

Strategies for Scaling to a Large Number of AI Models in production

Mattias Jonhede, Christian Beckers, Johnny Westerlund, Lex Avstreikh, Hamid Ahmadian, Wen Zhou, Robert Lundberg, Cansu Kavili Oernek, Daniel Jönsson, and Mattias Tiger

Executive summary

Organizations that already run a handful of machine-learning models can hit a wall when they try to scale to hundreds or thousands of models. Without automation, each additional model adds cost, coordination overhead, compliance risk, and operational fragility. This white paper proposes **fleet-native** MLOps practices that let enterprises scale model count **without** headcount increasing immensely.

What we did. Volvo, Hopsworks, Red Hat and Linköping University analyzed the problem in depth and synthesized clear descriptions of the challenges and candidate solutions. Based on this, the solution providers then implemented two complementary, production-oriented approaches: **(1) modular pipelines and GitOps promotion** on OpenShift (Red Hat), and **(2) data-centric parametrization with feature views and a unified model registry** (Hopsworks). We validated them through workshops and multi-month iterations focused on automation, orchestration, monitoring, and governance.

What you get. Practical **blueprints, templates, and patterns** to: (i) parameterize model lifecycles, (ii) reuse pipelines safely across many models, (iii) centralize lineage and governance, (iv) automate monitoring and closed-loop retraining, and (v) promote models using human-in-the-loop gates where needed.

Results in brief. These approaches show how to:

- Reduce the **human-per-model** ratio by shifting repetitive lifecycle work to automated pipelines.
- Shorten **time-to-retrain/promote** using event-driven triggers (e.g., drift → retrain) and GitOps promotion.
- Increase **reliability and auditability** via versioned data, features, models, and environments with clear lineage.
- Preserve **vendor flexibility** through open components (Kubernetes, Tekton, KubeFlow, KServe, MLflow, Feast/Hopsworks).
- Align with **European governance** expectations (traceability, monitoring, human oversight), building on AI Application Operations.

Who should read this. CTOs, product owners, ML leaders, MLOps/DevOps engineers, and compliance teams that need to scale the number of models while protecting quality, cost, and trust.

Table of content

Introduction	3
The Thousand-Model Challenge	4
Method	5
Automating Machine Learning Operations	6
MLOps: Organizational Processes	8
MLOps: Value-Driven Development	10
MLOps: Model Lifecycle	11
MLOps: Continuous Performance	13
MLOps for Continuous Operation	14
Continuous Monitoring for MLOps	15
Pipeline-Centric Scaling: “Assembly Line for Models”	18
MLOps for Continuous Improvement	18
What Must Be True Before Scaling to 1000 Models	20
Why the Needs Exist: From 1 Model → 50 → 1000	20
Core Needs for Fleet-Level Model Operations	20
Architectural Patterns for Scaling Model Fleets	21
Data-Centric Scaling: “Shared, Versioned Features as a First-Class Asset”	23
Relationship Between the Two Patterns	25
Business Context and Enterprise Motivation: High level solution to scale value extraction from ML models [Volvo]	25
Solution Perspective #1:	
Modular pipelines for scaling the number of models [Red Hat]	29
Solution Perspective #2:	
Data-Centrisism and Parametrization for Scaling AI Systems [Hopsworks]	35
Discussion	39
Conclusion and Recommendations	41
Appendix	42

Acknowledgements

Many people in the DDO project gave valuable comments or were included in discussions towards writing this white paper. We would like to thank Jens Augustsson, Håkan Stensby and Karl Sjöborg for their participation in multiple group discussions.

Introduction

This whitepaper focuses on solutions to automate deployment, training and management of model lifecycles in production environments at enterprise scale.

Many companies have successfully implemented machine learning models in production in their products and services, but still struggle to scale up the number of models without also having to scale up the number of people handling the models' lifecycles. In principle, their machine learning operations¹ (MLOps) teams (often a mix of Data science, DevOps, Data Engineering and Infrastructure teams) should focus on developing new or improved value-adding models and integrate them into real-world usage to reach fast added bottom-line value. Instead, many teams spend time on manual model and data life-cycle tasks when both creating and updating already productionised models.

The need to scale the number of machine learning models in production comes from various company related factors, as well as from the technical nature of the models themselves:

- **Internal factors** include organizations growing and diversifying their operations. Different teams may require tailored models to address specific use cases, such as customer segmentation, geographic locality, or sensor characteristics. Additionally, the adoption of MLOps practices encourages modularity and experimentation, leading to a proliferation of models across environments.
- **External factors** are related to evolving market demands, regulatory requirements, and competitive pressures that often necessitate rapid deployment of new models. To stay competitive in a global market, it is essential to move towards more granular and specialized ML models. Consequently, the number of models will keep on growing, and so does the need for automation of the model lifecycle.
- **Technical factors** are related to the machine learning models themselves; in a production environment they will encounter concepts and data drift (or shift) from their original source; their performance will decay over time as internal and external factors evolve and influence the model's quality. This leads to regular retraining and requirements on monitoring the models' output.

Concretely, if it takes 10 persons to manage 1 model today, it is paramount that it does not require 10000 persons to manage 1000 models.

In this white paper, we use real-world cases from Volvo to frame the problem, and present two architectural solution approaches that provide organisations with actionable practices, frameworks, and implementation blueprints.

What this paper delivers.

- A **problem framing** of the Thousand-Model Challenge and why classic “one-off” MLOps breaks at scale.
- A **Volvo perspective** connecting technical automation to **value extraction** in real products, with continuous retraining and observability.

¹ D. Kreuzberger, N. Kühn, and S. Hirschl, “Machine learning operations (MLOps): Overview, definition, and architecture,” IEEE Access, vol. 11, pp. 31 866–31 879, 2023.

- Two **solution perspectives**, pipeline modularization (assembly-line pattern) and **data-centric parametrization**, including architecture blueprints and example repos.
- **Checklists and recommendations** you can adopt incrementally without pausing current delivery.

Preliminary reading: For a broader view on organizational transformation for realizing an effective data-driven organisation, see *Building Blocks for Operationalization of AI²*. For a broader view on MLOps, and Monitoring, in a value-driven context, see *AI Application Operations - A Socio-Technical Framework for Data-driven Organizations*. These two works were done within the same project as this work, see details in the Appendix.

The Thousand-Model Challenge

To unlock the potential business value of deploying a large number of specialized machine learning models—such as improved accuracy, responsiveness to local conditions, and adaptability to evolving requirements—organizations must overcome a critical scalability challenge, which we refer to as **the Thousand-Models Challenge**: how to manage the lifecycle of thousands of models without unsustainably increasing the number of professionals. This challenge introduces a range of complexities across organizational, process, and technological dimensions. In this whitepaper, we focus specifically on the technological aspect, with an emphasis on automation as a key enabler for scalable model operations.

Unlike traditional software, machine learning models are inherently dynamic and sensitive to changes in data, environments, and business contexts³. Their performance can degrade over time due to phenomena such as data drift⁴, requiring continuous monitoring, retraining, and updating. Moreover, as a company's resources, processes, customer behavior, legal frameworks, global supply chains, and geopolitical conditions evolve, so too must the ML-powered software. This introduces a need for additional lifecycle capabilities such as data versioning, automated retraining, and adaptive deployment strategies.

Further complexities arise when the pace of model retraining does not align with the pace of other software functionality enhancements, potentially leading to asynchronous development and deployment cycles. Differences between training and inference environments can also introduce friction, requiring robust environment management and reproducibility mechanisms. Additionally, ML models often make complex multivariate decisions that are not easily understandable by humans, which increases the need for continuous business monitoring to ensure alignment with strategic goals and compliance requirements.

Finally, as complexity grows, new challenges arise in the management of those models. The propension for production skew⁵ (when data is processed differently during inference than during training) in complex ML systems leads to unreliable models. Feature inconsistency can escalate as models reuse overlapping data and transformations. Versioning of the models, their data,

² www.ai.se/ddo

³ Sculley, David, et al. "[Hidden technical debt in machine learning systems](#)." *Advances in neural information processing systems* 28 (2015).

⁴ Data drift: the statistical properties of the input data change over time, causing model performance to degrade.

⁵ Sun, Chong et al. "[Production Skew: Machine Learning Model management System at Uber](#)", Open Proceedings (2020).

other assets, and eventually a clear lineage across different -- sometimes disjointed -- systems, becomes paramount to production stability.

To address these challenges, **we draw inspiration from DevOps practices in software engineering**, which have successfully leveraged automation to streamline development and deployment processes. **Our goal is to explore how similar automation strategies can be applied to MLOps/ModelOps across the model lifecycle**—from development and deployment to monitoring and maintenance—to enable sustainable scaling.

Method

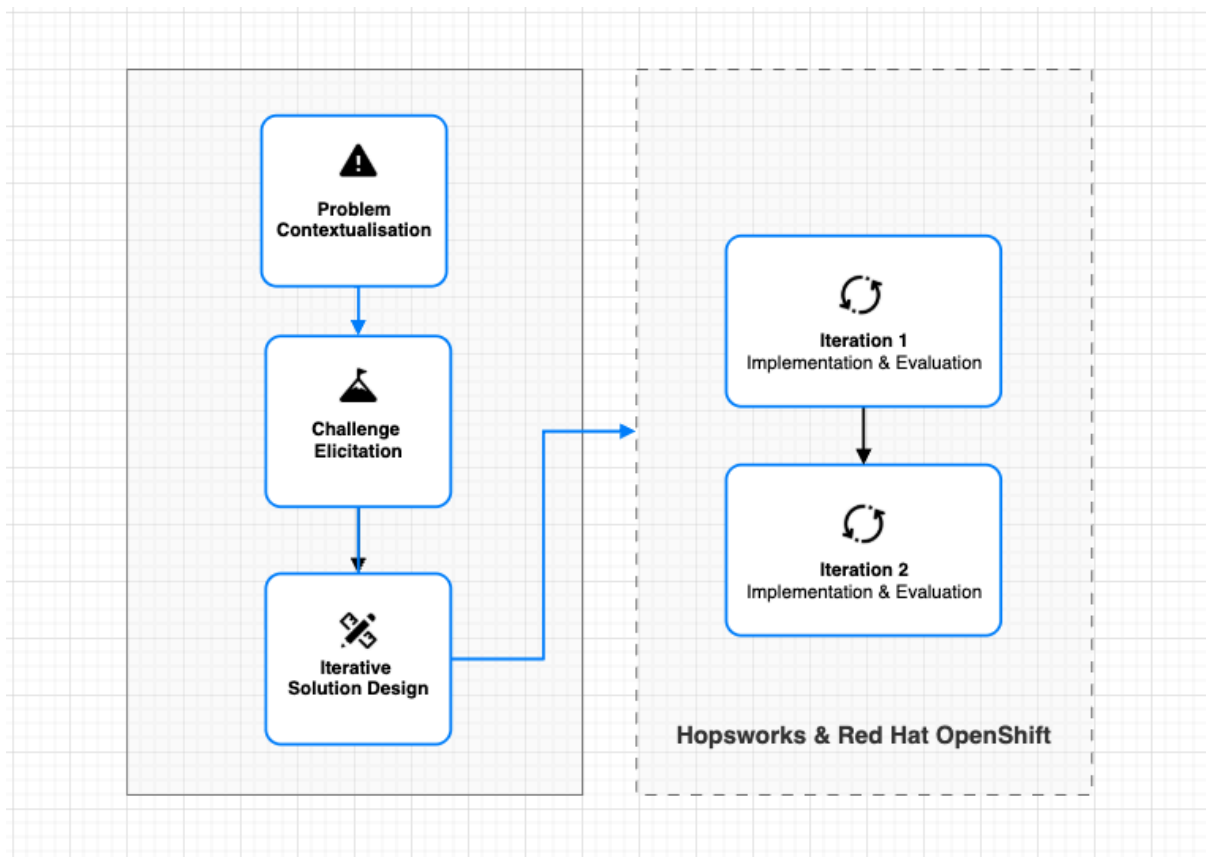


Figure 1. Collaborative and iterative method: industry, academia, and vendors refining and validating the Thousand-Model Challenge.

To investigate how automation can address the Thousand-Models Challenge, this study adopted a collaborative and iterative approach grounded in real-world industrial needs. The participant backgrounds included MLOps members from national and international companies (Volvo), AI researchers (Linköping University), as well as ML framework providers (Hopsworks, Red Hat). As shown in Figure 1, the work began with Volvo presenting the industrial context and outlining scaling challenges of machine learning development, deployment, and operations without increasing team size from the perspective of being a large international company with substantial experience in traditional software development. This initial step provided a shared understanding of the constraints and

objectives that guided the subsequent activities. To deepen this understanding, a collaborative workshop was held to identify and categorize challenges across the ML lifecycle, including data processing, model training, deployment, and operational monitoring⁶. These discussions helped prioritize the most critical bottlenecks and informed the design of scalable solutions.

Following the workshop, weekly meetings over about six months were conducted to refine problem statements and explore potential solutions. These sessions created an iterative feedback loop between stakeholders, ensuring that proposed approaches were both technically feasible and aligned with organizational goals. The discussions focused on automation, orchestration, and monitoring strategies that could reduce manual intervention while maintaining or improving system reliability.

Implementations of the discussed solutions were carried out in parallel with the weekly meetings. The implementations were performed in two iterations across two different platforms—Hopsworks and Red Hat OpenShift—to ensure platform independence and generalizability. The first iteration targeted scalability in data processing, model training, and deployment pipelines, leveraging platform-native capabilities for automation and resource management. The second iteration introduced monitoring mechanisms for model staging, including performance tracking and decision support for promoting models to production. Throughout both iterations, scalability was evaluated through discussions about computational efficiency, pipeline automation, and operational overhead, with the explicit constraint of maintaining the existing team size.

Automating Machine Learning Operations

Before an ML system can scale we need to make sure that we build from a solid foundation. To describe where we want to start, let's use an analogy: *an assembly line*. An assembly line enables a high degree of quality for the creation of an output in a structured and repeatable way. This is what we want for our models and is our assumption of the starting point of our readers; we have a process for creating a machine learning model. We want to make this process efficient and capable for thousands of models. If an assembly line can create a model with high quality, to scale to thousands of models, we need a way to scale the creation of assembly lines. Think of it as an assembly line for assembly lines. Our assembly lines must live in the context of the ML system operations depicted in Figure 2.

