavior additionally depends on a training dataset and a set of learned parameters, both of which must be versioned alongside the code that produced them, and a change in either can silently change the system's behavior [4]. Second, the statistical assumptions under which a model was trained can become invalid as the distribution of incoming data changes over time, a phenomenon known as data drift or concept drift [5], so a deployed model requires continuous monitoring, not merely continuous availability [6]. Third, the model training is typically resource-intensive and asynchronous relative to the request-response cycle of the serving system, which favors an architecture that decouples the training and inference execution paths rather than treating a model as an ordinary synchronous library call [7, 8].

This connects to the broader engineering task of building software systems that must simultaneously satisfy the reliability and maintainability expectations of conventional production software and the experimental, iterative workflow of data science [2, 9]. Universities training software engineers for this environment face a corresponding pedagogical problem: a course on "software design of intelligent systems" must teach not only modeling techniques but a reproducible engineering discipline for taking a model from a training script to a monitored, retrainable production service.

Literature review

The DevOps literature established continuous integration and continuous delivery (CI/CD) as an architectural and organizational discipline for conventional software, emphasizing automated testing, small incremental releases, and infrastructure as code [10]. MLOps extends this discipline with a 3-axis, continuous training (CT), which recognizes that an ML model is not a static artifact but requires periodic retraining as data evolves, and that the retraining process itself must be automated, reproducible, and auditable in the same way as a conventional build pipeline [4, 9].

A separate but related line of research addresses the serving architecture for trained models. Two recurring design decisions appear across the literature and industrial practice: the isolation of the model-serving process behind a well-defined API, most commonly implemented with a lightweight asynchronous web framework, and the delegation of long-running or resource-intensive inference and training tasks to a distributed task queue backed by a message broker, so that the request-handling process remains responsive under load [7]. Containerization, most commonly with Docker, is consistently recommended as the mechanism for ensuring that the environment in which a model was trained matches the environment in which it is served, addressing a well-documented class of production failures caused by dependency or environment mismatch [3, 8].

Because a model's behavior is a function of both code and data, researchers and practitioners have proposed treating datasets as versioned artifacts using dedicated data version control tooling and treating trained models, their hyperparameters, and their evaluation metrics as versioned artifacts within an experiment-tracking registry [4]. This allows a specific deployed model to be traced back to the exact code revision, dataset version, and training configuration that produced it, a property that is taken for granted for conventional binaries but is non-trivial to achieve for ML artifacts [1, 4].

A growing body of work addresses observability for ML systems specifically, extending conventional application monitoring (latency, throughput, error rates) with statistical monitoring of the input feature distribution and the model's prediction distribution over time, in order to detect data drift and concept drift before they manifest as a measurable degradation in business metrics [5, 6]. Related sentiment-analysis and public-opinion-monitoring pipelines built on social-media data illustrate the same underlying engineering need for continuous retraining and text-classification quality monitoring once a model is deployed against a live, non-stationary data stream [11].

While each of the practices described above (CI/CD/CT pipelines, containerized serving, asynchronous task-queue architectures, data and model versioning, and drift monitoring) is individually documented in the literature and in industrial engineering blogs [4-9], they are rarely presented as a single coherent architectural pattern with explicit layer boundaries and explicit interfaces between layers. Practitioner-oriented sources tend to describe individual tools (a specific experiment tracker, a specific orchestration framework) rather than the architectural roles those tools fill, which makes the resulting guidance difficult to transfer to a different tool stack or to teach as a generalizable design skill [3, 9]. Empirical studies of how software engineering teams actually build ML-enabled products confirm that this tooling fragmentation is a genuine friction point in practice, not only a gap in the literature [12]. In an academic software-engineering curriculum, an architecture is described in terms of layers and responsibilities, with a specific tool stack used only as a valid instantiation, so that the underlying design principles remain applicable as the tool ecosystem evolves [2]. This gap (the absence of a layered reference architecture for intelligent systems suitable for both engineering practice and academic instruction) is the specific unresolved problem addressed by this study.

Objectives of the Article

The objective of this article is to propose and justify a layered reference architecture for the software design of intelligent systems that separates concerns into a data layer, a model-training and versioning layer, an inference/serving layer, and an observability layer; specifies the responsibilities and interfaces of each layer independently of any single vendor's tooling; describes a concrete CI/CD/CT pipeline that automates the transition of

an artifact between layers, including the automatic triggering of retraining upon detection of data or concept drift; validates the key architectural claims with three small, reproducible Python experiments – a regression case study using experimental data, a drift-detection simulation, and a serving-latency simulation.

