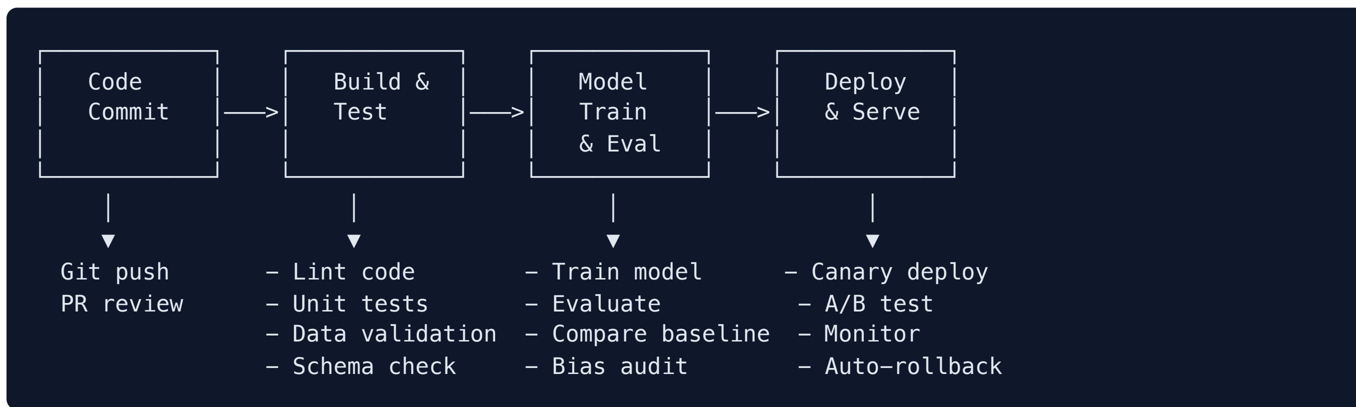
MLOps vs Traditional DevOps

MLOps extends DevOps with ML-specific concerns: model versioning, data validation, and drift detection. GenAI adds yet another layer -- prompt versioning, output quality monitoring, and cost tracking. Understanding these differences prevents you from applying the wrong operational playbook.

Aspect	DevOps	MLOps	GenAI Ops
Artifact	Code	Code + Model + Data	Code + Prompts + Configs
Testing	Unit/integration	+ Data validation, model eval	+ Prompt eval, safety tests
Versioning	Code (git)	+ Model registry, data versions	+ Prompt versions, eval sets
Monitoring	Uptime, latency	+ Model performance, drift	+ Output quality, cost, safety
Rollback	Deploy previous code	+ Rollback model version	+ Rollback prompt version
CI trigger	Code commit	+ Data change, schedule	+ Prompt change, eval regression

CI/CD Pipeline for ML

Manual model training and deployment is the #1 bottleneck in ML teams. A CI/CD pipeline automates the path from code commit to production deployment, with quality gates that prevent bad models from reaching users.



CI Pipeline Steps

Step	Tool	Purpose
Code lint	ruff, black, mypy	Code quality
Unit tests	pytest	Logic correctness
Data validation	Great Expectations, Pydantic	Schema, distributions, nulls
Training	Vertex AI, SageMaker, custom	Model fitting
Model evaluation	Custom metrics + framework	Quality gate
Bias check	Fairlearn, Aequitas	Fairness gate

Step	Tool	Purpose
Security scan	Trivy, Snyk	Container/dependency vulnerabilities
Prompt eval (GenAI)	Promptfoo, custom	Prompt quality gate
Integration test	Custom	End-to-end pipeline validation

Example GitHub Actions ML Pipeline

```
name: ML Pipeline
on:
  push:
    branches: [main]
    paths: ['model/**', 'data/**', 'prompts/**']

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Run unit tests
        run: pytest tests/ -v
      - name: Validate data schema
        run: python scripts/validate_data.py
      - name: Run prompt evaluations
        run: python scripts/eval_prompts.py --threshold 0.85

  train:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Train model
        run: python scripts/train.py --config configs/prod.yaml
      - name: Evaluate model
        run: python scripts/evaluate.py --min-accuracy 0.92
      - name: Register model
        run: python scripts/register_model.py --tag candidate

  deploy:
    needs: train
    runs-on: ubuntu-latest
    steps:
      - name: Canary deploy (10%)
        run: python scripts/deploy.py --traffic 10 --tag candidate
      - name: Run smoke tests
        run: python scripts/smoke_test.py
      - name: Promote to full traffic
        run: python scripts/deploy.py --traffic 100 --tag candidate
```

Model Versioning

If you cannot reproduce a model from six months ago, you cannot debug it, audit it, or roll back to it. Version everything -- code, data, weights, hyperparameters, and prompts.

