POLITECNICO DI TORINO

Master's Degree Course
in Computer Engineering

Master Degree Thesis

ML-Driven Log Analysis: A Practical Guide for Proactive Software Reliability Improvement.

Supervisor
prof. Marco Torchiano

Candidate
Gabriel Youbissi Kamdem

Academic Year 2025-2026

Acknowledgements

I would like to begin by expressing my sincere gratitude to Professor Marco Torchiano for accepting to be the supervisor of my thesis.
My journey at Politecnico di Torino was marked by moments of doubt, isolation, and fear. Yet, having my love and fiancée, Virginie Dsemta Wadji, by my side helped me stay focused. She gave me constant reassurance and support, always reminding me of what truly matters.
I am also deeply grateful for the brotherhood this journey brought me. I would like to start with Borel Ngahou, because without him welcoming me to Italy, this experience could have started as a nightmare. My thanks also go to Tchouta Duvier Dzeubou, whose morality and loyalty inspired me greatly, Christian Peyou Tessa, my computer science colleague since the very beginning, Wilfried Moussongo whom I like to call Matrix, and last but not least Jospin Tcheuffa Randolf, without whom this journey might never have begun. All of them, along with the many friends I probably forgot to add in this section, made my time at Politecnico di Torino truly meaningful.
I also feel fortunate to have encountered remarkable professors who not only guided me academically but also nurtured and strengthened my intellectual curiosity.
Finally, I cannot write this section without honoring the true heroes of my life, my brothers and sisters: Kenmogne Kamdem Celine Gloria, Kamhoua Kamdem Brice, Dinko Kamdem Raïssa, Nono Kamdem Grace, and Kountchou Maliedje Kamdem Pierre Kevin. Even though we were separated by thousands of kilometers, they never stopped thinking of me, caring for me, and encouraging me, thereby strengthening my determination to obtain this Master's degree.
A huge and heartfelt thank you to my beloved parents, Kamdem Jean Pierre and Demgne Bernadette Dorothée. I want to state this with deep sincerity: this Master's degree is also yours.

Summary

This thesis aims to provide a structured and practical guide to facilitate the integration of Machine Learning (ML) techniques into log analysis processes, with the ultimate goal of improving software reliability. In order to achieve this, the work first presents an overview of Machine Learning, introducing its fundamental concepts and the principal approaches that are relevant to log-based analysis.
Although the proposed guide is designed to be tool-agnostic, the thesis includes a concise discussion motivating the adoption of outsourced solutions for ML-driven log analysis. This discussion highlights their advantages in terms of scalability, reduced operational overhead, and ease of integration within existing information systems.
To demonstrate the practical applicability of the proposed guide, the Splunk platform is employed as a reference implementation. Accordingly, the thesis introduces the core Splunk concepts and components necessary for understanding and implementing the proposed workflow.
The resulting guide is structured as a workflow composed of the following steps:

- Problem formalization
- Data extraction
- Data preprocessing
- Model selection
- Model training and evaluation
- Alert creation and scheduling
- Take action
- Get Feedback and replicate

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context and motivation

Modern information systems form the backbone of critical services across nearly every industry, including banking, healthcare, and telecommunications. These systems generate massive volumes of log data, providing detailed records of operations, user interactions, and performance metrics. Traditionally, logs are analyzed reactively: engineers examine them only after failures occur or when users report service degradation. While effective for diagnosing known issues, reactive analysis is insufficient. This is primarily due to the sheer volume of logs and the growing complexity of modern environments, where microservices, cloud platforms, and interconnected APIs create intricate dependencies. In such environments, minor anomalies can rapidly propagate and lead to cascading failures. Static, rule-based monitoring approaches such as generating alerts when response times exceed a fixed threshold or a specific error code appears often fall short and could lead to alert fatigue because of the lack of pertinence of those alerts and the fact that they do not trigger any action. Their incapacity to detect unknown patterns that precede or indicate an error results in operational inefficiencies, which reduce software reliability, compromise system integrity, and erode user trust.

Consequently, there is a pressing need to move beyond reactive log monitoring toward intelligent, proactive maintenance. By applying Machine Learning (ML) to log data, organizations can detect subtle patterns, predict potential incidents, and significantly enhance monitoring capability. Proactive log analysis not only boosts system reliability but also enables IT teams to allocate resources more efficiently, allowing them to focus on higher-value tasks rather than manual troubleshooting.

The central challenge, then, is to design an effective system that transforms raw log data into structured features suitable for machine learning. This system must enable automated, proactive monitoring facilities capable of anomaly detection, failure prediction, extraction of failure-related features, and seamless integration with operational workflows, such as ticketing and alerting systems.

This work aims to guide readers and IT teams through the process of designing and evaluating an ML-driven workflow for proactive log analysis to enhance system reliability.

For this purpose, we will leverage the Splunk Platform, focusing on its Machine Learning Toolkit and Alert capabilities with the aim to:

- automate anomaly detecion and the extraction of features highly predictive of failures
- improve incident response by providing actionable alert, timely alert
- reduce dependence on reactive, manual log analysis
- integrate seamlessly with issue-traking platforms, such as Jira, to streamline remediation workflows or mail.

This thesis consists of the following chapters:

- Chapter 2 presents the background and related work necessary to understand the context of this work, including an overview of machine learning techniques and the Splunk platform.
- Chapter 3 introduces the proposed methodological guide, describing each phase of the machine learning workflow applied to analytics.
- Chapter 4 illustrates the guide through a detailed step-by-step explanation of its components.
- Chapter 5 applies the proposed methodology to a use case, describing the problem formulation, data extraction and preprocessing, model training and evaluation, and alerting strategy.
- Chapter 6 concludes the thesis and discusses possible future developments.

Chapter 2

Background and related work

Logs represent a primary source of information for monitoring software systems. In large-scale information systems, such as banking ecosystems, that may generate billions of log entries daily. These logs contain valuable insights into application behavior, system performance, and potential software issues. Traditionally, software engineers have relied on reactive monitoring techniques, which typically involve:

- Manual log inspection, where time is allocated to search for predefined keywords based on prior knowledge
- Incident-based response, where engineers wait for failures in the production environment before attempting to diagnose and resolve the issue quickly

While these reactive methods are effective for well-understood problems, they are insufficient in today's dynamic environments.
The emergence of machine learning (ML) has created opportunities to transit from reactive to proactive log analysis. ML models can derive patterns and rules directly from historical data, enabling anomaly detection and incident prediction without relying solely on human expertise. However, the application of ML in this domain faces several barriers. Building an ML pipeline from scratch using popular Python libraries, such as scikit-learn or TensorFlow, requires specialized ML engineering and Data analysis knowledge, and increases the operation's budget for infrastructure resources and maintenance. Furthermore, such an endeavor may involve coordination across different teams, potentially creating organizational friction.
Fortunately, several commercial solutions already address this gap. Tools such as Splunk which is the platform utilized in this guide, provide scalable log collection, monitoring capabilities, and alerting mechanisms. Aligning with the AIops principle, Splunk has introduced a Machine Learning Toolkit (MLTK) into its ecosystem. This integrated approach removes many of the previously explored barriers by offering an all-in-one solution. It only requires teams to have expertise only in the application of machine learning into the Splunk ecosystem, as the ML pipeline and alerting system are embedded within the platform itself.
As claimed on the Splunk website, the company serves over 15,000 customers across more

than 110 countries and holds an estimated 26.7% share of the Security Information and Event Management (SIEM) market. Vendor reports indicate that organizations adopting Splunk Security solutions achieve returns on investment (ROI) as high as 267% by reducing business risk, improving security team efficiency, and enhancing digital landscape visibility. Financial services organizations specifically achieve a remarkable 387% ROI and $15.3 million in average annual benefits by sharpening threat detection, streamlining operations, and ensuring compliance (Source: Splunk). These benchmarks suggest that leveraging the possibilities provided by Splunk solutions should be a mandatory consideration for all institutions dependent on complex information systems. Nevertheless, it still requires investment in training software engineers to apply ML effectively to log data.

2.1 Machine learning

Machine learning (ML) is a field of study within artificial intelligence that focuses on the design, development, and analysis of algorithms capable of learning from data. Rather than relying on explicit programming instructions, ML systems extract patterns, relationships, and decision rules directly from data, enabling them to generalize to unseen instances. As a result, machine learning is widely applied to tasks such as pattern extraction, prediction, classification, and anomaly detection.
Machine learning approaches are commonly divided into three main categories: Supervised learning, Unsupervised learning, and Reinforcement learning. This work focuses on the first two approaches, which are particularly relevant in data-driven analysis scenarios such as system log-driven analysis.

2.1.1 Supervised learning

Supervised learning involves training a model using features (inputs) and their corresponding labels (outputs) extracted from historical data. The primary goal is to enable the model to accurately predict labels for new, unseen data. In the context of log analysis, this might involve labeling historical log entries a *failure*, *warning*, or *normal* behavior.
This approach generally follows three main and interconnected steps:

- Training step:
  During this phase, the model is trained on historical labeled data to learn patterns and relationships between features and outputs. Common supervised learning tasks include classification, which assigns data points to discrete classes (e.g., error logs versus normal logs), and regression, which estimates continuous numerical values (e.g., predicting system response time based on CPU usage or input size).

- Evaluation step:
  The evaluation phase measures the model's performance using standard metrics such as accuracy, precision, recall, and F1-score. This step is essential to assess whether the model has learned meaningful patterns, to compare alternative models, and to tune hyperparameters. Moreover, evaluation helps detect overfitting, a situation in

which the model performs well on training data but fails to generalize effectively to new data due to excessive sensitivity to noise or specific training instances.

- Test step:
  Finally, the trained model is validated using completely unseen data. In the case of log analysis, new log entries are classified or predicted, and their outputs are compared with real observations. Additional statistical checks, such as comparing predicted and actual distributions, can further support the assessment of the model's robustness and reliability.

Several algorithms are commonly used in supervised learning, including:

- Logistic Regression: a supervised learning algorithm used mainly for binary classification tasks, such as predicting whether a log entry indicates a failure or not. It models the probability of a class using the logistic (sigmoid) function:

$$P(C = 1 \mid X) = \frac{1}{1 + e^{-(\mathbf{w}^\top X + b)}} \tag{2.1}$$

  where X is the feature vector, $\mathbf{w}$ is the weight vector learned during training, and b is the bias term. The output represents the probability that the input belongs to the positive class.

- K-Nearest Neighbors (KNN): a non-parametric, instance-based learning algorithm that classifies a log entry based on the labels of its k closest neighbors in the feature space. Distance is commonly computed using the Euclidean metric:

$$d(x_i, x_j) = \sqrt{\sum_{l=1}^{n} (x_{il} - x_{jl})^2} \tag{2.2}$$

  The class of a new log entry is determined by majority voting among its k nearest neighbors, making KNN effective for capturing local patterns without assuming a specific data distribution.

- Support Vector Machines (SVM): An algorithm that identifies optimal separating hyperplanes between classes, well-suited for high-dimensional log feature spaces.

- Naïve Bayes: a probabilistic classifier based on Bayes' theorem, particularly efficient for text-based log data such as error messages and event descriptions. It computes the posterior probability of a class given the observed features as:

$$P(C \mid X) = \frac{P(X \mid C)\, P(C)}{P(X)} \tag{2.3}$$

  where C represents the class label (e.g., failure or normal), X denotes the feature vector extracted from a log entry, $P(C)$ is the prior probability of the class, $P(X \mid C)$ is the likelihood, and $P(X)$ is a normalization term. The *naïve* assumption considers features to be conditionally independent given the class, allowing efficient computation even for high-dimensional data.