Research methods and tools

Layered Reference Architecture

The proposed architecture decomposes an intelligent system into four layers, illustrated schematically in Fig. 1: the data layer, the training and versioning layer, the serving layer, and the observability layer. Each layer communicates with its neighbors through an explicit, versioned artifact (a dataset version, a registered model, or a monitoring signal) rather than through direct code-level coupling [4]. This artifact-based interface is the key architectural decision that allows the data-science workflow (experimentation, retraining) to evolve independently of the production serving workflow (request handling, scaling, deployment) [7], while still guaranteeing that the second remain synchronized through automation rather than manual coordination [9, 10].

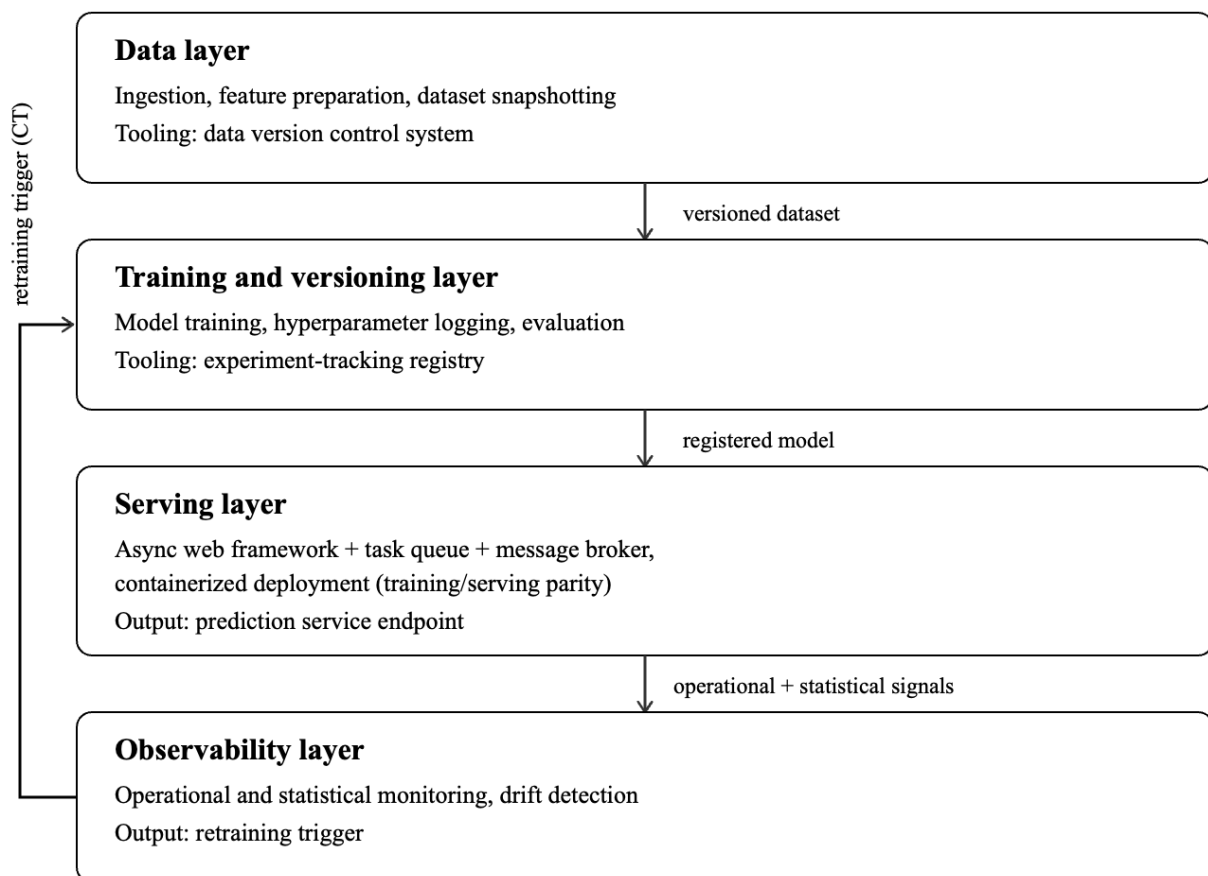


Fig. 1. Layered reference architecture for intelligent systems, with CT feedback path from the observability layer to the training and versioning layer

The data layer is responsible for ingesting raw data, applying feature transformations, and producing a versioned dataset snapshot. Using a dedicated data version control tool alongside the code version control system allows a dataset snapshot to be referenced by a content hash in the same way source code is referenced by a commit hash, so that a training run can be fully reproduced by checking out a specific code commit together with the corresponding dataset version [4].

