Proprietary and Confidential Information

Revision History

|  |  |
| --- | --- |
| **Date** | **Change Description** |
| Mar-29-2021 | Initial draft |

Table of Contents

[1 Top System Blocks 1](#_Toc67993212)

[2 Feature Extraction 2](#_Toc67993213)

[2.1 Candidates for Feature Extraction 2](#_Toc67993214)

[2.2 OSS Blocks for Feature Extraction 2](#_Toc67993215)

[2.3 Pre-Processing 3](#_Toc67993216)

[2.4 Clustering / Drain 3](#_Toc67993217)

[2.5 Error Set Building in Knowledge-base 3](#_Toc67993218)

[3 Vectoring and Matrixing 4](#_Toc67993219)

[4 LogLab Model Training 5](#_Toc67993220)

[5 The Contents of Targets / Classes 6](#_Toc67993221)

[6 Parameter Error Classification 7](#_Toc67993222)

[7 Prepare the Training Dataset 8](#_Toc67993223)

# Top System Blocks

We implemented three modules for log analyzing: OSS (Old School System), Loglizer (supervised learning model) and DeepLog (unsupervised learning model). The OSS analyzes each single log based on knowledgebase and has no context awareness in nature although we programmed some context technique for some kind of error logs. On the other hand, both Loglizer and DeepLog consider context but they are models for anomaly detection, say, false or true of the log after predicting.

LogLab is a comprehensive system for log analyzing by classifying **logs with error** (including its context) into different groups (targets), aka, a multiclass classification problem. Each class has description of issues, potential fix and optional CSPs/JIRAs related if exist.