2.1.2 Unsupervised learning

In many real-world scenarios, data is collected without predefined categories, or the process of manually extracting labels is time-consuming and costly. In such cases, unsupervised learning provides an effective solution to extract meaningful information from unlabeled data.
Unsupervised learning focuses on discovering the intrinsic structure of data by identifying hidden patterns, similarities, and groupings. Unlike supervised learning, the model does not learn from examples with known outcomes but it extracts relationships directly from the data distribution itself.
This learning paradigm is particularly valuable for exploratory data analysis, feature reduction and anomaly detection. By transforming raw and unorganized datasets into structured representations, unsupervised learning often serves as a preliminary step toward understanding complex systems and guiding further analysis.
Unsupervised learning typically involves the following steps:

- Pattern discovery (Model training):
  Algorithms analyze the dataset to identify underlying structures and regularities. Common techniques include:
  - K-Means clustering: a partition-based clustering algorithm that aims to group log entries into k clusters by minimizing the within-cluster variance. The objective function is defined as:

    $$\arg \min_{\{C_1,\ldots,C_k\}} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2 \tag{2.4}$$

    where C_i represents the i-th cluster and μ_i is its centroid. Logs that are distant from any centroid may indicate anomalous or rare system behaviors.
  - DBSCAN: a density-based clustering algorithm that groups together data points with a high density of neighbors while marking low-density points as noise. A point x is considered a core point if:

    $$|\{y \mid d(x, y) \leq \varepsilon\}| \geq \text{MinPts} \tag{2.5}$$

    where ε defines the neighborhood radius and *MinPts* is the minimum number of points required to form a dense region. This approach is effective for detecting arbitrarily shaped clusters and isolating sparse anomalies in log data.
  - Isolation Forest: an anomaly detection method based on the principle that anomalies are easier to isolate than normal data points. The anomaly score is computed as:

    $$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}} \tag{2.6}$$

    where $E(h(x))$ is the expected path length required to isolate point x in the forest, n is the number of samples, and $c(n)$ is a normalization factor. Higher scores indicate a higher likelihood of anomalous behavior.

- Interpretation and evaluation:
  Since no ground truth labels are available, evaluation relies on internal statistical measures such as the silhouette score or cluster cohesion metrics. Domain knowledge is often required to interpret the discovered patterns and assess their practical relevance.

2.2 Open and free resources for Machine Learning

The field of software engineering is constantly evolving, requiring practitioners to remain up to date with emerging tools, programming languages, and methodologies. In the context of this project, which focuses on the development of an ML-driven log analysis guide, acquiring practical knowledge in Python programming, data analysis, machine learning, and familiarity with Splunk was essential. These areas extended beyond prior academic experience and required additional self-directed learning.
Open and free educational resources proved invaluable in bridging this knowledge gap. They provided accessible, well-structured learning material that combined theoretical foundations with practical application. Among these resources, the *freeCodeCamp* platform offered comprehensive courses, tutorials, and coding exercises, enabling a progressive learning path from fundamental concepts to applied machine learning techniques. Key resources included:

- **Scientific Computing with Python**: Introduced core Python concepts such as algorithms, data structures, and object-oriented programming. This course equipped learners with the skills required to manipulate data, automate tasks, and implement computational logic efficiently.

- **Data Analysis with Python**: Covered essential data analysis libraries including NumPy, Pandas, Matplotlib, and Seaborn. These tools are widely used for data cleaning, transformation, visualization, and exploratory analysis, all of which are critical steps in preparing log data for machine learning models.

- **Machine Learning with Python**: Presented foundational machine learning concepts and demonstrated their practical implementation using frameworks such as TensorFlow and scikit-learn. This resource enabled learners to design, train, and evaluate predictive models on real-world datasets.

- **Machine Learning for Everybody (Kylie Ying)**: Provided intuitive explanations and practical live-coding examples, illustrating end-to-end machine learning workflows including data preprocessing, model training, and performance evaluation.

- **Scikit-learn documentation**: Served as an authoritative reference for machine learning algorithms, model selection, and evaluation techniques, supporting both implementation details and conceptual understanding.

- **Splunk documentation**: Offered detailed guidance on log ingestion, querying, and visualization, facilitating the integration of machine learning techniques with real-world log management and analysis tools.

Collectively, these open resources enable readers with a background in computer engineering, even from different specializations, to progressively onboard into machine learning concepts and techniques. Furthermore, they demonstrate how freely available learning material can support the practical application of ML methods in real-world scenarios such as log analysis and system monitoring.

Chapter 3

Splunk overview

Splunk is a powerful data platform designed to help organizations collect, index, analyze, and act on machine-generated data in real time. It enables solutions across multiple domains, including observability, security, IT operations, and business analytics. Splunk is most commonly used for log management, where it provides scalable and efficient tools for monitoring system behavior and extracting operational insights.
Among its key strengths, the following characteristics are particularly relevant:

- **Real-time visibility**: enables the ingestion and analysis of massive volumes of machine data with minimal latency, allowing rapid detection of events, anomalies, and failures.
- **Flexibility**: supports deployment across cloud-based, on-premises, and hybrid environments, making it adaptable to diverse infrastructure requirements.

To effectively harness the full potential of Splunk, engineers must understand its core concepts and key features, and, critically, the Splunk Processing Language (SPL). SPL provides a powerful query and transformation framework that enables users to search, filter, aggregate, and visualize log data efficiently. In the specific context of this work, familiarity with the Splunk Machine Learning Toolkit (MLTK) is also essential, as it extends SPL with machine learning capabilities that support predictive and proactive log analysis.
Readers are encouraged to deploy Splunk locally using the official Docker image in order to experiment with the platform. This approach allows hands-on exploration of Splunk features and testing with sample datasets provided by Splunk. A reference link to the Docker image is provided in the bibliography.

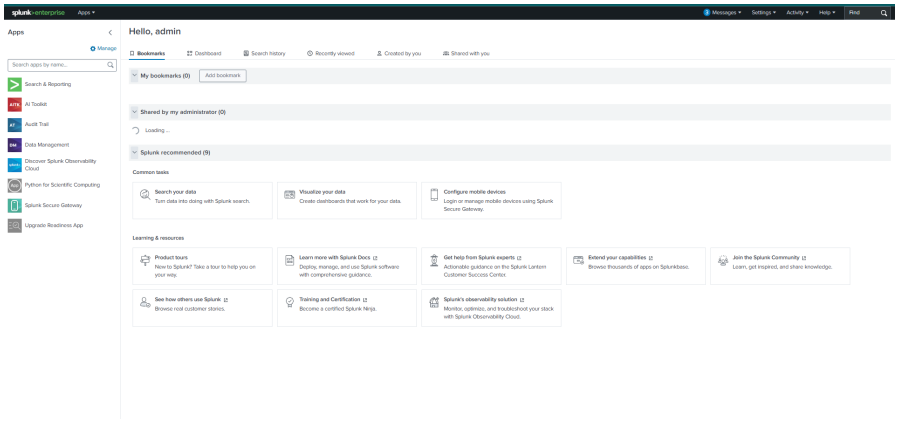


Figure 3.1. Screenshot of the Splunk homepage for a user with administrator access on a local deployment

3.1 Concept: Events

Although the Splunk documentation provides an extensive overview of platform concepts, this work focuses primarily on *events*, and more specifically on log data. These elements represent the fundamental units upon which machine learning will be applied to logs.
Within Splunk, an *event* represents a collection of values associated with a specific timestamp, capturing activity generated by machine data sources. An event may correspond to a single log line, a structured message, an entire stack trace, or any output produced by an information system, collectively referred to in this work as logs.
Each event is composed of multiple *fields*, defined by unique field names and their corresponding values. Fields may include metadata such as source, host, and severity level, as well as application-specific attributes extracted from the log content. These field-rich events form the primary data objects used throughout this work.
Consequently, the proposed framework operates directly on events and their associated fields. Examples of common operations include field extraction, data transformation, aggregation, and the application of machine learning techniques for tasks such as classification and anomaly detection.

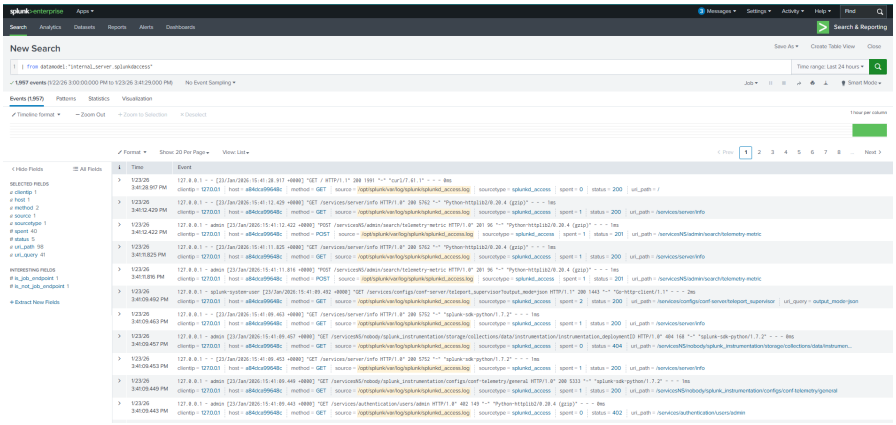


Figure 3.2. Example of log events generated by REST API calls, provided as a sample dataset within the Splunk Docker image

3.2 Features: Search and Alert

Two core features of Splunk that are essential for this work are its *Search* capability and its *Alerting* mechanisms. The search functionality enables users to query, filter, and explore indexed data using the Splunk Processing Language (SPL). SPL provides a flexible and expressive syntax for performing operations such as field extraction, aggregation, correlation, and visualization. Throughout this guide, practical examples of SPL usage are provided to progressively illustrate these capabilities.

Alerting mechanisms in Splunk are implemented through *alerts*, which are built directly on top of saved searches. By defining specific conditions, thresholds, or patterns on search results, Splunk can automatically trigger notifications or predefined actions when the specified criteria are satisfied. These actions may include sending emails, executing scripts, or integrating with external incident management systems.

Alerts play a critical role in enabling proactive monitoring. They allow organizations to detect abnormal behavior, potential failures, or security-relevant events as soon as they occur, reducing response time and improving system reliability.

Collectively, search and alerting features transform Splunk from a passive data repository into an active and dynamic platform for continuous monitoring, operational intelligence, and timely decision-making.

3.3 Splunk Processing Language (SPL)

The Splunk Processing Language (SPL) is the core query language used within Splunk, enabling direct interaction with indexed event data. It allows engineers to perform a wide range of operations, from simple keyword-based searches to complex data transformations, aggregations, and correlations. Through a rich set of commands, SPL supports filtering, grouping, and statistical computations on events, making it possible to identify patterns, trends, and anomalies within large volumes of machine-generated data.