⁶ Protschky, Dominik, et al. "What Gets Measured Gets Improved: Monitoring Machine Learning Applications in Their Production Environments." *IEEE Access* (2025).

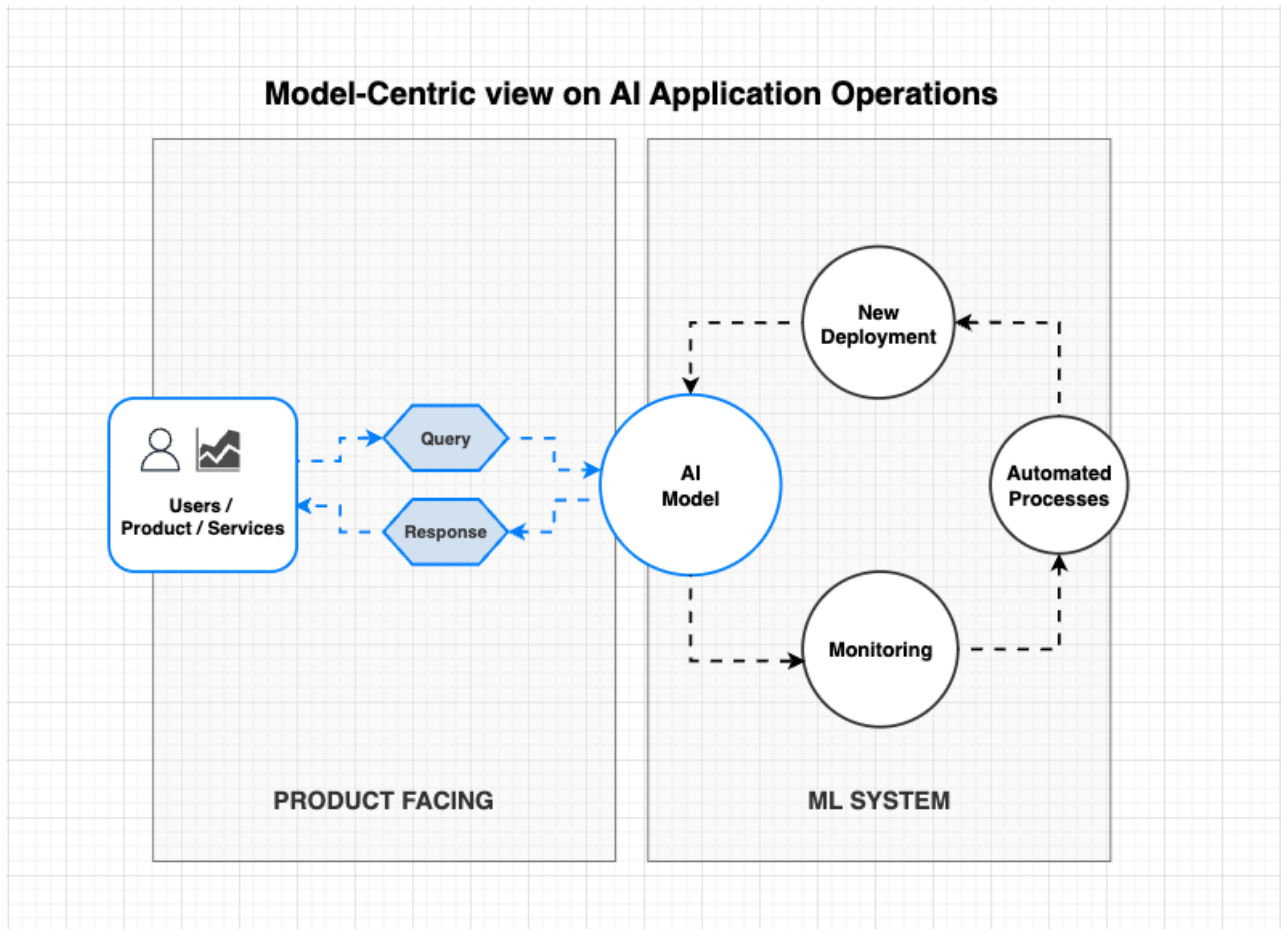


Figure 2. ML system context: training, deployment, inference, and feedback into products and services.

The goal of an ML system is to provide an answer when a stakeholder, user, service, or product sends a query. This makes modern machine learning systems dynamic; while data continuously flows in and a model is trained and deployed. Predictions are then being queried from the ML system. Therefore, we have two main requirements:

- **The system needs to maintain production quality;** the strategies are meant to keep the system functioning at an optimal state to continuously provide the prediction.
- **The system needs a positive feedback loop:** improving the output (for example model accuracy) towards a desired goal over time.

From the user/product perspective, we need to make sure **there is a technical output (prediction / answer) and a business outcome (of operational / strategical value) consistently being delivered**, and be sure that the quality can improve or at least not decrease over time. Without those strategies to maintain the system and increase its efficiency, the end product would not be able to exist. Those two patterns are what defines MLOps for ML Systems.

These are also characteristics of DevOps, but (similarly to DevOps) MLOps is not itself a system. MLOps is the collection of (ideally) **automated strategies and practices** to make the system work towards the intended goals.

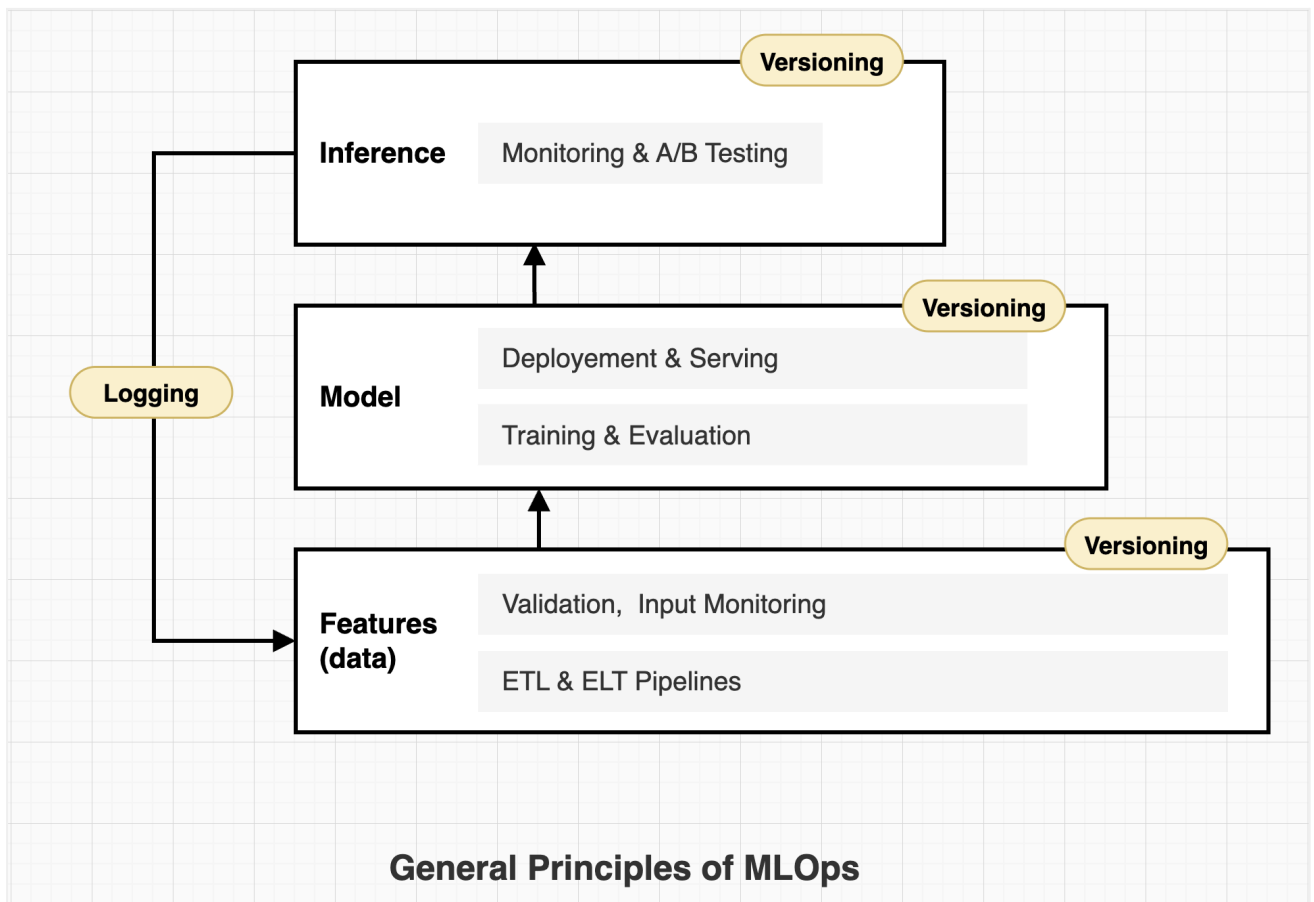


Figure 3. Core MLOps principles: versioning, automation, and observability across data → model → inference flow.

Foundational Concepts

The following elements are standard data engineering practices that underpin ML systems but fall outside the AI-specific scope of this paper:

ETL & ELT Pipelines: Extract, Transform, Load (or Load then Transform) are standard data engineering patterns for moving and preparing data between systems. AI/ML systems typically consume the same data sources as business analytics, inheriting these pipelines and their quality characteristics.

Validation & Input Monitoring: The process of checking data quality before it enters ML systems—verifying schema conformance, detecting anomalies, and ensuring values fall within expected ranges. Tools like Great Expectations automate these checks as part of the ingestion pipeline.

Assumption: The organization is familiar with standard MLOps practices (Figure 3). This includes data readiness, model training, deployment and inference, and using ML models to create real business value in the organization⁷.

When the lifecycle of a single model requires significant hands-on effort from data scientists, ML engineers, and IT operations, scaling beyond a few models becomes a problem of cost and complexity. The aim is to shift from primarily manual human involvement to a state where technology and automation manage the complexity.

MLOps: Organizational Processes

Organizational considerations in MLOps involve the management of the assets (data and models) and infrastructure (both hardware infrastructure such as GPUs and virtual machines but also software infrastructure) across different teams (DevOps, MLOps, Data engineering, Data Science) and organisational units.

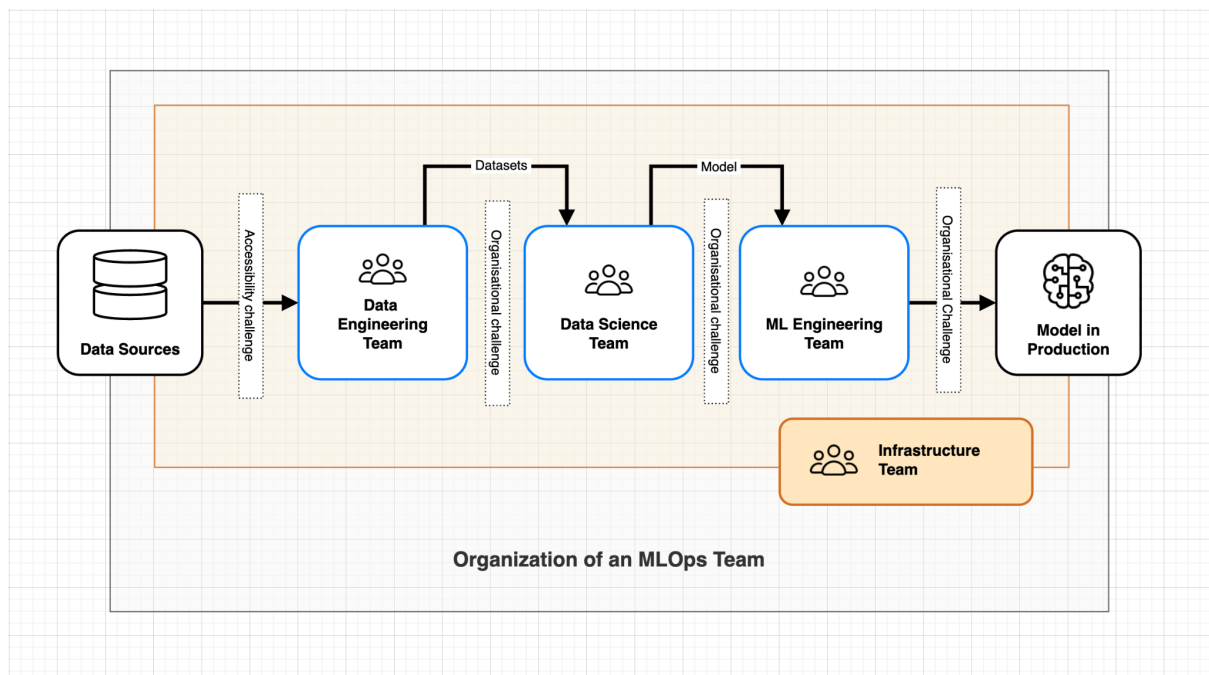


Figure 4; Organizational bottlenecks across Data, ML, and Infra teams in enterprise MLOps.

In standard organisations, the nature of machine learning models in production surfaces additional hurdles related to the structure of the organisations that go beyond the technical challenges (scale, monitoring, deployment, model management).

Team-Specific Issues:

1. Data Engineering:

- Data sources accessibility
- Unclear ownership of shared pipelines

⁷ Svärth Jönsson et al. *AI Application Operations - A Socio-Technical Framework for Data-driven Organizations*

- Feature changes for one model can break other models
- 2. Data Science:**
 - Data cataloguing and retrieval
 - Data duplication, source management
 - Training / Inference Skew
- 3. ML Engineering:**
 - Deployment consistency
 - Upstream data changes
 - Model versioning and lifecycle management
- 4. Infrastructure:**
 - Management of environments such as dev/staging/prod
 - Resource allocation, prioritisation

Cross-Team Hurdles:

1. **Handoffs:** Lost information between teams, accountability when things fail
2. **Standardization:** Heterogeneous tools, frameworks, data management, and processes

At Scale (1000 models):

- Every model touches all teams = bottleneck
- Coordination overhead grows exponentially
- Technical debt compounds

These organisational hurdles and challenges can be overcome, or at least be reduced, through clear framework standardisation, centralised storage and MLOps practices.

