Cairo University
Faculty of Engineering
Department of Computer Engineering

Aigile

The Intelligent Automation of the Scrum Workflow

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by

Moustafa Mohammed

Supervised by

Dr. Ahmed Darwish

July 2025

Chapter 4: System Design and Architecture

4.3.2. Modular Decomposition

The Acceptance Criteria Generation component is a self-contained part of the User Story Generation module, designed for modularity and scalability. I focused on its backend logic and frontend integration, ensuring seamless interaction with Jira. Below is the decomposition of my work:

Backend

- **Purpose**: Generate and refine acceptance criteria for a given user story.
- **How It Works**:
  - Employs a “Create-Update-Update” (CUU) approach, which I designed to ensure high-quality criteria:
    1. **Create**: Generates initial criteria using the LLM, based on a prompt incorporating the user story and context.
    2. **Update (QA Perspective)**: Refines criteria to ensure testability and alignment with QA standards.
    3. **Update (Team Perspective)**: Further refines criteria to incorporate team and business requirements, ensuring feasibility.
  - Uses the filterSimilarAC function, which I implemented, to remove duplicate or similar criteria using TF-IDF vectorization and cosine similarity for computational efficiency.
  - Incorporates few-shot examples in the prompt to ensure consistent, well-formatted outputs.
- **Implementation**:
  - Developed GenerateACByCUU and GenerateFinalAC functions in generate_ac.py and generate_acceptance_criteria.py.
  - Configured the backend to communicate with the Groq API, formatting outputs as JSON for frontend compatibility.
  - Hosted on a Flask server on PythonAnywhere, which I set up for asynchronous LLM request handling.

Frontend Integration

- **Purpose**: Enable user interaction with generated acceptance criteria within Jira.
- **How It Works**:
  - Fetches the issue summary (getIssueSummary) and existing criteria (get-all) for a Jira issue.
  - Allows users to trigger criteria generation via a “Generate Criteria” button, invoking the startGeneration resolver.
  - Polls the backend using checkGenerationStatus every 3 seconds to retrieve results.
  - Displays criteria as a checklist with Checkbox components and a ProgressBar for completion tracking.
  - Stores criteria per issue using storeACs for persistence in Jira.

- **Implementation**:
  - Contributed to index.jsx, using React and Atlassian's Forge UI framework for a reactive interface.
  - Utilized useState to manage issue key, criteria list, and user feedback messages.
  - Implemented useEffect for polling initialization and cleanup to prevent memory leaks.

Technology Choices

- **Backend**:
  - **Flask**: I selected Flask for its lightweight, asynchronous capabilities, ideal for handling multiple LLM requests on PythonAnywhere.
  - **Groq API (Llama-3.3-70B-Versatile)**: I chose this 70-billion-parameter LLM for its high accuracy and 131,072-token context window, optimizing it for criteria generation and refinement.
  - **TF-IDF Vectorization**: I implemented TF-IDF in filterSimilarAC over transformer-based embeddings for computational efficiency, enabling fast similarity checks.
- **Frontend**:
  - **React and Forge UI**: I integrated these for a reactive, Jira-compatible interface, ensuring seamless user interaction.
  - **Forge Bridge**: I used invoke and view for backend communication and Jira context access.
  - **No External CSS**: I adhered to Forge's styling tokens due to platform restrictions, ensuring consistent design.

Workflow Example

- A user selects a Jira issue with the summary "As a manager, I want to extract supplier PDFs, in order to streamline vendor data processing."
- In index.jsx, the user clicks "Generate Criteria," triggering startGeneration.
- My backend logic in GenerateACByCUU processes the story, generating initial criteria (e.g., "System processes PDFs"), refining them from QA ("System processes 10 PDFs in under 5 seconds") and team perspectives ("System supports PDF formats up to 10MB").
- The filterSimilarAC function, which I developed, removes duplicates, producing a final set of criteria.
- The frontend polls checkGenerationStatus, retrieves the JSON output, and displays criteria as a checklist. Users mark criteria as complete, updating the ProgressBar.

Error Handling and Feedback

- **Backend**: I implemented error handling for API failures, with retries and logging for debugging. JSON outputs are validated for frontend compatibility.
- **Frontend**: I used SectionMessage for success/error notifications (e.g., "Successfully generated 3 acceptance criteria!") and Spinner for loading states, with generationProgress tracking during polling.

4.4. Module 2: Story Point Estimation

4.4.2. Modular Decomposition

The Story Point Estimation module is decomposed into four sub-modules, each corresponding to one of the phases described above. These sub-modules work together to process data, experiment with models, incorporate historical context, and enable cross-project adaptability. Below, each sub-module is detailed to clarify its role and implementation.

4.4.2.1. Initial Exploration

Before any machine learning model can be applied, raw project data, often unstructured and inconsistent, must undergo rigorous preprocessing. This crucial phase transforms the diverse inputs, such as issue descriptions and attributes, into a clean, standardized, and numerical format suitable for algorithmic analysis. The goal is to maximize the predictive power of relevant features while mitigating noise and addressing inconsistencies.