Beyond data querying, SPL also enables the construction of dashboards, reports, and visualizations, which facilitate the communication of operational insights to both technical and non-technical stakeholders. Its flexibility and expressiveness allow engineers to adapt queries to diverse use cases, including operational monitoring, security analysis, and business intelligence. As such, SPL forms the backbone of any effective Splunk deployment and plays a central role in the implementation of advanced analytics workflows.
For practical reference and efficient usage, a link to the official Splunk documentation providing a concise cheatsheet of commonly used SPL commands is included in the annexes. This resource offers quick access to essential functions employed throughout data exploration and analysis workflows. In addition, practical SPL examples are presented throughout this guide to reinforce each concept and demonstrate real-world applications in the context of log analysis.

```
index="index_a" "api request" or "api response"
| rex field=_raw "(?i)Response\s*time\s*:\s*(?<response_time>\d+)\s*ms"
| rex field=_raw "(?<request_id>[a-zA-Z0-9]+)"
| rex field=_raw "API REQUEST * Body:\s*(?<request_body>[\s\S]+)"
| rex field=_raw "(?<request_id>[a-zA-Z0-9]+)"
| rex field=_raw "API RESPONSE * Body:\s*(?<response_body>[\s\S]+)"
| spath input=response_body
| eval status_code = tonumber(status_code)
| eval error_response = if(match(response_body,"error") OR status_code != 200, 1, 0)
| spath input=request_body
| foreach * [eval <<FIELD>>_present = if(isnotnull('<<FIELD>>'), 1, 0)]
| fields request_id, size, _time, response_time, *_present, error_response
| fillnull value=0
| outputlookup error_response.csv
```

Figure 3.3. Example of an SPL search pipeline for extracting, transforming, and exporting API log data

3.4 Splunk AI Toolkit (formerly MLTK)

The Splunk Machine Learning Toolkit (MLTK), recently rebranded as the *Splunk AI Toolkit*, extends Splunk's capabilities by enabling the seamless integration of machine learning techniques into operational monitoring workflows. Available through the Splunkbase application store, the toolkit simplifies adoption within existing IT environments by providing access to more than 30 commonly used machine learning algorithms. As documented by Splunk, the toolkit is built on top of well-established Python libraries for scientific computing, such as *scikit-learn*, thereby making advanced machine learning methods accessible to both Splunk users and data scientists.
By combining the expressive power of the Splunk Processing Language (SPL) with the AI Toolkit, users can extract meaningful features from incoming log data, preprocess and

transform them using SPL commands, and subsequently apply machine learning tasks directly within the Splunk ecosystem. Supported tasks include classification, regression, anomaly detection, and clustering, all of which can be executed on prepared log events without requiring external machine learning pipelines.
For practical guidance, a cheatsheet of the most relevant AI Toolkit (MLTK) commands available on the official Splunk documentation website is provided in the references. Additionally, practical SPL and MLTK examples are presented throughout this guide to reinforce key concepts and demonstrate real-world applications.

```
| inputlookup error_response.csv
| fit LogisticRegression error_response
  from response_time, *_present
  into error_response_model
  probabilities=true
```

Figure 3.4. Example of an SPL pipeline using the Splunk AI Toolkit to train a logistic regression model on extracted log features

Chapter 4

Guide illustration

The proposed guide is primarily oriented toward software maintenance teams and aims to transform raw event data (logs) into actionable intelligence. While the guide leverages Splunk and its AI Toolkit, the proposed methodology remains tool-agnostic. In practice, teams may adopt any platform capable of supporting the same workflow and objectives. For instance, the entire pipeline could be implemented from scratch using Python and widely adopted libraries such as *scikit-learn*. However, as previously discussed, such an approach may introduce friction in terms of setup, build, operability and maintenance. In contrast, integrated platforms like Splunk can reduce these barriers by providing a consolidated environment for data ingestion, preprocessing, model training, and operational monitoring. This integration can be particularly valuable when onboarding skeptical teams, as it demonstrates that improved reliability and automation can be achieved without requiring a complete redesign of existing operational processes.
The guide is structured into eight points, each contributing to the design, development, deployment, and continuous improvement of the machine learning pipeline.

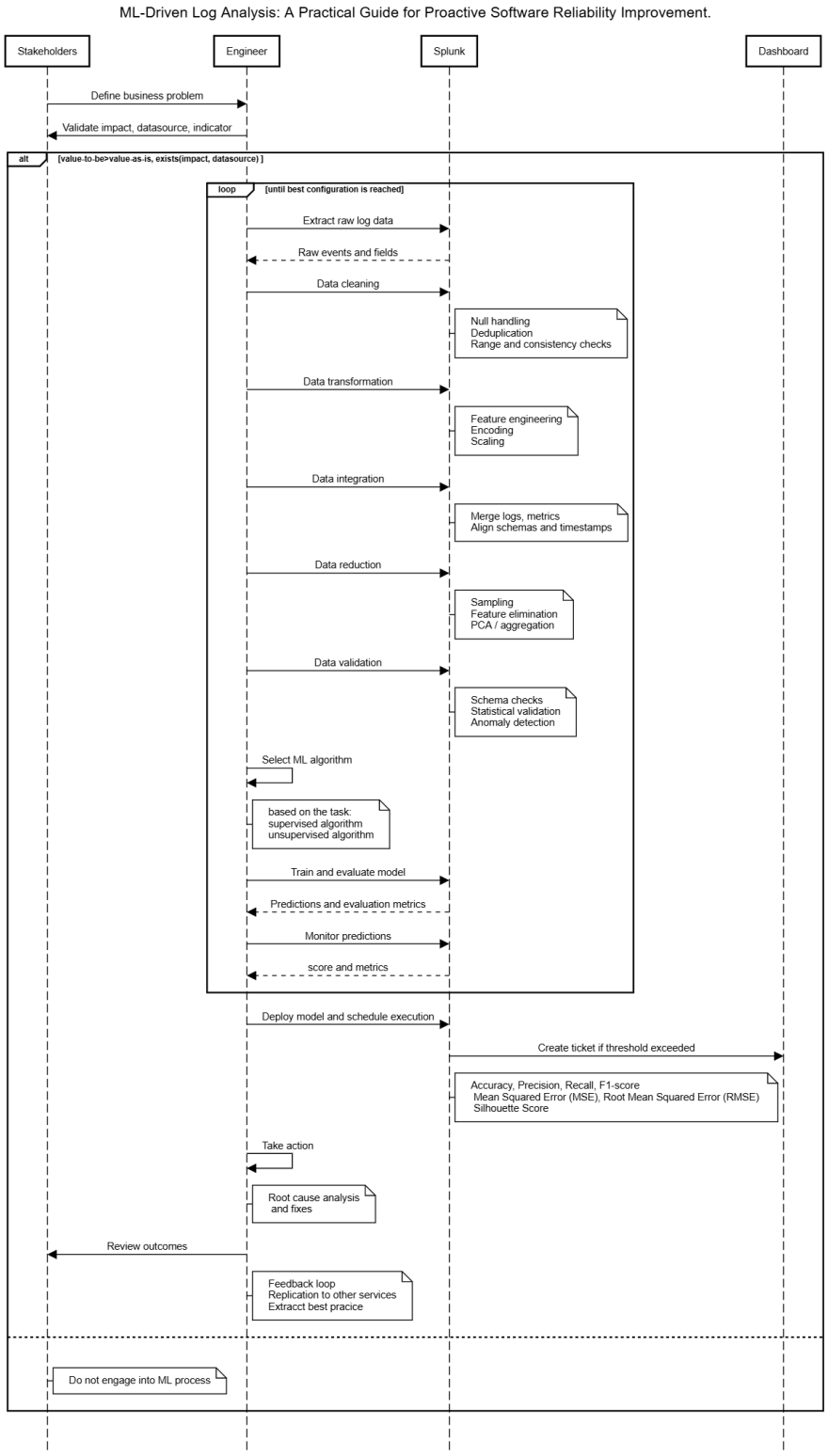


Figure 4.1. Sequence diagram illustrating the procedural steps of the ML-driven log analysis guide, from problem formalization through data extraction, data preprocessing, model training, alerting, remediation, and feedback.

4.1 Problem formalization: Establishing a repeatable and auditable process

Machine learning (ML) solutions are generally resource-intensive, requiring significant computational power, time, and specialized expertise. As a result, a clear and rigorous problem definition represents the most critical preliminary step in the entire ML pipeline. Without this clarity, organizations risk adopting machine learning merely to follow technological trends, leading to inefficient use of resources and limited return on investment. A well-defined problem statement directly influences the choice of the machine learning approach and, consequently, determines the specific task to be addressed, such as classification, regression, or clustering. It also clarifies the nature of the expected outputs and the type of actionable insights that the solution is intended to provide.
The primary objective of this preliminary phase must align with AIOps principles by ensuring that the proposed solution delivers measurable benefits throughout the Software Development Lifecycle (SDLC). In particular, it should support a shift from reactive to proactive software reliability improvement and maintenance, enabling early detection of issues, improved system stability, and more informed operational decision-making.

4.1.1 Formalizing problem identification

The process of problem identification requires a structured and explicit evaluation in order to ensure that machine learning efforts are both justified and effective. To this end, this work proposes a formal assessment based on three fundamental aspects, which collectively help determine whether a problem is suitable for machine learning-driven analysis:

Assessment Aspect	**Definition**	**Guiding Question**
Impact	This assesses whether solving the problem will have a high and measurable impact on the software reliability process.	Does the problem we want to solve have a high impact on software reliability (e.g., Mean Time to Repair/Detect, service availability, user experience)?
Indicator	This assesses the clarity and availability of the technical symptoms associated with the problem, which will serve as features for the machine learning model.	Are the indicators of the problem clear, well-defined, or do they require further analysis (e.g., feature engineering) to extract them?
Data source	This verifies the logical feasibility and trustworthiness of the solution by confirming the input data quality and access.	Are the data sources available, reliable, and clearly defined?

Table 4.1. Assessment criteria for formalizing a machine learning problem

By clearly defining this functional relationship between business impact, technical observability, and data availability, a repeatable and transparent problem identification process is established. This approach ensures that each identified problem or critical event is both business-relevant, reflecting tangible operational impact, and technically observable through measurable system data. Moreover, this formalization provides a consistent and auditable foundation for addressing new problems across different teams and environments. By applying the same evaluation criteria systematically, organizations can better prioritize high-value use cases and avoid allocating resources to low-impact or poorly defined machine learning initiatives.

4.2 Data extraction and preprocessing

The outcomes of the initial guiding questions and the data extraction phase help frame the problem and confirm that the selected data is relevant to the established business objectives. However, the interaction between engineers and stakeholders does not conclude at this stage. Additional iterations may be required to validate assumptions, refine indicators, and ensure that the extracted fields truly reflect operational needs.
Once the data structure is understood, the subsequent phase consists of preprocessing raw event data in order to make it suitable for machine learning algorithms. Given the need for high model efficiency and accuracy, this step is critical. It focuses on improving data quality and consistency by transforming raw, heterogeneous, and potentially noisy log events into a clean, structured, and model-ready dataset. In practice, preprocessing aims to retain only the most informative and reliable features, while correcting inconsistencies and reducing the impact of missing or irrelevant values. At this stage, the emphasis remains strictly on data usability and preparation rather than on the modeling itself.

