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
Mohammad Alomar
Mennatallah Ahmed
Mostafa Hani

Supervised by

Dr. Ahmed Darwish

July 2025

Abstract

Agile software development, particularly the Scrum framework, relies on manual, time-consuming tasks such as writing user stories, estimating effort, assigning tasks, and documenting meetings, which are prone to human bias and inconsistency. This project, "Aigile," addresses these challenges by introducing an AI-powered platform to automate and streamline key Scrum activities. The primary objective of Aigile is to enhance the efficiency, accuracy, and data-driven nature of agile project management by leveraging advanced Artificial Intelligence.

To achieve this, Aigile integrates four distinct modules into the Jira environment. Module 1 utilizes a Large Language Model (LLM), Llama-3.3-70B-Versatile, to automatically generate well-structured user stories and acceptance criteria from raw project requirements. Module 2 employs a combination of machine learning models, including a project-specific FastText + SVM model and a cross-project incremental learning model (SGDRegressor), to provide accurate and adaptive story point estimations. Module 3 introduces an online learning system for task assignment, using an AdaBoost ensemble model that adapts to changing team dynamics and developer expertise. Finally, Module 4 provides both extractive (TF-IDF, TextRank) and abstractive (fine-tuned BART) summarization of sprint review meetings.

The project's output is a suite of five integrated Jira plugins and two local GUI applications that provide a seamless user experience. Testing was conducted using a combination of real-world datasets (TAWOS, Apache Jira Issue Tracking), qualitative surveys with 30 Agile practitioners, and quantitative metrics (MAE, accuracy, ROUGE scores). The results confirm the system's effectiveness: the story point estimation model achieved a low MAE of 1.245, the task assignment model reached a peak accuracy of 74.7%, and the generative modules produced high-quality artifacts that were favorably evaluated by industry professionals. Aigile successfully demonstrates the feasibility and value of integrating AI into the agile workflow, offering a powerful tool to support modern software development teams.

الملخص

في عالم تطوير البرمجيات الحديث، تعتمد منهجيات العمل Agile، وخصوصًا إطار "سكرم"، على سلسلة من المهام اليدوية التي تتسم بكونها مُستهلِكة للوقت وعُرضة للأخطاء البشرية، مثل صياغة المهمة، وتقدير مدى صعوبة المهمة، وتوزيعها على الفريق، وتلخيص وقائع الاجتماعات. يأتي مشروعنا "Aigile" كحلٍ لهذه التحديات، عبر تقديم منصة ذكية تستخدم الذكاء الاصطناعي لأتمتة هذه العمليات المحورية وتبسيطها، بهدف تعزيز الكفاءة والدقة ودعم اتخاذ القرارات المبنية على البيانات.

لتحقيق هذه الغاية، يدمج "Aigile" أربعة أقسام من المشروع متخصصة ضمن بيئة "جيرا". القسم الأول من المشروع يختص بتوليد المهمة ومعايير القبول تلقائيًا من المتطلبات الأولية للمشروع، وذلك باستخدام النموذج اللغوي الكبير Llama-3.3-70B-Versatile. القسم الثاني من المشروع يقدم تقديرات دقيقة لمدى صعوبة المهمة من خلال الجمع بين نماذج تعلم الآلة المخصصة للمشروع (FastText + SVM) والنماذج التزايدية القادرة على التعلم من مشاريع متعددة (SGDRegressor). أما القسم الثالث من المشروع، فيقدم نظامًا ذكيًا لتوزيع المهام يعتمد على نموذج AdaBoost متعدد المصنفات، الذي يتأقلم مع تغير خبرات المطورين وديناميكيات الفريق. وأخيرًا، يوفر القسم الرابع من المشروع نوعين من التلخيص لنصوص اجتماعات مراجعة الاسبرنت: تلخيص استخلاصي (Extractive) قائم على خوارزميات إحصائية (TF-IDF, TextRank)، وتلخيص تجريدي (Abstractive) يعتمد على نموذج BART المتقدم والمُدرَّب خصيصًا لهذه المهمة.

تتجسد مخرجات المشروع في خمس إضافات برمجية (Plugins) متكاملة مع "جيرا"، بالإضافة إلى تطبيقين مكتبيين، لتقديم تجربة استخدام متكاملة. تم تقييم النظام عبر اختبارات شاملة استخدمت بيانات واقعية من مشاريع مفتوحة المصدر، واستطلاعات رأي مع ٣٠ خبيرًا في منهجيات العمل Agile، ومقاييس أداء كمية دقيقة. أثبتت النتائج فعالية النظام، حيث بلغ متوسط الخطأ المطلق في تقدير مدى صعوبة المهمة 1.245، ووصلت دقة توزيع المهام إلى 74.7%، بينما حازت المخرجات التوليدية للنظام على تقييمات إيجابية من المتخصصين في الصناعة. وبهذا، يبرهن مشروع "Aigile" على القيمة الحقيقية والجدوى العملية لدمج الذكاء الاصطناعي في صميم عمليات تطوير البرمجيات، مقدمًا أداة فعالة لدعم الفرق العصرية.