In an ideal scenario, all teams would be operating within the same infrastructure and frameworks, allowing them to build, deploy & operate AI systems at any scale.

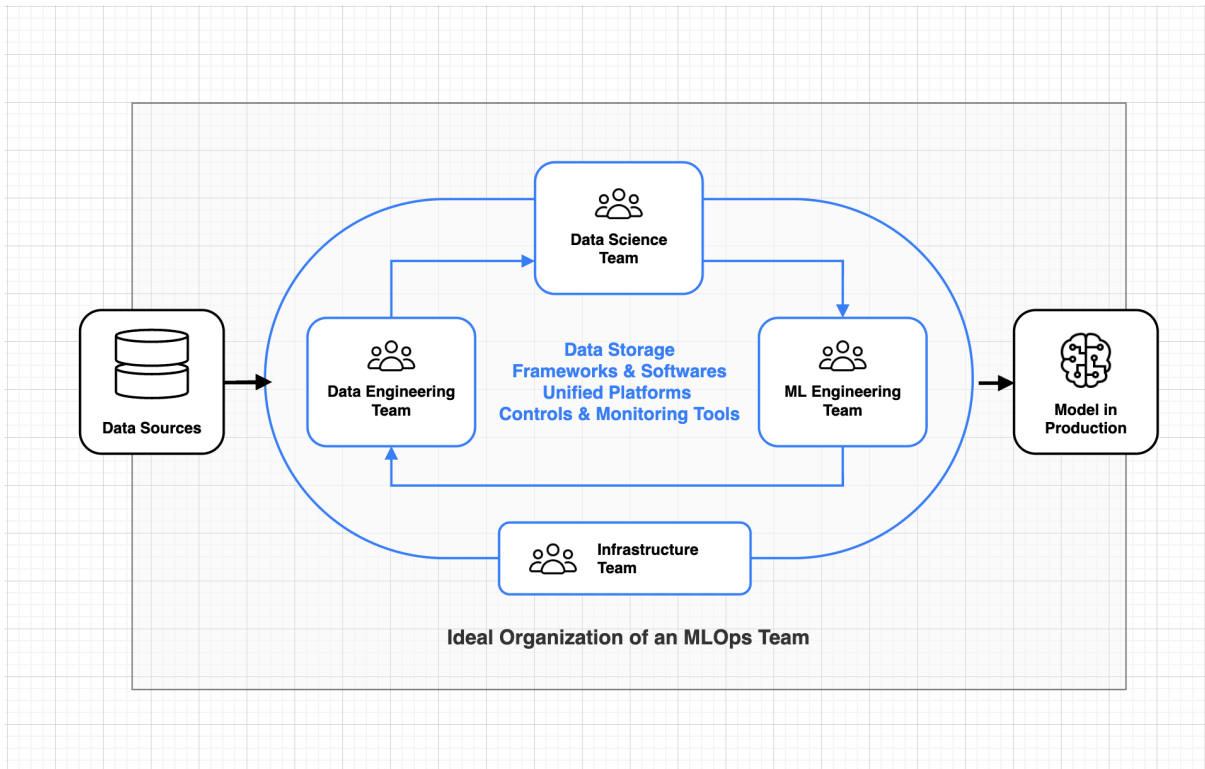


Figure 5; Unified MLOps ecosystem: shared infrastructure, standards, and repositories reduce handoffs. In an ideal paradigm all teams work within the same MLOps ecosystem to avoid hurdles.

MLOps: Value-Driven Development

In an ideal scenario, practitioners should strive to first develop an MVPS (minimal viable prediction service / Minimum Viable Product); to deliver a proof of value for the organisation or business. This, in turn, allows us to iterate over the quality of each pipeline component and improve the qualitative aspects of the project. Typically, such processes would follow the steps of:

1. Identify the Prediction Problem,
 - a. identify ROI and measurable business KPIs tied to ML **Proxy metrics**
2. Identify the Data Sources,
3. Write the Feature Pipeline(s) (and orchestrate them),
4. Write a Training Pipeline,
5. Write an Inference Pipeline (and orchestrate or serve)

Potentially, use a UI to showcase the value of your model to stakeholders

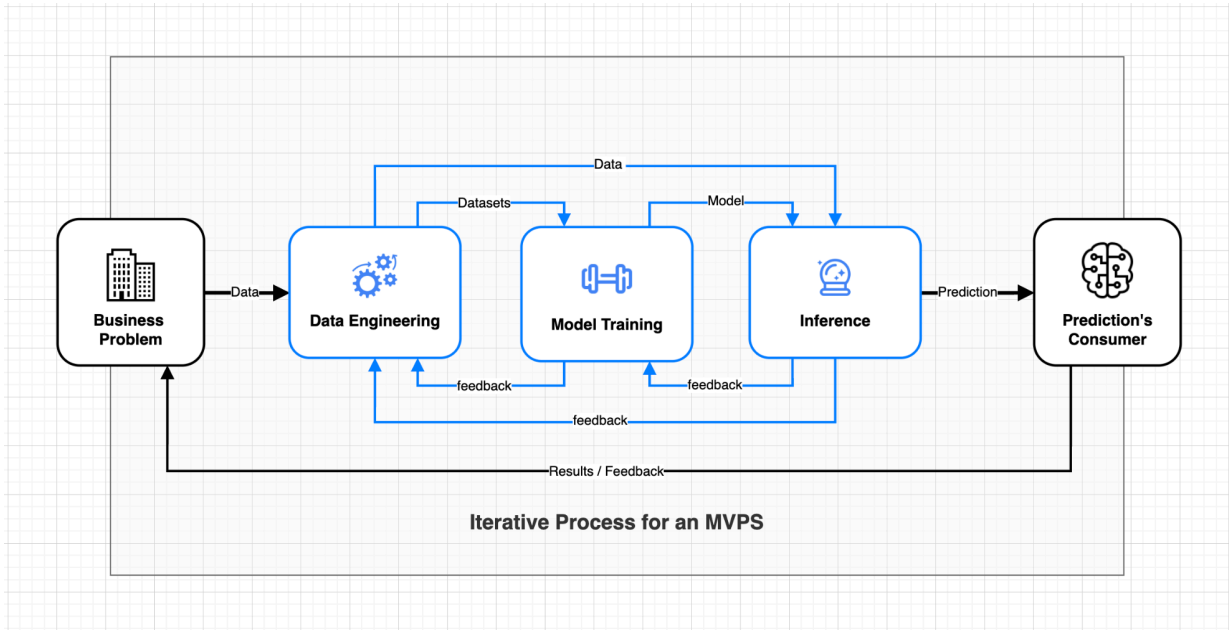


Figure 6. MVPS lifecycle: from prediction problem to business-validated deployment. For a minimum viable prediction service/system.

MLOps: Model Lifecycle

Due to the high reliance of machine learning models on data, traditional DevOps principles used for application development are, on their own, not sufficient. Overfitting, data drift, or model degradation make tested and validated models decline in performance and require consistent monitoring, retraining, or even completely rebuilding models. Therefore, input data and model monitoring are first-class elements of the lifecycle shown in Figure 7.

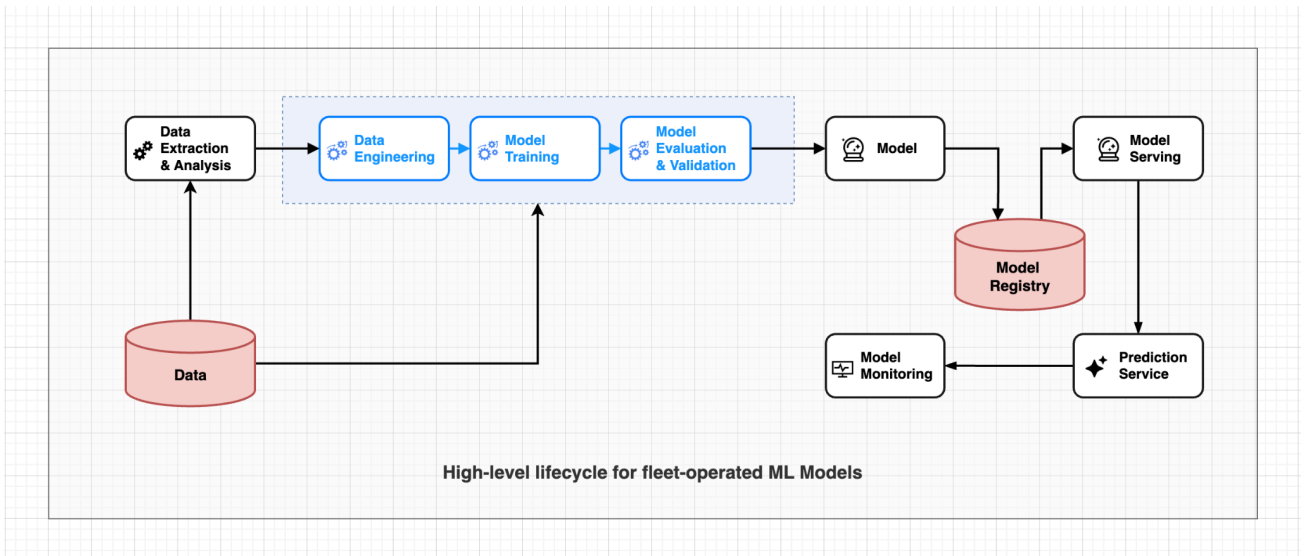


Figure 7: High-level lifecycle for fleet-operated ML models: from specification and data binding to training, evaluation, registration, staging, promotion, monitoring, and closed-loop adaptation. Covering the full multi-stage process of spec → data binding → training → evaluation → registry → staging → promotion → monitoring → adaptation.

Maintenance and model releases for the model registry can be handled manually by humans for a few models. However, as soon as the number of models grows, it is necessary to introduce automations to not grow the number of people involved at the same rate. In practice, this means treating each model as a productized, versioned asset and executing its lifecycle through policy-driven automation rather than bespoke engineering effort.

Goal. Make every model a first-class, versioned, and automatable asset whose lifecycle can be executed and audited while minimizing work per model.

Here is an example of a fleet-native lifecycle:

1. **Define the model spec.** Record owner, purpose, success criteria (SLOs/KPIs)⁸, constraints, and governance/risk class. This becomes the contract that downstream automation enforces.
2. **Bind the data.** Use **versioned feature views** (schema, transformations, and time-travel guarantees) to eliminate training/serving skew and make data lineage explicit.
3. **Train via reusable, parameterized pipelines.** Pin pipeline versions and pass only parameters (e.g., hyperparameters, feature view IDs), so the same pipeline can train hundreds of models reliably.
4. **Evaluate against business-aligned gates.** Go beyond accuracy: include latency, cost, robustness, and—where applicable—fairness and safety thresholds. Failing any gate blocks promotion.
5. **Register with full lineage.** Store the model artifact together with code commit, data snapshot/hash, environment/container image, and pipeline version to guarantee reproducibility.
6. **Stage safely.** Deploy using **shadow** or **canary** policies and environment-specific configs; collect telemetry before broad rollout.
7. **Promote to production.** Advance versions through pull requests and policy checks; keep a human-in-the-loop approval step where the risk class demands it.
8. **Monitor end-to-end.** Observe **data, model, infrastructure, pipelines, and business value**; tie alerts to runbooks so responders know the next action.
9. **Adapt automatically.** Use event-driven triggers (e.g., drift or SLO breaches) to retrain, restage, and, when safe, repromote—or to **rollback** to the last-known-good.

⁸ See the Glossary in *AI Application Operations - A Socio-Technical Framework for Data-driven Organizations*

10. **Audit and learn.** Capture deployment decisions, incidents, and post-mortems; feed those insights back into gates, policies, and pipeline templates.

By expressing the lifecycle as code (pipelines, policies, and manifests), and by versioning every artifact (data, features, models, environments), we shift from manual release work to **industrialized, repeatable operations**. The result is that adding the N+1th model reuses the same automated steps as the first one, keeping the **human-per-model** ratio low while improving traceability, resilience, and time-to-value.

MLOps: Continuous Performance

Once a model has been deployed, the challenge shifts from “*does it work?*” to “*does it keep working, at the required level of performance, under real production conditions?*”

While traditional MLOps often stop at deployment, **scaling to hundreds or thousands of models requires performance management to become an highly automated, first-class capability**—just like versioning, deployment, and retraining.

In practice, this means defining **service-level objectives (SLOs)** not only for the technical service (latency, availability, throughput, cost), but also for the model’s predictive behaviour (accuracy drift, calibration, fairness across cohorts, business impact). A model that is “healthy” in a lab environment but violates latency budgets, GPU quotas, or customer-facing expectations is not a usable model. Performance therefore spans both **system performance** and **prediction performance**, and both must be observable and enforceable at scale.

To support this, modern ML systems increasingly borrow patterns from cloud-native software operations:

- **Traffic control policies** such as canary deployments, shadow deployments, blue/green cutovers, and dynamic routing enable safe rollouts and controlled exposure of new models.
- **Autoscaling**—including scale-to-zero for rarely used or long-tail models—ensures that resource consumption grows sub-linearly with the number of models.
- **Performance budgets** (for compute cost, latency, CO₂ footprint, etc.) can be encoded directly in deployment manifests so that violations trigger automatic mitigation or rollback.
- **Fallback strategies** such as routing to a previous version, using cached predictions, or degrading gracefully to rule-based logic help maintain service continuity when a model misbehaves.

Continuous performance, in other words, is not a one-time validation step but an **operational loop**: observe → compare against SLOs → adapt. When combined with automated monitoring and retraining, this loop allows the platform—not individual engineers—to take first-line action in response to degradation.

As the number of deployed models increases, **manual supervision becomes infeasible**. A solution to this is to make performance management *declarative* (defined as policy and thresholds), *observable* (telemetry emitted consistently across all models), and *actionable*

(linked to automated responses such as retraining, reprovisioning, or rollback). This is what turns a collection of deployed models into a **model fleet that can be operated reliably at scale**.