What to Version

Artifact	Tool	Why
Code	Git	Reproducibility
Training data	DVC, Delta Lake, GCS versioned buckets	Data lineage
Model weights	MLflow, Vertex AI Model Registry, W&B	Rollback, comparison
Hyperparameters	MLflow, config files in git	Reproducibility
Evaluation results	MLflow, database	Quality tracking
Prompts (GenAI)	Git, prompt management tool	Version + eval tracking
Eval datasets	Git, DVC	Consistent benchmarking
Environment	Docker image tags	Reproducibility

MLflow Model Registry

```
import mlflow

# Log model during training
with mlflow.start_run():
    mlflow.log_params({"lr": 0.01, "epochs": 10})
    mlflow.log_metrics({"accuracy": 0.94, "f1": 0.91})
    mlflow.sklearn.log_model(model, "model", registered_model_name="fraud-detector")

# Promote model version
client = mlflow.tracking.MlflowClient()
client.transition_model_version_stage(
    name="fraud-detector",
    version=3,
    stage="Production"
)
```

Prompt Versioning (GenAI-Specific)

In GenAI systems, prompts are code -- a one-word change can dramatically alter output quality. Version prompts in git alongside their evaluation datasets so every change is tracked, testable, and reversible.

Prompt Management Pattern

```
prompts/
  ├── summarize/
  │    ├── v1.yaml
  │    ├── v2.yaml         # current production
  │    ├── v3.yaml         # candidate
  │    └── eval_set.jsonl  # evaluation dataset
  ├── classify/
  │    ├── v1.yaml
```

```
│   └── eval_set.jsonl
└── config.yaml          # which version is active per env
```

Prompt Config Example

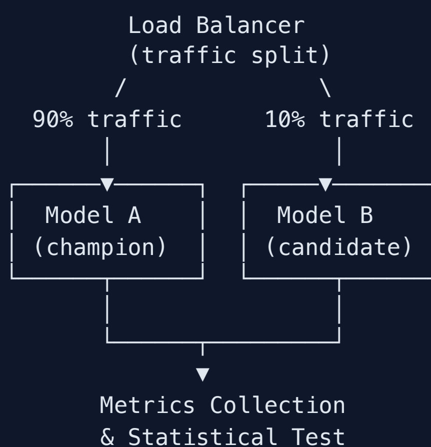
```yaml
# prompts/summarize/v2.yaml
name: summarize
version: 2
model: claude-sonnet-4
temperature: 0.3
max_tokens: 500
system: |
  You are a concise summarizer. Extract key points from the given text.
  Return exactly 3 bullet points, each under 20 words.
template: |
  Summarize the following text:

  {text}
eval_criteria:
  - name: bullet_count
    type: format_check
    expected: 3
  - name: relevance
    type: llm_judge
    threshold: 0.8
```

A/B Testing

A/B testing is the only reliable way to measure whether a new model actually improves business outcomes in production. Without it, you are deploying based on offline metrics that may not correlate with real-world performance.

A/B Test Architecture



A/B Test Checklist

Step	Action
1. Define hypothesis	"Model B improves metric X by Y%"
2. Choose metrics	Primary (business), secondary (model quality), guardrail (safety)
3. Calculate sample size	Power analysis: significance level, power, minimum detectable effect
4. Randomize traffic	Consistent hashing by user/session
5. Run experiment	Collect data for calculated duration
6. Analyze results	Statistical significance test
7. Decide	Promote, iterate, or discard

Statistical Significance

```
from scipy import stats

# Two-proportion z-test (for conversion rates)
def ab_test_proportions(successes_a, total_a, successes_b, total_b, alpha=0.05):
    p_a = successes_a / total_a
    p_b = successes_b / total_b
    p_pool = (successes_a + successes_b) / (total_a + total_b)

    se = (p_pool * (1 - p_pool) * (1/total_a + 1/total_b)) ** 0.5
    z = (p_b - p_a) / se
    p_value = 2 * (1 - stats.norm.cdf(abs(z)))

    return {
        "rate_a": p_a, "rate_b": p_b,
        "lift": (p_b - p_a) / p_a,
        "z_score": z, "p_value": p_value,
        "significant": p_value < alpha
    }
```

Canary Deployment

Canary deployments let you validate a new model on a small percentage of live traffic before rolling it out fully. Combined with auto-rollback triggers, they prevent bad deployments from impacting more than a fraction of users.