ACKNOWLEDGMENT

We would like to express our sincerest gratitude to the individuals and institutions whose support and guidance were invaluable to the completion of this graduation project.

First and foremost, we extend our deepest appreciation to our supervisor, **Dr. Ahmed Darwish**, for his unwavering support, insightful guidance, and expert mentorship. His encouragement and critical feedback were instrumental in shaping the direction of this project and navigating the complex challenges we faced.

We are also grateful to the faculty and staff of the **Department of Computer Engineering at Cairo University** for providing us with a strong academic foundation and the resources necessary to undertake this research.

We would like to thank the 30 anonymous Agile practitioners who generously gave their time to participate in our surveys. Their professional feedback was crucial for validating our work and providing real-world context to our results.

Finally, we wish to thank our families and friends for their constant encouragement, patience, and understanding throughout this demanding journey. This accomplishment would not have been possible without their support.

Dedication

This work is respectfully dedicated to our parents, whose support, guidance, and sacrifices have been instrumental throughout our academic journey. We also extend our sincere gratitude to our mentors and instructors, whose valuable insights and encouragement have greatly contributed to the realization of this project. Finally, we dedicate this work to the enduring pursuit of knowledge, which continues to inspire inquiry, discipline, and innovation.

Table of Contents

List of Figures

List of Tables

List of Abbreviation

ADF	Atlassian Document Format
ADWIN	ADaptive WINdowing
AI	Artificial Intelligence
API	Application Programming Interface
ARPU	Average Revenue per User
AWS	Amazon Web Services
BART	Bidirectional and Auto-Regressive Transformers
BERT	Bidirectional Encoder Representations from Transformers
Capex	Capital Expenditure
CUU	Create-Update-Update
Deep-SE	Deep Software Effort Estimation
GPT	Generative Pre-trained Transformer
GUI	Graphical User Interface
ILSVM	Incremental Learning Support Vector Machines
INVEST	Independent, Negotiable, Valuable, Estimable, Small, Testable
JQL	Jira Query Language
LDA	Latent Dirichlet Allocation
LLM	Large Language Model
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MdAE	Median Absolute Error
MLP	Multi-Layer Perceptron
MOSCOW	Must-have, Should-have, Could-have, and Won't-have
NLP	Natural Language Processing
Opex	Operational Expenses
PRDs	Product Requirement Documents
QA	Quality Assurance
RBF	Radial Basis Function
RHN	Recurrent Highway Networks
RNNs	Recurrent Neural Networks
ROCAUC	Receiver Operating Characteristic Area Under the Curve
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
S3	Simple Storage Service
SA	Standardized Accuracy
SBERT	Sentence-BERT
Seq2Seq	Sequence-to-Sequence
SGDRegressor	Stochastic Gradient Descent Regressor
SHAP	SHapley Additive exPlanations
SLR	Systematic Literature Review

SP	Story Points
SVM	Support Vector Machine
SVR	Support Vector Regression
T5	Text-To-Text Transfer Transformer
TELOS	Technical, Economic, Legal, Operational, and Scheduling
TF-IDF	Term Frequency-Inverse Document Frequency
UI	User Interface
XGBRegressor	eXtreme Gradient Boosting Regressor

List of Symbols

α	The weight of a classifier in an AdaBoost ensemble.
δ	The confidence parameter used in statistical tests like ADWIN and Hoeffding bound.
ε	The margin of tolerance in Support Vector Regression (SVR) that defines the error-insensitive tube.
λ	The decay factor used in the recency heuristic for online learning models.
μ	The mean value of a data window, used for drift detection.
ξ	A slack variable in SVM/SVR representing the magnitude of deviation or error beyond the margin.
σ2	The variance of values within a data window.
b	The bias term in a linear model.
C	A hyperparameter in SVM/SVR that controls the trade-off between model complexity and error tolerance.
c	A class or category, such as a developer assignee.
ct	The cell state of an LSTM, representing its long-term memory at time t.
d	The damping factor in the TextRank algorithm.
df	Document Frequency, the number of documents in which a term appears.
ft	The output of the forget gate in an LSTM at time t.
ht	The hidden state of an LSTM, representing its output and short-term memory at time t.
it	The output of the input gate in an LSTM at time t.
M	The total number of learners (e.g., trees) in an ensemble model.
m	The size of a data window in online learning algorithms.
N	The total number of documents in a corpus.
n	The number of observations or data points.
ot	The output of the output gate in an LSTM at time t.
R	The range of an attribute's values (max - min).
t	A term or word in a document.
vi	A node in a graph, typically representing a sentence in TextRank.
w	The weight vector in a linear model.
wd	The recency-based weight assigned to a developer.
x	An input feature vector.
xt	The input data at a specific time step t.
y	The true label or value for a data point.