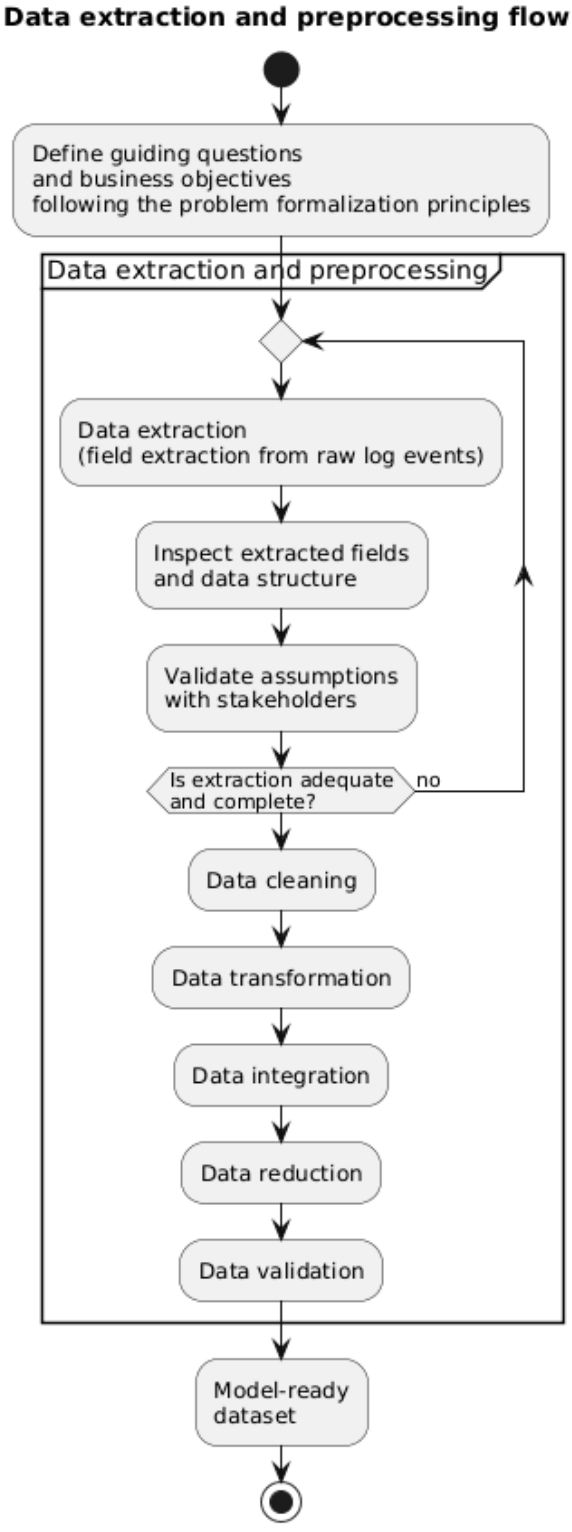


Figure 4.2. Data extraction and preprocessing workflow. The diagram illustrates the iterative refinement of data extraction guided by problem formalization and stakeholder validation, followed by a sequence of preprocessing steps that transform raw log events into a clean and model-ready dataset.

4.2.1 Data extraction

Once the Problem Identification phase has clearly defined the business impact, the relevant technical indicators, and the available data sources, the next essential step in the workflow is *Data Extraction*. This phase translates theoretical requirements into concrete data elements that can be consumed by the machine learning (ML) pipeline.
In the context of raw log analysis within Splunk, data extraction requires a thorough understanding of log structure and formatting. Engineers must often design custom regular expressions to isolate and extract relevant information embedded within unstructured log messages. This step is critical, as inaccuracies at this stage propagate throughout the pipeline and directly affect downstream model performance.
In SPL, the primary command used to extract fields from unstructured log data stored in the `_raw` field is `rex`. This command plays a central role in transforming free-text log messages into structured fields, which can be treated as columns or stream elements suitable for subsequent preprocessing and analysis.
The basic syntax for performing field extraction using `rex` is shown below:

```
| rex field=_raw "(?<field_name>regex)"
```

Figure 4.3. Basic syntax of the SPL `rex` command for extracting structured fields from raw log data

Once fields are successfully extracted, they move sequentially through the SPL pipeline and become available for downstream phases such as data preprocessing, including cleaning, integration, and feature engineering. This structured approach ensures that machine learning models operate exclusively on indicators that were explicitly identified during the problem formalization phase, thereby avoiding unnecessary computation on irrelevant or low-value data.
It is important to note that the introduction of machine learning techniques is not limited to a single step within the guide. While this phase primarily focuses on deterministic extraction, unsupervised learning methods may also be applied at this stage to explore the structure of log data, identify latent patterns, or assist in preliminary log classification.

4.2.2 Data cleaning

When extracting raw data from logs, it is common to encounter records that are incomplete, inaccurate, or irrelevant to the intended analysis. These data quality issues may originate from multiple sources, including inconsistencies in log generation across heterogeneous systems, insufficient application-layer validation, or erroneous user input. For instance, if a web application fails to enforce proper input constraints, a user may enter a conceptually invalid age of 1,000 years. Such anomalies can distort statistical summaries, bias machine learning models, and ultimately lead to unreliable predictions.

Ensuring data quality is therefore a foundational step in any data preparation process. Prior research and established best practices in data engineering identify a set of

systematic data constraints that can be used to assess, validate, and improve the quality of operational datasets before they are consumed by machine learning algorithms:

- **Data-Type Constraints**: Values in a given field must conform to an expected data type (e.g., Boolean, numeric, string, or date). Example: *response_time* should always be numeric.
- **Range Constraints**: Numeric or temporal values should fall within a logical or permissible range. Example: *response_time* should not be negative or exceed a plausible upper bound (e.g., 60 seconds).
- **Mandatory Constraints**: Certain fields must never be empty, as they are essential for interpretation or downstream processing. Example: *status_code* must always have a value.
- **Unique Constraints**: A field, or a combination of fields, must contain unique values to avoid ambiguity or duplication. Example: *request_id* should be unique per transaction.
- **Set-Membership Constraints**: Field values must belong to a predefined and valid set of options. Example: *method* should be one of [GET, POST, PUT, DELETE].
- **Foreign-Key Constraints**: Values should reference existing entities in an authoritative dataset. Example: a *user_id* appearing in a transaction log should exist in the user directory.
- **Regular Expression Patterns**: Textual fields should conform to a predefined syntactic pattern. Example: email addresses must match the pattern `[A-Za-z0-9._%+-]+@[A-Za-z0-9`
- **Cross-Field Validation**: Logical relationships between related fields must hold true. Example: a *discharge_date* cannot precede an *admission_date*.

Splunk's Search Processing Language (SPL) provides a comprehensive set of commands that allow engineers to detect, enforce, and remediate these constraints directly within the data processing pipeline. Once data quality rules are defined, appropriate cleaning strategies can be applied, including filtering invalid records, transforming values, or standardizing representations.

Below are common data cleaning use cases and their corresponding SPL implementations:

- **Handling missing or null values**:

```
| fillnull value="N/A" method status_code
```

Figure 4.4. Replacing null values in the *method* and *status_code* fields

- **Removing duplicates**:

```
| dedup request_id
```

Figure 4.5. Ensuring that each *request_id* is unique

- **Validating value ranges**:

```
| where age >= 0 AND age <= 150
```

Figure 4.6. Filtering records with invalid age values

- **Checking set membership**:

```
| where method IN ("GET","POST","PUT","DELETE")
```

Figure 4.7. Keeping only records with valid HTTP methods

- **Pattern matching using regular expressions**:

```
| regex email="^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$"
```

Figure 4.8. Keeping only records with valid email formats

- **Cross-field validation**:

```
| where discharge_date >= admission_date
```

Figure 4.9. Ensuring logical consistency between temporal fields

- **Standardizing or transforming data types**:

```
| eval response_time = tonumber(response_time)
```

Figure 4.10. Converting a string-formatted numeric field into a numeric data type

Through these commands, Splunk enables teams to systematically enforce data quality rules and transform raw log data into a clean, consistent, and analysis-ready dataset. This structured cleaning process not only improves machine learning model accuracy but also increases trust in the analytics pipeline and supports more reliable operational decision-making.

4.2.3 Data transformation

Raw data extracted from information systems frequently exists in formats that are unsuitable for direct consumption by machine learning algorithms. Typical issues include numeric values encoded as strings, timestamps expressed using inconsistent formats, or categorical fields represented non-uniformly across applications (e.g., *OK*, *Success*, *200*). The primary objective of data transformation is to convert these heterogeneous representations into consistent, standardized, and model-ready forms.
Since most machine learning algorithms operate exclusively on numerical inputs, this phase is essential to ensure that models can correctly interpret the semantic meaning of each variable. Proper data transformation improves model convergence, interpretability, and predictive performance.
Common transformation operations include normalization and scaling of numerical features, encoding of categorical variables, and feature engineering. Feature engineering refers to the process of deriving new variables from existing data in order to capture domain knowledge, behavioral patterns, or latent relationships. The Splunk Machine Learning Toolkit (MLTK) integrates scikit-learn preprocessing components through the `fit` and `apply` commands, enabling these operations to be performed directly within the SPL pipeline.
The following examples illustrate common data transformation operations implemented in Splunk:

- **Normalizing numeric values**:

```
| inputlookup cleaned_err_response.csv
| fit StandardScaler response_time into response_time_scaler
| apply response_time_scaler
| outputlookup engineered_response_latency.csv
```

Figure 4.11. Normalizing numeric values using *StandardScaler*. This transformation rescales *response_time* to have zero mean and unit variance, improving model stability and convergence.

- **Encoding categorical fields**: Automatically carried by the fit command, that will be introduced later in the paper

- **Feature engineering**:

```
| inputlookup cleaned_err_response.csv
| eventstats avg(response_time) as avg_response_time
| eval latency_ratio = response_time / avg_response_time
| outputlookup engineered_response_latency.csv
```

Figure 4.12. Feature engineering example deriving a deviation metric. The new feature captures how each request's response time differs from the global average, enabling models to detect abnormal behavior.

Data transformation is a pivotal bridge between raw information and machine learning readiness. By performing this step, teams can ensure their data is consistent, interpretable, and optimized for modeling.

4.2.4 Data integration

In real-world machine learning (ML) projects, data rarely originates from a single, clean, and self-contained source. Instead, relevant information is typically distributed across heterogeneous systems, including application logs, infrastructure metrics, CSV exports, API responses, and external databases. Within Splunk, these sources may be represented as indexes, lookup tables, summary indexes, or external data connections.
The diversity of these datasets in terms of schema, temporal resolution, and semantic meaning introduces a major challenge: combining multiple data streams into a single, coherent, and consistent dataset that accurately reflects overall system behavior. Data integration addresses this challenge by aligning, combining, and harmonizing data from multiple sources so that it can be reliably consumed by machine learning models.
This process generally involves the following key activities:

- **Aligning**:

```
| inputlookup cleaned_err_response.csv
| rename server AS host
| eval timestamp = coalesce(_time, event_time)
| fields host, timestamp, response_time, status_code, error_response
| outputlookup aligned_error_response.csv
```

Figure 4.13. Aligning fields that represent the same concept across different sources to ensure schema consistency. For example, normalizing the hostname field when it is recorded as *server* in one dataset and *host* in another.

- **Combining**:

```
| inputlookup aligned_error_response.csv
| join host timestamp
    [ search index=metrics
      | eval timestamp=_time
      | fields host, timestamp, cpu_usage, memory_usage ]
| outputlookup combined_error_system_metrics.csv
```

Figure 4.14. Combining datasets originating from different sources by synchronizing records on common keys such as *host* and *timestamp*. This operation enables the correlation of application-level errors with system-level metrics.

- **Harmonizing**:

```
| inputlookup combined_error_system_metrics.csv
| eval status_code = case(
      status_code=="OK" OR status_code=="Success", 200,
      status_code=="FAIL" OR status_code=="ERROR", 500,
      true(), status_code
  )
| eval cpu_usage = round(cpu_usage, 2)
| eval response_category = if(status_code==200, "SUCCESS", "FAILURE")
| outputlookup harmonized_error_metrics.csv
```

Figure 4.15. Harmonizing data semantics by unifying units, naming conventions, and categorical representations. For example, different success or failure indicators are mapped to standardized status codes and response categories.

- **Data Enrichment**:

```
inputlookup harmonized_error_metrics.csv
| lookup team_ownership.csv host OUTPUT team_name, business_unit
| outputlookup enriched_error_dataset.csv
```

Figure 4.16. Integrate complementary information such as linking the technical data with business or contextual information. For instance, mapping hosts to responsible teams.

By executing these steps, teams transform fragmented and heterogeneous raw data into a unified, machine-readable format. This integrated dataset forms the analytical backbone for all downstream processes, from feature engineering and model training to monitoring and predictive alerting in Splunk's MLTK.