The training and versioning layer consumes a specific dataset version and produces a trained model artifact, which is registered in an experiment-tracking registry along with its hyperparameters, evaluation metrics, and references to the training code commit and the dataset version that produced it [4]. Registering every training run, not only the ones ultimately selected for deployment, preserves an audit trail that supports later comparison and rollback [1, 3].

The serving layer exposes a registered model as a network-accessible prediction service. The serving layer is built around a lightweight asynchronous web framework that accepts prediction requests and immediately delegates any computation exceeding a latency budget to a distributed task queue backed by a message broker; the requesting

client receives a task identifier and polls or is notified upon completion [7, 13]. This decoupling of request acceptance from computation prevents slow inference or batch-scoring tasks from blocking the service's availability to other clients and allows the computational workers to be scaled independently of the request-handling process [13]. Both the request-handling process and the computational workers are packaged as containers built from the same base image used during training, which eliminates an entire class of "it worked in the notebook" failures caused by library version mismatches between the training and serving environments [8].

The observability layer collects two categories of signals from the serving layer: conventional operational metrics (request latency, throughput, error rate) and statistical metrics describing the distribution of input features and predictions observed in production [6]. Category 2 is specific to intelligent systems and enables the architecture to detect data drift (a change in the distribution of input features relative to the distribution seen during training) and concept drift (a change in the relationship between input features and the correct output) [5]. When either metric exceeds a configured threshold, the observability layer emits a retraining trigger, which is automatically consumed by the training and versioning layer, closing the loop between production behavior and the training process without requiring manual intervention [6].

Continuous Integration, Delivery, and Training Pipeline

The four layers are bound together by a CI/CD/CT pipeline. Continuous integration validates that code changes to the training or serving logic pass automated tests before merging [10]. Continuous delivery packages a newly registered model, once it passes an automated evaluation gate comparing its metrics against the currently deployed model, into the container image used by the serving layer and deploys it, typically behind a mechanism that allows a fraction of production traffic to be routed to the new model before a full rollout [9, 10]. Continuous training is the pipeline stage specific to intelligent systems: it is triggered either on a fixed schedule or, more informatively, by the retraining trigger emitted by the observability layer, and it re-executes the training layer against the most recent dataset version without requiring a developer to manually initiate the process [4, 6, 9].

Experiments

To move beyond a purely descriptive presentation of the architecture, three small, fully reproducible computational experiments were carried out in Python. The first situates a concrete intelligent-system component (a regression model) within the training and versioning layer using real experimental data; the second and third validate the observability and serving layers, respectively, using simulation. All 3 experiments are summarized in Table 1.

Table 1

Summary of experimental results

Experiment	Metric	Result
Regression case study	LOO-CV MAE, log10(ρ), Ohm·m	Linear: 2.14; Random forest: 1.88
	LOO-CV R-squared	Linear: 0.14; Random forest: 0.34
Drift detection	Detection delay (requests)	100 (window = 100, check interval = 20)
Serving latency	p95 latency, synchronous	348.8 s
	p95 latency, asynchronous (4 workers)	2.56 s (136x reduction)

Case study: a regression model as the intelligent-system component

The training and versioning layer of the architecture requires a concrete trainable component to validate against. Rather than a synthetic benchmark dataset, this experiment uses the author's own experimental dataset from a prior technological study of carbon-filled polyethylene composites, comprising 20 samples with known filler composition (copper-plated graphite, carbon nanotubes, and carbon black, individually and in binary mixtures at total filler contents of 5-30 wt. %) and the corresponding measured volume electrical resistivity. Predicting resistivity from formulation is a realistic regression task that an intelligent system in a materials-processing pipeline would need to learn, and the dataset's small size and non-linear percolation-type behavior make it a meaningful stress test for the evaluation-gate logic in the proposed CI/CD/CT pipeline [14].

The two candidate models were trained on the base-10 logarithm of resistivity (a standard transformation for percolation-type data spanning several orders of magnitude): an ordinary linear regression as the currently deployed baseline, and a random-forest regressor with 200 trees as a candidate for promotion. Both were evaluated with leave-one-out cross-validation (LOO-CV), the natural evaluation protocol for a 20-sample dataset and a reasonable standard for the automated evaluation gate that a continuous-delivery pipeline would run before promoting a newly trained model. The linear baseline achieved a LOO-CV mean absolute error of 2.14 (log10 Ohm·m) with a coefficient of determination R-squared of 0.14; the random-forest candidate achieved a lower LOO-CV mean absolute error of 1.88 with R-squared of 0.34. Fig. 2 plots the random forest's leave-one-out predictions against the measured values.