Contacts

Team Members

Name	Email	Phone Number
Mustafa Tawfiq	mustafa.tawfiq02@eng-st.cu.edu.eg	+20 101 420 8190
Mustafa Hani	mustafa.ali02@eng-st.cu.edu.eg	+20 100 905 6200
Mennatallah Ahmed	Mennatallah.abdo03@eng-st.cu.edu.eg	+20 115 924 0414
Mohammad Alomar	mohammad.alomar03@eng-st.cu.edu.eg,	+20 115 668 9177

Supervisor

Name	Email	Number
Dr. Ahmed Darwish	darwish@ieee.org	+20 122 224 7381

This page is left intentionally empty

Chapter 1: Introduction

In the landscape of modern software development, the Agile methodology—particularly the Scrum framework—has become the standard for teams aiming to deliver high-quality products efficiently and adaptively. However, the operational success of Scrum is heavily relies on a set of recurring, manual processes that, while essential, are often time-consuming and susceptible to human error and bias. Activities such as drafting user stories, estimating task complexity, assigning work to appropriate developers, and documenting key decisions from meetings are fundamental to the agile workflow, but can divert valuable time and energy from core development and strategic tasks. Recognizing this operational friction, the Aigile project was conceived to explore and implement an innovative solution that leverages the power of Artificial Intelligence to automate and enhance these critical Scrum activities.

This chapter introduces the Aigile project, beginning with the motivation behind automating agile processes and the justification for such a system. It then formally defines the project's objectives and the specific problems it aims to solve. Subsequently, the main outcomes of the project are detailed, followed by an overview of this document's organization to guide the reader through the report.

1.1. Motivation and Justification

The primary motivation for the Aigile project stems from the inherent inefficiencies involved in the manual execution of the Scrum framework. While Scrum provides an excellent structure for iterative development, its processes are labor-intensive. Project Managers, Product Owners, and Scrum Masters spend a significant portion of their time on administrative and organizational tasks, which include:

Writing User Stories and Acceptance Criteria: Translating high-level requirements into well-defined, actionable user stories that adhere to the INVEST (Independent, Negotiable, Valuable, Estimable, Small, Testable) criteria is a skilled and time-consuming task.

Effort Estimation: The process of assigning story points is often subjective, leading to inconsistent estimations, prolonged planning meetings (e.g., Planning Poker), and potentially inaccurate sprint forecasting.

Task Assignment: Manually assigning tasks requires a deep understanding of each developer's skills, current workload, and historical performance. In large or dynamic teams, this can be a complex challenge, and suboptimal assignments can lead to delays or reduced quality.

Meeting Documentation: Sprint review meetings generate a wealth of information, but manually transcribing and summarizing key decisions and action items is tedious and prone to omissions.

The Aigile project is justified by the clear need to address these challenges. By automating these tasks, Aigile aims to free up development teams to focus on what they do best: building software. The significance of this project lies in its potential to increase productivity, improve the accuracy and consistency of agile planning, reduce human bias in decision-making, and ensure that valuable knowledge is captured efficiently. Ultimately, the usefulness of Aigile lies in its ability to make the agile process itself more agile.

1.2. Project Objectives and Problem Definition

The central objective of the Aigile project is to design, implement, and validate an AI-powered platform that integrates seamlessly with the Jira project management tool to automate four key Scrum activities. The project's objectives as follows:

To automate the generation of high-quality user stories and acceptance criteria from unstructured project requirements, ensuring they align with agile best practices.

To develop an accurate and adaptive story point estimation model that can handle both project-specific and cross-project data, addressing the "cold start" problem in new projects.

To create a dynamic task assignment system that recommends the most suitable developer for each task by learning from historical data and adapting to changes in team structure and expertise.

To implement an effective summarization tool that can distill lengthy sprint review meeting transcripts into concise, actionable summaries.

Based on these objectives, the problem can be formally defined as:

How can Artificial Intelligence be leveraged to create an integrated system that automates the generation of user stories, estimation of story points, assignment of tasks, and summarization of meetings to improve the efficiency, consistency, and data-driven decision-making of software development teams operating within the Scrum framework?

1.3. Project Outcomes

The Aigile project has produced a suite of fully functional software tools and machine learning models that collectively form the Aigile platform. The primary outcomes are:

A Suite of Five Integrated Jira Plugins: A set of applications built with Atlassian Forge that embed Aigile's functionalities directly into the Jira user interface.These includes plugins for user story generation, acceptance criteria generation, story point prediction, story point model training, and task assignment.

