MANTECH Publications

Journal of Software Engineering & Software Testing
Volume 11, Issue 1, January-April 2026
ISSN: 2457-0516 (Online)

Software Engineering for Machine Learning Systems (ML-SE)

Kavita Menon[1], Arjun Prakash[2], Meera Thomas[3], Surender Chatarjee[4]

Assistant Professor, Associate Professor

Department of CSE

Vishwakarma Institute of Technology

Email ID: *Kavitadmenonk2@yahoo.com[1], prakasharjun5y@gmail.com[2], thomas00m@rediffmail.com[3]*

DOI: *https://doi.org/10.5281/zenodo.19641967*

ABSTRACT

Machine Learning (ML) systems are increasingly integrated into critical domains such as healthcare, finance, transportation, and public services. However, traditional software engineering principles are often insufficient to manage the complexity, uncertainty, and data-dependency inherent in ML-based systems. Software Engineering for Machine Learning Systems (ML-SE) has emerged as an interdisciplinary approach that integrates classical engineering practices with data-centric development processes. This paper reviews the foundations, lifecycle models, challenges, tools, quality attributes, and best practices associated with ML-SE. It discusses how ML systems differ from conventional software, highlights the need for reproducibility, data versioning, monitoring, and continuous retraining, and examines frameworks supporting MLOps. The study also addresses ethical, reliability, scalability, and maintainability concerns. Finally, the paper outlines research directions to strengthen ML-SE methodologies.

KEYWORDS: *Machine Learning Systems, Software Engineering, MLOps, Model Lifecycle, Data Versioning, Continuous Integration, AI Deployment*

INTRODUCTION

The rapid advancement of machine learning technologies has changed the way software systems are designed and deployed. From recommendation engines to autonomous systems, ML components are now embedded into mainstream software applications. Organizations such

as Google, Microsoft, and Amazon heavily rely on ML-driven architectures to enhance user experiences and optimize operations.

Traditional software engineering focuses on deterministic logic where requirements are translated into code with predictable outputs. In contrast, ML systems learn patterns from data and produce probabilistic outputs. The behavior of an ML system depends not only on code but also on training data, model parameters, and environmental conditions. Because of this, conventional development methodologies must be extended or adapted.

Software Engineering for Machine Learning Systems (ML-SE) seeks to address these differences. It integrates software engineering best practices—such as requirement analysis, version control, testing, and maintenance—with data science workflows. Without proper engineering discipline, ML systems often suffer from issues such as data drift, model degradation, reproducibility challenges, and deployment failures.

This paper aims to present a comprehensive review of ML-SE, exploring lifecycle stages, challenges, engineering frameworks, tools, and research opportunities.

DIFFERENCES BETWEEN TRADITIONAL SOFTWARE AND ML SYSTEMS

Machine Learning (ML) systems differ fundamentally from traditional software systems in terms of development philosophy, behavior, validation, and maintenance. While conventional systems rely on explicitly programmed logic, ML systems depend on statistical learning from data. This shift from rule-based logic to data-driven modeling creates significant engineering implications.

Organizations such as Google and Microsoft have reported that maintaining ML-based products requires more effort in data engineering and monitoring than in core algorithm implementation. This itself shows that ML systems are not just “another software module,” but a different paradigm of system design.

The following sections elaborate key differences in depth.

1. Nature of Logic: Explicit Programming vs Learned Behavior

In traditional software development, engineers write explicit rules that determine system behavior. For example, a payroll system calculates salary based on clearly defined formulas. If

the logic needs modification, developers update the code.

In contrast, ML systems do not contain fully predefined logic. Instead, developers select algorithms and train them using datasets. The internal decision-making rules are derived automatically from data patterns. For instance, in an image classification system built using frameworks like TensorFlow or PyTorch, the model learns features from thousands of images rather than relying on manually coded rules.

This learned behavior introduces uncertainty because the model's reasoning is often not transparent and may change when retrained on new data.

2. Deterministic vs Probabilistic Outputs

Traditional systems produce deterministic outputs. Given the same input and system state, the output will always be identical. For example, a sorting algorithm will consistently return the same ordered list for a given input.

ML systems, however, produce probabilistic outputs. A classifier may output a probability score such as “80% likelihood of fraud.” Additionally, retraining the same model with slightly different data or initialization seeds may produce slightly different results.

This probabilistic nature makes debugging and validation more complex. Engineers must think in terms of statistical confidence rather than absolute correctness.

3. Data-Centric vs Code-Centric Development

Traditional software is primarily code-centric. Once requirements are defined, development revolves around implementing and optimizing code logic.