The preprocessing pipeline involves several key steps, starting with an initial exploration of the available attributes. For this module, we utilized the TAWOS dataset, a comprehensive collection of open-source issue data from agile web-hosted projects [20], as provided by the SOLAR-group/TAWOS repository (https://github.com/SOLAR-group/TAWOS).

Correlation Analysis and High Cardinality Columns

Our initial data exploration focused on identifying attributes with potential correlation to story points and examining the unique value counts within high-cardinality categorical columns. This analysis was critical for understanding the data's structure and deciding on appropriate cleaning strategies. Upon inspecting columns like 'Type', 'Priority', and 'Status', we observed a significant number of unique values, indicating high cardinality:

- **'Type' Column:** Exhibited 157,222 unique values.
- **'Priority' Column:** Showed 112,339 unique values.
- **'Status' Column:** Contained 246,183 unique values.

Further investigation into the top 15 most common values for each of these columns revealed important insights:

Table 4.2: *Top 15 Values for 'Type', 'Priority', and 'Status' Columns*

Type	Count	Priority	Count	Status	Count
Closed	111775	Unknown	91619	Closed	61796
Bug	39344	Fixed	60509	Bug	25774

Suggestion	19258	Done	20397	Suggestion	13936
Done	13669	Won't Fix	12890	Gathering Interest	5592
Improvement	8223	Low	12629	Done	5249
Open	6540	Minor	10839	Unknown	4627
Gathering Interest	6016	Medium	10083	Improvement	4608
Story	5650	Duplicate	9501	Open	4472
Unknown	5136	Major - P3	9125	Story	2971
Task	4967	Major	6082	Task	2320
Resolved	4118	Won't Do	4142	Resolved	2096
Complete	2565	High	4105	Fixed	1799
New Feature	2429	Closed	3483	New Feature	1483
Sub-task	1982	Timed out	3239	Gathering Impact	1369
To Do	1844	Complete	3019	Sub-task	991

Initially, we hypothesized that the 'Type' and 'Priority' columns could serve as potential features for predicting story points. However, this inspection confirmed that while these columns held significant potential for predictive modeling, they required extensive cleaning and consolidation due to the presence of status-like values and ambiguous categories. The 'Status' column, on the other hand, was deemed less useful for predicting story points for new issues.

Detailed Column-by-Column Analysis and Recommendations

Based on the observations, a specific strategy was formulated for each key column:

'Type' Column:

- Potentially Very Helpful. This column directly reflects the nature of the work (e.g., Bug, Story, New Feature), which might correlates with effort.
- Categories like 'Bug', 'Suggestion', 'Improvement', 'Story', 'Task', and 'New Feature' might be highly relevant for estimation.
- **Data Cleaning Needed:** Many values (e.g., 'Closed', 'Done', 'Resolved', 'Complete') describe the *state* of a task rather than its original *type*. These needed to be cleaned or consolidated to retain the original issue type.

'Priority' Column:

- Potentially Helpful. Issue priority often indicates complexity or urgency, which can influence story points.
- 'Low', 'Minor', 'Medium', 'High', and 'Major' directly provide a scale for urgency/importance.

  Data Cleaning Needed:
- A high number of 'Unknown' values (91,619) required careful handling, potentially through imputation or exclusion.
- Similar to 'Type', this column contained status-like values ('Fixed', 'Done', 'Won't Fix', 'Closed'), which needed to be addressed.
- Ambiguities like 'Major' and 'Major - P3' needed consolidation into a single 'Major' category to avoid redundancy and improve consistency.

'Status' Column:

Probably Not Helpful for New Issues as the most frequent values ('Closed', 'Done', 'Resolved', 'Fixed') represent a state *after* an issue has been estimated and completed. When estimating a *new* issue, its status will typically be 'To Do' or 'Open'. Therefore, historical status data provides little predictive value for future estimations. Many values in 'Status' (e.g., 'Bug', 'Suggestion') were also present in the 'Type' column, indicating mixed semantics and suggesting that 'Type' was the more appropriate feature for this information.

Conclusions based on analysis:

- **Focus on 'Type' and 'Priority':** These were identified as the most valuable features for story point prediction.
- **Perform Data Cleaning and Consolidation:** A key step was to create a mapping to consolidate similar values in 'Type' and 'Priority'. For instance, in 'Priority', values like 'Fixed', 'Done', 'Won't Fix', 'Closed', and 'Complete' could be grouped or removed if they didn't contribute to the initial estimation.
- **Feature Engineering:** After cleaning, these categorical columns would be converted into a numerical format, typically using one-hot encoding, to make them compatible with machine learning models.
- **Ignore 'Status':** For the purpose of predicting story points on *new* issues, the 'Status' column was deemed unsuitable and was dropped from the feature set.