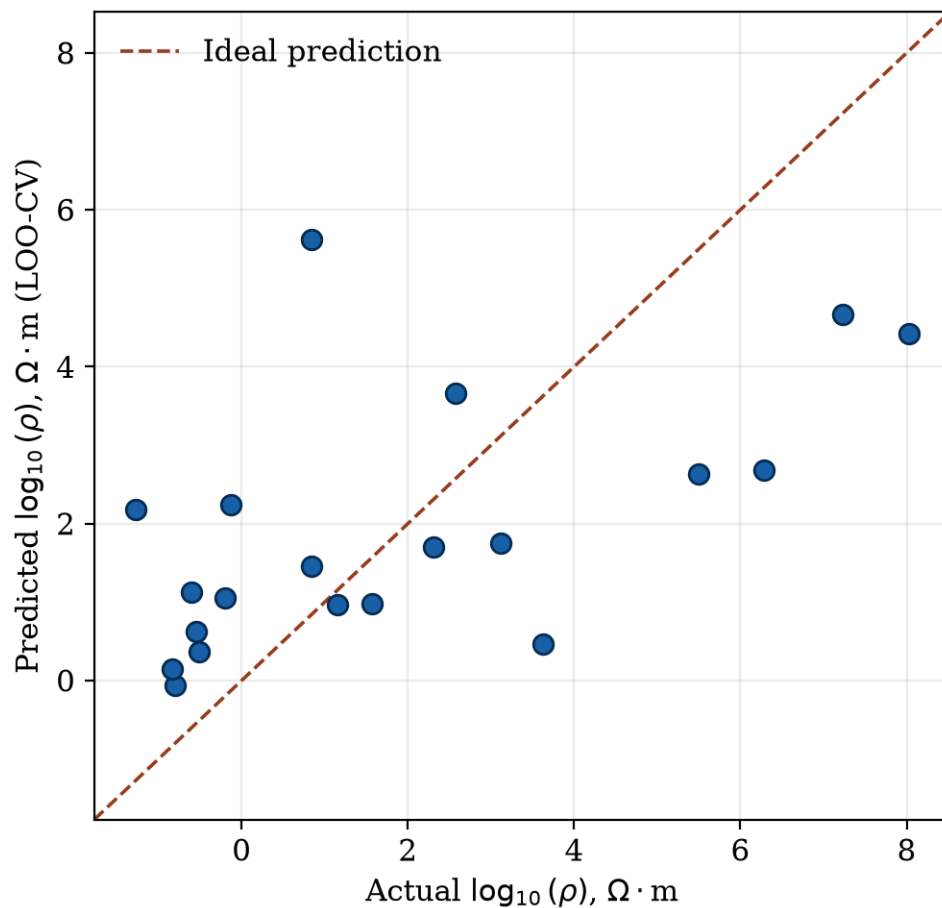


Fig. 2. Leave-one-out predictions of the random-forest resistivity model against measured values from the dataset (perfect prediction would lie on the dashed line)

Applying the evaluation-gate rule used in the simulated pipeline (promote the candidate only if its LOO-CV error improves on the deployed model), the random forest is promoted. The scatter around the ideal-prediction line in Fig. 2 also reveals a clear limitation: with only 20 samples spanning a physically discontinuous percolation transition, no model achieves high absolute accuracy, and the R-squared of 0.34 is modest. This is itself an argument for the proposed architecture rather than against it: the same evaluation-gate mechanism that promoted this candidate today would, under continuous training, automatically re-evaluate future candidates as more experimental samples are registered in the data layer, without requiring a developer to determine when the model has become stale manually.

Simulation: observability-layer drift detection and the CT trigger

This experiment validates the retraining-trigger mechanism described for the observability layer. A synthetic input stream of 2000 requests was generated: the first 1000 monitored requests are drawn from a fixed reference distribution (matching a 200-sample training-time reference set), after which the distribution's mean drifts linearly over the remaining 1000 requests, modeling a gradual covariate shift of the kind that motivates continuous training in practice [5]. A sliding window of 100 requests was compared against the reference sample every 20 requests using a 2-sample Kolmogorov-Smirnov test [15], with the retraining trigger firing at the first window whose p-value fell below a significance threshold of 0.01.