MLOps for Continuous Operation

This section focuses on platform and service reliability—the infrastructure and runtime capabilities that keep prediction services healthy—complementing the previous section’s focus on model performance against SLOs. Those capabilities are what empower the product(s) with machine learning capabilities. Data is needed to train/re-train a model. Models need to be trained and improved, and the prediction needs to be produced (inferred) for the service to be able to use those predictions; to provide better recommendations in a recommender system, better text output in the case of a LLM system, more accurate fraud detection, etc..

There are 3 core principles that contribute to a healthy ML system;

- **Automated Versioning** of assets and artifacts. This allows the engineers and data scientists to easily implement new models, find assets, and **rollback** if errors are made during new deployments. Effective versioning of the data itself also allows time-travel, the ability to retrieve the data as it was at a specific point in time; useful for both debugging and auditing.
- **Automated Testing** of models and data. More in line with the classic principles of DevOps, here we consider the testing of individual engineering functions; does this input always provide the appropriate output, can the system handle large amounts of requests, etc..
- **Automated Monitoring** of systems and data. Monitoring in machine learning systems can be divided into two main categories. The monitoring of the system and the monitoring of its output; models and features (data). The continuous monitoring of the input and output data allows for preventing decay of the model. As the data keeps changing and models are trained on a snapshot at a specific time, the new data may cause the model to lose its effectiveness. For example, foundation models (such as GPT) require periodic retraining or augmentation (e.g., RAG or fine-tuning) to mitigate staleness as real-world data evolves.

Core Infrastructure (what);

- **Source & Version Control:** A centralized repository (e.g., Git) for all code, data (using tools like DVC), and model configurations. This ensures reproducibility and collaboration.
- **Feature Store:** A centralized repository to store, retrieve, and manage ML features for both training and serving. It prevents training-serving skew and encourages feature reuse.
- **Experiment Tracking:** A system for logging and comparing all aspects of training runs, including parameters, metrics, and model artifacts.
- **Model Registry:** A versioned repository for storing, managing, and governing trained models, tracking their lineage from data to deployment.

- **Model Serving & Deployment:** Infrastructure to deploy models as scalable and reliable prediction services (e.g., REST APIs), often supporting strategies like A/B testing or canary deployments.
- **Workflow Orchestration:** An engine that automates, schedules, and manages the end-to-end ML workflow, from data ingestion to model monitoring.

As might be expected, integrating all of these components together into a complete system is hard enough to depict in a diagram and can be a tough challenge in practice.

Continuous Monitoring for MLOps

As the number of deployed models grows, monitoring must evolve from isolated dashboards and manual checks into a **standardized, automated, and fleet-wide capability**. The purpose of monitoring is not just to “observe what is happening,” but to provide the signals that allow a model fleet to remain reliable, performant, and economically valuable over time. In the context of operating hundreds or thousands of models, the question is no longer *whether* to monitor, but *how much of the monitoring pipeline can be automated*—and where human decision-making remains essential.

What to Monitor

A scalable monitoring strategy spans multiple dimensions:

<u>Dimension</u>	<u>Examples of Signals</u>
Data	Drift (covariate, feature, label), schema changes, quality issues (nulls, outliers, duplicates), recency/latency of updates, emerging bias patterns
Model	Performance deltas vs. baseline, calibration error, uncertainty distributions, cohort-level degradation, concept drift
Infrastructure	Latency, throughput, saturation, error rates, autoscaling behaviour, GPU/CPU cost burn, carbon/power budgets
Pipelines	Schedule adherence, failed tasks, reproducibility violations, artifact corruption, dependency mismatch
Governance	Who deployed what, when, and where; compliance with deployment policies; existence of documentation and approvals
Business Value	Leading indicators (conversion, SLA hits, saved cost), lagging KPIs, uplift vs. control group, feature-level contribution signals

This scope ensures that monitoring is not reduced to *just* model accuracy, but instead reflects the full technical and business lifecycle of the model.

Why Monitor

Monitoring serves multiple operational intents:

- **Quality assurance** – detect performance degradation before it affects customers
- **Responsiveness** – ensure inference services meet latency and availability targets
- **Robustness** – detect failures and automatically fall back to safe baselines
- **Proactive alerts** – identify early indicators such as data drift, schema changes, or cost spikes
- **Incident response** – enable rapid triage with traceable model + data lineage
- **Human validation where needed** – especially when business value or fairness cannot be reduced to a single numerical metric
- **FinOps for MLOps** - for per-model cost attribution, GPU optimization, or automated budget enforcement.
- **Legal compliance** - legal frameworks such as GDPR and the AI act require continuous control with evolving data and clear lineage.

The key shift at scale is that **monitoring moves from being a passive activity to an active driver of lifecycle decisions**—including retraining, rollback, canarying, or decommissioning.

How to Monitor: Levels of Automation

Monitoring maturity can be expressed as four operational tiers:

<u>Level</u>	<u>Description</u>
L0 – Manual	Dashboards and periodic reviews; no automatic alerts
L1 – Assisted	Alerts exist and runbooks document how to react, but intervention is manual
L2 – Automated Guardrails	Alerts trigger predefined actions (e.g., scale-down, route to previous model, disable feature)
L3 – Closed Loop	Events (e.g., drift, SLA breach) trigger retraining, validation, and staging of a new model; promotion still subject to policy approval

Many organizations today operate at L0/L1. Operating a fleet of 500+ models reliably requires moving toward L2/L3—where the monitoring system not only *detects* problems, but *initiates corrective action*.

Example from this white paper: in the Red Hat implementation, TrustyAI detects drift, fires a webhook, which in turn triggers a Tekton retraining pipeline. This exemplifies L3 closed-loop monitoring, with final promotion gated by policy-as-code.

Instrumentation Patterns

To make fleet-scale monitoring possible:

- Standardize metric, log, and trace schemas across teams and models

- Emit model-aware telemetry (predictions, confidence, feature stats) while respecting privacy & minimization
- Tag every prediction with **model version + data snapshot ID + feature view version**, enabling full auditability
- Store all monitoring output in a queryable observability backend (e.g., Prometheus, OpenTelemetry, feature-level timeseries store)

Business-Value Telemetry

Technical success does not guarantee business success. At scale, every production model should also emit **business-aligned signals**, such as:

- Conversion uplift, reduced scrap, improved forecast accuracy
- Energy use, cost per inference, impact on SLA violations
- Counterfactual or A/B slices to avoid false attribution (“model looked good, business impact did not move”)

This allows automated systems to detect not just *model drift*, but *value drift*.

Human-In-The-Loop Monitoring

Automation does not eliminate the need for human judgment. The scalable pattern is:

- Schedule periodic review cycles based on model **risk class** (e.g., safety-critical vs. low-impact)
- Auto-generate monitoring reports summarizing drift, incidents, fairness, cost, business impact
- Require a documented decision: *keep / retrain / canary / rollback / retire*
- Store decisions in the same audit trail as model versions and deployment actions

This preserves governance while avoiding manual day-to-day babysitting.

With monitoring in place as a continuous, automated source of truth, the next step is to connect those signals to **lifecycle actions**—so that models do not merely report when they are failing, but can actively evolve, retrain, or roll back without requiring manual intervention. In technical terms, this is the shift from monitoring as a passive dashboard to monitoring as an event stream that can trigger automated responses such as retraining, redeployment, or routing traffic to safer versions. Operationally, this marks the transition from human-driven oversight to **closed-loop execution**, where the platform can detect drift, launch a retraining job, validate the resulting model, and stage it for release—while still allowing policy-gated approval for high-risk use cases. And from a business perspective, the value of monitoring is only realized when it supports **continuous improvement**, meaning that the system not only protects against degradation, but also enables measurable uplift over time, based on business-aligned KPIs rather than just technical metrics. In other words: monitoring tells us what is happening; continuous improvement ensures the fleet adapts and keeps delivering value at scale.

For more on Monitoring, see *AI Application Operations - A Socio-Technical Framework for Data-driven Organizations*.

MLOps for Continuous Improvement

To prevent the systems from decaying over time, it is necessary to introduce feedback loops into the MLOps process. These feedback loops can operate in real time or batch, triggered by new data, scheduled intervals, or events such as drift or SLO breaches. Continuous reinforcement strategies for MLOps can be divided into three main categories:

- **Automated Re-training of the model** to prevent the model from losing performance due to new characteristics in the data. Mechanics need to be put in place to trigger regular retraining, e.g., by monitoring data distributions. Note that this is not accounting for training of the models that are due to improvement of the model architecture themselves.
- **Automated Logging and Feedback** to adapt the system based on actual usage and previous predictions.
- **A/B Testing of Models for Redeployment.** Quite simply the process of comparing a model against another. This may be done after retraining, or it can be a part of the training itself. A/B testing ensures that the best model, given a specific goal, is always in production. Tests can include traffic-split experiments (A/B, multi-armed bandit), offline backtesting against holdouts, and cohort-specific evaluations.

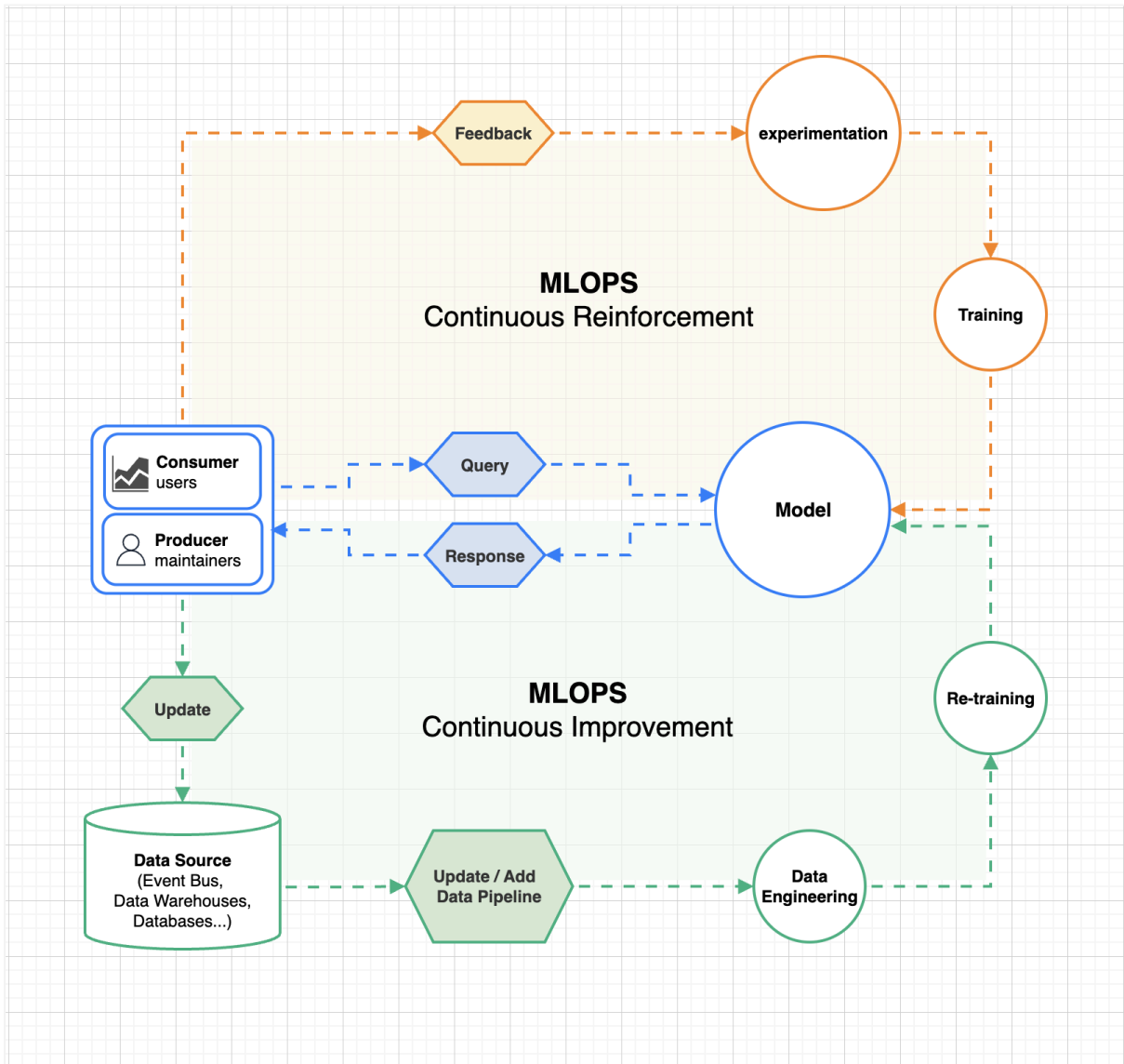


Figure 8. Operation and Reinforcement in MLOps: Continuous improvement loop linking feedback, retraining, and A/B testing

A critical enabler for continuous improvement at scale is to treat **model promotion as policy-as-code**. Instead of relying on ad-hoc judgment or manual experimentation, the decision to replace a production model is expressed as a set of explicit, machine-checkable conditions: minimum performance thresholds, statistical confidence levels, cohort-level fairness and robustness checks, latency or cost ceilings, and any domain-specific guardrails. When these conditions are satisfied, the platform can automatically stage and validate a challenger model. The **degree of automation depends on the assigned risk class**: low-impact models may be promoted fully automatically, while high-risk or business-critical models require a final human approval step before traffic is rerouted. In this way, improvement becomes continuous, controlled, and scalable—without losing the accountability and oversight demanded by regulated or safety-critical environments.

What Must Be True Before Scaling to 1000 Models

The earlier sections established **why** organizations struggle to scale beyond a small number of machine-learning models. Before we turn to concrete solution architectures, this section clarifies **what capabilities an organization must have in place in order to scale model count without scaling headcount, risk, or operational fragility**. These are *not* tools, platforms, or vendors—they are **non-negotiable requirements** for fleet-level AI operations.

Why the Needs Exist: From 1 Model → 50 → 1000

With 1–5 models, almost any process works:

- Retraining can be scheduled manually
- Owners can be tracked in a spreadsheet
- A data scientist can push a model to production
- Monitoring dashboards can be reviewed weekly
- Documentation, approvals, and troubleshooting are social, not systemic

At 50 models, these processes begin to strain capacity. At 500–1000 models, they **fail mathematically**: every manual task becomes a scaling multiplier on cost, delay, and operational risk. If one model requires 10–20 hours/month of human attention, **1000 models require 30–50 people just for maintenance**—unless the lifecycle becomes automated, observable, and policy-driven.