4.2.5 Data reduction

In large-scale systems, raw datasets collected from logs, metrics, and events can rapidly grow to millions of records and hundreds of features. While such data volumes provide rich analytical potential, they also introduce challenges related to computational cost, model complexity, and noise. Excessive data can slow down training, reduce model interpretability, and in some cases degrade performance due to redundancy or irrelevant information.
Data reduction addresses these challenges by simplifying the dataset while preserving the essential information required for accurate and meaningful analysis. The objective is to reduce either the number of observations, the number of features, or both, thereby enabling faster model training, improved generalization, and lower computational overhead.
Within the context of Splunk and the Machine Learning Toolkit (MLTK), data reduction relies on a combination of statistical and algorithmic techniques applied directly within the SPL pipeline. These techniques allow teams to condense datasets without significantly compromising signal quality or analytical value.
Data reduction typically involves the following key activities:

- **Sampling**:

```
| inputlookup enriched_system_data.csv
| sample 0.1
```

Figure 4.17. Selecting a representative subset of the data when the full dataset is too large to process efficiently. In this example, a random 10% sample is extracted to support rapid prototyping and exploratory modeling.

- **Feature elimination**:

```
| fit RandomForestClassifier target from * into rf_model

| sort - importance
| eventstats perc80(importance) as threshold_p80
| where importance >= threshold_p80
| fields - threshold_p80
```

Figure 4.18. Removing features with low predictive relevance based on model-derived importance scores. In this example, only features above the 80th percentile of importance are retained after training a Random Forest classifier.

- **Dimensionality reduction**:

```
| fit PCA bytes_sent response_time cpu_usage into reduced_pca_model
| fields PC*
```

Figure 4.19. Applying Principal Component Analysis (PCA) to reduce correlated numerical features into a smaller set of orthogonal components that preserve most of the data variance.

- **Aggregation**:

```
| bin _time span=5m
| stats avg(response_time) as avg_response_time
        count as event_count
        by host
```

Figure 4.20. Aggregating fine-grained event data into coarser time windows. This transformation reduces data volume while preserving temporal trends relevant for system-level analysis.

By applying these data reduction techniques, teams can transform large, noisy, or redundant datasets into compact and high-quality representations optimized for machine learning. This reduction not only accelerates computation but also improves model stability, interpretability, and generalization, making it a critical step in production-ready ML pipelines.

4.2.6 Data validation

After data has been cleaned, transformed, integrated, and potentially reduced, it must undergo a final data validation phase to ensure that the resulting dataset is both reliable and representative of the real-world system being modeled. In the context of machine learning, data validation acts as the final quality gate before model training, verifying that data is complete, consistent, and statistically sound. Without this step, even well-designed preprocessing pipelines may propagate silent errors into models, leading to inaccurate predictions or unstable behavior in production environments.
Within Splunk, data validation is achieved by combining logical rule checks, statistical validation techniques, and model-based anomaly detection. Together, these approaches confirm that the dataset adheres to predefined structural constraints, remains within plausible operational ranges, and behaves consistently with historical patterns.
The following techniques illustrate common data validation strategies implemented using SPL and the Splunk Machine Learning Toolkit (MLTK):

- **Schema validation**:

```
| inputlookup reduced_pca_model.csv
| eval valid_fields = if(isnull(response_time) OR isnull(status_code), 0, 1)
| where valid_fields = 1
```

Figure 4.21. Schema validation ensuring that required fields are present and non-null before model training. Records missing mandatory attributes are filtered out.

- **Range and constraint checks**:

```
| inputlookup reduced_pca_model.csv
| where response_time >= 0 AND response_time <= 60
| where status_code IN (200, 201, 400, 404, 500)
```

Figure 4.22. Validating numeric ranges and categorical constraints to ensure that values fall within plausible operational limits.

- **Statistical validation and anomaly detection**:

```
| inputlookup reduced_pca_model.csv
| fit OneClassSVM response_time bytes_sent into validation_model
| apply validation_model
| where isOutlier = 1
```

Figure 4.23. Statistical validation using a One-Class SVM to detect anomalous records that deviate significantly from historical behavior.

By combining structural validation, constraint enforcement, and statistical analysis, teams can confidently verify both the integrity and behavioral consistency of their datasets before applying machine learning algorithms. When implemented systematically, data validation helps ensure that training inputs remain trustworthy, model predictions are stable, and automated decisions accurately reflect real operational conditions.
It is important to note that these validation techniques are not necessarily applied sequentially; their use depends on the specific characteristics of the data and the requirements of the analysis. The outcome of the data preprocessing phase should be a validated, structured dataset containing the extracted and transformed fields, ready to be consumed by the subsequent model training stage.

4.3 Model selection

The model selection phase represents a critical step in the machine learning pipeline, as it directly influences the quality, reliability, and operational usefulness of the resulting insights. While achieving high predictive accuracy is an important objective, it is not sufficient on its own. In the context of proactive log analysis and operational monitoring, models must also produce results that are actionable, explainable, and robust enough to support automated alerting and decision-making processes.
Effective model selection therefore requires balancing technical performance with practical constraints. Models should be interpretable enough for engineers to understand why a specific alert or prediction was generated, and stable enough to behave consistently when exposed to evolving system conditions. These requirements are particularly important in production environments, where false positives, unexplained predictions, or unstable models can quickly erode trust in machine learning-driven solutions.
This necessity leads directly to the identification of the appropriate machine learning task. In the context of proactive log analysis, the most commonly encountered task categories include:

- **Supervised learning tasks**: such as classification and regression, which rely on labeled historical data. These tasks are typically used to predict known outcomes, for example classifying log events as normal or anomalous, or estimating continuous values such as response time or resource utilization.
- **Unsupervised learning tasks**: such as clustering and anomaly detection, which operate without predefined labels. These approaches are particularly valuable for exploratory analysis, pattern discovery, and detecting previously unseen behaviors in log data.

Just as the objective of the analysis determines the type of machine learning task, the characteristics of the data strongly influence the choice of algorithm. For instance, algorithms well-suited for clustering unstructured textual information, such as exception messages or error descriptions, differ significantly from those used to model continuous numerical variables like response time or throughput. Factors such as data volume, feature dimensionality, class imbalance, and noise must all be considered during this selection process.
The Splunk Machine Learning Toolkit provides access to a wide range of algorithms that address these diverse requirements. While a comprehensive list of available models is provided in the official documentation referenced in this work, several commonly used algorithms have already been introduced in the machine learning overview chapter. These algorithms serve as practical building blocks for implementing reliable and explainable models tailored to operational log analysis.

4.4 Model training and evaluation

After selecting an appropriate algorithm and defining the relevant features, the next phase of the pipeline consists of training and evaluating the machine learning (ML) model. In

the Splunk ecosystem, this process is performed using the Search Processing Language (SPL) in combination with the Splunk Machine Learning Toolkit (MLTK), allowing models to be trained, evaluated, and applied directly on operational data without leaving the platform.

This phase follows a standard and rigorous ML lifecycle. The prepared dataset is first partitioned into distinct subsets for training, evaluation (or validation), and testing. The training dataset is used to fit the selected algorithm and allow the model to learn underlying patterns and relationships between features and targets. The evaluation dataset is then used to assess intermediate performance, tune hyperparameters, and compare alternative model configurations. Finally, the testing dataset, which remains unseen during training, is used to estimate the model's true generalization capability before it is considered suitable for production use.

The `fit` command is the primary MLTK function used to train a model. When executed, it operates on a *copy* of the search results produced by the preceding SPL pipeline. The resulting table is pulled into memory and parsed into a Pandas DataFrame, while the original indexed data remains unchanged. The `fit` command then performs an embedded preprocessing phase to prepare the data for machine learning: fields that are null across all events are discarded, events containing one or more null values are removed, and non-numeric categorical fields with more than 100 distinct values are dropped by default. Remaining categorical fields are converted into dummy variables using one-hot encoding.

The prepared dataset is then transformed into a numeric matrix, which is used by the selected algorithm to learn model parameters. The trained model is instantiated in memory and immediately applied to the prepared data, appending one or more result columns for example predicted labels, numerical estimates, or anomaly scores, back into the SPL pipeline. If the `into` clause is specified, the trained model is persisted as a Splunk knowledge object; otherwise, the temporary model is discarded at the end of the search. At the time of writing, the `LocalOutlierFactor` algorithm does not support model persistence and therefore cannot be saved as a reusable knowledge object.

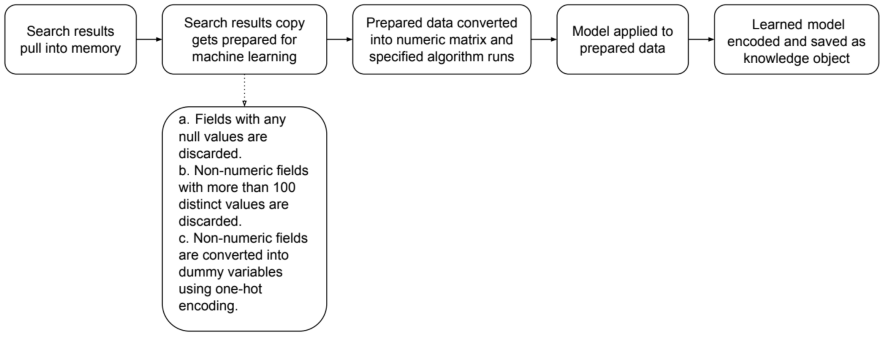


Figure 4.24. Internal operations triggered by the execution of the `fit` command. The limit for distinct categorical values is set to 100 by default and can be modified through Splunk configuration.

Once a model has been trained and optionally persisted, the `apply` command is used to load the learned model and execute it on new, unseen, or streaming data. To ensure compatibility with the learned feature space, `apply` repeats key preparation steps: it removes fields that are null across all events, discards high-cardinality categorical fields, applies one-hot encoding, removes dummy variables that were not present during training, and fills missing dummy variables with zeros. The prepared data is then converted into a numeric matrix, the model is applied, and the resulting output columns are returned to the SPL pipeline.

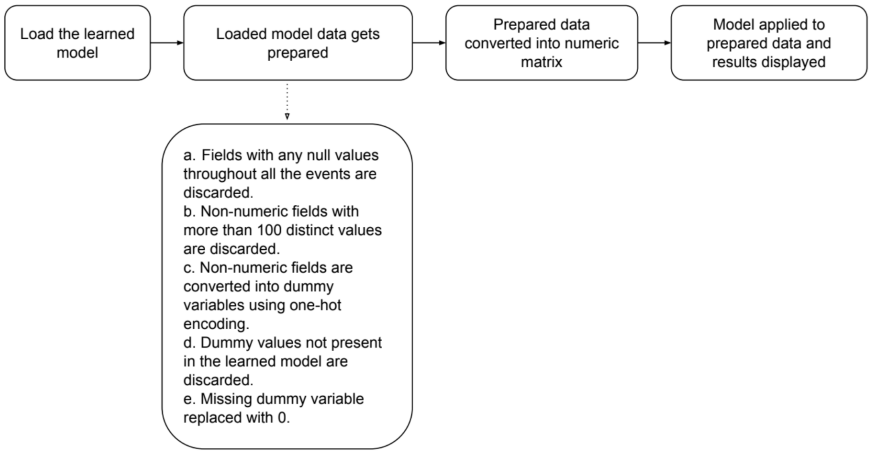


Figure 4.25. Internal operations triggered by the execution of the `apply` command. The same preprocessing constraints applied during training are enforced to ensure feature compatibility.

To support transparency and interpretability, MLTK also provides the `summary` command, which exposes metadata about a trained model. This command allows engineers to inspect key information such as the algorithm used, feature list, training parameters, and model statistics. The `summary` command is particularly useful for model auditing, debugging, and documentation in order to share the information about the current step.

```
| summary failure_predictor_model
```

Figure 4.26. Using the `summary` command to inspect metadata and configuration details of a persisted machine learning model.

