jozu

TECHNICAL GUIDE

End-to-End Kubernetes ML

Index

Executive Summary

Most enterprises deploy AI/ML using makeshift combinations of git repositories, object storage, and manual scripts. When models fail in production, teams waste hours determining which code version, data file, and configuration were actually deployed together. KitOps ModelKits solve this by packaging complete AI/ML projects (models, code, data, configs, dependencies) as single, versioned OCI artifacts that work with existing container registries, Kubernetes clusters, and CI/CD pipelines. One immutable reference replaces coordinating multiple systems to track project state.

This guide shows platform engineers and ML practitioners how to package projects as ModelKits, deploy them on Kubernetes, and integrate them into production workflows. A European logistics company managing hundreds of models eliminated multi-day debugging sessions by switching from custom scripts to ModelKits. A US research laboratory chose self-hosted ModelKit registries for air-gapped CI/CD with sensitive data. The pattern: teams reduce operational overhead and improve deployment reliability and security by treating AI/ML projects as versioned units rather than loosely coordinated files, using infrastructure and practices they already have.

Introduction

Moving AI/ML projects from development to production remains one of the hardest problems in enterprise technology. Data scientists build models that work perfectly on their laptops, but platform engineers struggle to deploy them reliably. Teams waste weeks debugging why a model that worked in staging fails in production.

The core problem isn't the model itself - it's everything around it. Models don't run in isolation. They need specific versions of code, particular datasets (or references to them), exact configurations, and compatible dependencies. Change any piece and behavior changes. Most teams handle this with a combination of git repositories, model and feature stores, container registries, and manual deployment scripts.

Throughout this guide, we use "AI/ML project" to refer to the complete set of versioned artifacts required to reproduce a working system: the model weights, training and inference code, datasets (or references to them), configuration files, prompts, dependencies, and environment specifications. This matters because deploying AI/ML successfully means deploying all of these components together, not just a model file.

When we refer specifically to "models," we're talking about the trained weights and architecture - the narrow technical artifact that gets loaded into memory for inference or training.

This guide shows you how to use KitOps and ModelKits to package, version, and deploy complete AI/ML projects on Kubernetes. You'll learn how to move from experimental workflows to production-grade deployment practices using tools your platform team already understands.

What is a ModelKit?

A ModelKit is a packaged AI/ML project stored as an OCI (Open Container Initiative) artifact - the same standard used for container images. It contains everything needed to reproduce your project's state at any point: model files, code, datasets, configuration, and dependency specifications.

A note on terminology:
Most ML tooling talks about "models" because that's what data scientists focus on during development. But operationalizing AI/ML means dealing with everything that surrounds the model - code, data, configs, dependencies. We use "AI/ML project" throughout this guide because getting value from AI requires not only the model, but the agents, workflows, and services around it that drive the customer outcome. The whole project, then, is what needs to move from development to production. ModelKits are unique in versioning and packaging the entire project, not just the model weights or dataset.

Think of a ModelKit as a deployment unit. Instead of tracking "which model version, which code commit, which config file" separately, you track one immutable reference. Your laptop, your colleague's laptop, staging, and production can all pull the exact same project state.

Why OCI Artifacts?

OCI is the standard for distributing container images: every container registry supports them; your security systems already scan them; your CI/CD pipelines already move them. Using OCI for AI/ML projects means you leverage existing infrastructure instead of building parallel and disjointed systems.

ModelKits package AI/ML projects just like container images package applications. Both use the same distribution mechanism, which means:

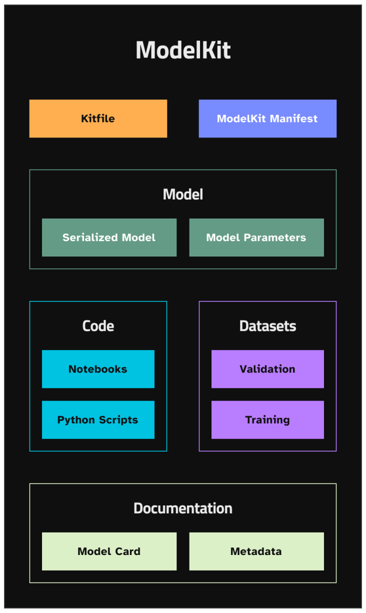
- Standard (and always consistent) registry authentication and access controls
- Built-in content addressing guarantees integrity
- The registry's content addressable storage automatically deduplicates assets saving storage and egress costs
- Native support comes for free with Kubernetes and other orchestration platforms

Best of all, ModelKits can be used with any container registry whether cloud hosted, private, on-premises, or open source.