The monitor correctly held its p-values well above the threshold throughout the pre-drift period and fired the retraining trigger 100 requests after the true onset of drift, as shown in Fig. 3. This detection delay is an explicit, tunable property of the architecture that was governed by the window size and the check interval rather than an unknown or unbounded quantity, which is precisely the operational guarantee that distinguishes the proposed explicit observability layer from an architecture that lacks one: without a monitor of this kind, the corresponding degradation would only become visible once it affected a downstream business metric, with no comparable bound on detection delay.

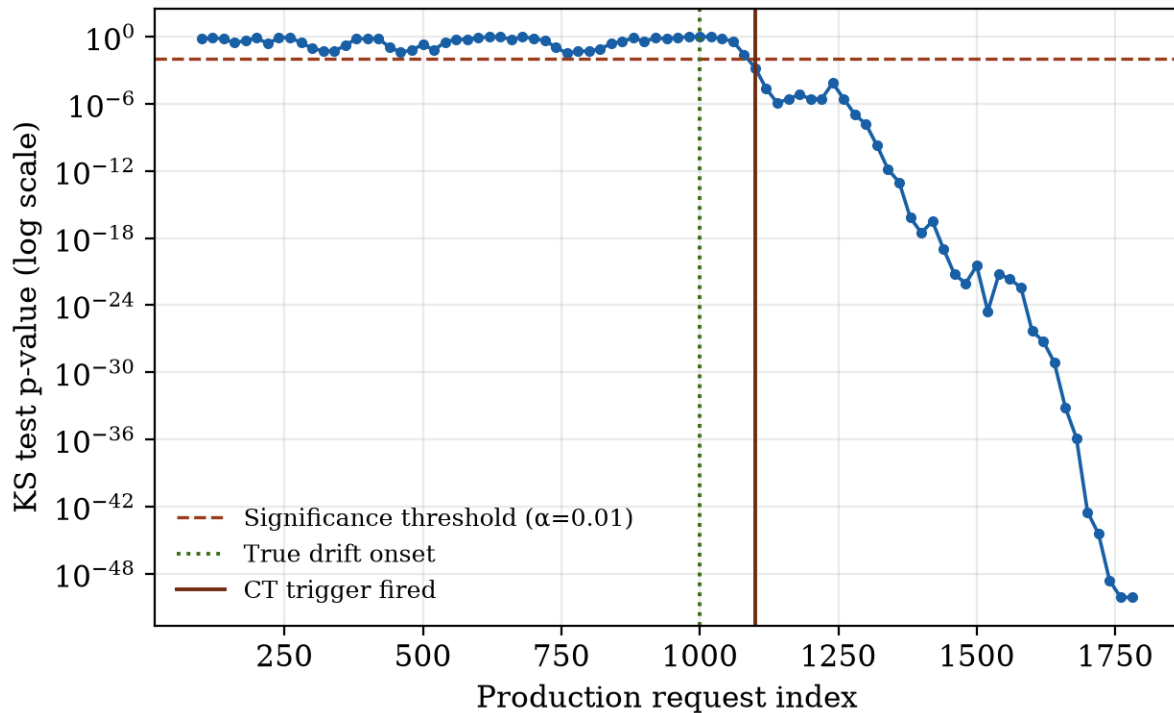


Fig. 3. Kolmogorov-Smirnov p-value monitored over the simulated production stream, showing the true drift onset and the resulting continuous-training trigger

Simulation: serving-layer latency under load

This experiment quantifies the engineering motivation for the task-queue-based serving layer [7, 13]. A discrete-event simulation (built with the SimPy library) modelled 600 seconds of request traffic arriving as a Poisson process with a mean inter-arrival time of 0.35 s and a log-normal inference-time distribution with a mean of 0.9 s, under two serving designs sharing identical arrival and inference-time statistics: a synchronous design in which a single request-handling process serves 1 request at a time, and the proposed asynchronous design in which requests are placed on a task queue served by a pool of four independent workers.