ML systems are data-centric. The quality, quantity, and representativeness of training data often have greater impact than algorithm selection. Even a sophisticated algorithm will perform poorly if the dataset is biased or incomplete.

This means that data pipelines, labeling processes, and feature engineering become core engineering activities. In many ML projects, data preparation consumes more than half of total development time.

4. Testing and Validation Approaches

Testing traditional software involves unit testing, integration testing, and system testing. Engineers verify correctness by checking if outputs match expected results.

Testing ML systems is more complex. Since outputs are probabilistic, engineers evaluate performance using metrics such as:

- Accuracy
- Precision
- Recall
- F1-score
- ROC-AUC

Rather than checking a single output, validation involves analyzing performance across entire datasets. Additionally, fairness testing and bias detection are necessary in sensitive applications.

Unlike conventional bugs, ML failures may arise from data distribution shifts rather than code errors.

5. Debugging Complexity

Debugging traditional software often involves tracing execution paths, inspecting variable values, and identifying faulty logic. Tools and debuggers make this relatively straightforward.

In ML systems, debugging may involve:

- Investigating mislabeled data
- Analyzing feature importance
- Checking training instability
- Identifying overfitting or underfitting

Because models operate as high-dimensional mathematical functions, interpreting why a model made a specific decision can be difficult. Explainability techniques are often required.

.***Table 1: Comparison between Traditional Software and ML-Based Systems***

Aspect	Traditional Software	ML-Based Systems
Logic	Explicitly coded rules	Learned from data
Output	Deterministic	Probabilistic
Testing	Unit and integration testing	Statistical validation and performance metrics
Maintenance	Code updates	Model retraining and data updates
Debugging	Code tracing	Data and model analysis

In traditional systems, developers write clear instructions that define program behavior. In ML systems, developers design algorithms that learn from data, often using libraries such as TensorFlow and PyTorch. The unpredictability of data introduces additional complexity.
Another major difference is that ML systems degrade over time due to data distribution shifts. Therefore, continuous monitoring and retraining are essential components of ML-SE.

LIFECYCLE OF ML SYSTEMS

The lifecycle of Machine Learning (ML) systems is significantly more iterative, experimental, and data-intensive compared to classical software development. In traditional systems, development generally moves from requirement gathering to coding, testing, deployment, and maintenance in a relatively structured manner. In ML systems, however, the lifecycle includes continuous feedback loops between data, models, and deployment environments.

Unlike deterministic software, ML systems evolve over time because data changes, user behavior shifts, and real-world conditions are dynamic. Therefore, the ML lifecycle is not linear; it is cyclical and adaptive.

1. Requirement Analysis

Requirement analysis in ML systems includes both **functional** and **non-functional** requirements, but defining them is often more challenging than in traditional software projects.

Functional Requirements

Functional requirements describe what the ML system is expected to do. Examples include:

- Classify emails as spam or not spam

- Predict customer churn probability
- Detect fraudulent transactions
- Recommend products to users

In ML projects, these tasks are typically framed as classification, regression, clustering, or ranking problems. However, unlike traditional systems where requirements are precise and measurable from the beginning, ML systems involve uncertainty.

For example, a stakeholder may request "90% accuracy," but the actual achievable accuracy depends on:

- Data quality
- Feature representation
- Model complexity
- Noise and bias in data

Thus, requirement analysis in ML systems often begins with feasibility studies and pilot experiments. Teams may need to perform exploratory data analysis (EDA) before confirming whether the objective is realistic.

Non-Functional Requirements

Non-functional requirements are even more critical in ML systems. These include:

- **Fairness:** The model should not discriminate against specific groups.
- **Interpretability:** The system should provide explainable outputs when needed.
- **Scalability:** It should handle large volumes of data.
- **Latency:** Predictions must be delivered within acceptable time limits.
- **Security:** The model should resist adversarial attacks.

For example, a fraud detection model may need to respond within milliseconds to prevent financial loss. Therefore, performance constraints must be considered from the design stage.

In ML systems, requirement analysis is iterative. Initial assumptions are frequently refined after early experiments reveal practical limitations.

2. Data Collection and Preparation

Data is the foundation of any ML system. Unlike traditional software, where logic determines behavior, in ML systems, data largely determines behavior.

Data Collection

Data can come from multiple sources:

- Databases
- Sensors
- APIs
- User logs
- Surveys

In industrial environments, companies such as Amazon collect large-scale user interaction data to improve recommendation systems. However, raw data is rarely clean or directly usable.

Data Cleaning

Data cleaning involves:

- Removing duplicates
- Handling missing values
- Correcting inconsistencies
- Eliminating outliers

If these issues are ignored, the model may learn incorrect patterns. For example, biased training data can result in biased predictions.