For organizations requiring enhanced security and compliance capabilities, Jozu Hub provides an on-premises registry specifically designed for AI/ML workloads.

Unlike generic container registries, Jozu Hub adds automated security scanning for model-specific vulnerabilities, compliance reporting for regulated industries, and deployment controls that prevent tampered or unapproved models from reaching production. Teams use the same Kit CLI and workflows while gaining enterprise-grade security features.

ModelKit Structure



A ModelKit contains:

Model files: Weights, architectures, and model-specific artifacts in standard formats (ONNX, PyTorch, TensorFlow, etc.)

Code: Training scripts, inference handlers, preprocessing logic, and any custom code needed to use the model

Datasets: Training data, validation sets, or references to data locations with versioning information

Configuration: Hyperparameters, environment settings, feature definitions, and deployment specifications

Dependencies: Language runtimes, libraries, and system packages with exact versions

Documentation & Prompts: System and base prompts, README, installation and testing guides, or anything else

Metadata: Ownership and lineage information, training metrics, model cards, and documentation

Each component is versioned together. When you reference ModelKit version 1.2.3, you get all of these pieces in their exact state from that version.

Not all ModelKits need to contain all components - it's common to "layer" ModelKits. For example, in a fine-tuned RAG context that project's ModelKit may reference a canonical dataset ModelKit and a foundational model's ModelKit.

Development Workflow

Creating Your First ModelKit

Start with a trained model and its associated project files. The Kit CLI packages everything into a ModelKit.

Install the Kit CLI:

```
brew install jozu-ai/kitops/kit
```

Create a Kitfile describing your project:

```
manifestVersion: 1.0.0
package:
  name: fraud-detection
  version: 1.0.0
  description: Transaction fraud detection model
  authors: ["ML Team"]

model:
  name: fraud-detector
  path: ./models/fraud_model.onnx
  framework: onnx
  version: 1.0.0

code:
  - path: ./src/inference.py
  - path: ./src/preprocessing.py

datasets:
  - name: training-data
    path: ./data/training_set.parquet
  - name: validation-data
    path: ./data/validation_set.parquet

config:
  - path: ./config/model_config.yaml
  - path: ./config/feature_definitions.json
```

Pack and push the ModelKit:

```
# Pack the ModelKit and store it in the local registry
kit pack . -t local-registry.com/fraud-detection:1.0.0

# Push the ModelKit from local to a remote registry for sharing
kit push local-registry.com/fraud-detection:1.0.0 remote-registry.com/fraud-detection:1.0.0
```

Your complete project is now in your registry as an immutable, versioned artifact.

Why Not Just Use Experiment Trackers?

Experiment tracking tools (MLflow, Weights & Biases, Neptune) excel at comparing model performance during development. They version model weights and log hyperparameters effectively. But they weren't designed to version the complete project required for production deployment.

The gap shows up when you promote a model to production. Your experiment tracker tells you which model performed best. It doesn't tell you which version of the preprocessing code to deploy with it, which configuration file controlled feature engineering, or which dataset version influenced model behavior. Teams bridge this gap manually - tracking code versions in git, config versions in separate systems, and hoping everything aligns correctly in production.

ModelKits package the complete project state that experiment trackers don't capture. You still use MLflow or W&B for comparing experiments, but add ModelKits when you need to deploy, version, and reproduce the entire project - not just the model weights.

Local Development and Testing

You interact with ModelKits like a container - by pushing and pulling from a registry.

To pull a ModelKit from the registry onto your local file system:

```
kit pull myregistry.com/fraud-detection:1.0.0
```

Then unpack only the model to the local fraud-project directory:

```
kit unpack myregistry.com/fraud-detection:1.0.0 -f=model -d ./fraud-project
```

The unpack operation recreates the exact project structure from the ModelKit. Every file, every dependency version, every configuration setting matches what was packed. Optionally, you can extract only those parts of the ModelKit you need using filters.

```
kit unpack myregistry.com/fraud-detection:1.0.0 ./fraud-project
```

This solves the "works on my machine" problem. When a colleague pulls the same ModelKit, they get identical project state. When you deploy to staging or production, those environments get the same state. No drift, no surprises.

One government SI managing hundreds of models across their supply chain optimization platform switched from scattered packaging repositories to ModelKits. Their biggest win wasn't faster deployments - it was eliminating the "which version of which data file went with which model" debugging sessions that previously consumed hours or days of engineering time.