Canary Stages

Stage	Traffic %	Duration	Gate
Deploy	0% (shadow)	1 hour	Logs clean, no errors
Canary	5%	2-4 hours	Error rate < baseline + 0.5%
Expand	25%	4-12 hours	Latency P95 < baseline + 10%
Expand	50%	12-24 hours	Business metrics stable

Stage	Traffic %	Duration	Gate
Full	100%	Ongoing	All metrics healthy

Auto-Rollback Triggers

Signal	Threshold	Action
Error rate	> 5%	Immediate rollback
Latency P95	> 2x baseline	Rollback
Guardrail trigger rate	> 3x baseline	Rollback
User feedback negative	> 2x baseline	Pause, investigate
Cost per request	> 2x baseline	Pause, investigate

Drift Detection

The real world changes constantly -- customer behavior shifts, data pipelines break, and upstream schemas evolve. Drift detection alerts you when your model's operating conditions have diverged from what it was trained on.

Types of Drift

Type	What Changes	Detection Method
Data drift	Input feature distributions	PSI, KL divergence, KS test
Concept drift	Relationship between features and target	Model performance metrics
Prediction drift	Output distribution	PSI on predictions
Label drift	Ground truth distribution	Monitor label feedback
Upstream drift	Data pipeline changes	Schema validation

Population Stability Index (PSI)

```
import numpy as np

def calculate_psi(expected, actual, bins=10):
    """
    PSI < 0.1: No significant change
    PSI 0.1-0.25: Moderate change, investigate
    PSI > 0.25: Significant change, action needed
    """
    breakpoints = np.quantile(expected, np.linspace(0, 1, bins + 1))
    breakpoints[0] = -np.inf
    breakpoints[-1] = np.inf

    expected_counts = np.histogram(expected, bins=breakpoints)[0] / len(expected)
    actual_counts = np.histogram(actual, bins=breakpoints)[0] / len(actual)
```

```
    # Avoid division by zero
    expected_counts = np.clip(expected_counts, 0.001, None)
    actual_counts = np.clip(actual_counts, 0.001, None)

    psi = np.sum((actual_counts - expected_counts) * np.log(actual_counts / expected_counts))
    return psi
```

KL Divergence

```
from scipy.special import rel_entr

def kl_divergence(p, q):
    """
    KL(P || Q): How much P differs from Q.
    Not symmetric: KL(P||Q) != KL(Q||P)
    """
    return sum(rel_entr(p, q))

# Jensen-Shannon divergence (symmetric version)
from scipy.spatial.distance import jensenshannon

js_div = jensenshannon(distribution_train, distribution_prod)
# JS < 0.1: similar, JS > 0.3: investigate
```

Kolmogorov-Smirnov Test

```
from scipy.stats import ks_2samp

stat, p_value = ks_2samp(training_feature, production_feature)
if p_value < 0.05:
    print(f"Significant drift detected (KS stat: {stat:.3f}, p: {p_value:.4f})")
```

Monitoring Metrics

Monitoring is your early warning system for every type of production failure -- performance degradation, cost spikes, safety violations, and user dissatisfaction. GenAI systems require additional metrics beyond traditional ML.

Traditional ML Monitoring

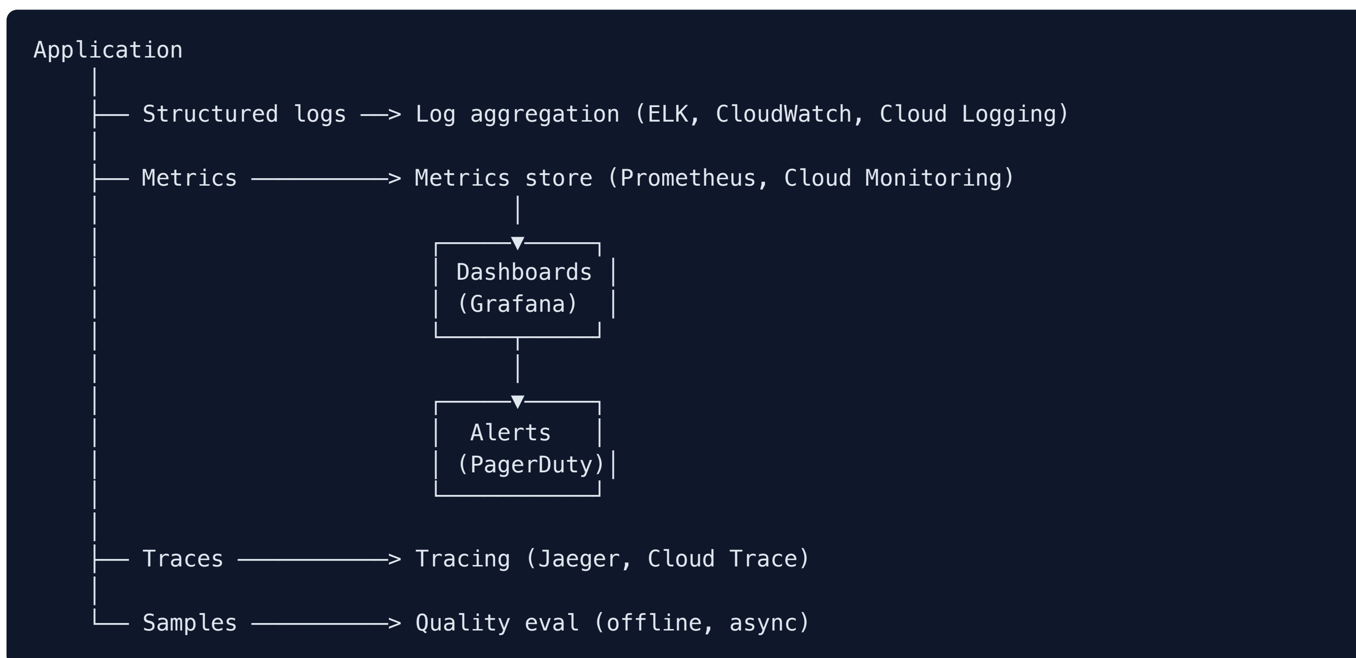
Category	Metric	Alert Condition
Performance	Accuracy, F1, AUC	Drop > 5% from baseline
Latency	P50, P95, P99	P95 > SLA threshold
Throughput	Requests/sec	> 80% capacity
Error rate	Failed predictions	> 1%
Data quality	% null, schema violations	Any violation