Two Standalone GUI Applications: Local desktop applications developed with Python for initiating the training of the task assignment model and for summarizing sprint review transcripts, providing flexibility for offline or specialized use.

A Robust Backend System: A scalable backend architecture comprising a Flask server hosted on PythonAnywhere for serving generative AI tasks and a serverless infrastructure on AWS (API Gateway, Lambda, S3) for hosting and running the predictive machine learning models.

A Set of Trained AI Models: The project delivers several custom-trained and fine-tuned models, including an AdaBoost ensemble for task assignment, an SGDRegressor for story point estimation, and a fine-tuned PEGASUS model for abstractive summarization.

1.4. Document Organization

This report is structured into six chapters and three appendices to provide a comprehensive overview of the Aigile project.

- **Chapter 1: Introduction** provides the motivation, objectives, and outcomes of the project.
- **Chapter 2: Market Visibility Study** surveys the market, identifies target customers, and analyzes competitive products.
- **Chapter 3: Literature Survey** discusses the theoretical background of the technologies used and reviews previous work in related fields.
- **Chapter 4: System Design and Architecture** details the design and implementation of each of the four modules of the Aigile system.
- **Chapter 5: System Testing and Verification** presents the testing methodologies, test cases, and results for each module, as well as the end-to-end integration testing.
- **Chapter 6: Conclusions and Future Work** summarizes the project's achievements, limitations, and potential directions for future research.
- **Appendix A: Development Platforms and Tools** lists the hardware and software used in the project.
- **Appendix B: User Guide** provides step-by-step instructions for using the Aigile platform.
- **Appendix C: Feasibility Study** evaluates the project's viability across technical, economic, legal, operational, and scheduling domains.

Chapter 2: Market Visibility Study

The integration of artificial intelligence into agile project management tools is no longer a futuristic concept, but a rapidly accelerating reality. The market is converging on a powerful value proposition: AI saves time, automates repetitive administrative work, and provides data-driven insights that empower teams to deliver better results, faster. This chapter surveys this dynamic market to situate the Aigile platform within the current competitive landscape. It begins by identifying the target customers best suited to leverage Aigile's unique capabilities. It then provides a detailed market survey of the key commercial competitors, analyzing their AI strategies and products. This analysis highlights a clear distinction between the broad, workflow-oriented AI features offered by major platforms and the deep, specialized, and predictive models that define Aigile's core innovation. This distinction justifies the need for Aigile, not as a replacement for these tools, but as a potential source of high-performance "engine" technology for data-mature organizations.

2.1. Targeted Customers

Aigile is not designed for every agile team. Its value is most pronounced for specific user profiles and organizational contexts where predictability, efficiency, and data-driven decision-making are paramount.

The primary target customers for the Aigile platform are **data-mature, large-scale enterprise organizations** that are already deeply invested in the agile methodology and use platforms such as Jira or Azure DevOps. These organizations often manage multiple complex projects with dozens of teams, where even minor improvements in estimation accuracy or task assignment efficiency can lead to significant cost savings and more reliable delivery timelines. They possess the large, structured historical datasets required to train and fine-tune Aigile's predictive models effectively.

Within these organizations, Aigile is intended for several key roles:

- **Product Owners and Product Managers:** These users benefit directly from **Module 1 (User Story Generation)**, which automates the creation of backlogs from raw requirements, and **Module 4 (Summarization)**, which provides concise, actionable summaries of sprint reviews. This frees them from time-consuming administrative work, allowing them to focus on strategy and stakeholder management.
- **Scrum Masters and Engineering Leads:** These roles benefit from **Module 2 (Story Point Estimation)** and **Module 3 (Task Assignment)**. By providing accurate, unbiased estimates and intelligent assignment recommendations, Aigile helps these leaders facilitate smoother sprint planning, balance workloads, and mitigate risks. The adaptive nature of these models is particularly valuable for handling the dynamic changes in team composition and project scope common in large organizations.

- **Developers:** While not the primary users of the administrative features, developers are the ultimate beneficiaries of a more efficient agile process. More accurate estimations and better task assignments lead to more realistic sprint goals, reduced context-switching, and a more sustainable development pace.

2.2. Market Survey

The competitive landscape for AI in agile project management is dominated by a few large incumbents rapidly integrating AI features, alongside a vibrant ecosystem of niche innovators. The primary battleground is not just the quality of the underlying algorithms, but the seamlessness of their integration into the user's daily workflow.

2.2.1. Atlassian (Jira Intelligence)

Atlassian, the market leader with Jira, has a strategy to deeply embed "Atlassian Intelligence" across its entire product suite.