Updating and Versioning

When you update your project, create a new ModelKit version:

```
# Make changes to code, retrain model, update configs
kit pack . -t myregistry.com/fraud-detection:1.1.0
kit push myregistry.com/fraud-detection:1.1.0
```

The previous version (`1.0.0`) remains available. You can run both in production, roll back instantly, or compare behavior across versions. Because ModelKits are immutable, version `1.0.0` will always contain exactly what it contained when you first pushed it.

Handling Large Datasets

OCI registries handle multi-gigabyte artifacts efficiently - they're designed for container layers that can reach tens of gigabytes. ModelKits leverage the same content-addressable storage and layer deduplication that makes container distribution fast and inexpensive.

For datasets under 50GB, include them directly in the ModelKit. The registry's deduplication means you only store each unique chunk once. If 90% of your training data stays constant between versions, you only pay storage costs for the 10% that changed.

For datasets over 50GB or datasets that change frequently, use references instead of embedding.

The ModelKit stores the reference, version, and content hash. Your deployment pulls data from the source (S3, data lake, feature store) but the ModelKit still captures which exact dataset version the model expects. This gives you versioning without storage duplication.

Production deployments commonly use dataset references. Development and testing environments often embed smaller validation datasets directly for reproducibility.

Jozu Hub tracks both embedded datasets and dataset references, maintaining lineage even when data lives in external systems. When a ModelKit references an S3 bucket or feature store, Jozu Hub logs the reference details, version hash, and access timestamp. During audits, you can prove which exact data version trained each model without storing duplicate copies.

Development Patterns

Teams use three main patterns with ModelKits, each suited to different workflows and compliance requirements:

Automatic Packaging Post-Training:
Create a new ModelKit automatically when training runs complete successfully. The SDK packages resulting assets and pushes them to the registry without manual intervention. This pattern works for teams wanting immediate packaging or needing all runs packaged for compliance reasons.

Selective Packaging of Champion Models:
Package only models meeting performance thresholds. The SDK can query experiment tracking tools for runs matching specific criteria - accuracy above threshold, loss below target - then package only qualifying models. This keeps production registries focused on production-ready candidates.

Milestone-Based Packaging:
Most teams create ModelKits at key milestones: when validation passes, metrics meet thresholds, or models are ready for promotion to staging. This balances governance benefits with development velocity. During active experimentation, artifacts remain in existing tools (MLflow, feature stores, S3). When ready for promotion, package everything into a ModelKit for reliable handoff to downstream teams.

For regulated industries, automatic packaging of all runs provides complete auditability. For non-regulated industries, milestone-based packaging minimizes workflow disruption while enabling reliable production deployments.

Best Practices

- Version everything together from the start - retroactive versioning is painful
- Use semantic versioning for ModelKits to signal breaking changes
- Tag ModelKits with meaningful metadata for searchability
- Automate ModelKit creation in CI pipelines rather than manual packing
- Test unpacked ModelKits in staging before production deployment

The goal isn't to replace your existing ML infrastructure - it's to give you a better way to package and distribute AI/ML projects through infrastructure you already have. Container registries, Kubernetes, CI/CD pipelines - ModelKits work with the tools your platform team already maintains.

Securing the Project

Registry Integration and Authentication

Jozu Hub extends your existing OCI registry with enterprise governance capabilities. Most organizations attach Jozu Hub to their current registry infrastructure—Amazon ECR, JFrog Artifactory, or similar—rather than deploying a separate system. This keeps authentication and authorization unchanged. ModelKits use the same RBAC policies, identity management, and access controls already governing container images.

Teams typically separate projects through registry namespaces. The fraud-detection team maintains their ModelKits independently from customer-segmentation, yet both inherit the same security scanning, policy enforcement, and audit logging. This structure provides autonomy within governance boundaries without fragmenting infrastructure.

Automated Security Scanning

Every ModelKit pushed to Jozu Hub triggers multi-layered security analysis targeting AI/ML-specific attack vectors for generative AI and machine learning models. This addresses the unique security challenges that emerge when models become production systems - threats that traditional vulnerability scanning misses entirely.

The scanning framework employs five specialized scanners operating in concert. Each addresses specific attack vectors while maintaining complete on-premises deployment capability - critical for organizations with strict data sovereignty requirements. All tools run self-hosted with no dependency on external APIs or cloud services, ensuring compliance with regulatory requirements while keeping ModelKit data entirely within organizational infrastructure.