Core Needs for Fleet-Level Model Operations

<u>Need</u>	<u>What Works at Small Scale</u>	<u>Why It Breaks at Large Scale</u>
Ownership & accountability	One person “just knows” who owns a model	Ownership is forgotten, unclear, or lost across org boundaries
Reproducible pipelines	Copy-paste scripts, Jupyter notebooks	1000 versions = drift, skew, untraceable failures
Automated promotion & rollback	Manual PR + Slack approval	Dozens of releases per week → bottleneck + risk exposure
Unified lineage (data → model → deployment)	Folder structure, wiki pages	Impossible to trace what data/model/env is in prod
Standardized monitoring & alerting	Single dashboard per team	1000 dashboards = no one sees critical failures

Event-driven retraining	Retrain “when we remember”	1000 models degrade on different timelines → chaos
Policy-based governance	“We check with legal when needed”	Compliance cannot scale through meetings and documents
Decoupling models from data plumbing	Each model has its own data prep	1000 duplicated pipelines → brittle and expensive
Robust and holistic security controls	“We react on security issues when there’s a CVE”	At the 1000 model scale the security posture is massive

These needs form the **evaluation lens** for the solution architectures that follow. Each of the three solution perspectives (Red Hat, Hopsworks, Volvo) addresses these same needs—but with **different architectural principles and implementation paths**.

Key takeaway: The need is not “better MLOps,” but **MLOps that does not collapse under scale**. Scaling AI is not about running more models; it is about running more models **without multiplying humans, meetings, dashboards, and emergencies**.

Architectural Patterns for Scaling Model Fleets

The needs defined in the previous section describe what an organization must be able to do before it can operate machine-learning models at fleet scale. This section introduces two architectural patterns that have emerged as effective ways to meet those needs in practice. They do not prescribe specific tools or platforms; instead, they describe the *structural shifts* that differentiate a “handful of models” approach from a “hundreds or thousands of models” approach.

Both patterns address the same core scaling pressures—reproducibility, lifecycle automation, monitoring, drift response, traceability, and governance—but they do so from different architectural centers of gravity:

<u>Pattern</u>	<u>Primary unit of reuse</u>	<u>When it is most valuable</u>
Pipeline-centric scaling	Reusable, parameterized lifecycle pipelines	When many models follow similar lifecycle shapes and differ mainly in configuration
Data-centric scaling	Reusable, versioned feature definitions and data contracts	When models consume shared data assets and the main scaling challenge is data evolution

These patterns are not mutually exclusive; many enterprise-grade MLOps platforms combine elements of both. The following subsections outline the core ideas behind each pattern and how they enable model fleet operations.

Pipeline-centric scaling

In this pattern, the model lifecycle is treated like an **industrial assembly line**: once the process is defined, many models can be produced by changing the inputs (**parameters**) rather than rebuilding the factory each time. But scaling to hundreds or thousands of models does not just require **one assembly line** — it requires an **assembly line for creating assembly lines**, where pipeline templates themselves are versioned, parameterized, and deployed repeatedly. This is what turns MLOps from a craft process into a manufacturing process. Instead of building one pipeline per model, the organization develops a **small number of composable, reusable pipeline templates** that can be instantiated repeatedly with different configuration values (model type, dataset location, hyperparameters, deployment target, etc.).

This approach mirrors industrial manufacturing: once an assembly line exists, many units can be produced by changing only the inputs, not the factory itself.

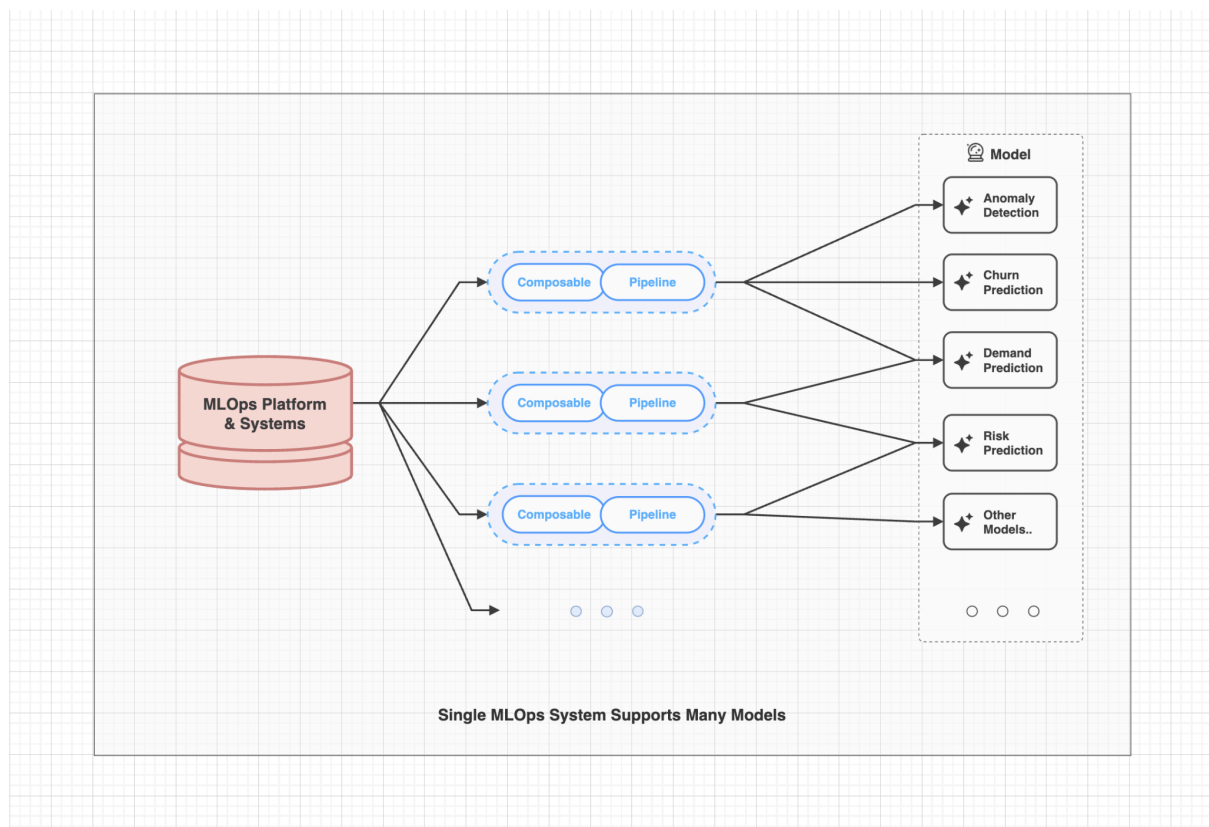


Figure 9. illustrates this idea: a single MLOps system supports many models, not by cloning pipelines manually, but by creating multiple instances of the same pipeline definition with different parameters.

Why this pattern scales

- **Reduces per-model engineering cost:**
new model → new configuration, not new code

- **Enables consistent governance and logging:**
every model follows the same lifecycle stages
- **Standardizes** promotion, rollback, retraining, and monitoring logic
- **Allows new models to be added with low marginal effort**
(a critical requirement once model count passes ~50)

When this pattern is most useful

- When the organization has many similar model types (e.g., per-business-unit churn models, per-market forecasting models)
- When repeatability and compliance require the *same lifecycle gates* across all models
- When the bottleneck is *engineers writing and maintaining pipelines*, not *data diversity*

Pipeline-centric scaling solves a major scaling problem: without it, every model becomes a snowflake, and human cost grows linearly with model count.

Data-Centric Scaling: “Shared, Versioned Features as a First-Class Asset”

Where the previous pattern treats pipelines as the reusable unit, this pattern treats **data definitions and feature transformations** as the primary source of reuse. Instead of each model building (and rebuilding) its own data preparation logic, the organization defines **maintained, versioned, time-aware feature assets** that many models can consume.

In this view, models are not *pipelines with data attached*—they are *parameterized consumers of shared data products*.

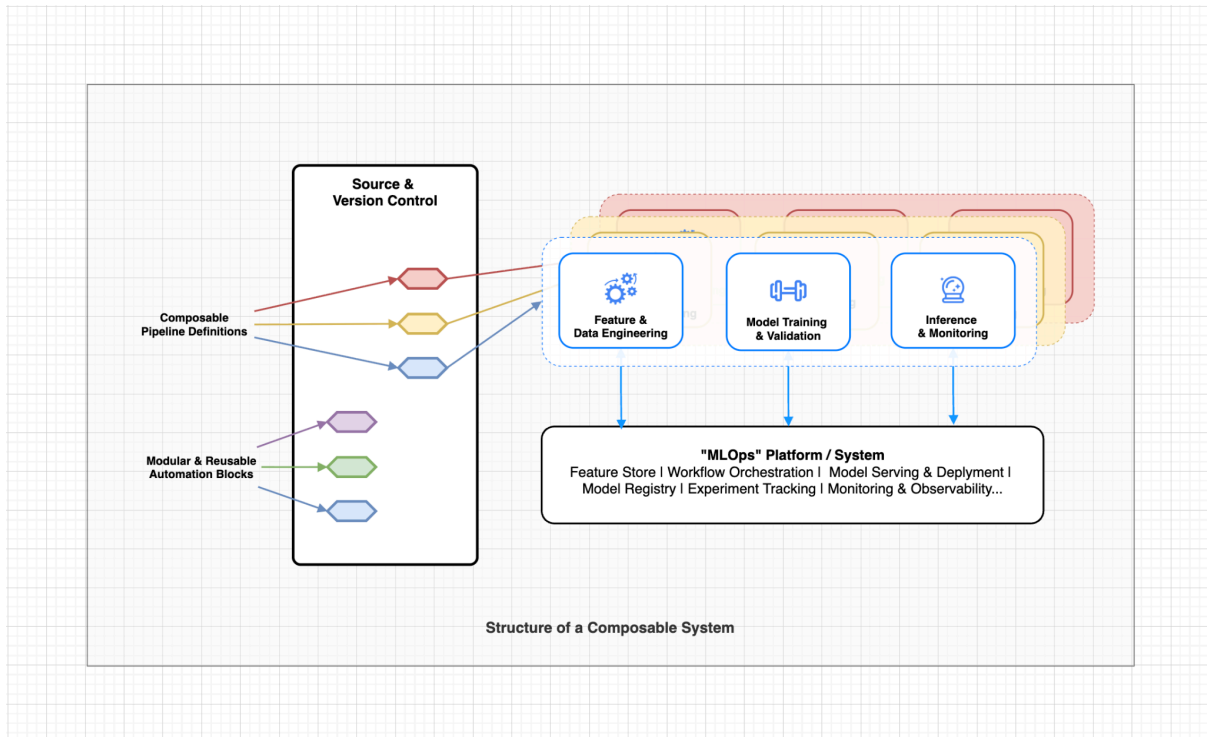


Figure 10. The structure of such a system: data engineering, training, and inference pipelines are connected through reusable data and metadata assets, governed by a central platform.

Why this pattern scales

- **Eliminates duplication** – no more “10 teams each implementing `customer_age_bucket()` differently”
- **Makes training–serving parity enforceable** – features are versioned and queryable, not re-computed ad-hoc
- **Reduces model decay rate** – data evolution can be managed centrally, rather than per model
- **Supports large model fleets that share a common domain** (e.g., fraud, pricing, IoT telemetry, risk scoring)

When this pattern is most useful

- When models rely heavily on shared enterprise data sources
- When the rate of **data change** is higher than the rate of **model architecture change**
- When data availability, quality, and lineage are the biggest blockers to deployment at scale

Data-centric scaling addresses the other major scaling law: models degrade because data changes. At scale, the costliest failure mode is not pipeline breakage but **silent accuracy loss due to drifting data semantics**.

Relationship Between the Two Patterns

These patterns solve different scaling bottlenecks:

<u>If your bottleneck is...</u>	<u>The pattern that helps most is...</u>
Too much custom code per model	Pipeline-centric scaling
Too many inconsistent data transformations	Data-centric scaling
High cost of onboarding new models	Pipeline-centric
High cost of maintaining existing models	Data-centric
Compliance requiring full lineage from data → model → prediction	Both, but data-centric is foundational
Frequent regeneration of similar models across teams or regions	Pipeline-centric
Frequent regeneration of data across models	Data-centric

Most mature organizations end up combining both patterns:

- Pipelines handle lifecycle automation, promotion, rollback, retraining
- Feature/data assets provide reuse, time-travel, lineage, and cross-model consistency

The three solution perspectives in the next section illustrate different emphases:

<u>Solution Perspective</u>	<u>Emphasis</u>
Red Hat	Pipeline-centric model scaling (assembly-line automation)
Hopsworks	Data-centric model scaling (shared, versioned feature assets)
Volvo	Value-centric integration of model fleets into product ecosystems

Business Context and Enterprise Motivation: High level solution to scale value extraction from ML models by Volvo

In the business context a model has zero bottom line value until it is implemented and used in the operational business process. The high level solution focuses on modularity and

automating as much as possible from model initial experimentation to value extraction at inference in the business process.

This means the models need to be integrated into the operational digital products where customer value is created. However, the landscape of digital business products can be diverse, hence to be able to scale, the MLOps solution needs to be use case/application/model agnostic and have several deployment types. The solution includes three major deployment types, deploying the model artifacts, the model as code for online learning and model as API endpoint – to cater for variations in requirements from the business products.

To be able to scale dynamically the solution is set up in a modular way to facilitate if certain parts in the solution need to be scaled individually. Every part of the model life-cycle flow is automated to be able to scale in number of models without having to scale in number of people. Freeing up time for Data scientists to focus on lab work coming up with new challenger models or new use cases as well as letting ML Engineers focus on implementing the value extraction from the models together with product teams – will secure long term value extraction from a larger number of models.

Additionally the solution will facilitate securing value over time (as opposed to one time value) by adding proactive observability (refer to this or other whitepaper) and automations enabling the continuous re-training to keep or increase e.g. accuracy of models.