Correlation Analysis

Table 4.3: *Top 20 Positive Correlations.*

Feature	Value
Story_Point	1
In_Progress_Minutes	0.438474
Type_Other_Type	0.00098
Priority_Other_Priority	0.000422

Resolution_Time_Minutes	0.000038
Timespent	0.000008
Total_Effort_Minutes	0.000006
Type_Sub-task	-0.000035
Type_Suggestion	-0.000038
Priority_Low	-0.000083
Type_New Feature	-0.000148
Priority_Minor	-0.000175
Priority_Medium	-0.000181
Type_Improvement	-0.000249
Priority_Major	-0.000263
Type_Task	-0.000268

During correlation analysis, the In_Progress_Minutes attribute showed a strong positive correlation with Story_Point (0.438). This seemed promising initially, as it logically suggests that issues with higher story points tend to spend more time in progress.

However, a critical realization emerged: In_Progress_Minutes is data that becomes available *only after* an issue has been completed. For predicting story points on a *new* issue that has not yet been started, this information is inherently unavailable. Including such a feature would constitute "data leakage," leading to an artificially inflated model performance during training that would not generalize to real-world predictions.

Conclusion: Despite its high correlation, In_Progress_Minutes was identified as an unusable feature for our model due to data leakage.

Final Choice: Combining 'Title' and 'Description'

After analyzing various numerical and categorical features, it became evident that the 'Title' and 'Description' columns were likely the most potent predictors of story points. These columns, despite being unstructured text, contain the core information about an issue's intent and complexity.

A decision was made to combine the 'Title' and 'Description' columns into a single text feature. This strategy offers several benefits:

- **Title Provides Intent:** The 'Title' offers a concise, high-level summary, immediately conveying the core "what" of the task. For example, "Fix stream failover" provides a quick understanding of the issue.

- **Description Provides Context and Complexity:** The 'Description' elaborates on the 'Title', providing crucial details, technical specifications, stack traces, and steps to reproduce. A lengthy, detailed description often signals a more complex problem requiring higher story points.
- **Synergy:** By combining them, the machine learning model gains a comprehensive text document for each issue, allowing it to learn patterns from both the high-level intent and the underlying technical complexities.

Preprocessing Steps

To prepare the combined 'Title' and 'Description' text for machine learning models, the following steps were applied:

1. **Removing Punctuations:** All punctuation marks were removed from the text to reduce noise and standardize word representations.

2. **Filter Story Points (SP):** The target variable, Story_Point, was filtered to retain only standard Fibonacci values commonly used in Scrum: {1, 2, 3, 5, 8}. Non-Fibonacci values (e.g., 4, 6, 7) and story points exceeding 8 were removed, as these often represent outliers or non-standard estimations.

3. **Resolve Inconsistencies:** User stories with identical text content but conflicting story point values were excluded to ensure data integrity and model consistency.

4. **Encode SP Values:** For machine learning compatibility, the filtered Fibonacci story point values were mapped to numerical labels: {1, 2, 3, 5, 8} were encoded as {0, 1, 2, 3, 4} respectively.

5. **Combining 'Title' + 'Description':** As discussed, the content of the 'Title' and 'Description' columns was concatenated into a new column, typically named 'Text', which served as the primary input feature for text-based models.

6. **Token Limit:** To manage computational resources and focus on the most relevant information, the input text for each story was truncated to the first 500 tokens.

7. **TF-IDF Required Preprocessing:** Before applying Term Frequency-Inverse Document Frequency (TF-IDF) vectorization, additional text preprocessing steps were performed:

    - **Convert to Lowercase:** All text was converted to lowercase to treat words uniformly regardless of capitalization.

    - **Remove Numbers and Punctuation:** Numeric digits and remaining punctuation were stripped.

    - **Tokenize:** The text was broken down into individual words or tokens.

- **Remove Stop Words:** Common words (e.g., "the", "is", "and") that carry little semantic meaning for prediction were removed.
- **Apply Lemmatization:** Words were reduced to their base or dictionary form (e.g., "running" to "run") to group inflected forms together and reduce vocabulary size.
- **Rejoin Tokens:** The processed tokens were rejoined into a clean string for TF-IDF vectorization.

This preprocessing pipeline ensures that the textual and numerical data are optimized for the subsequent machine learning stages, laying a robust foundation for accurate story point estimation.