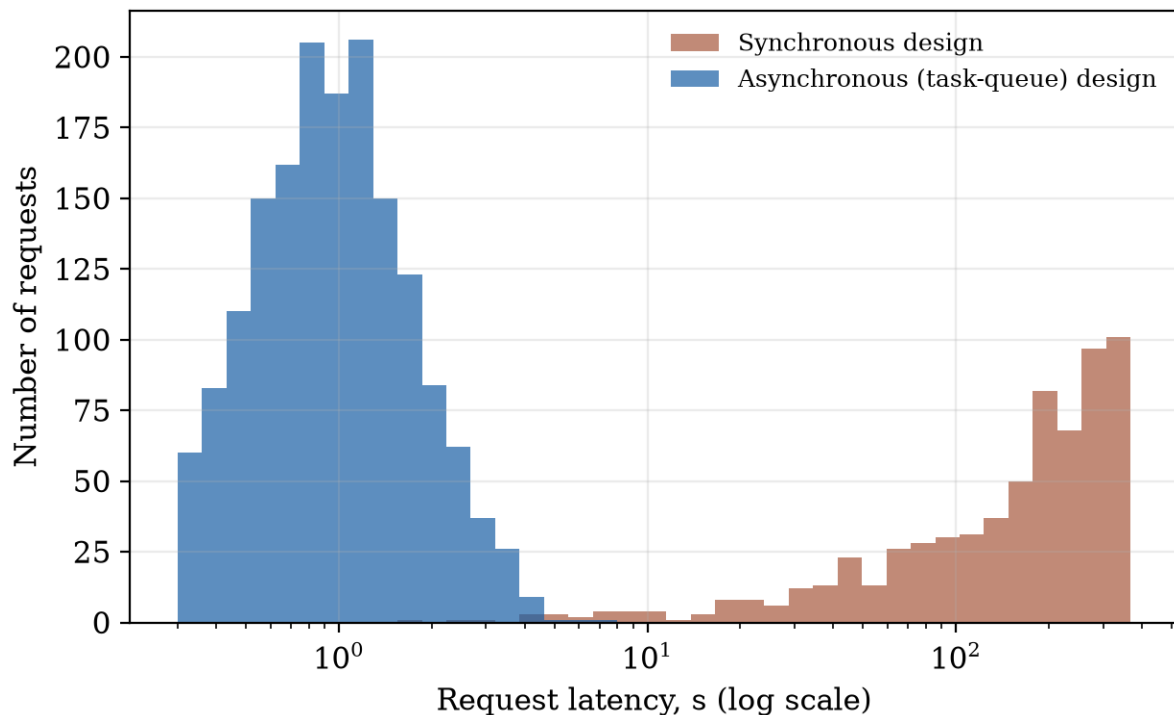


Fig. 4. Simulated request-latency distributions for synchronous versus asynchronous (task-queue-based) serving-layer designs under identical offered load

Under this offered load, the synchronous design's single handler cannot keep pace with the arrival rate: queued requests accumulate throughout the simulation, and the 660 requests it completed show a median latency of 186 s and a 95th-percentile latency of 349 s. The asynchronous design, with the same total inference workload distributed across 4 workers, completed 1718 requests with a median latency of 0.92 s and a 95th-percentile latency of 2.56 s – a 136-fold reduction at the 95th percentile. Fig. 4 shows the resulting latency distributions on a logarithmic scale. The result is unsurprising queueing-theory behavior for a single-server system operating near or above capacity [16]. Still, it makes concrete, in the vocabulary of the architecture proposed in this study, exactly the production failure mode (such as unbounded queueing and request starvation for other clients) that motivates delegating inference to an independently scalable worker pool rather than serving it synchronously within the request-handling process [7, 13].

Discussion

Compared with a monolithic design in which a single service both trains and serves a model synchronously, the proposed layered architecture offers three principal engineering advantages: First, it reduces coupling between the experimental, iterative work of model development and the reliability-oriented work of production serving, since each layer can be modified, tested, and scaled independently as long as its artifact-based interface to neighboring layers is preserved [2, 7]. Second, it provides an explicit and automatable mechanism for keeping a deployed model aligned with the current data distribution, rather than relying on ad hoc manual retraining decisions [5, 6]. Third, because every training run and every dataset version is registered, the architecture provides an audit trail that supports reproducibility and rollback, properties that are difficult to retrofit onto a system that was not designed with artifact versioning from the outset [1, 4].

The main limitation of the proposed architecture, as implemented and evaluated to date, is that the drift-detection rule used to trigger continuous training in the laboratory implementation is a comparatively simple statistical threshold rather than a more sophisticated online monitoring model, and that the evaluation gate governing automated promotion of a newly trained model compares only a small set of offline metrics rather than a full online experiment. These simplifications were appropriate for an instructional setting in which the priority is to demonstrate the architectural pattern rather than to optimize the specific decision rules, but they identify concrete directions for further research.