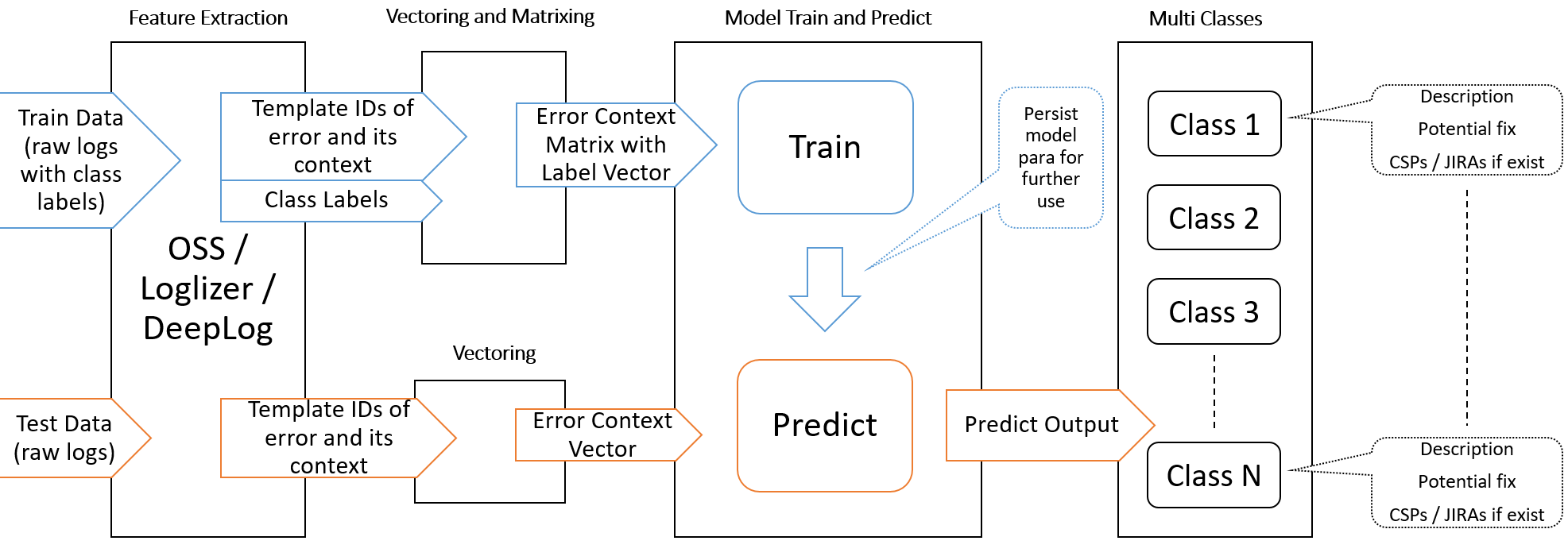


Figure 1: LogLab System Blocks

# Feature Extraction

## Candidates for Feature Extraction

LogLab classifies logs of error including its context into different classes, so we need detect and identify the logs of error and its context firstly. We have OSS, Loglizer and DeepLog to do this. The OSS wins based on the analysis below.

**Loglizer** needs labeled training data to update its model and the labeled data should cover anomalies as many as possible. We don’t have enough training dataset for Loglizer, especially when the host system (log producer like CM) is evolving.

**DeepLog** doesn’t need labeled training data but FP (False Positive) detection is always annoyable when the model is not fully trained and updated. We can overcome this shortage in some way but it’s a challenge.

**OSS** depends on the knowledge base in which we store the typical templates of error. In the process of training data collecting, we can update the knowledge base by the way if we find new typical templates. Another advantage of OSS is the parameter anomaly detection ability.

## OSS Blocks for Feature Extraction

We can reuse the basic parts of the standalone OSS we already implemented. The part of pre-processing has to consider the LogLab class labels. When matching the typical template in the knowledge base, we also need get the context template IDs (aka. Event IDs) in a window.