Attack Vector Coverage

Supply Chain Protection:
Serialized model files in pickle, joblib, HDF5, ONNX, and TensorFlow SavedModel formats can execute arbitrary code during loading. Static analysis examines these files for malicious deserialization payloads and embedded code before they reach runtime environments. This prevents compromised models from entering your pipeline disguised as legitimate artifacts. Behavioral validation provides secondary verification by testing whether models exhibit suspicious runtime behavior patterns.

Content Safety Validation:
Models and their associated data undergo safety checks across multiple dimensions. Prompt templates get scanned for injection patterns that could override system instructions.

Training data gets analyzed for PII exposure, toxic content, and embedded secrets. Configuration files get checked for exposed credentials. Code artifacts get examined for malicious patterns. This multi-layer scanning ensures ModelKits don't inadvertently contain sensitive information or harmful content that could compromise production systems or violate data protection regulations.

Behavioral Red-Teaming:
Automated probes test how models respond to adversarial inputs through systematic attack simulation. Prompt injection testing attempts to override system instructions with malicious prompts. Policy evasion probes try to trick models into violating guidelines. Data exfiltration testing checks whether attackers can extract training data through carefully crafted queries. System prompt leakage detection verifies whether users can discover internal instructions. Each probe generates risk scores that inform deployment decisions—models failing critical probes get blocked from production automatically.

Adversarial Robustness Testing:
Sophisticated attacks attempt to fool models through carefully crafted inputs. Evasion attack testing evaluates whether adversarial examples can cause misclassification. Poisoning vulnerability detection analyzes training data for evidence of manipulation. Model extraction resistance testing measures whether attackers can steal model logic through repeated querying. Membership inference testing determines if attackers can discover whether specific data was in the training set. This deep analysis is critical for models making security decisions, processing financial transactions, or handling untrusted user input.

Privacy Protection:
Models can leak sensitive information through multiple channels. PII scanning examines training data, prompts, and configuration files for exposed personal information—names, addresses, financial data, health records—that shouldn't be in production systems. Membership inference testing determines whether attackers can discover if specific individuals' data was included in training sets, a critical concern under GDPR and similar privacy regulations. Data leakage detection identifies whether models inadvertently memorize and reproduce training data when queried with specific prompts. This dual-layer approach catches privacy risks before they become compliance violations or security incidents.

Model Integrity:
Compromised models pose existential risks to AI systems. Backdoor detection searches for hidden triggers embedded in model weights that could activate malicious behavior when specific inputs appear. These backdoors can survive training and remain dormant until exploited. Poisoning vulnerability analysis examines training data for evidence of manipulation—mislabeled examples, adversarial samples, or biased data injected to corrupt model behavior. Statistical analysis detects anomalies indicating poisoning attempts. While detection is moderate (sophisticated backdoors and poisoning can evade analysis), the scanning provides baseline protection against known attack patterns and obvious manipulation attempts.

Regression Prevention:
Reproducible test suites measure jailbreak resistance and prompt robustness across model iterations. This ensures models maintain consistent security posture as they evolve and that updates don't introduce new vulnerabilities. Tests run automatically on every ModelKit push, catching degradation before it reaches production.

Policy-Driven Scanning Tiers

Not every ModelKit requires identical scrutiny. The policy engine routes ModelKits to appropriate scanning tiers based on configurable YAML policies that evaluate model characteristics and deployment context.

Essential Tier provides rapid baseline security for experimental models during active development.
Supply chain attack checks verify model file integrity. Optional content safety scanning adds basic validation for PII and secrets. This tier completes within 5 minutes, enabling fast iteration without sacrificing fundamental security. It targets models that won't leave development environments but still need protection against obvious threats.

Standard Tier balances thoroughness with efficiency for staging deployments and integration testing.
Required checks include supply chain protection and comprehensive content safety validation. Optional assessments add behavioral red-teaming and adversarial testing based on model type and risk profile.

This 15-minute scan provides solid coverage for most production candidates. It's the default tier for models moving from development to QA or staging environments where they'll face more realistic testing conditions.

Comprehensive Tier applies maximum scrutiny to critical production models serving real users or handling sensitive data.
All security checks execute with extensive test suites. Adversarial robustness testing runs with GPU acceleration to handle computationally intensive attack simulations within acceptable timeframes. While requiring up to 60 minutes, this tier ensures models meet the highest security standards for regulated deployments. It's mandatory for models in healthcare, financial services, or other domains where model failures carry serious consequences—both technical and regulatory.