Conclusions

This study proposed a layered reference architecture for the software design of intelligent systems that separates a data layer, a training and versioning layer, a serving layer, and an observability layer, connected through explicit versioned artifacts rather than direct code coupling, and described the CI/CD/CT pipeline that automates the transition of artifacts between these layers, including automated triggering of continuous training upon detection of data or concept drift. The architecture's key claims were validated with three reproducible Python experiments: a regression case study on real experimental composite-materials data showed the automated evaluation gate correctly selecting the better-performing candidate model while honestly exposing the accuracy ceiling imposed by a small dataset; a drift-detection simulation showed the observability layer's statistical monitor identifying an injected distribution shift within a bounded, tunable delay of 100 requests; and a serving-latency simulation showed a 136-fold reduction in 95th-percentile latency for the proposed asynchronous, task-queue-based serving design compared with a synchronous alternative under identical load. Unlike prior descriptions of individual MLOps tools and practices, the contribution of this study organizes these practices into a layer-and-interface architecture that is explicitly tool-agnostic and therefore transferable across technology stacks, supports each architectural claim with a runnable, reproducible experiment rather than a qualitative argument alone, and demonstrates the architecture's suitability as the organizing structure for MLOps practices.

Prospects for further research include extending the observability layer with more sophisticated online drift-detection models capable of distinguishing gradual drift from transient anomalies, extending the serving layer toward the orchestration of multiple concurrently deployed models to support online comparison, and extending the continuous delivery stage with automated rollback strategies driven directly by online quality metrics rather than by offline evaluation gates alone.

ADDITIONAL INFORMATION

AUTHOR CONTRIBUTIONS

Conceptualization, D.N.; methodology, O.P.; validation, D.N.; formal analysis, O.P.; investigation, O.P.; data curation, D.N.; writing-original draft preparation, O.P.; writing-review and editing, D.N.; visualization, O.P.; supervision, D.N.; project administration, D.N. All authors have read and agreed to the published version of the manuscript.

DECLARATION ON THE USE OF GENERATIVE ARTIFICIAL INTELLIGENCE TOOLS

In preparing this work, the author used DeepL Translate and Grammarly for: grammar and spelling checks, paraphrasing, and rephrasing. After using these tools/services, the author reviewed and edited the content and takes full responsibility for the content of this publication.

REFERENCES

1. Eremenko V., Zaporozhets A. (Eds.), Advanced information-measuring technologies and systems II, Springer, Cham, Switzerland, 2025; 160 p.
20. Hough P., Environmental security: an introduction, 2nd Edition, Routledge, 2021; 220 p.
1. Sculley D., Holt G., Golovin D. et al. Hidden technical debt in machine learning systems // Advances in Neural Information Processing Systems. 2015. Vol. 28. P. 2503–2511.
2. Lewis G. A., Ozkaya I., Xu X. Software architecture challenges for ML systems // Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2021. P. 634–638.
<https://doi.org/10.1109/ICSME52107.2021.00071>
3. Paleyes A., Urma R.-G., Lawrence N. D. Challenges in deploying machine learning: a survey of case studies // ACM Computing Surveys. 2022. Vol. 55, No. 6. Art. 114. P. 1–29.
<https://doi.org/10.1145/3533378>
4. Zaharia M., Chen A., Davidson A. et al. Accelerating the machine learning lifecycle with MLflow // IEEE Data Engineering Bulletin. 2018. Vol. 41, No. 4. P. 39–45.
5. Gama J., Žliobaitė I., Bifet A. et al. A survey on concept drift adaptation // ACM Computing Surveys. 2014. Vol. 46, No. 4. Art. 44. P. 1–37.
<https://doi.org/10.1145/2523813>
6. Breck E., Cai S., Nielsen E. et al. The ML test score: a rubric for ML production readiness and technical debt reduction // Proceedings of the 2017 IEEE International Conference on Big Data. 2017. P. 1123–1132.
<https://doi.org/10.1109/BigData.2017.8258038>
7. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 612 p.
8. Merkel D. Docker: lightweight Linux containers for consistent development and deployment // Linux Journal. 2014. Vol. 2014, No. 239. Art. 2.
9. Kreuzberger D., Kühl N., Hirschl S. Machine learning operations (MLOps): Overview, definition, and architecture // IEEE Access. 2023. Vol. 11. P. 31866–31879.
<https://doi.org/10.1109/ACCESS.2023.3262138>
10. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston: Addison-Wesley, 2010. 512 p.
11. Lynnyk R., Vysotska V., Chyrun L. Sentiment analysis of public opinion regarding the war in Ukraine based on Reddit data using NLP and machine learning methods // Computer Systems and Information Technologies. 2026. № 2. P. 214–228.
<https://doi.org/10.31891/csit-2026-2-18>
12. Amershi S., Begel A., Bird C. et al. Software engineering for machine learning: A case study // Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). 2019. P. 291–300.
<https://doi.org/10.1109/ICSE-SEIP.2019.00042>
13. Celery Project. Celery: Distributed Task Queue. Version 5.6 documentation. URL: <https://docs.celeryq.dev/>
14. Novak D., Guida O. Image-Driven Modeling of Percolation Networks in Carbon-Based Conductive Polymer Composites // Scientific Notes of Taurida National V. I. Vernadsky University. Series: Technical Sciences. 2026. Vol. 37(76), № 1, Part 2, 269–273.
<https://doi.org/10.32782/2663-5941/2026.1.2/34>
15. Massey F. J. The Kolmogorov-Smirnov test for goodness of fit // Journal of the American Statistical Association. 1951. Vol. 46, No. 253. P. 68–78.
<https://doi.org/10.1080/01621459.1951.10500769>
16. Kleinrock L. Queueing Systems. Volume 1: Theory. New York: Wiley-Interscience, 1975. 448 p.