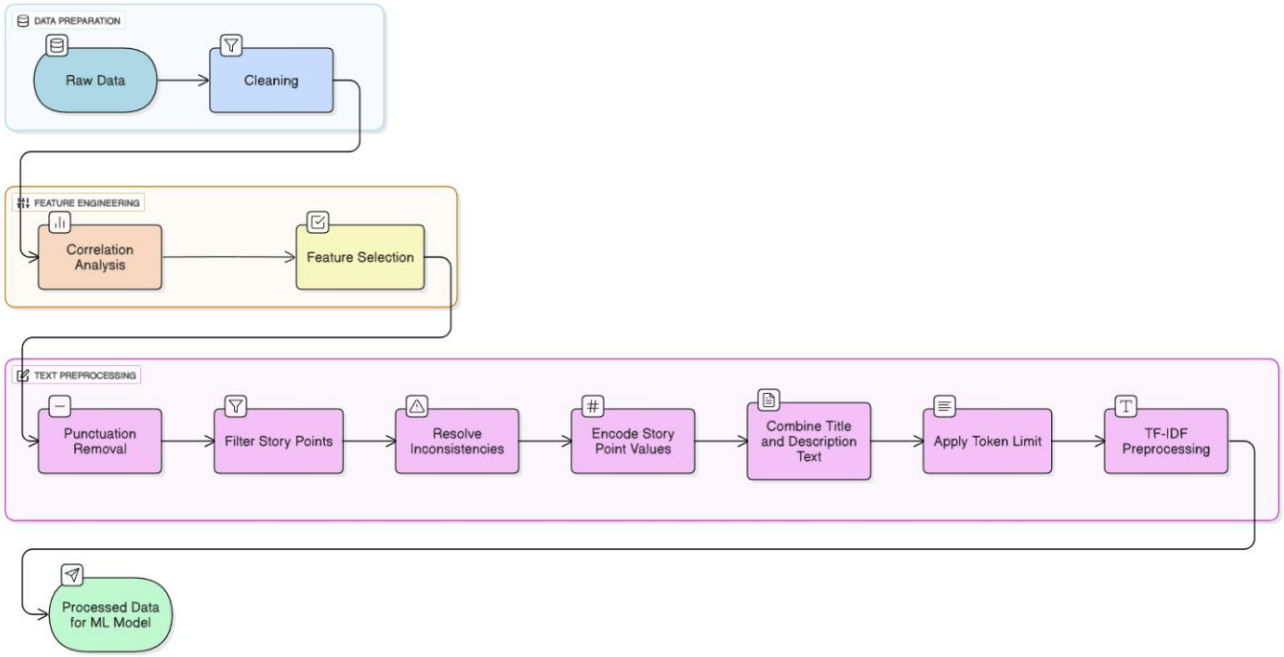

Figure 4.6: *A flow diagram illustrating the preprocessing steps.*

4.4.2.3. Project-Specific Estimation with Historical Context

In this section, we describe the project-specific estimation part, where we developed and implemented four models: an Enhanced Multi-Layer Perceptron (MLP), a Standard MLP, DEEP SE, and FastText + SVM. Our main contribution is the Enhanced MLP, which uses context-aware features to improve predictions.

1. **Enhanced MLP**: Our main contribution, which uses extra information from similar past user stories to improve predictions.
2. **Standard MLP**: A simpler version that only uses the user story's text, serving as a baseline.
3. **DEEP SE**: A complex deep learning model that uses Long Short-Term Memory (LSTM) networks and attention mechanisms.
4. **FastText + SVM**: A classical machine learning approach using FastText embeddings and Support Vector Machines (SVM).

How the Enhanced MLP Works

The Enhanced MLP is our primary model for project-specific Story Point Estimation. It uses a combination of the user story's text and information from similar past user stories within the same project to make predictions. Below, we explain the flow of the process step by step, as shown in the following Figure.

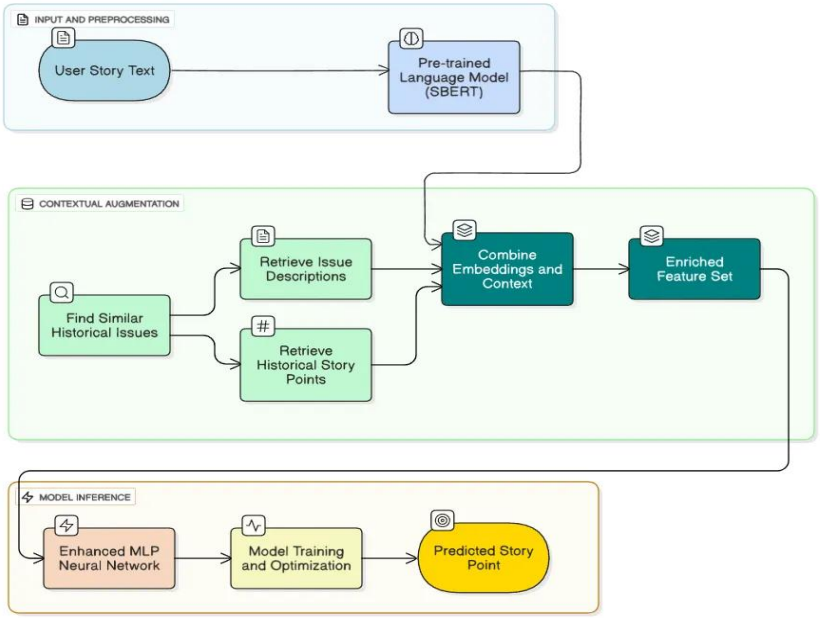

Figure 4.8: *Flow of the Enhanced MLP process for story point estimation.*

Step 1: Converting Text to Numbers (SBERT Embedding)

We start with the user story's title and description, which are text. To process this text with machine learning, we convert it into numbers using SBERT (Sentence-BERT). SBERT turns text into a 384-dimensional vector (a list of 384 numbers) that captures the meaning of the text. Similar user stories have similar vectors.