In terms of technology choices the best fit technology within the frame of company internal technology appointments are done for each module.

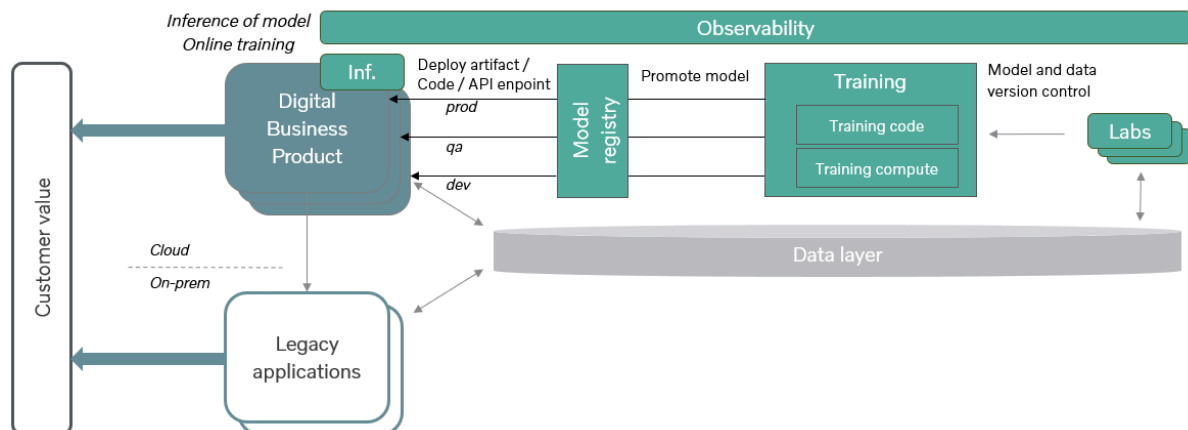


Figure 11. High level architecture for value extraction oriented MLOps showing the connections between the main components.

Here, the solution is built on Microsoft Azure. The entire model lifecycle is automated using **Azure DevOps CI/CD pipelines**. We package this as a reusable template that each product team can adopt to accelerate the path from model development to production.

Because everything should be automated, the first step is infrastructure provisioning. Using **Terraform** as the IaC, all ModelOps prerequisites are created in Azure, such as Storage

Accounts, Azure Machine Learning (Azure ML), Key Vault, and other resources, so that when it's time to work with models, everything is already in place.

Beyond infrastructure, the operating model for the lifecycle is equally important. We focus on **Azure ML** and **MLflow**. Azure ML's asset model helps ModelOps track and version not only code, but also datasets used for training and testing, environments and libraries, components, and pipelines. Treating these pieces as first-class assets increases governance and traceability across the lifecycle. With MLflow, packaging is straightforward, and synchronizing code and models (along with versioning and tagging) makes models traceable, which is crucial for reproducibility and debugging during QA.

We also use **Application Insights** to monitor model behavior and detect data drift, allowing us to trigger retraining when needed.

In our case, we needed to promote models from our lab environment into our ModelOps environment. To support this, we created a dedicated Azure DevOps pipeline that can be called from a Jupyter notebook. Because these actions modify model artifacts in storage, we designed approval gates to keep the process secure for both model updates and promotions.

In addition to Azure ML, we leverage Azure ML Registry as a unified, shared hub across environments. Since model transitions (e.g., promoting from dev → QA → prod) are part of the lifecycle, we store our MLflow models in the registry to standardize and streamline promotion workflows.

Overall, this approach, Azure-based infrastructure automated with Terraform, lifecycle governance via Azure ML + MLflow, observability with Application Insights, secure promotions through Azure DevOps, and a shared registry, gives us a consistent, reliable, and fully automated path from experimentation to production.

Automated Pipelines

Lab to Dev ModelOps

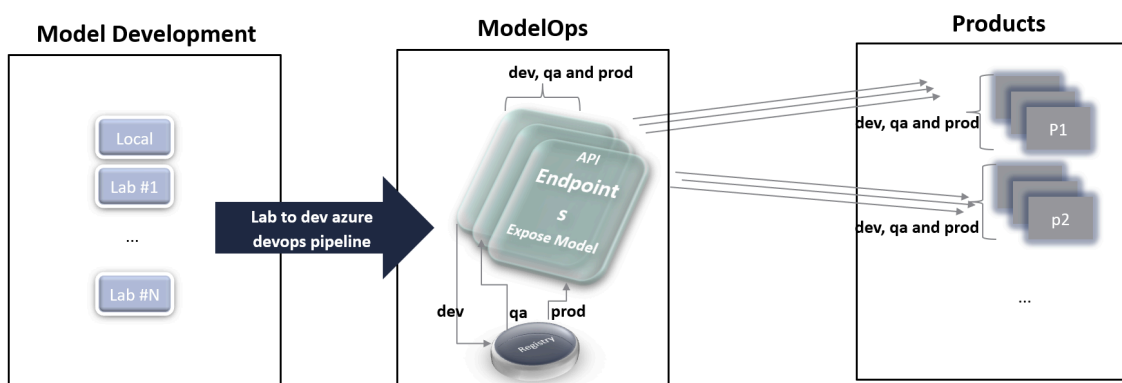


Figure 12. Automated pipelines showing lab-to-dev transitions, promotions across different environments, and model deployment.

As mentioned earlier, we define ModelOps as a reusable template. Below is a sample from one of our use cases. The template includes:

- **Data:** Pointers to data sources (or the data itself, e.g., CSV files).
- **Components** : Each with its own managed asset name and version, enabling us to reproduce any combination of code, environments, and dependencies.
- **Infra:** Primarily Terraform files for provisioning and configuring cloud resources.
- **Pipelines:** Orchestration logic that automates the ModelOps workflow and streamlines the journey from lab to production.

Together, these pieces make the model’s lifecycle, from experimentation to production, consistent, and repeatable.

MLOps v2 template Combine 4 modules.

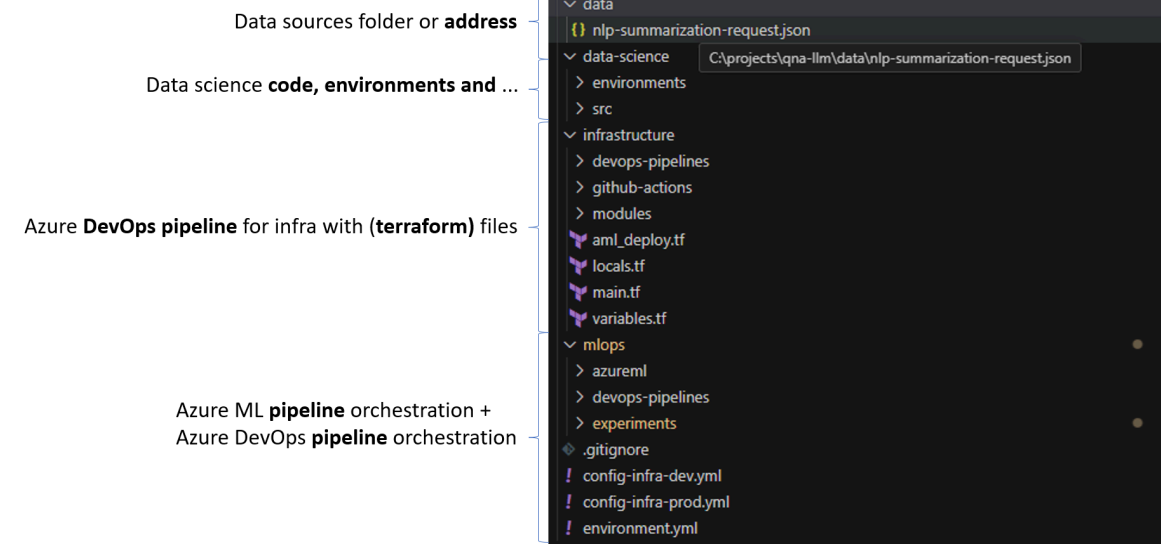


Figure 13. Code templates and folder structure, including data, data science pipelines, MLOps, infrastructure, and configurations.

The Volvo case highlights the business need and partial automation already in place. The next two sections present **generalizable architectures** that solve this scaling problem fully, at a deeper technical level.

Solution Perspective #1:

Modular pipelines for scaling the number of models [Red Hat]

We will be using Red Hat OpenShift AI which is an AI/ML platform that provides tools to build, deploy, and manage GenAI and machine learning models on OpenShift. OpenShift being Red Hat's enterprise kubernetes application platform. Think of it as the underlying "factory" environment that can host and run the system for our "assembly lines"

The solution brings to life the "assembly lines" concept by implementing a systematic and automated workflow. This implementation is not a single, monolithic tool, but rather a flexible architecture built from a set of five specialized source code management / Git repositories. Each repository is given a distinct responsibility, creating a clear separation of concerns that is key to managing complexity at scale.

This separation of concerns allows data scientists to define models using simple configuration files, while the system automates the training, packaging, and deployment. It also enables a great degree of transparency and tracking of changes to the system as a whole, which are additional but important aspects to any system of this scale.

The five core repositories are:

- Model Configurations Repository:
 - This repository is the main interface for data scientists. It contains lightweight configuration files (e.g., JSON) that define each model.
 - A model's configuration file specifies what data to use (e.g., a DVC hash from the data management repo), which training pipeline to run, what version of that pipeline to use, and any model-specific hyperparameters.
 - Changes to these files can trigger the entire automated workflow.
- Pipelines Repository:
 - This repository defines the "glue" that connects the system. It contains the high-level CI/CD pipelines (e.g., using Tekton) that orchestrate the entire process.
 - This includes the Continuous Training Pipeline, which triggers on a model configuration change, runs the corresponding training pipeline, scans/packages the model, and updates the model registry.
 - It also includes a Training Pipeline CD Pipeline, which manages the deployment and versioning of the training pipelines themselves.
- Training Pipelines Repository:
 - This repository stores the actual machine learning workflows (e.g., as Kubeflow pipelines). These are the pipelines that execute the ML-specific steps.
 - A typical training pipeline here would be responsible for fetching data, preprocessing it, training the model, evaluating its performance, and registering the final model artifact.

- These pipelines are versioned, allowing different models to be trained on different, stable versions of the training logic.
- Data Management Repository:
 - This repository handles data lineage and reproducibility. It does not store the raw data itself, but rather the metadata and versioning information.
 - It uses tools like DVC (Data Version Control) to track datasets. The model configuration files point to specific DVC hashes in this repo to ensure that training runs are reproducible.
- Deployment Repository:
 - This repository uses GitOps principles (e.g., with Argo CD) to manage the deployment of models to different environments (test, production).
 - It contains the model serving configuration files (e.g., YAML) for each model. These files point to the specific versioned model artifact (or "ModelCar") that should be served.
 - Promoting a model to production is done via a pull request in this repository, providing a human-in-the-loop approval gate.

Diving into these repositories for additional technical details.

Model Configurations Repository:

This is where the model configurations are stored. They identify what data the model should use for training, what pipeline should be used to train it, which parameters we send into the pipeline, and what runtime it should be served with. Here's a small example JSON file:

```
JSON
{
  "data_source": {
    "type": "dvc",
    "dataset": "datasets/churn-predictor/data.parquet",
    "repo_url":
    "https://github.com/nine-thousand-models/nine-thousand-data",
    "dvc_hash": "5c2145f3f900c69fb20c0af2657b7876"
  },
  "model_name": "churn-predictor-model",
  "kubeflow_pipeline": "classification-training-pipeline",
  "kubeflow_pipeline_version": "classification-training-pipeline-v1.23.0",
  "runtime_name": "tensorflow",
  "model_hyperparameters": {
    "epochs": 1
  }
}
```

Figure 14. Example config.json file.

Even though this is a small example, you can see how to extend this approach to gain more flexibility. By adding or adjusting hyperparameters in the JSON file, we can reuse the same training pipeline for many different models with different behaviors or performance

characteristics. This also makes it simple for data scientists to define new models or update existing ones whenever needed. Making it simple becomes an important aspect around onboarding not only new models but also people to work with the system.

Pipelines Repository

In this repo, we define the pipelines that glue everything together.

Continuous training pipeline

The continuous training pipeline kicks off our individual training pipelines, scans and packages the model, and ensures the metadata is updated to reflect the changes (see image below). We set this pipeline to trigger automatically on any change to our model JSON files, but you can also trigger it on other events or based on a schedule.

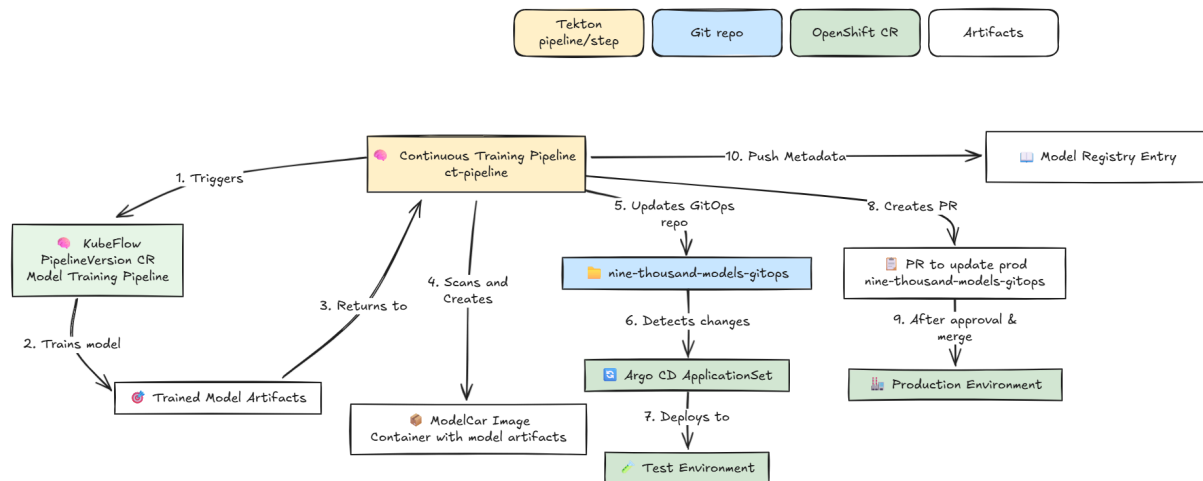


Figure 15. Flow of the continuous training pipeline.