The policy engine selects tiers by evaluating multiple factors: model size (larger models receive more thorough scanning), deployment target (production tags trigger comprehensive assessment), model type (LLMs and multimodal models require specialized testing), and custom organizational policies. For example, a policy might route all LLMs to Comprehensive Tier regardless of size, or require Standard Tier for any model tagged for customer-facing deployment. Teams configure these rules through YAML policies that adjust as requirements evolve, enabling governance without blocking development velocity.

Attack Vector Completeness and Limitations

This multi-dimensional scanning approach provides strong to excellent coverage across most AI/ML security threats. However, upload-time scanning has inherent limitations that highlight the importance of defense in depth. Stateful attacks involving multi-turn conversations or memory manipulation cannot be fully tested without deployment context. These require runtime monitoring to detect. Integration attacks targeting plugin interactions or API abuse need actual integrations to manifest. Performance attacks attempting denial-of-service through resource exhaustion require actual infrastructure to measure impact. Behavioral drift (models changing behavior over time) only emerges during operation.

These limitations don't diminish the framework's value, they define its role in a broader security strategy. Upload-time scanning shifts security testing earlier in the cycle and prevents known vulnerabilities from reaching production. Runtime monitoring catches behavioral issues that only emerge during operation. Together, they provide comprehensive protection across the model lifecycle.

Security Attestations

Every scan generates cryptographically signed attestations documenting results in a standardized JSON format.

These attestations become immutable records attached to ModelKits using Cosign, providing auditable evidence of security assessments that can't be tampered with or forged.

The attestation captures comprehensive details: tool identification with versions, ModelKit reference with SHA-256 digest, metric definitions with pass/fail thresholds, individual findings with severity levels (info, anomaly, concern, error), and execution metadata including timing and configuration. Each security check contributes one evaluation run to a unified document, creating a complete security picture for the ModelKit.

These attestations serve multiple purposes throughout the deployment lifecycle. Deployment systems check attestations before allowing production releases—no attestation or failed checks mean no deployment, enforced at the infrastructure level. Compliance audits use them as evidence of security diligence, demonstrating that models underwent thorough testing before serving users. Incident investigations reference them to understand model vulnerabilities and trace security decisions back to specific findings. The cryptographic signatures ensure attestations haven't been tampered with, maintaining chain of custody from scanning through deployment and into audit review.

The framework evolves continuously as new attack vectors emerge. The modular architecture enables adding new security checks without disrupting existing workflows. Scanner capabilities update automatically to detect the latest attack patterns. Policies adjust to reflect changing organizational requirements. The result: security that improves over time while maintaining the same interface—push a ModelKit, receive comprehensive security assessment.

Policy Enforcement and Governance

Approval workflows transform deployment from uncontrolled progression to governed transition. A data scientist pushes a model—it enters quarantine. Security scanning passes—it awaits business approval. Manager approves—it becomes eligible for staging. Each gate is enforced at the registry level through policy evaluation.

Environment restrictions prevent models from jumping stages. Development models cannot deploy to production. Healthcare models require HIPAA attestations. Financial models need SOX compliance checks. Jozu Hub blocks non-compliant deployments automatically.

Compliance attestations attach to ModelKits as cryptographically signed statements verifiable by auditors. "This model was trained on anonymized data." "This model passed fairness testing." "This model complies with GDPR." These aren't code comments—they're signed claims with chain of custody.

Production Deployment on Kubernetes

Basic Deployment Pattern

You can deploy a ModelKit to Kubernetes using standard patterns. Here's a deployment for a model serving endpoint:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fraud-detection
spec:
  replicas: 3
  selector:
    matchLabels:
      app: fraud-detection
  template:
    metadata:
      labels:
        app: fraud-detection
        modelkit-version: "1.0.0"
    spec:
      initContainers:
      - name: unpack-modelkit
        image: ghcr.io/jozu-ai/kit:latest
        command: ['kit', 'unpack', 'myregistry.com/fraud-detection:1.0.0', '-d', '/mnt/modelkit']
        volumeMounts:
        - name: modelkit-storage
          mountPath: /mnt/modelkit

      containers:
      - name: inference-server
        image: myregistry.com/inference-runtime:latest
        volumeMounts:
        - name: modelkit-storage
          mountPath: /mnt/modelkit
        env:
        - name: MODEL_PATH
          value: /mnt/modelkit/models/fraud_model.onnx
        - name: CONFIG_PATH
          value: /mnt/modelkit/config/model_config.yaml

      volumes:
      - name: modelkit-storage
        emptyDir: {}
```

The init container unpacks the ModelKit before the inference server starts. The inference server loads the model, code, and configuration from the unpacked location. Everything needed to run the model is present and versioned together. More importantly, everything can be tied back to the ModelKit and even to the experiment run or notebook that the ModelKit was built from!

Immutable Deployments

Traditional model deployment often looks like this: update a model file in object storage, restart pods, hope the code still matches. With ModelKits, you reference a specific version in your deployment spec. That version never changes.

To update the ModelKit reference and apply the new deployment:

```
- image: ghcr.io/jozu-ai/kit:latest
  command: ['kit', 'unpack', 'myregistry.com/fraud-detection:1.1.0', '-d', '/mnt/modelkit']
```

Kubernetes handles the rollout. If something breaks, roll back by reverting the deployment to reference the previous ModelKit version. No hunting through logs to figure out which S3 bucket had which model file.

Integration with Model Serving Platforms

KServe's storage initializer architecture enables direct ModelKit integration without custom serving code. A ModelKit-specific storage initializer handles the kit:// protocol, pulling ModelKits from your registry, verifying their SHA-256 digests, and unpacking artifacts into serving containers. This maintains the security chain through to inference - the same immutable artifacts that passed security scanning are exactly what serves predictions.

```yaml
apiVersion: serving.kserve.io/v1alpha1
kind: ClusterStorageContainer
metadata:
  name: kitops
spec:
  container:
    name: storage-initializer
    image: ghcr.io/kitops-ml/kitops-kserve:latest
    imagePullPolicy: Always
    env:
      - name: KIT_UNPACK_FLAGS
        value: "" # Additional flags for `kit unpack`
    resources:
      requests:
        memory: 100Mi
        cpu: 100m
      limits:
        memory: 1Gi
  supportedUriFormats:
    - prefix: kit://
```

Authentication leverages Kubernetes-native patterns. Service accounts control which clusters can access which ModelKits. Development clusters cannot pull production models. Production clusters cannot access experimental models. This enforces environment boundaries at the infrastructure level.

The ModelKit URI (kit://myregistry.com/fraud-detection:1.0.0) replaces traditional S3 URIs pointing to mutable buckets. This single change transforms deployments from hopeful to deterministic. The ModelKit structure ensures the URI points to model weights, preprocessing code, configuration, and documentation as a complete package.

Performance and Automation

ModelKit manifests (Kitfiles) can include resource requirements from training, enabling KServe to allocate appropriate CPU, memory, and GPU resources without manual specification.

Performance optimization through caching reduces registry load and deployment time. KServe's persistent volume caching stores unpacked ModelKits locally. Subsequent deployments check cache validity against SHA-256 digests from your registry. Changed models trigger fresh pulls. Unchanged models serve from cache. This balances security verification with operational efficiency - every deployment confirms artifact integrity without repeatedly downloading gigabytes of unchanged model weights.

Tracking Across Environments

Experiment tracking tools like MLflow or Weights & Biases version models well. They're less helpful for versioning the code that loads those models, the data that trained them, or the configuration that controls behavior. When you promote a model from staging to production, you're not just moving model weights - you're moving an entire project.

ModelKits make this explicit. Your staging environment tests ModelKit version `x.y.z`. If tests pass, production deploys the same version `x.y.z`. Not a new model file with "hopefully the same code" - the exact same immutable project package.

Several research laboratories working with sensitive data chose self-hosted ModelKit registries specifically because their CI/CD runs in air-gapped environments. They needed to version and distribute complete AI projects without external dependencies or cloud services. The OCI standard made this straightforward with existing container registry infrastructure.

CI/CD Integration

Automated pipelines benefit most from project-level versioning. When your training pipeline produces a new model, you want to test the entire project - not just the model weights.

GitHub Actions Example

```yaml
name: Train and Deploy Model
on:
  push:
    branches: [main]

jobs:
  train-and-package:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Train model
      run: python train.py

    - name: Run validation tests
      run: pytest tests/

    - name: Install Kit CLI
      run: |
        curl -L https://github.com/jozu-ai/kitops/releases/latest/download/kit-linux-amd64.tar.gz | tar xz
        sudo mv kit /usr/local/bin/

    - name: Package ModelKit
      run: |
        kit pack . -t ${{ secrets.REGISTRY }}/fraud-detection:${{ modelkit.SHA }}
        kit push ${{ secrets.REGISTRY }}/fraud-detection:${{ modelkit.SHA }}