- **Pros:** Atlassian's key advantage is its "Teamwork Graph," a unique data layer that connects people, work, and knowledge across Jira, Confluence, and Bitbucket. This allows its AI to provide highly contextual assistance, such as translating natural language into complex Jira Query Language (JQL), generating subtasks from a parent issue, and summarizing long comment threads. By positioning AI as a platform-wide utility, Atlassian enhances the value of its entire ecosystem for its massive existing enterprise customer base.
- **Cons:** Atlassian's AI features are primarily "assistive" rather than fully automated for core predictive tasks. For instance, it does not offer a native feature for automated story point estimation or predictive task assignment based on developer expertise. Its focus is on broad, workflow-enhancing features rather than deep, specialized predictive models, creating an opportunity for more advanced tools like Aigile to provide enhanced functionality as a plugin.

2.2.2. ClickUp

ClickUp's market position is built on being a single, unified platform for all types of work, and its AI offering, "ClickUp Brain," reinforces this "all-in-one" strategy.

- **Pros:** ClickUp AI focuses on breadth and ease of use, offering over 100 role-based AI templates to accelerate common tasks, from writing Product Requirement Documents (PRDs) to generating marketing campaign briefs. It excels at content generation, summarization, and automating workflows through user-defined rules, appealing to teams looking to consolidate their tools and automate as much administrative work as possible.
- **Cons:** Like Jira, ClickUp's AI is geared towards general-purpose productivity and lacks the specialized, data-driven models for core agile challenges like story point estimation or drift-aware task assignment. Its value proposition is in breadth of functionality, which may not meet the needs of data-mature teams requiring high-precision predictive insights.

2.2.3. Asana

Asana's AI strategy is tightly aligned with its core brand promise of providing organizational clarity and aligning work with strategic goals.

- **Pros:** "Asana Intelligence" focuses on structuring work and automating project setup. Features like "Smart Projects" (generating a project plan from a simple description) and "Smart Status" (drafting project updates by analyzing progress) help enforce best practices and reduce manual setup. This appeals to cross-functional teams who value structured planning and clear communication.
- **Cons:** Asana is less focused on the specific, granular needs of software development teams. It does not offer features for story point estimation or developer task assignment, which are central to the Scrum process. This makes it a less direct competitor to Aigile, which is explicitly designed for the software engineering workflow.

2.2.4. Niche Innovators: Zenhub and Tara AI

Beyond the major players, several innovative companies are targeting the specific needs of software development teams.

- **Zenhub AI:** Tightly integrated with GitHub, Zenhub is a favorite among developer-first organizations.
    - **Pros:** It offers AI-powered features designed to streamline the developer workflow, including AI-generated acceptance criteria and automated sprint reviews. Its roadmap includes AI-assisted estimation and task assignment, positioning it as a direct competitor to Aigile's core functionalities. Its strength is its deep integration within the developer's primary environment (GitHub).
    - **Cons:** Zenhub's planned estimation and assignment features are described as "AI-assisted," where the AI acts as a virtual team member to provide a benchmark for human discussion. This is less ambitious than Aigile's goal of fully automated prediction, which offers a higher potential for time savings.
- **Tara AI:** This is a product delivery platform focused on providing engineering analytics and intelligence.
    - **Pros:** Tara AI uses AI to analyze data from sources like GitHub and Jira to provide insights into team performance, identify bottlenecks, and automate processes like sprint planning. Its value proposition is centered on data-driven leadership and engineering efficiency.
    - **Cons:** Tara AI is more of an analytics and intelligence platform than a direct project management tool. It provides insights *about* the work, but it does not have the same depth of features for automating the creation and management of the work itself, such as user story generation or abstractive summarization.

2.3. Business Case and Financial Analysis

This section outlines a hypothetical business case for launching "Aigile" as a commercial entity. The proposed strategy is to position Aigile as a premium plugin for the Atlassian Marketplace, focusing on its most innovative and defensible features: **Module 2 (Story Point Estimation)** and **Module 3 (Task Assignment)**.

2.3.1 Business Case

The go-to-market strategy is to target data-mature enterprise customers already using Jira. These organizations feel the pain of estimation inaccuracy and suboptimal task assignment most acutely and have the historical data needed for Aigile's models to provide significant value.

- **Product Offering:** Aigile will be offered as a single, integrated Jira plugin with two main features: AI Story Point Prediction and AI Task Assignment.
- **Pricing Strategy:** A tiered subscription model will be used to cater to teams of different sizes.
  - Pro Plan: $10 per user/month. Includes all features for a single team.
  - Enterprise Plan: Custom pricing (e.g., starting at $5,000/year for 50 users). Includes advanced features like cross-project model training, priority support, and a dedicated customer success manager.
- **Sales Forecast:** The forecast is based on a conservative market penetration strategy, targeting an initial set of early adopters and scaling up as the product gains testimonials and case studies. The projection assumes an average revenue per user (ARPU) of $120/year.