Дмитро НОВАК, Олександр ПАНАСЮК
Київський національний університет технологій та дизайну

АРХІТЕКТУРНІ ПАТЕРНИ ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ

Проектування програмного забезпечення інтелектуальних систем дедалі більше виходить за межі навчання моделі та охоплює повний життєвий цикл компонента машинного навчання (МН) у складі промислової програмної системи: підготовку даних, формування ознак, обслуговування моделі, моніторинг і повторне навчання. Традиційні архітектурні патерни програмного забезпечення, розроблені для детермінованих систем, не враховують специфічних інженерних проблем, породжених компонентами МН, а саме: недетермінованості результатів, залежності від розподілів даних, що змінюються з часом, та зв'язності між артефактами коду й артефактами даних чи моделі. Це дослідження розглядає проблему формування цілісної архітектурної методології проектування програмного забезпечення інтелектуальних систем, за якої модель МН розглядається як повноцінний версійований програмний компонент, а не як зовнішній «чорний ящик», що викликається звичайним прикладним кодом. Проаналізовано сучасні дослідження з MLOps, конвеєрів безперервного навчання (СТ), мікросервісної декомпозиції сервісів інференсу та моніторингу дрейфу даних, визначено недостатньо розкриті проблеми інтеграції цих окремих практик у єдиний відтворюваний архітектурний патерн. Було запропоновано та обґрунтовано багаторівневу еталонну архітектуру для інтелектуальних систем, яка розділяє рівень даних, рівень навчання моделей та

управління версіями, рівень інференції/обслуговування та рівень спостережуваності. Також було описано супровідний конвеєр безперервної інтеграції, безперервної доставки та безперервного навчання (CI/CD/CT), який об'єднує ці рівні. В основній частині дослідження детально викладено архітектуру та проведено її валідацію 3 відтворюваними Python-експериментами: регресійним кейсом прогнозування питомого об'ємного електричного опору струмопровідних поліетиленових композицій на основі експериментальних даних, використаних як конкретний компонент інтелектуальної системи в рівні навчання та версіонування; дискретним моделюванням детектора дрейфу на основі критерію Колмогорова-Смірнова в рівні спостережуваності, який виявив штучно внесений поступовий зсув розподілу із затримкою 100 запитів; імітаційним моделюванням рівня обслуговування, яке порівнює синхронну та асинхронну (на основі черги завдань) архітектури за однакового навантаження і демонструє 136-кратне зменшення затримки на 95-му перцентилі для асинхронної архітектури. Результати свідчать, що розділення завдань за запропонованими 4 рівнями зменшує зв'язок між дослідницькою діяльністю з обробки даних і промисловою інженерією програмного забезпечення, підвищує відтворюваність розгортань і забезпечує явний, емпірично обґрунтований механізм безперервного навчання, який утримує розгорнуті моделі узгодженими з поточним розподілом даних. Переваги багаторівневого підходу порівняно з монолітними архітектурами МН-сервісів окреслюють перспективи подальших досліджень, зокрема, розширення архітектури для підтримки багатомодельної оркестрації та автоматизованих стратегій відкату, керованих онлайн-метриками якості.

Ключові слова: інтелектуальні системи; архітектура програмного забезпечення; безперервне навчання; версіонування моделей; моніторинг дрейфу даних; мікросервісна архітектура.