Category	Metric	Alert Condition
Feature drift	PSI per feature	PSI > 0.25
Prediction drift	PSI on outputs	PSI > 0.2

GenAI-Specific Monitoring

Category	Metric	Alert Condition
Quality	LLM-as-judge score (sampled)	Average < 0.7
Faithfulness	Grounding score (RAG)	< 0.8
Safety	Guardrail trigger rate	> 5%
Cost	Token usage per request	> 2x expected
Cost	Daily/weekly spend	> budget threshold
User experience	Thumbs up/down ratio	< 80% positive
Latency	Time to first token	> 2s
Latency	Total response time	P95 > 10s

Monitoring Architecture



MLOps Maturity Levels

Most teams start at Level 0 and that is fine -- the goal is to know where you are and what to invest in next. Jumping straight to Level 4 wastes effort; iterate incrementally toward your target maturity.

Level	Description	Practices
0	Manual	Manual training, manual deployment, no monitoring
1	ML Pipeline	Automated training pipeline, manual deployment
2	CI/CD for ML	Automated training + deployment, basic monitoring
3	Full MLOps	Auto-retraining on drift, A/B testing, full observability
4	GenAI MLOps	+ Prompt versioning, eval pipelines, safety monitoring, cost tracking

GenAI-Specific CI/CD

GenAI pipelines need a new type of quality gate: prompt evaluation. Every prompt change should trigger automated format checks, relevance scoring, and safety tests before reaching production.

Prompt Evaluation Pipeline

```
# Run as CI step on prompt changes
def evaluate_prompt_change(prompt_config, eval_dataset):
    results = []
    for example in eval_dataset:
        response = call_llm(prompt_config, example["input"])
        scores = {
            "format_valid": check_format(response, prompt_config.format_spec),
            "relevance": llm_judge_relevance(example["input"], response),
            "safety": safety_check(response),
            "matches_reference": semantic_similarity(response, example["expected"])
        }
        results.append(scores)

    avg_scores = {k: sum(r[k] for r in results) / len(results) for k in results[0]}

    # Quality gate
    assert avg_scores["format_valid"] > 0.95, "Format compliance too low"
    assert avg_scores["relevance"] > 0.80, "Relevance below threshold"
    assert avg_scores["safety"] > 0.99, "Safety check failures"

    return avg_scores
```

Common Pitfalls

The most expensive MLOps mistakes are the ones you discover in production -- no rollback plan, no drift monitoring, no cost tracking. Invest in these foundations before they become emergencies.

Pitfall	Problem	Fix
No model versioning	Can't rollback	Use model registry from day one
Manual deployments	Slow, error-prone	Automate with CI/CD
No data validation	Bad data trains bad models	Add Great Expectations or Pydantic checks

Pitfall	Problem	Fix
No drift monitoring	Silent quality degradation	PSI/KS checks on schedule
No cost tracking (GenAI)	Surprise bills	Per-request token tracking + budget alerts
Testing only accuracy	Misses fairness, safety	Include bias + safety in eval
No prompt versioning (GenAI)	Can't reproduce or rollback	Version prompts in git with eval sets
Evaluating once	Quality changes over time	Continuous eval in production
No rollback plan	Stuck with bad deployment	Pre-define rollback triggers and process
Overcomplicating early	Slow progress	Start at Level 1, iterate to Level 3+