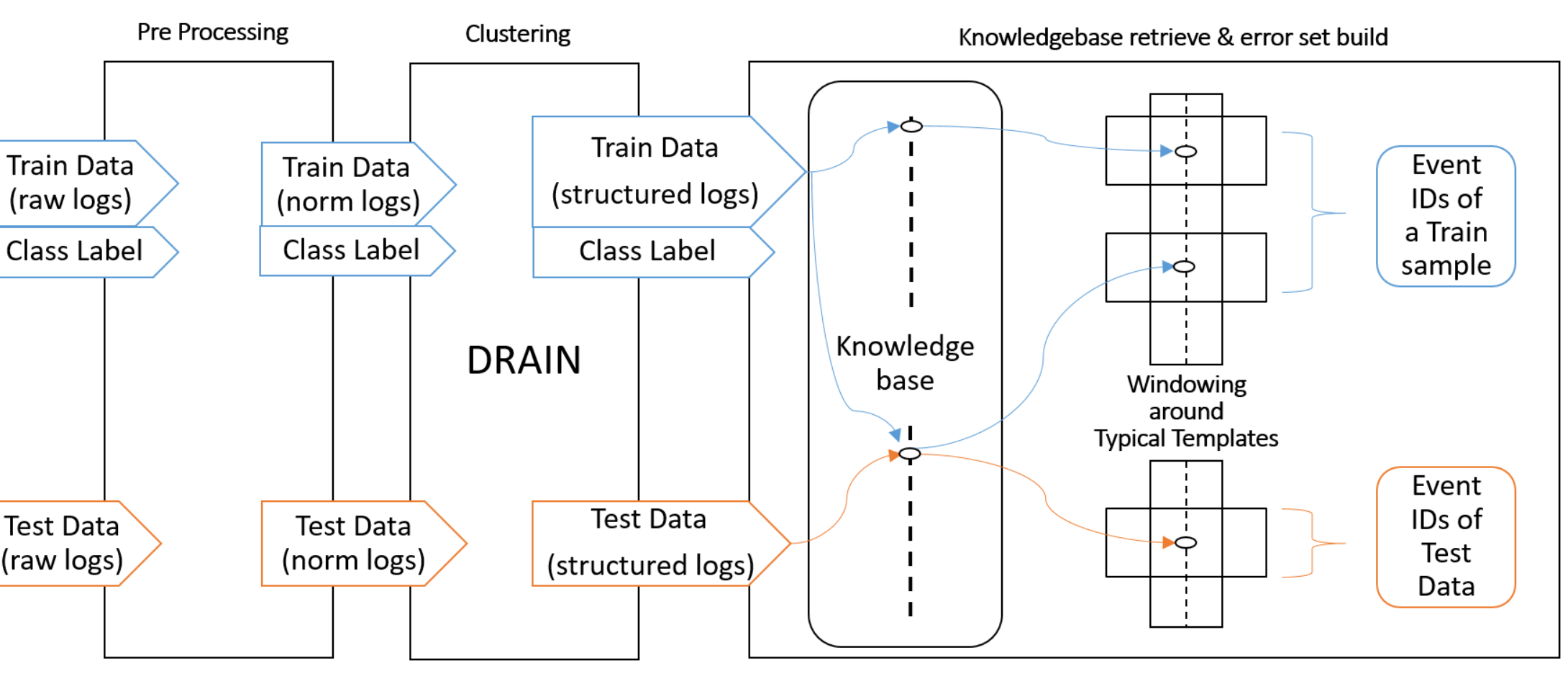


Figure 2: Revised OSS Blocks

## Pre-Processing

We store a sample (from the perspective of LogLab) of logs in a file separately for training dataset. Each training file has a label which tells us the target class of the file. This label will pass through Pre-Processing block, Clustering block and reach KB block for further use.

Besides the LogLab class label, another change is to handle arbitrary format of timestamp as the training data may come from CSP and JIRA system as well as reference system under different kind of error simulations.

At last, a top script will control all the training data files (each file is a sample) to form the final matrix for LogLab model training.

## Clustering / Drain

Suppose no changes.

## Error Set Building in Knowledge-base

We mainly need to do two things. The **first** one is to capture the event IDs in the window around the typical event ID matched in the KB. Most likely there are more than one typical event ID matching in the sample file. So the **second** one is to remove the duplicates to form a unique event ID set for one sample.

Algorithm 2-1 in pseudo code (typical event ID windowing)

TBD.

Algorithm 2-2 in pseudo code (build a unique set of event IDs)

TBD.

# Vectoring and Matrixing

It is similar with the event count matrix in Loglizer, but there are multiple labels. We define N as 1999 currently.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sample | Template0 | Template1 | Template2 | … | TemplateN | Label |
| 1 | 11 | 3 | 20 | … | 0 | 1 |
| 2 | 0 | 6 | 20 | … | 0 | 1 |
| 3 | 0 | 1 | 0 | … | 0 | 2 |
| 4 | 9 | 1 | 0 | … | 0 | 3 |
| 5 | 1 | 10 | 0 | … | 0 | 3 |

It is TBD whether we should count the event ID or not. This needs some tests.

# LogLab Model Training

It is a multiclass classification problem. Deep learning usually has a better precision than the classical machine learning on this problem, but the result is based on the availability of extensive train dataset.

TBD for the model selection.

# The Contents of Targets / Classes

The training data come from three ways. 1) System error simulation on Reference SW / Boards. E.g. Errors because of environment factor, CMTS and CM settings, etc. 2) Logs in some CSPs. 3) Logs in some JIRAs.