The choice of evaluation metrics must align with the operational objective of the model. For instance, in failure detection scenarios, recall (measuring how many relevant failures are correctly identified) may be prioritized to minimize missed incidents. Conversely, in alerting systems, precision (measuring how many triggered alerts correspond to actual issues) is often more critical to reduce alert fatigue.

Model evaluation is performed using the `score` command, which computes specific performance metrics directly within SPL. The following example illustrates how classification metrics can be calculated after applying a trained model:

```
| apply error_response_model
| score precision recall f1 accuracy
```

Figure 4.27. Computing classification performance metrics using the `score` command after model inference.

The following table showcase some metrics that could be computed using the score command:

ML Task	**Common Evaluation Metrics**
Classification	Accuracy, Precision, Recall, F1-score
Regression	Mean Squared Error (MSE), Root Mean Squared Error (RMSE)
Clustering	Silhouette Score

Table 4.2. Common evaluation metrics used for different machine learning task types in Splunk MLTK

Finally, an important design decision concerns whether to compute machine learning results on demand or to persist trained models for repeated use. Because ML computations are resource-intensive, exploratory or diagnostic use cases such as identifying which request-body fields correlate with errors in a Spring application are often better served by ad hoc analysis. In contrast, when the application schema is stable and the objective is continuous monitoring or predictive alerting, persisting and reusing trained models provides greater efficiency and operational value. This flexibility allows teams to balance computational cost, responsiveness, and long-term maintainability within the ML pipeline.

4.5 Alert creation and scheduling

Once the Machine Learning model is trained and validated, the next crucial step is its operationalization. This involves ensuring the model runs automatically, monitors live data, and triggers alerts when specific conditions are met, or shares the results of the analysis. This process establishes continuous monitoring of the information system. In Splunk, this operationalization is achieved through scheduled searches and alert configurations. These components are often integrated into broader operational workflows using webhook integration or linked directly to ticket management systems such as JIRA. Given that

the Splunk materials comprehensively cover the technical implementation of this step, relevant links will be provided in the appendix to guide the reader.

4.6 Take action

After an alert has been triggered and, when applicable, a corresponding JIRA ticket has been automatically created, the process transitions to the remediation phase. This stage marks the shift from automated detection to manual or automated response, during which operational or development teams act upon the insights provided by the machine learning model.
The specific actions taken depend on the nature of the detected critical event and the severity of its predicted impact. For example, if an alert indicates a surge in predicted slow web requests, teams may initiate a root-cause analysis by examining correlated logs, metrics, and traces. This investigation aims to identify underlying performance bottlenecks, infrastructure constraints, or configuration issues affecting system behavior.
By establishing this integrated workflow, the system evolves from reactive monitoring toward proactive operations. This transition contributes to a significant reduction in the Mean Time To Detection (MTTD) and supports higher overall service reliability.
Effective remediation efforts must adhere to several core principles:

- **Timeliness**: Acting promptly upon alerts to minimize potential impact.
- **Relevance**: Ensuring that automated or manual interventions are appropriate for the detected anomaly.
- **Traceability**: Logging and documenting every action taken in response to an alert within the corresponding JIRA issue, enabling accountability and knowledge sharing.

4.7 Get feedback and replicate

Data analysis and machine learning are not one-time implementations but part of a continuous improvement cycle. Once actions have been taken and incidents resolved, the system must learn from its own performance; this objective is achieved through a feedback loop.
Following each incident, engineering teams should evaluate the accuracy and effectiveness of the model's predictions. This evaluation should address several critical questions:

- Did the model correctly identify a real issue (true positive)?
- Did it generate an unnecessary alert (false positive)?
- Did it miss a real event that later occurred (false negative)?
- Did the intervention (action taken) demonstrably improve software reliability?
- Could we define standard and best-practice from the process?

The insights gained from this evaluation are invaluable for refining alert thresholds, retraining models, or redefining features in order to improve predictive accuracy and operational relevance. The feedback loop also includes updating the corresponding JIRA tickets or notification threads with final resolution details, explicitly linking identified root causes back to the model's predictive features. These outcomes can subsequently be reused as labeled data for future retraining or for systematic model comparison.
Beyond continuous improvement, this phase also enables replication. Once the methodology for identifying, modeling, and operationalizing critical events has proven effective, the same approach can be extended to other services or system components within the organization. For example, the pipeline can be replicated to monitor additional metrics such as database query latency, message queue backlogs, or API failure rates, thereby scaling the benefits of the learning framework across the broader infrastructure.

Chapter 5

Use Case Scenario

Due to the absence of a real industrial context, this chapter presents a representative example designed to emulate a realistic operational scenario. This illustrative use case serves to demonstrate the application of the proposed guide, even if the final two steps will be impacted because of the lack of real context.

5.1 Problem statement

The company has observed that certain requests sent to specific microservices fail even though they appear to conform to the documented API specifications. This behavior suggests that some operational scenarios have not been fully anticipated or reviewed during the design and analysis phases, resulting in reduced software reliability and increased application maintenance effort.
To address this issue proactively, the company seeks to identify combinations of request fields and other request characteristics that are likely to trigger failures. By detecting these high-risk patterns in advance, development teams can gain actionable insights that support defect prevention rather than post-incident debugging, ultimately improving system robustness and operational stability.

5.2 Use case solution

5.2.1 Problem formalization

The objective of this use case is to apply Machine Learning techniques to proactively identify combinations of request fields and other relevant features that are likely to cause failures in specific microservices. This enables developers to mitigate issues during the Quality Assurance (QA) phase or to predict failing requests in production. Such predictions help determine whether these failures represent true defects or expected behaviors (i.e., false positives by design).
The problem can be formalized according to the following dimensions:

- **Impact**: Proactively identifying request parameter combinations that lead to failures allows the development team to reduce the Mean Time to Detection (MTTD) and improve overall software reliability.
- **Indicator**: Technical events or log messages that reliably signal errors or exceptions serve as observable indicators of software issues and can be associated with specific request fields.
- **Data Source**: The relevant microservices and business processes involved in the observed failures or defects must be identified in order to collect the appropriate datasets for analysis.

5.2.2 Data extraction

The preceding step provides insight into the type of information that must be extracted from the log data. Based on these requirements, the following SPL query is designed to extract key fields relevant to failure prediction, including request body attributes, request URI, response time, HTTP status code, request size, and event type (i.e., whether the log entry corresponds to a request or a response). In order to achieve this purpose, we make use of the `rex` command, which was introduced in Chapter 4.2.1 and is specifically related to the data extraction phase.
These extracted fields constitute the core feature set used in subsequent stages of data preprocessing and model construction, enabling both exploratory analysis and supervised learning techniques to identify request patterns associated with service failures.

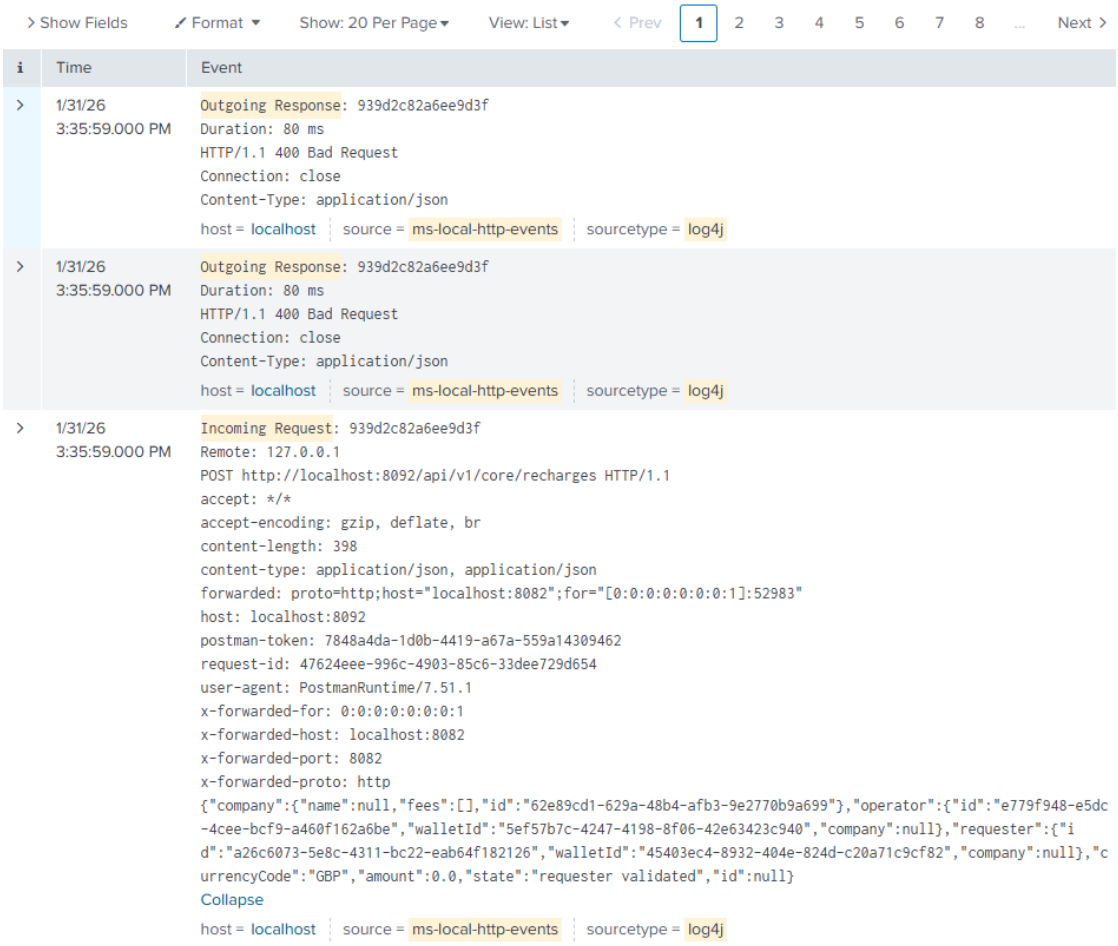
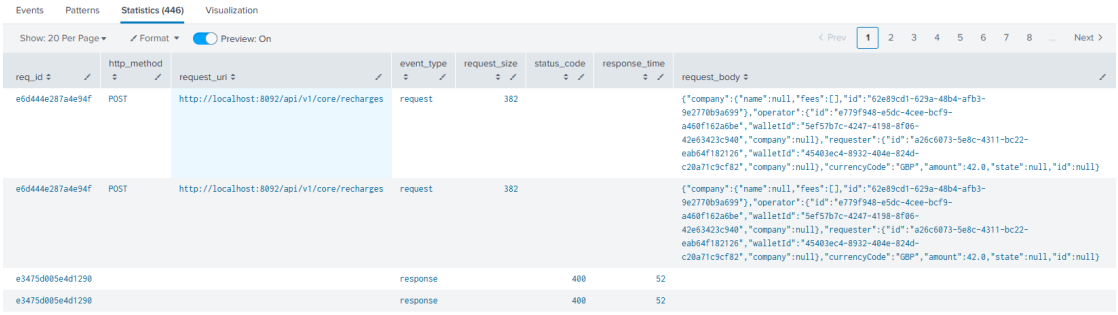


Figure 5.1. The screenshot shows representative request and response log entries associated with a POST API invocation, including HTTP metadata, request payload, response status code, and processing duration. These raw events constitute the unstructured input data from which relevant fields are extracted during the data extraction phase.

```
index="*"
source="ms-local-http-events"
sourcetype="log4j"
("Incoming Request" OR "Outgoing Response")
| rex "(Incoming Request|Outgoing Response):\s+(?<req_id>[a-f0-9]+)"
| rex field=_raw "(?<http_method>GET|POST|PUT|DELETE|PATCH|OPTIONS|HEAD)\s+(?<request_uri>\S+)"
| eval event_type=if(match(_raw,"Incoming Request"),"request","response")
| rex field=_raw "(?i)content-length:\s+(?<request_size>\d+)"
| rex field=_raw "HTTP/1\.1\s+(?<status_code>\d{3})"
| rex field=_raw "Duration:\s+(?<response_time>\d+)\s*ms"
| rex field=_raw "(?<request_body>\{.*\}$)"
| table req_id, http_method, request_uri, event_type, request_size,
        status_code, response_time, request_body
| outputlookup extracted_data.csv
```

Figure 5.2. SPL query used to extract structured request and response attributes from raw microservice logs of a local Spring application. The application of the `rex` command, combined with the corresponding regular expressions for each field, enables the extraction of the attributes that will be used in subsequent processing stages.