And to make this concrete, below shows the Tekton tasks that implement this workflow.

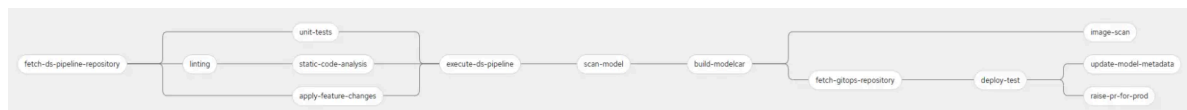


Figure 16. Flow of the overall workflow using Tekton/OpenShift Pipelines.

After the continuous training pipeline finishes, we will have a newly trained model packaged as a ModelCar and served in our test environment, a pull request (PR) that promotes that model into our production environment, and an entry in our model registry that reflects this new model version

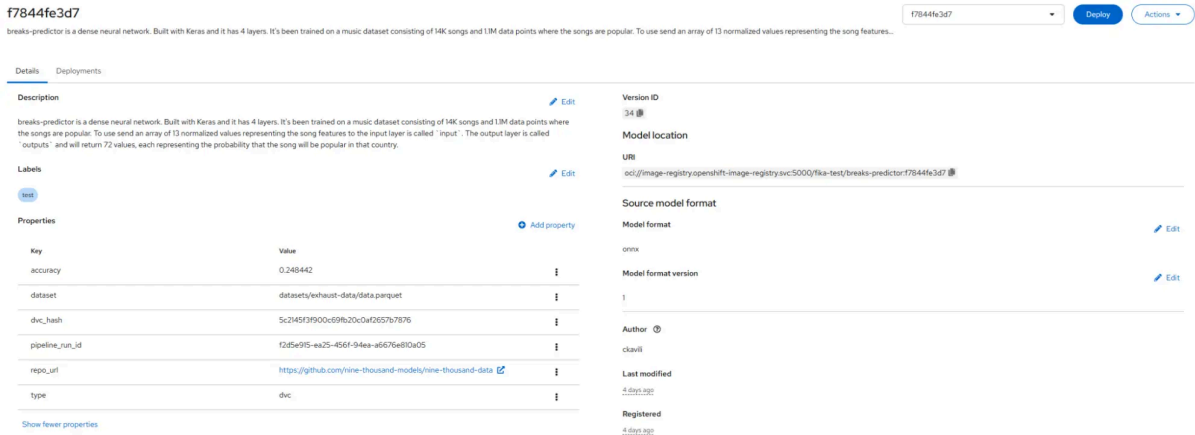


Figure 17. Screenshot of the model registered in the model registry.

Training pipeline CD pipeline

The training pipeline CD pipeline compiles and deploys new versions of the individual training pipelines so that they are ready to be executed by the continuous training pipeline. See below.

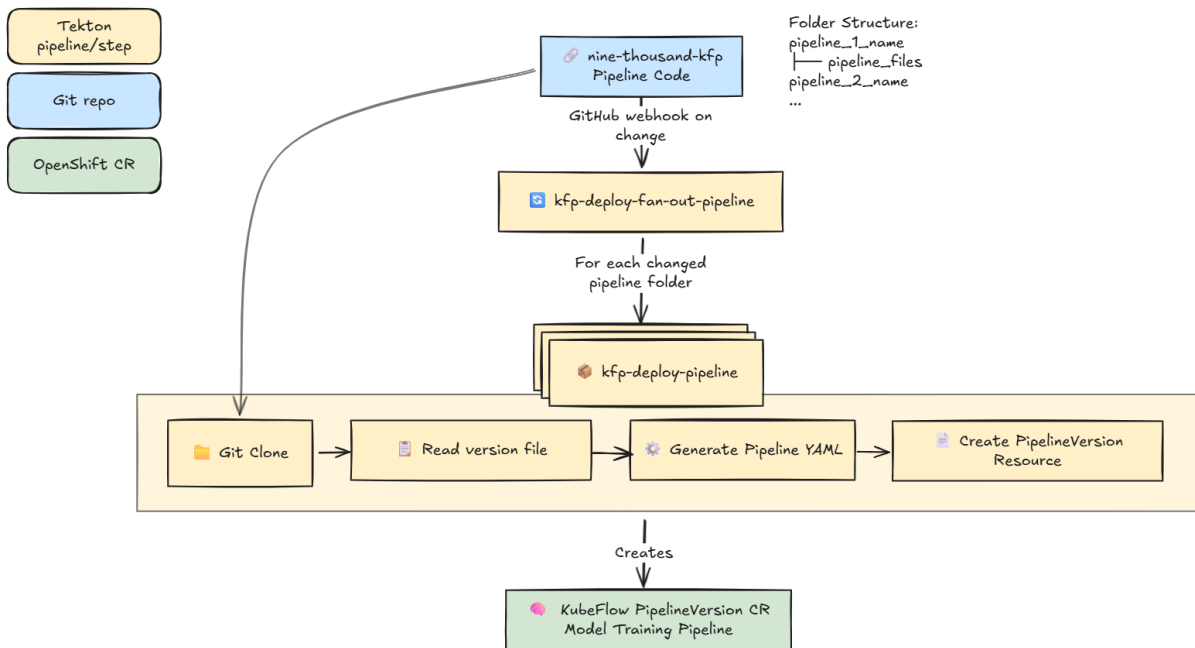


Figure 18. Diagram of the Continuous Deployment pipeline.

The training pipeline versions that this pipeline deploys are the ones we were referring to in the sample model JSON. This allows us to keep different models training on different versions of our training pipelines.

Data pipeline

We added a data pipeline just to showcase how data versioning could be managed. This is a small pipeline that detects when new data comes in, versions it through DVC (Data Version Control), and updates our data management repository (which just stores the DVC files; see below). It also updates our model repository for the models associated with the specific data

that was updated, and we now know what happens when we update a model JSON. We will get a new model trained on the new data specified in that JSON file.

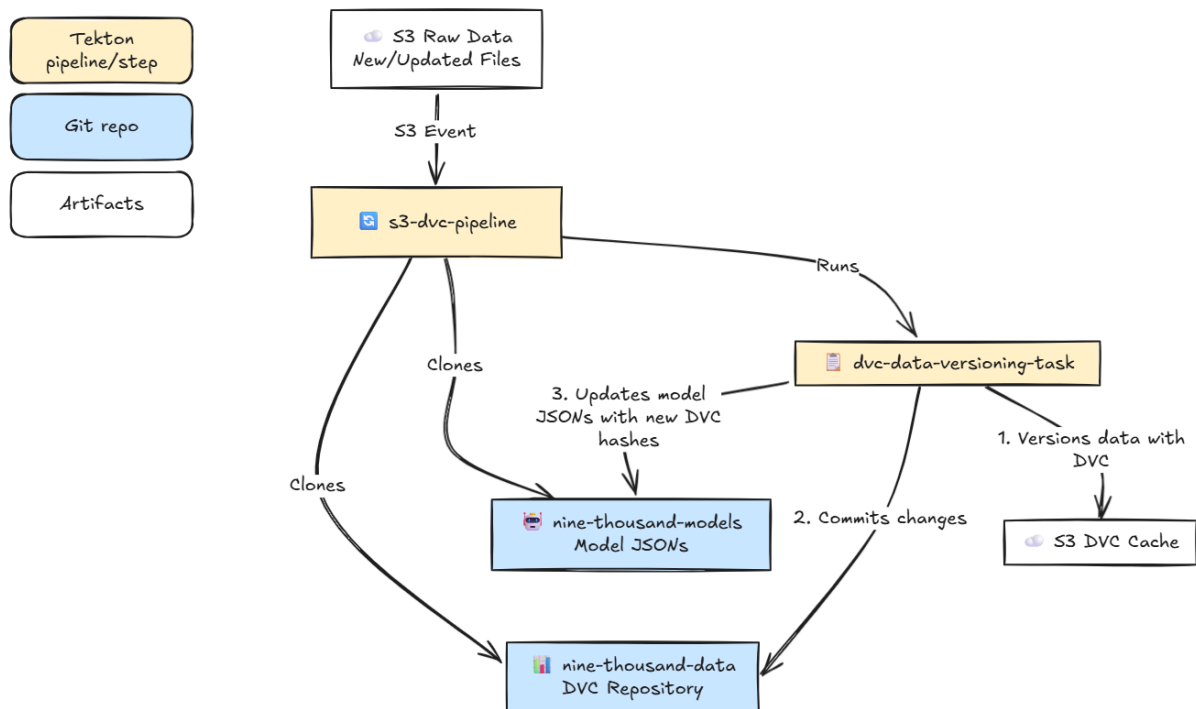


Figure 19. Flow of the data pipeline.

Training pipeline Repository

The training pipeline is where we store our individual training pipelines. In our case, these are Kubeflow pipelines that are built in Python so it's easy for a data scientist or machine learning engineer to create new ones or modify existing ones.

These are the pipelines that execute the actual machine learning (ML) flow: fetch data, preprocess, train, evaluate, validate, and register models.

These pipelines can be defined based on what kind of training steps you want to take. Ours looks like below.

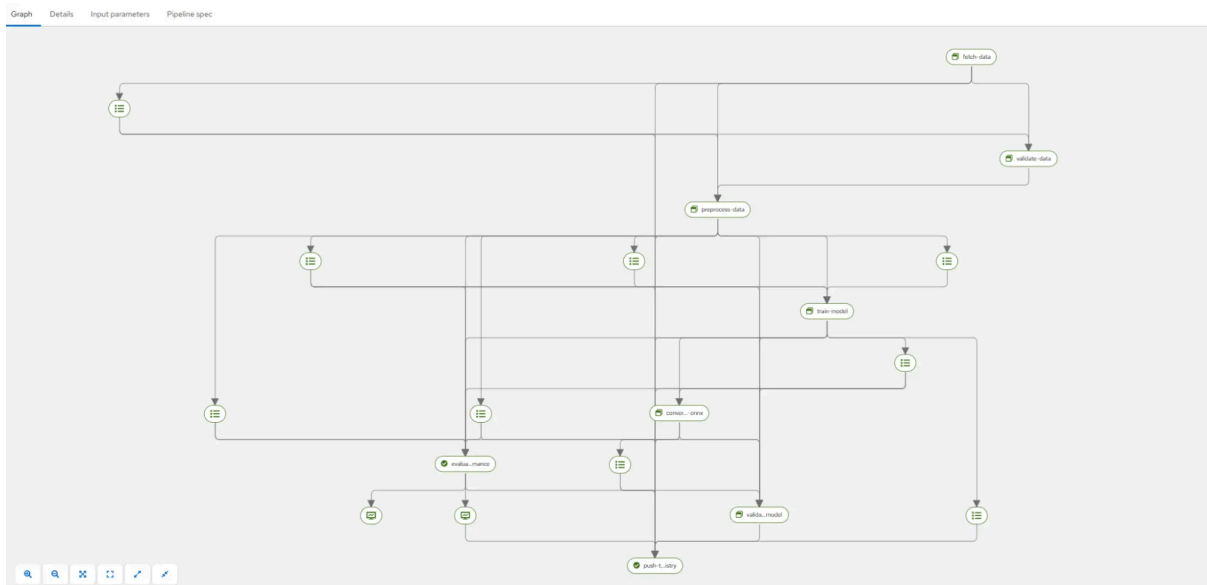


Figure 20. Screenshot of a kubeflow pipeline run.

Data management Repository

The data management repo simply holds our Data Version Control (DVC) information. If you don't use DVC, you might not need this repo.

Deployment Repository

We use Argo CD to automatically deploy our models, keeping the cluster synced with this GitOps repo. It contains YAML files for our model serving configurations in test and production (in two separate folders), which look something like this:

```
None
chart_path: charts/model-deployment/simple
name: anomaly-detector
image_repository: image-registry.openshift-image-registry.svc:5000
image_namespace: fika-test
version: f7844fe3d7
```

Figure 21. Configuration for deployment parameters.

The interesting part here is the version, **f7844fe3d7**, which points out which ModelCar we should serve as our model.

These connect together in a seamless flow, where all the data scientist/ML engineer needs to interact with are:

- The training pipelines repository, to build new pipelines or update existing ones.
- The models repository, to quickly modify or train new models.

Note: We chose these tools because they are open, cloud-native, and fit well into our Red Hat OpenShift AI platform. But the real point isn't which tools you use, it's the principles.

Describing an example retraining flow caused by Model/Data Drift.

Trusty AI monitors the model under inference and has an alertmanager configuration to trigger a webhook when detecting drift. The continuous training pipeline has an eventlistener that the alertmanager will call when the alert is raised, triggering a retraining of the model.

A note on security. To further evolve this architecture from the "assembly lines" concept, into a high-assurance, "Secure Model Factory." We can transform the "ModelCar" from a simple container into a verifiable, immutable, and trusted software artifact that is secure-by-default. This transformation is accomplished by systematically integrating a "security plug-in" of cloud-native, automated controls:

Tekton Chains & Sigstore (Red Hat Trusted Artifact Signer): For automated, keyless, and non-repudiable provenance and cryptographic signing, providing the "unified lineage" that was previously only a metadata entry.

Vulnerability Scanners (Red Hat Advanced Cluster Security): As a mandatory, automated Task in the CI pipeline, acting as a "shift-left" quality gate to block defects.

Red Hat Quay: As the unified, secure OCI registry for images, signatures, and attestations, becoming the concrete "Model Registry."

Admission Control (Red Hat Advanced Cluster Manager or RHACS Admission Control): As a non-negotiable, policy-as-code enforcement gate at deployment, replacing the "human-in-the-loop" security check.

Zero Trust, (Red Hat Zero Trust Workload Identity Manager): To secure the pipeline itself with dynamic, least-privilege credentials, protecting against the compromise of the "assembly line."

Through the lens of security, the "Thousand-Model Challenge", is at a minimum, two distinct problems: (1) How do we operate 1,000 models efficiently?. (2) How do we trust 1,000 models continuously? By embedding security into the "assembly line" as an automated, non-negotiable, and policy-driven component, this architecture achieves both scale and trust.