For example, a user story like "Add login button to homepage" becomes a vector of 384 numbers. We create embeddings for all user stories in a project.

Step 2: Finding Similar Past User Stories

We look at past user stories in the same project that are similar to the new one. We use cosine similarity to measure how close two embeddings are. For each new user story, we find the top 3 most similar past user stories (k=3).

For example, if the new user story is about adding a login button, we might find past user stories about adding a signup button or fixing a login page. These similar stories provide context about typical effort.

Step 3: Building Context-Aware Features

From the 3 most similar user stories, we create extra features:

- **Weighted Context Vector**: We combine the embeddings of the similar user stories into one vector, giving more weight to the most similar ones. This vector (384 numbers) captures the context of similar tasks.
- **Statistical Features**: We calculate three numbers from the story points of the similar user stories:
    - Mean (average) story point value.
    - Median (middle) story point value.
    - Maximum story point value.

For example, if the similar user stories have story points of 2, 3, and 5, the mean is 3.33, the median is 3, and the maximum is 5.

Step 4: Combining All Features

We combine three sets of numbers to create a final feature vector:

- The original SBERT embedding (384 numbers).
- The weighted context vector (384 numbers).
- The statistical features (3 numbers).

This results in a 771-number vector (384 + 384 + 3), which we feed into the neural network.

Step 5: Predicting Story Points with the MLP

The Enhanced MLP is a neural network with three layers:

- **First layer**: Reduces the 771-number vector to 256 numbers.
- **Second layer**: Reduces the 256 numbers to 128 numbers.
- **Third layer**: Outputs 1 number, the predicted story point.

The network uses ReLU (for non-linear patterns), dropout (to prevent overfitting), and Adam optimization (for effective learning). The predicted number (e.g., 2.7) is rounded to the nearest valid story point (1, 2, 3, 5, or 8), so 2.7 becomes 3.

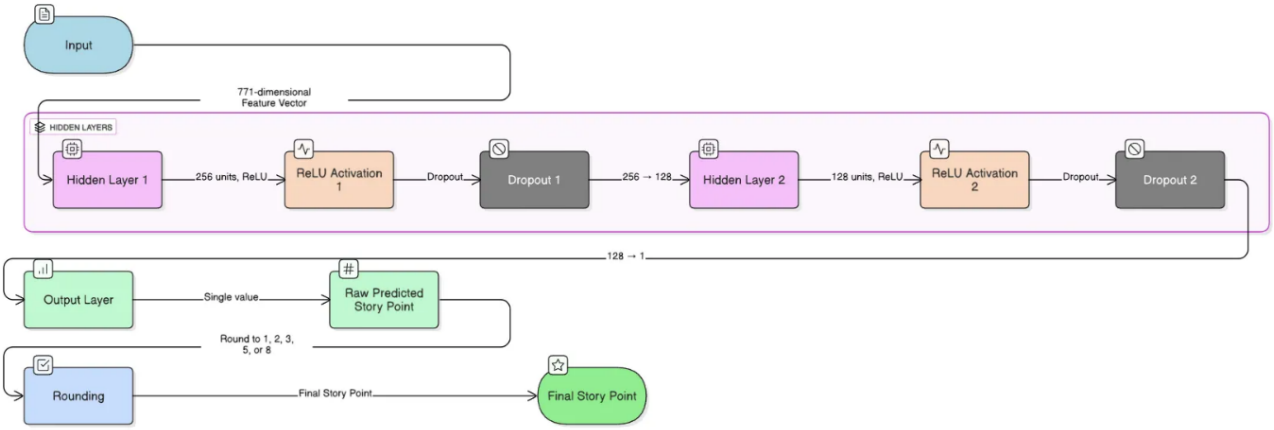

Figure 4.9: *The Enhanced MLP Architecture*

Training the Enhanced MLP

To make the Enhanced MLP accurate, we trained it using data from 35 software projects, each with many user stories and their story points. Here's how we trained it:

1. **Data Splitting**: For each project, we split the data into:
    - Training set (60%): To teach the model.
    - Validation set (20%): To tune settings.
    - Test set (20%): To check final performance.
2. **Hyperparameter Tuning**: We tested settings to find the best ones:
    - **k**: Number of similar user stories (3 or 5). k=3 was best.
    - **Learning rate**: How fast the model learns (0.001 or 0.0005). We used 0.0005.
    - **Dropout rate**: To prevent overfitting (0.3 or 0.5). We used 0.3.
    - **Weight decay**: To keep the model simple (1e-5).
3. **Training Loop**: We trained for up to 50 rounds (epochs), stopping early if the model stopped improving on the validation set to prevent overfitting.
4. **Evaluation**: We used two metrics:
    - **Mean Absolute Error (MAE)**: Average difference between predicted and actual story points.
    - **Accuracy**: Percentage of predictions matching the actual story point.