***Table 2.1**: Five-year sales forecast for the Aigile platform.*

Year	*Target New Customers (Enterprise)*	*Target New Users*	*Total Active Users*	*Projected Revenue*
Year 1	5	250	250	$30,000
Year 2	15	750	1,000	$120,000
Year 3	30	1,500	2,500	$300,000
Year 4	50	2,500	5,000	$600,000
Year 5	80	4,000	9,000	$1,080,000

2.3.2 Financial Analysis

This analysis outlines the initial investment (Capex) and recurring costs (Opex) required to launch and operate Aigile as a startup. All figures are in USD.

- **Capex (Capital Expenditure - Year 1, One-Time):**
  - Company Registration & Legal Fees: $5,000
  - Website & Branding Development: $10,000
  - Initial Cloud Infrastructure Setup: $5,000
  - Total Capex: $20,000
- **Opex (Operational Expenses - Annual):**
  - Salaries: A small founding team of 4 (2 Engineers, 1 Product/Marketing, 1 Support) with an average salary of $60,000/year. Total: $240,000.
  - Marketing & Sales: $50,000 per year, focused on content marketing, developer outreach, and participation in agile conferences.
  - Cloud Hosting & API Costs: (AWS, Groq API, etc.) Estimated at $15,000 per year, scaling with usage.
  - Software & Subscriptions: $5,000 per year.
  - Total Annual Opex: $310,000

Cash Flow Projection & Break-Even Analysis

Table 2.2: *Five-year cash flow projection and break-even analysis for Aigile.*

Year	*Projected Revenue*	*Total Expenses (Opex + Y1 Capex)*	*Annual Profit / (Loss)*	*Cumulative Profit / (Loss)*
Year 1	$30,000	$330,000	($300,000)	($300,000)
Year 2	$120,000	$310,000	($190,000)	($490,000)
Year 3	$300,000	$310,000	($10,000)	($500,000)
Year 4	$600,000	$310,000	$290,000	($210,000)
Year 5	$1,080,000	$310,000	$770,000	$560,000

Break-Even Point: Based on this projection, the company operates at a loss for the first three years as it invests in development and market penetration. The business becomes profitable in **Year 4**. The cumulative losses are recovered, and the business achieves its overall break-even point sometime in **Year 5**, at which point it becomes a truly profitable venture. This timeline reflects a typical startup trajectory, requiring initial funding to cover the operational losses in the early years.

Chapter 3: Literature Survey

This chapter provides the foundational context for the Aigile project by surveying essential background knowledge and reviewing current state-of-the-art research. The goal is to situate our work within the broader academic and industry landscape, establish its theoretical underpinnings, and justify the technical approaches implemented in our modules.

The chapter begins with an overview of core concepts, including the Scrum framework and the fundamental machine learning and natural language processing techniques used throughout the project. It then presents a comparative study of recent publications, critically examining previous work in AI-driven backlog generation, story point estimation, task assignment, and meeting summarization to identify current trends and research gaps.

Finally, this survey culminates in a discussion of our implemented approach for each module, demonstrating how the Aigile platform builds upon established research while introducing novel solutions to enhance agile project management.

3.1. Background on Agile Software Development and Scrum Framework

Agile software development is a flexible approach to building software, emphasizing teamwork, adaptability, and delivering working products in short cycles. Scrum, a widely adopted agile framework, organizes work into sprints, typically lasting 1–4 weeks, to deliver incremental improvements.

Key Components of Scrum

- **Roles**:
  - *Product Owner*: Prioritizes tasks in the product backlog based on stakeholder needs.
  - *Scrum Master*: Guides the team, ensuring Scrum practices are followed and obstacles are removed.
  - *Development Team*: Builds the product collaboratively.
- **Artifacts**:
  - *Product Backlog*: A prioritized list of all tasks needed for the product.
  - *Sprint Backlog*: Tasks selected for a specific sprint.
  - *Increment*: The working product delivered at the end of a sprint.
- **Events**:
  - *Sprint Planning*: Planning tasks for the upcoming sprint.
  - *Daily Scrum*: A short meeting to discuss progress and daily goals.
  - *Sprint Review*: Demonstrating completed work and gathering feedback.
  - *Sprint Retrospective*: Reflecting on the sprint to improve future work.

Story Points and Estimation

User stories describe features or requirements in Scrum, and each is assigned *story points* to estimate effort, complexity, and time, often using the Fibonacci sequence (1, 2, 3, 5, 8). Story point estimation is vital for sprint planning but can be subjective and time-consuming when done manually, such as through *Planning Poker*. *Aigile* uses AI to automate this process, making it faster and more consistent.