Data Labeling

Supervised learning requires labeled data. Labeling is often expensive and time-consuming. In some domains like healthcare, expert annotation is required, increasing costs significantly.

Data Transformation and Feature Engineering

Feature engineering converts raw data into meaningful representations. Examples include:

- Encoding categorical variables
- Normalizing numerical features
- Generating derived features

Feature engineering directly impacts model performance. In many real-world projects, feature engineering consumes more effort than model selection.

Data Versioning and Governance

Since datasets evolve over time, version control for data is essential. Each model must be linked to a specific dataset version to ensure reproducibility.

Additionally, organizations must comply with privacy regulations. Data governance policies ensure that sensitive data is handled securely and ethically.

3. Model Development

Model development is often considered the "core" ML activity, but in practice it is only one part of the lifecycle.

Algorithm Selection

Engineers experiment with different algorithms depending on the problem type. For traditional ML tasks, libraries such as Scikit-learn provide implementations of algorithms like:

- Logistic regression
- Decision trees
- Support vector machines
- Random forests

For deep learning tasks, frameworks like TensorFlow and PyTorch are widely used.

Hyperparameter Tuning

Hyperparameters control model behavior, such as learning rate or tree depth. Engineers perform experiments to identify optimal configurations.

Techniques include:

- Grid search
- Random search
- Bayesian optimization

Hyperparameter tuning is computationally expensive and requires systematic experiment tracking.

Experiment Tracking and Reproducibility

Reproducibility is a critical concern. Minor differences in environment, random seeds, or data splits can produce different outcomes.

Therefore, teams use experiment tracking tools to record:

- Dataset version
- Hyperparameters
- Model architecture
- Performance metrics
- Hardware configuration

Without proper tracking, reproducing results becomes nearly impossible, especially in large teams.

4. Testing and Validation

Testing ML systems differs significantly from testing traditional software. Instead of verifying exact outputs, engineers evaluate statistical performance.

Performance Metrics

Common evaluation metrics include:

- **Accuracy:** Percentage of correct predictions
- **Precision:** Proportion of true positives among predicted positives
- **Recall:** Proportion of true positives among actual positives
- **F1-score:** Harmonic mean of precision and recall
- **ROC-AUC:** Trade-off between sensitivity and specificity

Different applications require different metrics. For example, in medical diagnosis, recall may be more important than accuracy to avoid missing positive cases.

Cross-Validation

Cross-validation ensures that model performance generalizes beyond the training dataset. It reduces the risk of overfitting.

Robustness Testing

Models must be tested against:

- Noisy inputs
- Adversarial attacks
- Edge cases

Even high-performing models can fail in unexpected situations. For instance, a model trained on urban data may perform poorly in rural settings.

Bias and Fairness Evaluation

Bias testing ensures that predictions are equitable across demographic groups. This step is increasingly important in regulated industries.

5. Deployment

Deployment transitions the model from experimentation to production use. This stage introduces operational challenges not present during development.

Packaging and Serving

Models are often exposed as REST APIs or integrated into web/mobile applications.
Containerization tools like Docker ensure consistent runtime environments. Orchestration systems such as Kubernetes manage scaling and fault tolerance.

CI/CD and MLOps

Traditional Continuous Integration/Continuous Deployment (CI/CD) pipelines are extended into MLOps pipelines. These pipelines automate:

- Model training
- Testing
- Validation
- Deployment

Unlike conventional software releases, ML deployments may occur frequently due to retraining requirements.

Infrastructure Considerations

Deployment environments may require specialized hardware such as GPUs. Additionally, prediction latency must meet business constraints.

6. Monitoring and Maintenance

Once deployed, ML systems require continuous monitoring. Unlike static software, model performance can degrade over time due to environmental changes.

Performance Monitoring

Key metrics are tracked in real time:

- Prediction accuracy
- Error rates
- Response time
- Throughput

Alerts are triggered when performance drops below predefined thresholds.

Data Drift Detection

Data drift occurs when the distribution of incoming data differs from the training data. This can significantly reduce model performance. Automated drift detection mechanisms compare statistical properties over time.

Retraining Pipelines

When performance declines, retraining pipelines are activated. Retraining may occur:

- On a fixed schedule
- When drift is detected
- After significant business changes

Retrained models undergo validation before redeployment.

Model Retirement

In some cases, models must be retired due to regulatory updates or technological advancements. Proper documentation ensures smooth transitions.

MLOPS AND ENGINEERING PRACTICES

MLOps extends DevOps principles to ML workflows. It integrates automation, monitoring, and collaboration across teams.