Solution Perspective #2: Data-Centrisism and Parametrization for Scaling AI Systems [Hopsworks]

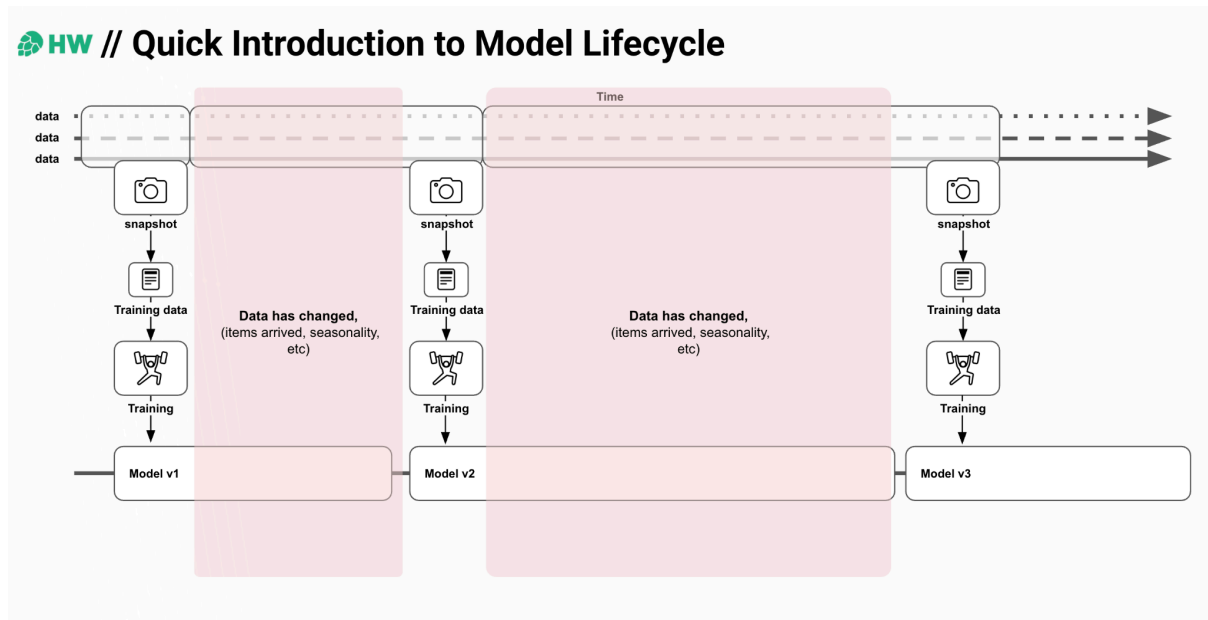


Figure 22. Timeline and models snapshots

In this perspective of the scaling challenge, we aim to separate actions that are programmatic (joining the data, training the models, etc...) from the data and infrastructural layer itself. We demonstrate this in the Hopsworks platform which, through its architecture illustrated below, separates the model-centric frameworks and the engineering frameworks responsible for the pre-processing of the data itself.

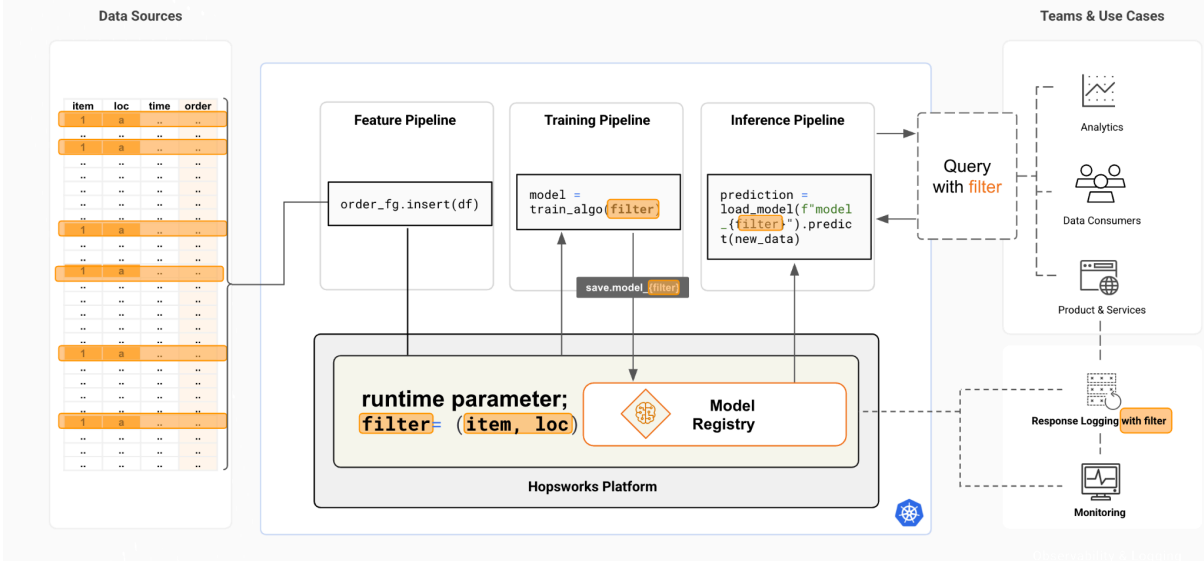


Figure 24. Hopsworks parameters in the production workflow.

Python

```
# Building complex queries through the Query abstraction
sales_fg = fs.get_feature_group("sales")
inventory_fg = fs.get_feature_group("inventory")
weather_fg = fs.get_feature_group("weather")

# Lazy query building - nothing executed yet
query = sales_fg.select(["item", "sales"]) \
    .join(inventory_fg.select(["stock"])) \
    .join(weather_fg.select(["temp"]))

# Define transformations separately (attached to specific features)
transformations = {
    "sales": fs.get_transformation_function(name="min_max_scaler"),
    "item": fs.get_transformation_function(name="label_encoder"),
    "temp": fs.get_transformation_function(name="standard_scaler")
}

# Query + transformations defined but not executed
feature_view = fs.create_feature_view(
    name="forecast_features",
    query=query,
    transformation_functions=transformations
)

# Execution + transformations happen here
training_data = feature_view.get_batch_data() # Data retrieved AND transformed
```

The code above showcases how the initial table is not altered by the query for the generation of the data; both joins and transformation are defined as part of the feature view

object and only applied during the retrieval for training; this allows for a clear programmatic way to generate new models without changing any source or deployed infrastructure.

Where does model management live?

As this is inherently a solution for production systems, the question of the model development could be raised; but quite simply, the same query that has been done for a production infrastructure can be done for development using the very same data structure, types, schemas and transformation as the production environment; abstracting the data management layer allows for users, enterprise and otherwise to manipulate dataframes; they could be queried from the same tables without altering the production system.

There is a direct benefit to this type of solution; it makes the iteration loop of model development much faster; same tables, same code; promoting a pipeline to production is inherently just code, not infrastructure changes, which allows the system to be able to rollback easily and a much better lineage/versioning of the individual assets.

Python

```
# Development registers models
model = mr.create_model("forecast", metrics={"accuracy": 0.95})
model.save("model_files")

# Staging deployment
staging = model.deploy(name="forecast_staging")

# Promote to production - same model, different deployment
production = model.deploy(name="forecast_prod")
# Roll back - just deploy a different version
old_model = mr.get_model("forecast", version=1)
production = old_model.deploy(name="forecast_prod") # Replaces existing
```

Finally, in hopsworks the model assets (artifact) are separated from the model serving infrastructure; users can deploy a different model in serving while simply choosing a different version as seen in the pseudo code above.

What's in the box?

Built on Kubernetes, the Hopworks platform is organized in the following Platform capabilities;

- **Data Source Management**
 - Connectors for existing data warehouses and databases.
 - External table support; allowing for data to stay where it is and be queried and computed **on-demand** when a model needs to be trained; this also supports real-time data that would live in the online feature store.
 - Event and Real-time streaming (support for Flink, and newer pythonic frameworks for real-time) and events with Kafka

- **Feature Store**, Hopsworks at its core has been built around its feature store technology; a dual-database system that prevents leakage and skew. The online and offline storage are kept consistent by the metadata layer; abstracting the need for managing two different databases.
 - Feature Views; are views over tables, they are the interface/input to models and retain the schema, version and transformation logic as metadata. A feature view can be used to filter sub-set of data for the generation of data sets and are unique to the Hopsworks platform.
- **Query engine**; Hopsworks Query Service (HQS), is a fast Python API built for performance, allowing 10-45 times higher throughput for data read from offline data. The service allows for efficient point in time correct training data from source tables and removes the need to write/generate/maintain complex temporal queries for time series data; it also transparently transpiles python into SQL. Finally, it supports federated queries across multiple backend without duplication.
- **Model registry**; with support for KServe. The registry is where developers publish their models during the experimentation phase. The model registry can also be used to share models with the team and stakeholders.
- **Model Serving**; Hopsworks Model Serving is built on Open-source Kserve, with its own library (HSML) to easily abstract and integrate with the rest of the platform while benefiting from the components of KServe.
- **Other Framework Support**; Integrated support for vLLM server and Ray framework

Discussion

The three solution perspectives presented in this white paper—Red Hat’s modular pipeline architecture, Hopsworks’ data-centric parameterization, and Volvo’s value-extraction-oriented ModelOps template—do not represent competing approaches, but rather **complementary patterns that address different layers of the Thousand-Model Challenge**.

<u>Challenge dimension</u>	<u>Red Hat</u>	<u>Hopsworks</u>	<u>Volvo</u>
Scaling the number of deployable models	Modular, containerized pipelines that support 100→1000 models with isolated execution	Parameterized models that share data pipelines + feature views instead of duplicating logic	Template-driven delivery enabling consistent model onboarding & infra reuse
Scaling without a linear increase in human labor	Pipeline assembly lines + GitOps → “model lifecycle as code”	Centralized feature store + shared metadata → reduces duplication & drift	Governance automation + environment replication reduce manual approvals

Avoiding operational entropy	Clear separation of concerns across 5 Git repos	Strong state management (data → model → version lineage)	Explicit lifecycle ownership and promotion rules
Reducing long-term technical debt	Everything versioned, auditable, repeatable	Data & model lineage tied to time, schema, and storage → no orphaned objects	Business value tied to each model → avoids zombie models in production
Business and compliance alignment	Policy-based promotion + reproducibility	Feature-level governance + data contracts	Traceable, risk-aware deployment and rollback strategy

Taken together, these three perspectives illustrate a **repeatable pattern for scaling model fleets**:

1. **Standardize the lifecycle** (from model spec to promotion)
2. **Automate the lifecycle** (pipelines, registries, validation gates)
3. **Decouple data from models** (shared feature objects, versioning, backfills)
4. **Enable closed-loop evolution** (monitoring → retraining → republishing)
5. **Attach business value to model ownership** (no model exists “just because it’s possible”)

In fact, **a mature platform will incorporate aspects from all the three different aspects**. A key observation is that **technology alone does not unlock scale**. A fully working MLOps stack still fails if:

- Nobody owns the model after deployment
- Retraining triggers are manual or undefined
- Model metrics are not connected to business KPIs
- Feature stores are bypassed and ad-hoc data wrangling reappears
- Compliance processes are still document-driven instead of policy-as-code

The Thousand-Model Challenge is therefore not just a matter of **running more models**, but of **running more models without multiplying overhead, drift, debt, or risk**.

In that sense, the “maturity jump” is not from *model* → *product*, but from *model* → *fleet*.

Conclusion and Recommendations

Companies that succeed in large-scale AI operations are not the ones that build the most advanced models, but the ones that **treat models as continuously evolving software artifacts with business accountability, not as static research outputs.**

The central finding of this initiative is:

It is possible to scale from tens to thousands of models without an unsustainable increase in human labor; provided the model lifecycle is standardized, automated, and connected to shared data assets and policy-driven governance.

A second important result is that **the scaling bottleneck is not exclusively compute.** It is also:

- Manual monitoring
- Manual promotion and rollback
- Manual retraining decisions
- Model-specific pipeline logic
- Model-specific data prep
- Human coordination across Data/ML/Infra/Business roles

Once these steps are turned from *meetings and documents* into *automation and code*, the human-per-model ratio decreases dramatically.

This white paper has shown three viable paths to get there: one infrastructure-driven (Red Hat), one data-centric (Hopsworks), one business-first (Volvo). All three demonstrate that the real question is no longer *Can we deploy ML models?* but:

Can we deploy, monitor, adapt, and retire hundreds of models sustainably, reliably, and at business speed?

Organizations that cannot answer yes will reach a point where **their models scale faster than their ability to manage them**; creating risk, decay, duplication, and wasted cost.

Organizations that can answer yes will treat **AI as a fleet**, not a set of one-off projects; unlocking higher value per model, shorter lead times, and a defensible operational advantage.

Appendix

The project “Data-driven organizations – Best practices for operationalization of AI in Sweden”

This material has been produced as part of the Vinnova-funded project *Data-driven organizations – Best practices for operationalization of AI in Sweden* (DDO), a project lasting just under two years with twenty participants from private sector, public sector, and academia. Together, they tackled issues concerning large- and small-scale operation of AI solutions and how to enable and use AI broadly across an organization.

The work focused on three specific use cases: Local sustainable operation of AI, legal and technical prerequisites for effective infrastructure, and how to create the best conditions for keeping thousands of AI models in operation.

A compilation of all material produced within the framework of DDO is available on [AI Sweden's website](#).

The organizations that participated in DDO were [Aixia](#), [Hewlett Packard Enterprise](#), [Hopsworks](#), [IBM](#), [Linköping University](#), [NetApp](#), [Predli](#), [Proact](#), [RISE](#), [RedHat](#), [Region Halland](#), [Sahlgrenska University Hospital](#), [Statistics Sweden](#) (Statistiska Centralbyrån), [The Swedish Tax Agency](#) (Skatteverket), [Stormgrid](#), [The Swedish Transport Administration](#) (Trafikverket), [Volvo Parts](#), [Region Västra Götaland](#), [Santa Anna](#), and [AI Sweden](#).

The project was funded by the participating organizations and Vinnova. AI Sweden is in part financed by the EU.