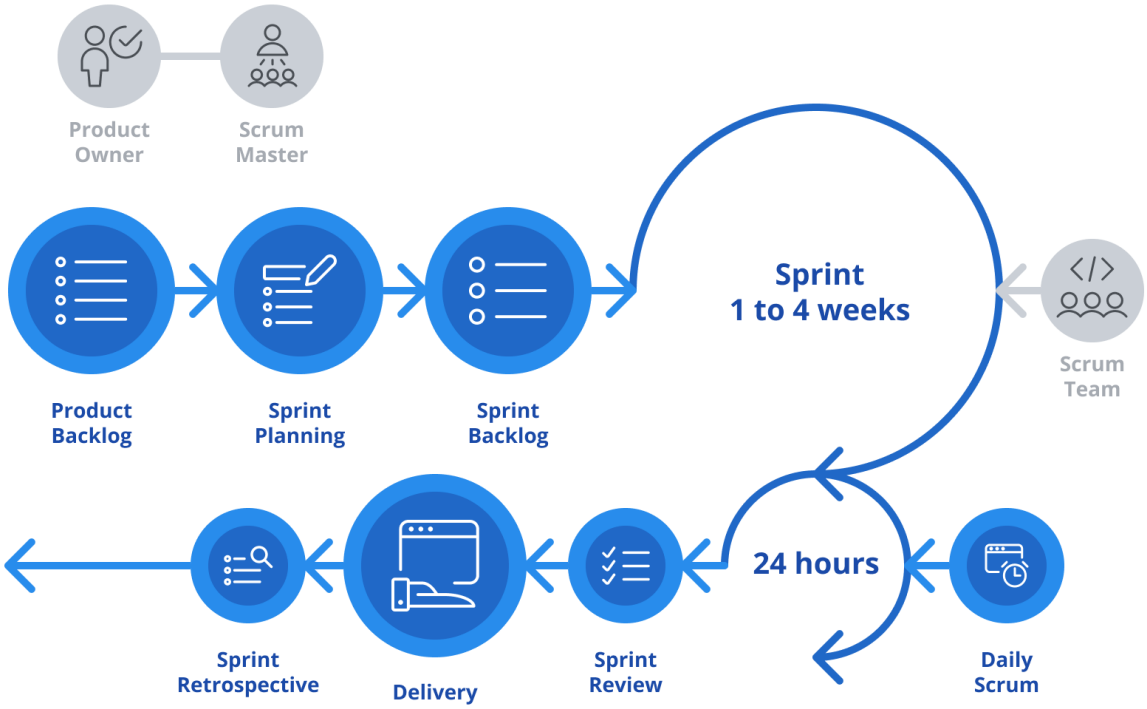


Figure 3.1: *Scrum Framework Overview*

3.2. Technical Background

3.2.1 TF-IDF

TF-IDF (Term Frequency- Inverse Document Frequency) converts text into numerical vectors by measuring the importance of words based on their frequency in a document relative to the entire dataset. It reduces the weight of common words and emphasizes unique ones.

Algorithm:

$$TF(t,d) = \frac{number\ of\ times\ t\ appears\ in\ d}{total\ number\ of\ terms\ in\ d}$$

$$IDF(t) = log\frac{N}{1+df}$$

$$TF-IDF(t,d) = TF(t,d) * IDF(t)$$

Relevance:

TF-IDF is used as a lightweight, effective method to prepare text data, especially for models requiring numerical inputs.

3.2.2. Support Vector Regression (SVR):

Overview:

SVM is a supervised learning algorithm that finds the optimal hyperplane to separate data points into classes, maximizing the margin between them.

Algorithm:

Support Vector Regression (SVR), a powerful algorithm that extends the concepts of Support Vector Machines to regression problems. Unlike conventional regression models that aim to minimize the sum of squared errors for all data points, SVR introduces the concept of an **epsilon-insensitive tube**.

The central solid line, defined by the function y=wx+b, represents our regression model. This line is flanked by two dashed lines at a vertical distance of ±ϵ, forming a channel or "tube." The parameter ϵ defines a margin of tolerance. Any data point (xi ,yi) that lies within this tube is considered to be predicted with zero error.

The model's objective is to find a function that is as "flat" as possible while tolerating deviations up to ϵ. Deviations beyond this margin are penalized using **slack variables**, denoted by ξi for points above the tube and ξi∗ for points below it.