Events Patterns **Statistics (446)** Visualization

Show: 20 Per Page ✓ Format Preview: On

< Prev 1 2 3 4 5 6 7 8 ... Next >

req_id	http_method	request_uri	event_type	request_size	status_code	response_time	request_body
e6d444e287a4e94f	POST	http://localhost:8092/api/v1/core/recharges	request	382			{"company":{"name":null,"fees":[],"id":"62e89cd1-629a-48b4-afb3-9e2770b9a699"},"operator":{"id":"e779f948-e5dc-4cee-bcf9-a460f162a6be","walletId":"5ef57b7c-4247-4198-8f06-42e63423c940","company":null},"requester":{"id":"a26c6073-5e8c-4311-bc22-eab64f182126","walletId":"45403ec4-8932-404e-824d-c20a71c9cf82","company":null},"currencyCode":"GBP","amount":42.0,"state":null,"id":null}
e6d444e287a4e94f	POST	http://localhost:8092/api/v1/core/recharges	request	382			{"company":{"name":null,"fees":[],"id":"62e89cd1-629a-48b4-afb3-9e2770b9a699"},"operator":{"id":"e779f948-e5dc-4cee-bcf9-a460f162a6be","walletId":"5ef57b7c-4247-4198-8f06-42e63423c940","company":null},"requester":{"id":"a26c6073-5e8c-4311-bc22-eab64f182126","walletId":"45403ec4-8932-404e-824d-c20a71c9cf82","company":null},"currencyCode":"GBP","amount":42.0,"state":null,"id":null}
e3475d005e4d1290			response		400	52	
e3475d005e4d1290			response		400	52	

Figure 5.3. The table shows the structured representation of request and response events obtained after field extraction, including request identifiers, HTTP method and URI, event type, request size, response status code, response time, and parsed request payload. This intermediate view is used to verify the correctness and completeness of the extracted attributes before applying further preprocessing and feature engineering steps.

5.2.3 Data preprocessing

For this step, only data transformation was necessary. I wrote the script that creates the field presence matrix, engineer features in order to have columns that represent whether a field is numeric or not, the size of an array if there is one in the request body, and compute string length. This lead me to the following script:

```
| inputlookup extracted_data.csv
| dedup req_id
| stats
    latest(request_uri)       AS request_uri
    latest(request_size)      AS request_size
    latest(status_code)       AS status_code
    latest(response_time)     AS response_time
    latest(request_body)      AS request_body
    values(sourcetype)        AS sourcetype
    by req_id
| eval
    request_size  = tonumber(request_size),
    response_time = tonumber(response_time),
    status_code   = tonumber(status_code)
| spath input=request_body
| foreach * [
    eval <<FIELD>>_missing = if(isnull(<<FIELD>>),1,0)
]
| foreach request_body.* [
    eval <<FIELD>>_numeric = if(isnum(<<FIELD>>),1,0)
]
| foreach request_body.* [
    eval <<FIELD>>_strlen = if(isstr(<<FIELD>>),len(<<FIELD>>),0)
]
| eval is_failure=if(status_code>=400,1,0)
| fillnull value=0
| table
    req_id
    request_uri
    request_size
    response_time
    status_code
    sourcetype
    request_body.*
    *_missing
    *_numeric
    *_strlen
    is_failure
|  outputlookup preprocessed_data.csv
```

Figure 5.4. Feature engineering pipeline for transforming request payloads into a structured feature matrix. The query starts by loading the previously extracted dataset and removing duplicate requests based on the request identifier. It then aggregates request- and response-level attributes into a single record per request and enforces numerical typing for quantitative fields. The `spath` command is used to parse the request body, after which a set of iterative transformations generates features capturing field presence, data type, and string length for each payload attribute. Finally, a binary failure label is derived from the HTTP status code, missing values are normalized setting them to 0, and the resulting feature table is exported for subsequent model training.

Events Patterns Statistics (113) Visualization

Show: 20 Per Page Format Preview: On < Prev 1 2 3 4 5 6 Next >

req_id	request_uri	request_size	response_time	status_code	sourcetype	amount_missing	company.id_missing	company.name_missing
81b108b959735b68	0	0	43	400		1	1	1
82af9a852b58995d	0	0	69	400		1	1	1
83281e4d8a6cb0c4	0	0	72	400		1	1	1
84b37ebb8774b6aa	0	0	111	400		1	1	1
853289318ee9cf55	0	0	76	400		1	1	1
8536ee27d9f4f05c	0	0	75	400		1	1	1
8771ec68b6ecb927	0	0	77	400		1	1	1
87ac90c1537c4b7c	0	0	67	400		1	1	1
8915f93cc85039c0	0	0	62	400		1	1	1
89a0a648bc0e881a	0	0	145	200		1	1	1
8ae30b1f3c8aec44	0	0	325	400		1	1	1
8c448b1bf394a9ae	0	0	95	500		1	1	1
8c7cbcae76ce0ac0	0	0	64	400		1	1	1
90e23340ea1c72cd	0	0	88	400		1	1	1
91d5d38c5a7efa8d	0	0	96	400		1	1	1
9262a4548a43e856	0	0	64	400		1	1	1
92b24951f4351b7c	0	0	87	400		1	1	1

Figure 5.5. The table shows a subset of the engineered features derived from the extracted log data, including numerical request attributes, response indicators, and binary flags capturing missing fields within the request payload. For readability, only a portion of the available feature columns is displayed. This representation is used to validate the outcome of the preprocessing and feature engineering steps prior to model training.

5.2.4 Model selection

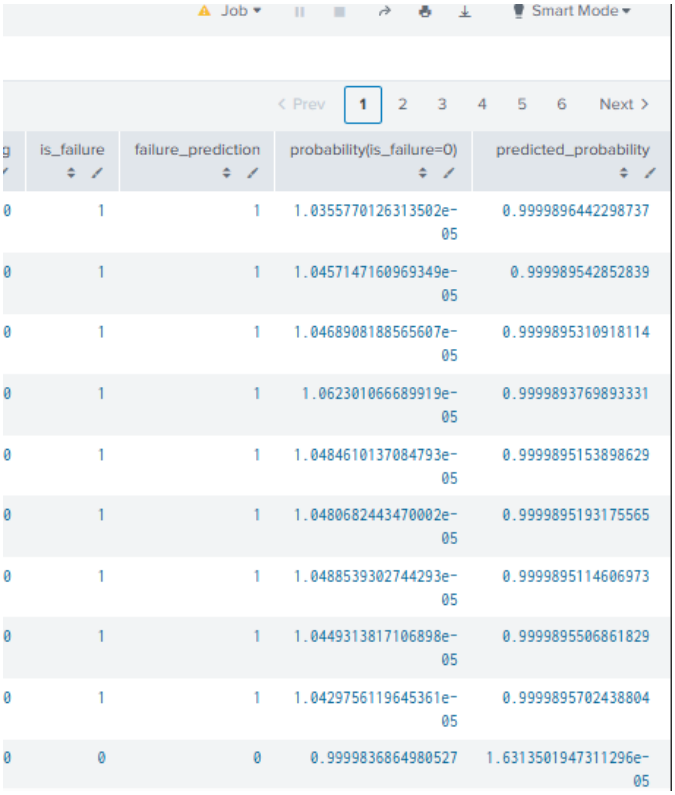
Since our data is clearly split into features and a label, where the label is the is_failure field and the others are the features, it's clear that we should use a supervised learning approach. Then comes the choice of the model we want to use, and for the purpose of this work, we will go for Logistic Regression, which perfectly fits the case of a binary label.

5.2.5 Model training and evaluation

The model selection phase gives us all we need to build the script for model training but we still have to split our logs into training and evaluation dataset. This was achieved by splitting the logs on a time based logic. Note that after the data preprocessing step you can also use the smart assistant which has a dashboard more suitable for the configuration of the machine learning model.
So, the application of the fit command that train the model is the following:

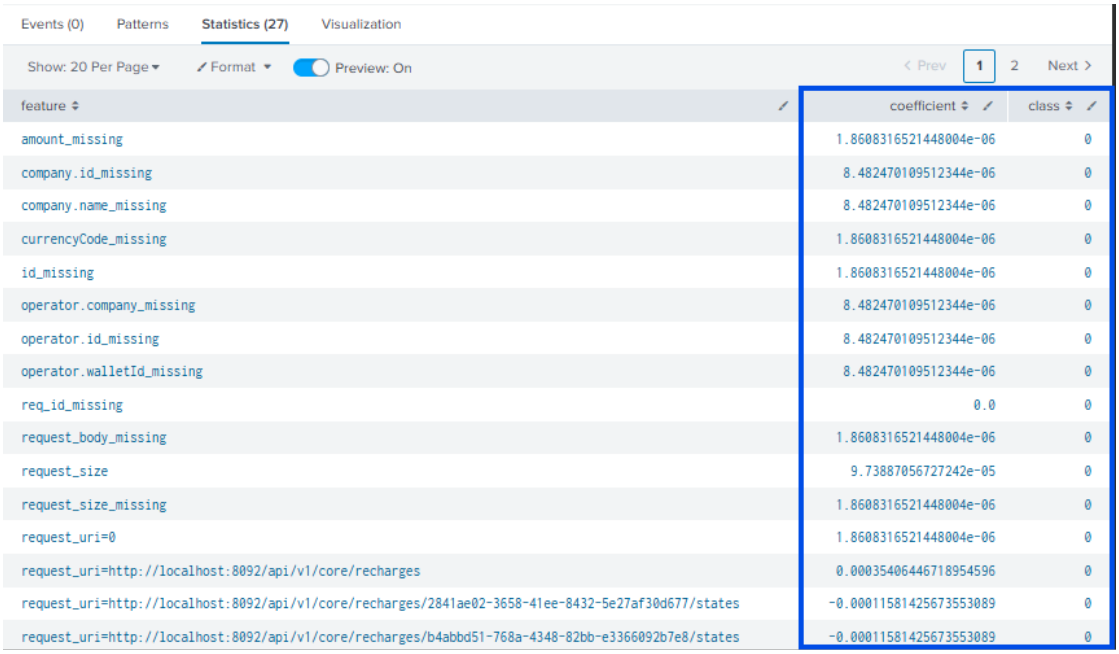
```
| inputlookup preprocessed_data.csv
| fit LogisticRegression is_failure
  FROM req_id
    request_uri
    request_size
    response_time
    status_code
    sourcetype
    request_body.*
    *_missing
    *_numeric
    *_strlen
    is_failure INTO failure_predictor_model probabilities=true
| rename probability(is_failure=1) as predicted_probability
```

Figure 5.6. Training of a Logistic Regression model for request failure prediction using preprocessed request attributes and engineered structural features. The query loads the feature table generated in the preprocessing stage and fits a supervised classification model using the failure label as the target variable. All request-level, payload-derived, and structural features are provided as input to the learning algorithm. The trained model is stored as a reusable Splunk knowledge object, and class probabilities are enabled to allow continuous failure likelihood estimation, with the predicted failure probability explicitly extracted for subsequent analysis and alerting.



g	is_failure	failure_prediction	probability(is_failure=0)	predicted_probability
0	1	1	1.0355770126313502e-05	0.9999896442298737
0	1	1	1.0457147160969349e-05	0.999989542852839
0	1	1	1.0468908188565607e-05	0.9999895310918114
0	1	1	1.062301066689919e-05	0.9999893769893331
0	1	1	1.0484610137084793e-05	0.9999895153898629
0	1	1	1.0480682443470002e-05	0.9999895193175565
0	1	1	1.0488539302744293e-05	0.9999895114606973
0	1	1	1.0449313817106898e-05	0.9999895506861829
0	1	1	1.0429756119645361e-05	0.9999895702438804
0	0	0	0.9999836864980527	1.6313501947311296e-05

Figure 5.7. The table displays the predicted failure label for each log event together with the associated class probabilities produced by the trained Logistic Regression model. In particular, the predicted probability represents the model's confidence in the occurrence of a failure event and is subsequently used for evaluation and alerting purposes. Only a subset of the available records is shown for readability.