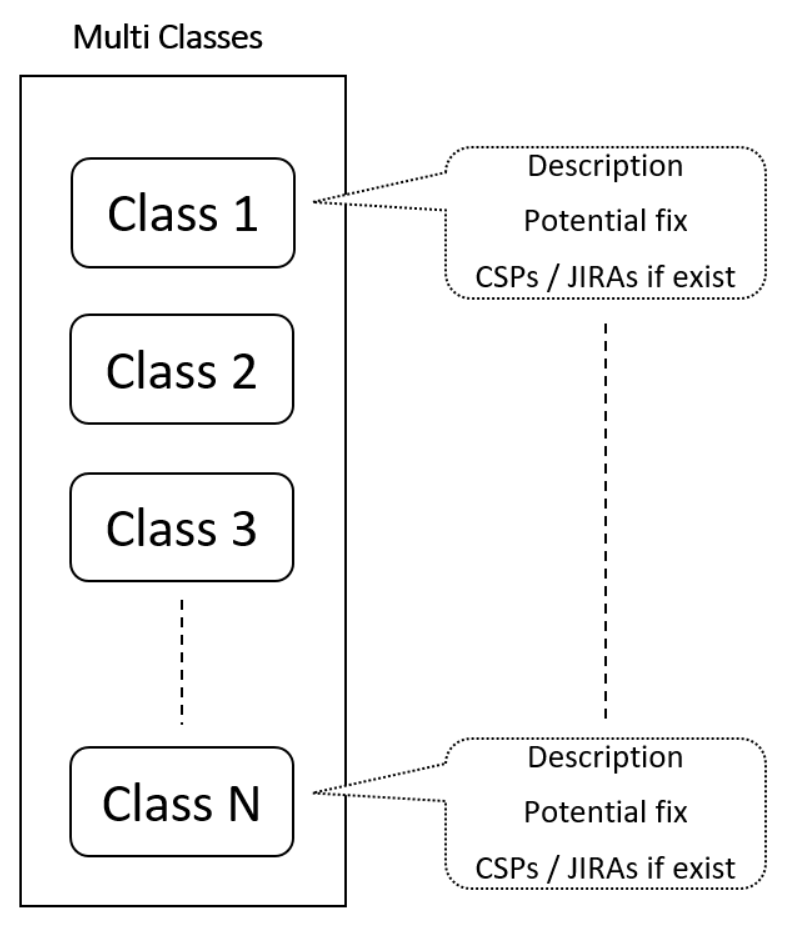


Figure 3: Multi-Class Contents

The CSP and JIRA number are optional in each class’s contents. If one or more of the sample files of a class come from CSP or JIRA, we can fill out the number there. More than one CSP / JIRA number can be in one class. No CSP / JIRA number for the class if we only have training data of simulation. The logs in CSP / JIRA are supplementary to the simulation data.

# Parameter Error Classification

If a parameter in a log has some problem, e.g. downstream snr / rx power or upstream tx power, the former model MAY not classify the error into the correct target class. This is because we remove all the parameter values for the LogLab model training. To overcome this issue, we can add an exception path in the OSS blocks (Figure 2.)