Table 4.21*: Performance of the Enhanced MLP on selected projects.*

Project	MAE	Accuracy
JAVA	0.119	0.850
USERGRID	1.242	0.380
DURACLOUD	0.830	0.520
CXX	0.964	0.410
EVG	0.591	0.600
Average	**1.389**	**0.412**

Comparing with Other Approaches

To understand how good our Enhanced MLP is, we implemented and tested three other models: Standard MLP, DEEP SE, and FastText + SVM. Below, we explain how each model works and compare their performance.

Standard MLP

The Standard MLP is a simpler version of our Enhanced MLP, used as a baseline to show the value of context-aware features. It only uses the SBERT embedding of the user story's text (384 numbers) without any information from similar past user stories. The process is as follows:

- **Step 1: SBERT Embedding**: Like the Enhanced MLP, it converts the user story's text into a 384-dimensional vector using SBERT.
- **Step 2: Neural Network**: The vector is fed into a three-layer neural network:
  - First layer: Reduces 384 numbers to 256.
  - Second layer: Reduces 256 numbers to 128.
  - Third layer: Outputs 1 number, the predicted story point.
- **Step 3: Prediction**: The predicted number is rounded to the nearest valid story point (1, 2, 3, 5, or 8).

The network uses the same techniques (ReLU, dropout, Adam optimization) as the Enhanced MLP but lacks the context vector and statistical features.

Performance: The Standard MLP had an average MAE of 1.524 and accuracy of 0.385, worse than our Enhanced MLP (MAE 1.389, accuracy 0.412). This shows that adding context-aware features improved predictions by about 9%. The Standard MLP struggled on projects with varied story point patterns, as it couldn't use historical context.

DEEP SE

DEEP SE is a complex deep learning model designed for story point estimation. Unlike our MLP models, it processes text differently and uses advanced techniques to capture patterns. Here's how it works:

- **Step 1: Text Processing**: Instead of SBERT, DEEP SE builds a custom vocabulary from the project's user stories and converts text into sequences of word indices. Each word is represented by a number based on its position in the vocabulary.
- **Step 2: Embedding Layer**: The word indices are turned into dense vectors (embeddings) that capture word meanings, learned during training.
- **Step 3: LSTM Layers**: The sequence of word embeddings is fed into Long Short-Term Memory (LSTM) layers, which process the text in order, remembering important parts over long sequences. This helps capture the structure of user story descriptions.
- **Step 4: Attention Mechanism**: An attention layer focuses on the most relevant words in the text, giving them more weight in the prediction. For example, words like "complex" or "database" might be emphasized.
- **Step 5: Deep Layers**: The output from the attention layer goes through several dense layers (like in an MLP) to predict a story point.
- **Step 6: Prediction**: The final number is rounded to the nearest valid story point.

DEEP SE is computationally intensive, requiring a GPU for efficient training, and learns complex patterns automatically without manual feature engineering.

Performance: DEEP SE had an average MAE of 1.456, slightly worse than our Enhanced MLP (1.389). It performed better on larger projects like CXX (MAE 0.304 vs. 0.964) but worse on smaller projects like JAVA (MAE 0.142 vs. 0.119). Our Enhanced MLP is simpler, faster to train, and more interpretable, making it more practical for most projects.

FastText + SVM: FastText + SVM combines FastText embeddings with a Support Vector Machine (SVM) for prediction. It's a classical machine learning approach that uses the same context-aware features as our Enhanced MLP but processes them differently. Here's the flow:

- **Step 1: FastText Embedding**: FastText converts the user story's text into a vector by breaking words into smaller parts (subwords). For example, "login" might be split into "lo", "log", "ogi", "gin". This helps capture meanings of similar words (e.g., "login" and "logging"). The output is a 300-dimensional vector.
- **Step 2: Context-Aware Features**: Like the Enhanced MLP, we find the top 3 similar user stories using cosine similarity. We create:
    - A weighted context vector (300 numbers) from the FastText embeddings of similar user stories.
    - Statistical features (3 numbers: mean, median, max story points).
- **Step 3: Combining Features**: We combine the original FastText embedding (300 numbers), context vector (300 numbers), and statistical features (3 numbers) into a 603-number vector.
- **Step 4: SVM Prediction**: The vector is fed into an SVM with a Radial Basis Function (RBF) kernel. The SVM finds a boundary that best separates user stories with different story points, predicting a number that is rounded to the nearest valid story point.

SVM uses convex optimization, which is stable and naturally regularizes the model to avoid overfitting.

Performance: FastText + SVM had the best average MAE of 1.245, beating our Enhanced MLP (1.389). It performed better on most projects, like JAVA (MAE 0.043 vs. 0.119) and CXX (MAE 0.217 vs. 0.964). Its success comes from FastText's ability to handle software-specific terms and SVM's robustness. However, our Enhanced MLP is more interpretable and easier to adjust.

Table 4.22: *Comparing the performance of MLP, DEEP SE, and SVM.*

Approach	***Average MAE***	***Average Accuracy***
Enhanced MLP	1.389	0.412
Standard MLP	1.524	0.385
DEEP SE	1.456	0.390
FastText + SVM	1.245	0.435

Implementation Details: We implemented all models using Python. The Enhanced MLP and Standard MLP used PyTorch for neural networks, DEEP SE used TensorFlow, and FastText + SVM used the FastText library and scikit-learn. Below are the key details.