1. CI/CD for ML

In ML projects, CI/CD pipelines must handle code, data, and models. Version control systems manage both source code and model artifacts. Automated testing includes validation datasets and performance thresholds.

2. Model Versioning

Model versioning ensures traceability. Each version includes:

- Training dataset snapshot
- Hyperparameters
- Performance metrics
- Deployment configuration

This allows rollback if performance issues occur.

3. Experiment Tracking

Experiment tracking tools document experiments systematically. They reduce duplication and improve collaboration among data scientists.

QUALITY ATTRIBUTES OF ML SYSTEMS

ML-SE emphasizes several quality attributes:

1. **Reliability:** Stable performance under different conditions.
2. **Scalability:** Ability to handle large data volumes.
3. **Maintainability:** Easy retraining and updates.
4. **Security:** Protection against adversarial attacks.
5. **Ethical Compliance:** Fairness and transparency.

Security concerns include adversarial inputs that manipulate model outputs. Engineering defenses must be integrated early in the design stage.

CHALLENGES IN ML-SE

1. Data Dependency

Model performance heavily depends on training data. Inconsistent data pipelines can break production systems.

2. Reproducibility Issues

Results may differ due to randomness, hardware differences, or library updates. Ensuring reproducibility requires strict documentation and environment control.

3. Technical Debt

ML projects accumulate hidden technical debt, especially when quick experiments are deployed without proper validation.

4. Ethical and Legal Risks

Bias in datasets may result in unfair decisions. Regulatory compliance is essential in domains such as healthcare and finance.

TOOLS AND FRAMEWORKS SUPPORTING ML-SE

Table 2 presents commonly used tools in ML-SE environments.

Table 2: Tools in ML-SE Ecosystem

Category	Example Tools	Purpose
ML Frameworks	TensorFlow, PyTorch	Model training
Data Processing	Apache Spark	Large-scale data processing
Containerization	Docker	Deployment
Version Control	Git	Code management
Orchestration	Kubernetes	Scalable deployment

Apache Spark enables large-scale distributed data processing. Orchestration tools such as Kubernetes manage containerized ML services in production.

RESEARCH DIRECTIONS

Future research in ML-SE may focus on:

- Automated testing frameworks for ML
- Standardized lifecycle models

- Explainability integration
- Robustness verification methods
- Green AI and energy-efficient training

As ML systems expand into safety-critical domains like autonomous vehicles, engineering rigor becomes even more important.

CASE ILLUSTRATION

Consider a predictive healthcare model deployed in hospitals. The system predicts patient readmission risk. Initially, performance metrics are high. However, over time, new patient demographics change the data distribution. Without monitoring and retraining, accuracy declines.

This example shows why ML-SE practices—monitoring, retraining pipelines, and version control—are not optional but necessary.

CONCLUSION

Software Engineering for Machine Learning Systems (ML-SE) represents a critical evolution in modern computing. Traditional engineering principles alone cannot address the probabilistic and data-driven nature of ML systems. By integrating structured lifecycle models, MLOps practices, quality assurance techniques, and monitoring frameworks, organizations can build robust and scalable ML applications.

Despite progress, several challenges remain including reproducibility, data governance, fairness, and technical debt management. Future work should focus on creating standardized methodologies and automated validation systems to ensure reliability and ethical compliance.
In conclusion, ML-SE bridges the gap between experimental data science and dependable production systems. With proper engineering discipline, machine learning solutions can achieve long-term sustainability and societal trust.

REFERENCES

1. Amershi, S., et al. (2019). Software engineering for machine learning: A case study. IEEE Software.

2. Sculley, D., et al. (2015). Hidden technical debt in machine learning systems. NIPS.
3. Breck, E., et al. (2017). The ML test score: A rubric for ML production readiness.
4. Polyzotis, N., et al. (2018). Data validation for machine learning.
5. Zaharia, M., et al. (2018). Accelerating the machine learning lifecycle with MLflow.
6. Chen, J., et al. (2020). Continuous training in ML systems.
7. Rahman, A., et al. (2021). Reproducibility challenges in ML pipelines.
8. Mitchell, M., et al. (2019). Model cards for model reporting.
9. Zhang, H., et al. (2022). Monitoring ML systems in production.
10. Nguyen, T., et al. (2023). Advances in MLOps engineering practices.

Cite as:

Kavita Menon, Arjun Prakash, Meera Thomas, Surender Chatarjee. (2026). Software Engineering for Machine Learning Systems (ML-SE). Journal of Software Engineering & Software Testing, 11(1), 59-73.

https://doi.org/10.5281/zenodo.19641967