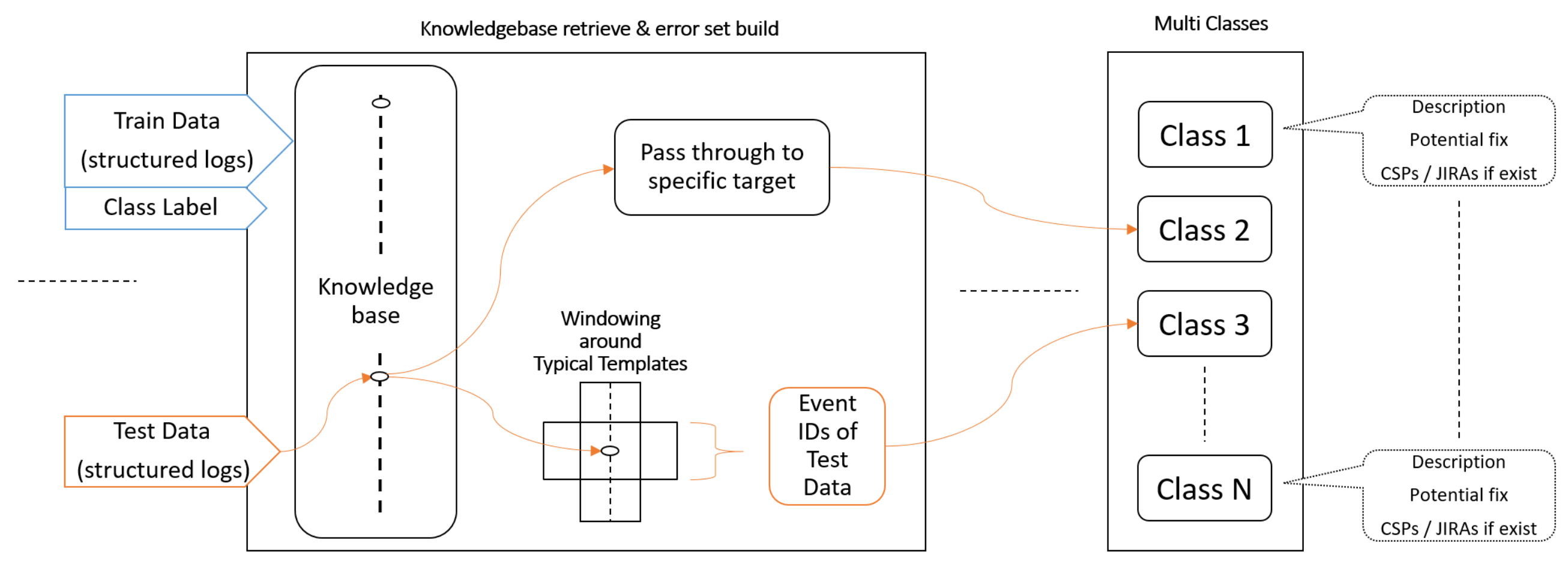


Figure 4: Parameter Error Classification

This scheme can deal with some tough cases like ranging abort, tx power low high threshold, etc., which is usually enough for classification with single log or along with one or two context logs.

# Prepare the Training Dataset

It is hard work to prepare the training data. For simulations, it is relatively easy to control what we want, e.g. a sample of logs saves to a separate file in a dedicated folder which maps a target class.

For training data from CSP or JIRA, 1) we have to use domain knowledge to classify different logs into different target classes. 2) Identify typical error logs / templates. 3) Split one whole log into sample files.

Update the knowledgebase incrementally in the process of either doing simulation or processing the logs in CSP / JIRA.

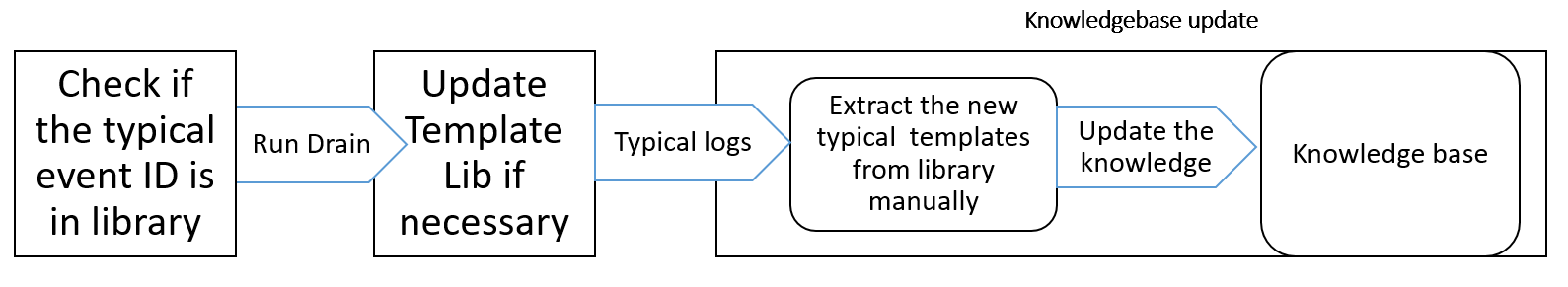


Figure 5: Update KB during Training Data Collecting