Data Preparation: Using the TAWOS dataset we mentioned earlier, and after applying preprocessing, we created embeddings (SBERT for MLPs and DEEP SE, FastText for SVM). We skipped projects with fewer than 20 user stories.

Core Code: For each model:

- **Enhanced MLP**: Used cosine similarity for similar issues, created context-aware features, and trained a 3-layer MLP.
- **Standard MLP**: Used only SBERT embeddings and a 3-layer MLP.
- **DEEP SE**: Built a custom vocabulary, used LSTM and attention layers, and trained deep layers.
- **FastText + SVM**: Created FastText embeddings, used context-aware features, and trained an SVM with an RBF kernel.

Results Analysis: Our Enhanced MLP performed well, with an average MAE of 1.389 and accuracy of 0.412. Key findings:

- **Project Size**: It worked best on small projects (< 200 user stories), like JAVA (MAE 0.119). FastText + SVM was better on larger projects.
- **Context Window (k)**: k=3 gave better results than k=5 or k=7, as larger windows added noise.
- **Feature Importance**: The context vector improved MAE by 5.2%, and statistical features added 3.9%.

Lessons Learned: We learned several important things:

1. **Context Matters**: Using similar past user stories improved predictions by over 20% in Enhanced MLP and FastText + SVM.
2. **Keep It Simple**: Smaller context windows (k=3) worked better.

3. **Project-Specific Models**: Each project needed its own model due to unique story point patterns.
4. **Classical Methods Are Strong**: FastText + SVM outperformed neural models, showing simple methods can be powerful.

4.4.2.4. Cross-Project Generalization with Incremental Learning

- This section highlights the final approach of our module: **cross-project generalization using incremental learning**. This novel methodology was formalized in a research paper, a full version of which is included in Appendix D for reference [23].

We found that earlier methods had problems:

- **Random projects don't make sense**: Testing on random projects isn't realistic because each project has unique patterns, making predictions unreliable.
- **Regular project-specific learning isn't practical**: Splitting a project's data into 60% training, 20% validation, and 20% testing doesn't work for new projects with no data. Also, retraining a model from scratch each time new data arrives is too slow for agile sprints, which happen every 1–2 weeks.

To solve these issues, we explored **incremental learning**, where the model updates itself with new data without starting over. We discovered that Support Vector Machines (SVMs), a common model, don't support incremental learning. Instead, we used two models: **PassiveAggressiveClassifier** for classification (predicting story points as categories) and **SGDRegressor** for regression (predicting story points as numbers). These models are fast, adapt to changing data, and work well for real-world agile projects. This approach makes our module practical for new projects, ensuring accurate and efficient story point predictions.

How Incremental Learning Works

Our incremental learning approach simulates a real-world scenario where a team starts a new project and needs story point estimates from day one. The model begins with knowledge from other projects and updates itself as new data arrives during sprints. The following Figure shows the process.

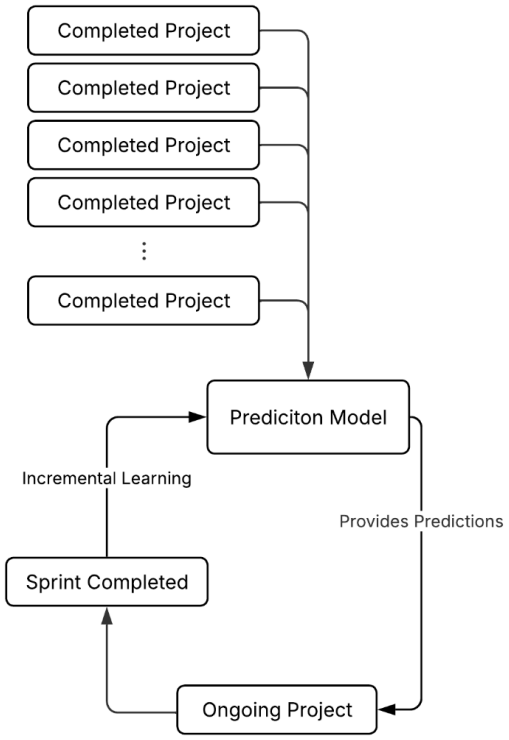

***Figure 4.10**: Flow of the incremental learning process.*

Step 1: Initial Training

We train the model on historical data from multiple past projects. This gives the model a general understanding of story point patterns across different software projects.

Step 2: Batch Updates (Simulating Sprints)

In agile projects, work happens in sprints (1–2 weeks). We treat each sprint's user stories as a "batch" of new data. For each batch:

- The model updates itself using only the new batch's data. This is fast because it doesn't need the entire historical dataset.
- The model predicts story points for the batch before updating, simulating real-time estimation.