Events (0) Patterns **Statistics (27)** Visualization

Show: 20 Per Page ▾ Format ▾ Preview: On < Prev 1 2 Next >

feature ⇕	coefficient ⇕	class ⇕
amount_missing	1.8608316521448004e-06	0
company.id_missing	8.482470109512344e-06	0
company.name_missing	8.482470109512344e-06	0
currencyCode_missing	1.8608316521448004e-06	0
id_missing	1.8608316521448004e-06	0
operator.company_missing	8.482470109512344e-06	0
operator.id_missing	8.482470109512344e-06	0
operator.walletId_missing	8.482470109512344e-06	0
req_id_missing	0.0	0
request_body_missing	1.8608316521448004e-06	0
request_size	9.73887056727242e-05	0
request_size_missing	1.8608316521448004e-06	0
request_uri=0	1.8608316521448004e-06	0
request_uri=http://localhost:8092/api/v1/core/recharges	0.00035406446718954596	0
request_uri=http://localhost:8092/api/v1/core/recharges/2841ae02-3658-41ee-8432-5e27af30d677/states	-0.00011581425673553089	0
request_uri=http://localhost:8092/api/v1/core/recharges/b4abbd51-768a-4348-82bb-e3366092b7e8/states	-0.00011581425673553089	0

Figure 5.8. The table reports the learned feature coefficients associated with the failure class, indicating the relative contribution of each engineered feature to the model's prediction. Positive and negative coefficient values reflect feature's influence on failure likelihood. For readability, only a subset of the available features is shown.

Evaluation of the model is then performed by analysing the confusion matrix using the following spl:

```
| rename predicted(is_failure) as failure_prediction
| score confusion_matrix is_failure against failure_prediction
```

Figure 5.9. Computation of the confusion matrix using the `score` command to evaluate binary classification performance by comparing true failure labels with model predictions.

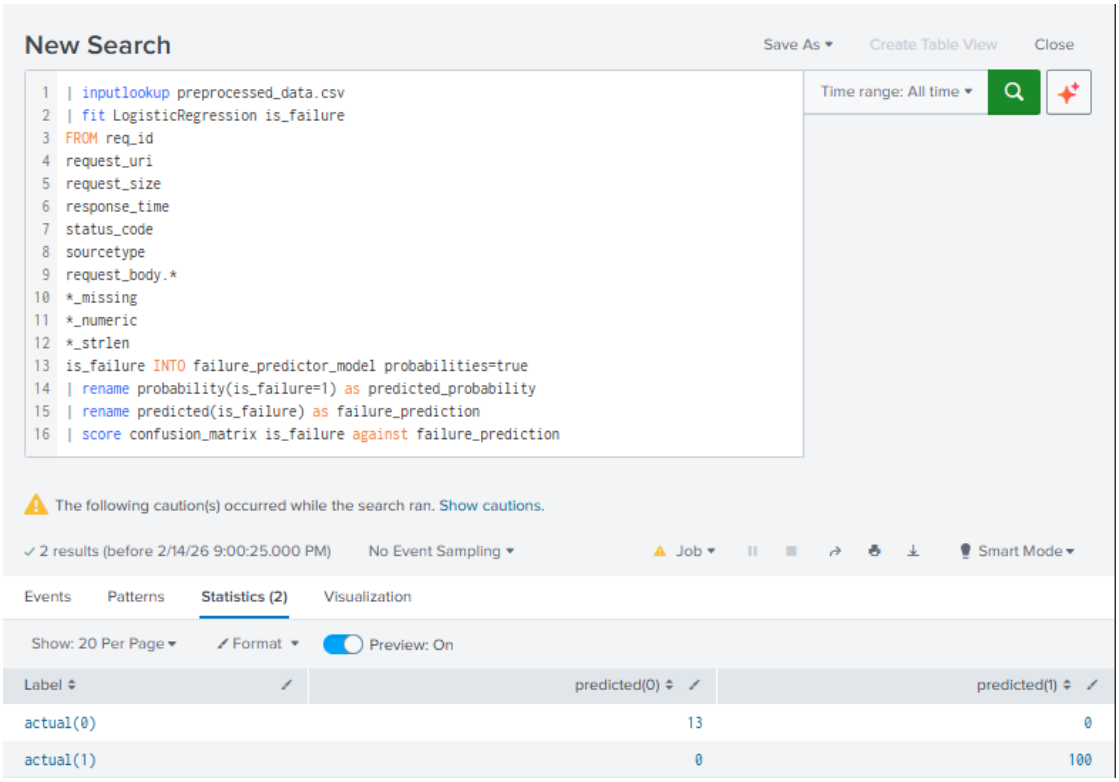


Figure 5.10. The visualization summarizes the distribution of predicted versus actual failure labels, reporting true positives, true negatives, false positives, and false negatives computed on the available dataset. The confusion matrix is used to provide a qualitative assessment of the model's behavior under the controlled experimental conditions described in this use case.

Note that the apparent high predictive performance is largely attributable to the controlled nature of the dataset, in which failure events were deliberately induced. Consequently, the class distribution is highly imbalanced and differs significantly from that of real production environments.

5.2.6 Alert creation and scheduling

Scheduling the alert requires writing the SPL that applies the model, this is done by copying the data extraction and preprocessing part and then instead of fitting the model, we write the following for new unseen data: Then, we will be ready for alert scheduling.

```
| apply failure_predictor_model
```

Figure 5.11. Applying the persisted failure prediction model to unseen log events to generate inferred failure labels

Here for simplification we can choose a set of people that will receive alerts related to our model application by the means of mail. This is configured in Splunk search functionality by saving the search, defining a trigger condition (in our case prediction with 90% probabilities) and configuring other parameters such as the periodicity of the alert. The following is a screenshot of the configuration for our use case:

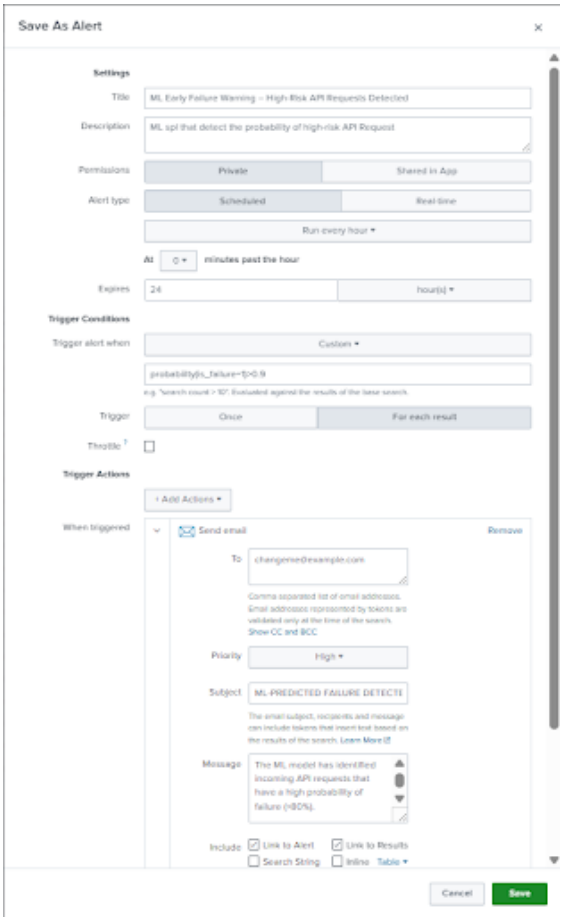


Figure 5.12. Configuration of a scheduled Splunk alert based on machine learning probability predictions.

5.2.7 Take action

This part requires the dev team in charge of improvement of the software, to investigate, mitigate the issue that our alert is raising.

5.2.8 Get feedback and replicate

This step follows the guideline illustrated in the previous chapter allowing the continuous improvement of our AIops.

Chapter 6

Conclusion

This journey began with an exploration of the field of Machine Learning (ML), significantly extending my prior knowledge and technical perspective. The solid theoretical and methodological foundations acquired during my studies at Politecnico di Torino provided the confidence and analytical rigor necessary to approach this complex challenge.
As the learning phase progressed, a key principle emerged: ML is both computationally and operationally expensive, and its adoption must therefore be justified by clear and measurable value. This realization motivated the development of a formal framework to rigorously define problems and assess whether the application of ML would deliver a meaningful return on investment. Building on this conceptual groundwork, the majority of the technical effort was dedicated to mastering Splunk's Search Processing Language (SPL) in order to fully leverage the capabilities of the Splunk AI Toolkit (formerly MLTK) and integrate machine learning workflows into an operational log analysis environment. This work culminated in the definition of an eight-step guide designed to proactively enhance software reliability through structured and explainable ML-driven monitoring.
In conclusion, introducing ML into software maintenance operations can provide significant value by improving system reliability and enabling more proactive operational practices. However, due to the inherent complexity of ML, the availability of a clear and practical guide that accelerates technology adoption represents a substantial advantage for engineering teams. This thesis aims to address that need. A natural extension of this work would be to enrich the proposed framework with AI-assisted code remediation, allowing models not only to detect issues but also to suggest corrective actions that engineers can validate before deployment. Such an evolution would further automate and streamline the software maintenance lifecycle.

Bibliography

Maren Westermann Yao Xiao Arturo Amor, Lucy Liu. scikit-learn: Machine learning in python, 2024. URL https://scikit-learn.org/stable/. Official documentation.

freeCodeCamp. freecodecamp learning platform, 2024. URL https://www.freecodecamp.org/learn. Online educational resource.

Splunk Inc. Configuring alerts in splunk, 2023a. URL https://www.youtube.com/watch?v=8jvEmAmQNug. Video tutorial.

Splunk Inc. Splunk machine learning toolkit cheat sheet, 2023b. URL https://www.splunk.com/en_us/pdfs/training/splunk-machine-learning-toolkit-app-cheat-sheet.pdf. Training documentation.

Splunk Inc. Splunk spl cheat sheet: Query, spl & regex commands, 2023c. URL https://www.splunk.com/en_us/blog/learn/splunk-cheat-sheet-query-spl-regex-commands.html. Splunk Blog.

Splunk Inc. Splunk cloud platform documentation, 2024a. URL https://help.splunk.com/en/splunk-cloud-platform. Accessed: 2025-01.

Splunk Inc. Splunk docker image, 2024b. URL https://hub.docker.com/r/splunk/splunk/. Docker Hub repository.

Splunk Inc. Sending splunk alerts to jira, 2024c. URL https://help.splunk.com/en/splunk-observability-cloud/manage-data/available-data-sources/supported-integrations-in-splunk-observability-cloud/notification-services/send-alerts-to-jira. Notification service integration guide.

Kylie Ying. Machine learning for everybody, 2022. URL https://www.youtube.com/watch?v=i_LwzRVP7bg. YouTube tutorial.