    - name: Deploy to staging
      run: |
        kubectl set image deployment/fraud-detection \
          unpack-modelkit=ghcr.io/jozu-ai/kit:latest \
          --env="MODELKIT_REF=${{ secrets.REGISTRY }}/fraud-detection:${{ modelkit.SHA }}"
```

This pipeline trains a model, packages the complete project into a ModelKit, and deploys it to staging. Adding the specific SHA to the ModelKit reference (`modelkit.SHA`) ties the deployment to a specific ModelKit version, giving you complete traceability.

ModelKits can be used with any pipeline that can consume containers or OCI Artifacts.

Advanced Deployment Patterns

A/B testing through traffic splitting requires unambiguous model identification. Tag-based promotion enables clear experiment definition using KServe's traffic management:

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: fraud-detection
spec:
  predictor:
    canaryTrafficPercent: 20
    model:
      modelFormat:
        name: onnx
      storageUri: kit://myregistry.com/fraud-detection:champion
  canary:
    model:
      modelFormat:
        name: onnx
      storageUri: kit://myregistry.com/fraud-detection:challenger
```

The `fraud-detection:champion` ModelKit receives 80% of traffic while `fraud-detection:challenger` receives 20%. Metrics collection happens at the ModelKit level, enabling direct comparison between versions. When the challenger outperforms the champion, promotion means retagging - the same immutable artifact that served 20% of traffic now serves 100%.

Canary Deployments with Automated Rollback

Canary deployments reduce risk for critical models through gradual rollout with automated rollback. New versions receive minimal traffic initially - 1% or less - with automatic rollback triggers if error rates spike or latencies degrade. ModelKit immutability guarantees rollback returns to an exact previous state, not a reconstructed approximation.

KServe monitors metrics, compares against thresholds defined in deployment policies, and reverts to the previous ModelKit version if the canary fails validation. Because ModelKits are immutable, these rollbacks complete in seconds rather than minutes or hours.

Multi-Model Ensemble Serving

Complex AI systems often combine multiple models for ensemble predictions. A fraud detection system might combine transaction analysis, user behavior modeling, and network graph analysis. Each model exists as a separate ModelKit, versioned independently but deployed together:

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: fraud-ensemble
spec:
  predictor:
    containers:
    - name: transaction-analyzer
      image: myregistry.com/inference-runtime:latest
      env:
      - name: MODELKIT_URI
        value: kit://myregistry.com/transaction-analyzer:v3
    - name: behavior-model
      image: myregistry.com/inference-runtime:latest
      env:
      - name: MODELKIT_URI
        value: kit://myregistry.com/behavior-model:v7
    - name: ensemble-orchestrator
      image: myregistry.com/ensemble-server:latest
```

Metadata or tagging ensures compatible versions deploy as a unit `- transaction-analyzer:v3` only deploys with `behavior-model:v7`, preventing version mismatches that could degrade ensemble performance. Each component ModelKit maintains its own security scanning results and audit trail while functioning as part of a larger system.

Team Collaboration and Governance

Shared Project State

Making an AI project successful takes a village: data scientists create ModelKits during development, ML engineers test them in staging, platform engineers deploy them in production, DevOps teams manage the registry, and Security teams scan the artifacts. Everyone references the same versioned project.

When someone asks "what's running in production," the answer is an immutable ModelKit reference: `myregistry.com/fraud-detection:1.0.0`. That reference tells you everything: which model, which code, which data, which configuration. Stop using spreadsheets to track deployment versions and Slack messages asking "did we deploy the model trained on the new data or the old data?"

Audit and Compliance

Regulated industries need to answer questions like: "What model version processed this transaction? What data trained that model? Who approved the deployment?"

ModelKits provide project-level lineage. The Kitfile metadata includes training information, model metrics, and authorship. Registry access logs show who pushed and pulled which versions, while Kubernetes labels tie running pods to specific ModelKit versions.

When auditors ask for evidence, you can point to immutable artifacts with complete provenance and lineage rather than trying to reconstruct history from scattered logs and git commits.

Troubleshooting and Debugging

Version Comparison

When a newly deployed model behaves differently than expected, you don't need to hunt through multiple logs, just compare ModelKit versions:

```
kit diff myregistry.com/fraud-detection:1.0.0
myregistry.com/fraud-detection:1.1.0
```

The `kit diff` command shows what changed between versions in the terminal.

Jozu Hub provides a UI for visual comparison with line-by-line diffs of changes: updated model files, modified preprocessing code, different hyperparameters, dataset changes, or prompt rewrites. Security scans run automatically on each version, surfacing new vulnerabilities immediately. You're not guessing what changed - you're examining the immutable record of project state with full security context.

Reproducing Issues Locally

To pull the exact ModelKit version running in production:

```
kit pull myregistry.com/fraud-detection:champion
kit unpack myregistry.com/fraud-detection:champion -vv ./debug
```

Now you have the production project state on your laptop. Run the inference code with the production model and configuration. Feed it the problematic input. Debug with full context instead of trying to reproduce production conditions from memory.

Rollback Procedures

Rolling back to the previous ModelKit version can be done through your serving platform:

```
kubectl set image deployment/fraud-detection \
unpack-modelkit=ghcr.io/jozu-ai/kit:latest \
--env="MODELKIT_REF=myregistry.com/fraud-detection:1.0.0"
```

Kubernetes performs a standard rolling update back to the working version. Because ModelKits are immutable, you know exactly what you're rolling back to.

Production Monitoring and Drift Detection

Prometheus metrics collection extends beyond standard HTTP metrics to capture model-specific behavior. Prediction latency distributions, confidence score patterns, feature value ranges, and inference throughput all link back to ModelKit versions, enabling version-specific analysis:

```
# Sample Prometheus metrics using ModelKit SHA as IDs
modelkit_prediction_latency_seconds{modelkit="fraud-detection${SHA}"} 0.045
modelkit_prediction_confidence{modelkit="fraud-detection${SHA}"} 0.87
modelkit_prediction_count{modelkit="fraud-detection${SHA}",result="fraud"} 42
```

When metrics degrade after deploying `fraud-detection:SHA`, teams can use the SHA to compare the in-production ModelKit and diff against previous known good deployments. Or, using the SHA and ModelKit trace back to the experiment run that generated the model and check runtime metrics against tested metrics to isolate the regression.

Drift detection requires baseline comparisons that ModelKits provide naturally. The validation datasets included in ModelKits during development establish performance baselines - expected accuracy, precision, recall, and confidence distributions. Production predictions get compared against these baselines through statistical tests that identify significant deviations.

When drift exceeds configurable thresholds, automated retraining pipelines trigger, creating feedback loops back to development workflows. Automatic rollback triggers prevent prolonged outages when models degrade in production. When error rates spike, latencies increase beyond SLAs, or drift accelerates past emergency thresholds, KServe can automatically revert to previous ModelKit versions.

This closes the loop from development through security to production operations. Models packaged as ModelKits, secured through scanning, promoted through pipelines, now serve predictions with full observability and automated recovery. The atomic unit established at the beginning - the ModelKit - maintains its integrity and traceability through every stage of the lifecycle.

Conclusion:

Moving AI/ML projects from development to production requires more than deploying model weights. Teams must coordinate models, code, datasets, configurations, and dependencies across environments while maintaining security, auditability, and reproducibility. The traditional approach of managing these components separately creates operational overhead, increases debugging time, and introduces deployment risk.

ModelKits solve this by treating the complete AI/ML project as a single, versioned unit. By packaging everything needed to reproduce a working system into an OCI artifact, teams leverage existing container infrastructure instead of building parallel systems. The same registries that store container images now store complete AI/ML projects. The same CI/CD pipelines that deploy applications now deploy models. The same security scanning that protects code now protects AI workloads.

This matters because it reduces the risk of production issues and shortens time-to-recovery for when outages do happen. The fact that it typically speeds time-to-production is a bonus.

The operational benefits compound over time. Debugging sessions that once took days now take hours because you can compare exact project states across versions. Audit preparation that once took weeks now takes days because you have complete provenance for every deployed model.

Deployment failures that once required hunting through multiple systems now point to a single ModelKit reference.

For organizations requiring enhanced security and compliance, Jozu Hub extends the ModelKit pattern with automated vulnerability scanning, compliance reporting, and deployment controls specifically designed for AI/ML workloads in regulated industries. From five-layer security scanning that catches AI-specific vulnerabilities to KServe integration with automated drift detection, ModelKits transform every stage of the ML lifecycle from fragile coordination into reliable operations.

Next Steps

Start with a single AI/ML project. Package it as a ModelKit. Deploy it to a test Kubernetes cluster. Experience the difference between managing versioned project state versus manually coordinating model files, code, and configuration.

Get started:

- Install Kit CLI: https://github.com/jozu-ai/kitops
- Review example Kitfiles in the repository
- Join the KitOps community on Discord for technical support
- Learn more about Jozu: jozu.com
- Schedule a Jozu POV: https://jozu.com/fast-and-secure/
- Get enterprise support for KitOps: https://jozu.com/kitops-modelpack-support/