Step 3: Prediction

The model predicts story points for each user story in the batch, using the text (title and description) as input. For classification, it predicts a category (e.g., 1, 2, 3, 5, or 8). For regression, it predicts a number, which is rounded to the nearest Fibonacci value.

Models Used

- **PassiveAggressiveClassifier**: Treats story points as categories (1→0, 2→1, 3→2, 5→3, 8→4). It updates its weights only when it makes a mistake, staying “passive” if correct and “aggressive” if wrong. This makes it efficient and good at adapting to changes.
- **SGDRegressor**: Treats story points as numbers. It uses Stochastic Gradient Descent to update weights incrementally, optimizing a loss function to minimize prediction errors.

Experimental Setup

We designed our experiments to mimic a real-world scenario where a new project starts with no data. For each experiment:

- **Test Project**: We picked one project (e.g., TIMOB) as the test project.
- **Training Data**: We trained the initial model on the remaining 39 projects.
- **Batch Processing**: We processed the test project in batches, like sprints, updating the model after each batch.

We compared our incremental models against two baselines:

- **Static Model**: Trained once on historical data and never updated. This is like using a fixed model without adapting to the new project.
- **Full Retrain Model**: Retrained from scratch after each batch, using all historical data plus all batches so far. This is accurate but very slow.

For classification, the baseline was an SVM with a linear kernel. For regression, it was a Support Vector Regressor (SVR) with an RBF kernel (C=1, gamma=scale).

Hyperparameter Tuning

We used **RandomizedSearchCV** to find the best settings for our models, testing different values for:

- TF-IDF: max_features, ngram_range, min_df, max_df.
- PassiveAggressiveClassifier: C, max_iter, class_weight, loss function.
- SGDRegressor: alpha, max_iter, learning_rate, eta, loss function.

We used **GroupKFold** for cross-validation to ensure all data from one project stayed in either the training or validation set, preventing data leakage and mimicking real-world conditions.

Results and Analysis

We tested our approach on several projects, including TIMOB. The incremental models were compared to the static and full retrain models based on accuracy (for classification), Mean Absolute Error (MAE, for regression), and training time.

Classification Results

Table 4.23: *The results of Incremental Learning for the TIMOB project (classification).*

Batch	Incremental Accuracy	Static Accuracy	Full Retrain Accuracy	Incremental Time (s)	Full Retrain Time (s)
1	0.2385	0.2176	0.2929	0.0246	605
2	0.2762	0.1632	0.2678	0.0240	609
5	0.3264	0.1841	0.3264	0.0280	629
8	0.3556	0.2176	0.3431	0.0221	665
10	0.3264	0.2176	0.3640	0.0245	710

- **Accuracy**: The incremental model (PassiveAggressiveClassifier) had accuracy close to or matching the full retrain model (e.g., 0.3264 vs. 0.3264 in batch 5) and always beat the static model (e.g., 0.3264 vs. 0.2176 in batch 10).
- **Training Time**: The incremental model updated in ~0.02–0.03 seconds per batch, while the full retrain model took ~600–710 seconds. This makes incremental learning over 99% faster.
- **Concept Drift**: The static model's accuracy dropped over time (e.g., 0.2176 to 0.1632), showing it couldn't handle changes in data patterns (concept drift). The incremental model adapted, maintaining or improving accuracy.

Regression Results

Table 4.24: *The results of Incremental Learning for the TIMOB project (regression).*

Batch	*Incremental MAE*	*Full Retrain MAE*	*Incremental Time (s)*	*Full Retrain Time (s)*
1	1.8185	1.9070	0.0557	923.3
2	1.8359	1.9705	0.0578	951.0
5	1.8627	1.9389	0.0519	974.7
8	1.7813	1.8470	0.0521	1037.5
10	1.7832	1.8856	0.0563	1067.8

- **MAE**: The incremental model (SGDRegressor) had lower MAE (better performance) than the full retrain model in most batches (e.g., 1.7832 vs. 1.8856 in batch 10).
- **Training Time**: Incremental updates took ~0.05–0.06 seconds, while full retraining took ~923–1067 seconds, again over 99% slower.
- **Concept Drift**: The static model (not shown) performed poorly over time, while the incremental model adapted to data changes, keeping MAE stable or improving it.

Advantages of Incremental Learning

Our incremental learning approach is ideal for agile projects because:

1. **Handles New Projects**: By starting with historical data from other projects, it provides reasonable estimates from day one, unlike project-specific models that need local data.
2. **Adapts to Change**: It handles concept drift, where data patterns change over time, by updating with each batch. Static models fail here, as seen in their dropping accuracy.
3. **Fast and Scalable**: Updates take seconds, not minutes or hours, unlike full retraining. This fits the fast pace of agile sprints and scales to large datasets.
4. **Efficient**: It uses less memory, as it doesn't need the entire dataset for updates, unlike full retraining.

Implementation Details: We implemented the incremental models using Python and scikit-learn:

- **PassiveAggressiveClassifier**: For classification, with TF-IDF vectors as input.
- **SGDRegressor**: For regression, with TF-IDF vectors and rounded outputs.