The optimization problem, as stated in the image, is to **minimize** the objective function:

- The term $\frac{1}{2} \; || \, w \, || \; ^2$ is the regularization term. Minimizing the norm of the weight vector, $|| \, w \, ||$, corresponds to maximizing the "flatness" of the function, which helps prevent overfitting.
- The other term represents the empirical error. It is the sum of the slack variables—the cost of the deviations larger than ϵ. The hyperparameter C controls the trade-off between the flatness of the function and the amount up to which deviations larger than ϵ are tolerated.

This optimization is performed under the **constraints** that ensure the predicted value for each data point is within ϵ of its true value, or else the appropriate slack variable is assigned:

$$y_i - wx_i - b \leq \epsilon + \xi_i$$

$$wx_i + b - y_i \leq \epsilon + \xi_i^*$$

$$\xi i, \xi i^* \geq 0$$

Ultimately, the model is defined not by all the data points, but only by those that lie on the boundary of or outside the epsilon-insensitive tube. These are the **support vectors**, which "support" the final regression function.

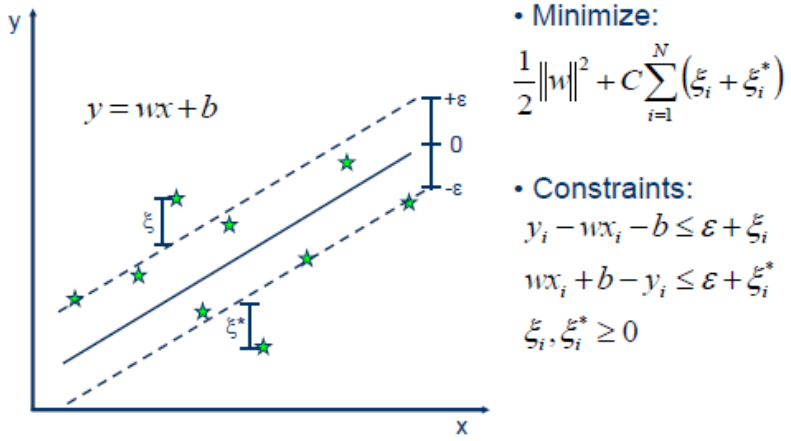


Figure 3.2: *Support Vector Regression (SVR) with epsilon-insensitive tube.*

- **Relevance**: SVR is selected for its ability to predict continuous story point values, which are rounded to Fibonacci numbers.

3.2.3. Support Vector Machine (SVM):

The key Differences from SVR for Classification is that the goal of SVR is **prediction**. It seeks a continuous function to approximate the data. In contrast, the goal of an SVM for classification is **separation**. It seeks a decision boundary to divide data into distinct categories.

1. **The Margin:**
   - In **SVR**, the dashed lines form a "tube" which is a **zone of tolerance**. The model's success is measured by how many points it can fit *inside* this tube.
   - In **SVM for classification**, the margin is a "street" that acts as a **separating barrier** between two classes. The goal is to make this street as wide as possible and to have *no points* within it.
2. **The Error (ξ):**
   - In **SVR**, the slack variable ξ measures the *magnitude of the prediction error* for a point that falls outside the tolerance tube.
   - In **SVM for classification**, ξ measures the *degree of misclassification* for a point that falls within the separating margin or on the wrong side of the decision boundary.
3. **The Function (wx + b):**
   - In **SVR**, y = wx + b is the **regression line** itself, used to predict a value.
   - In **SVM for classification**, wx + b = 0 defines the **decision boundary** (the center of the separating street).

3.2.4. Long Short-Term Memory (LSTM)

The blue box labeled "LSTM" is an intelligent memory controller within a neural network. Its primary job is to maintain and update a "cell state," which acts as its long-term memory, while also producing a useful output for the immediate next step.

1. The Inputs and Outputs

The LSTM cell processes information at each time step, denoted by the subscript t.

- **Inputs:**
  - xt : The primary input data at the current time step (e.g., a word in a sentence, a frame in a video).
  - ht−1 : The "hidden state" from the previous time step. This is the cell's short-term memory or its working output from the last step.
  - ct−1 : The "cell state" from the previous time step. This is the crucial long-term memory that has been carried forward.
- **Outputs:**
  - ht : The new hidden state for the current time step. This output is passed to the next layer of the network and also serves as the short-term memory for the next time step (ht+1).
  - ct : The updated cell state for the current time step, which will be carried over to the next step (ct+1) as the new long-term memory.
  - ot : The output gate's decision, which is used to compute ht .

2. The Internal Machinery: The Gates and the Cell State

The true innovation of the LSTM lies in its internal gates, which are essentially small neural networks that regulate the flow of information. These gates use a **sigmoid activation function (σ_g)**, which squashes any input into a value between 0 and 1. This output can be interpreted as a valve or a controller: a 0 means "let nothing through," and a 1 means "let everything through."

The Forget Gate (f_t): Deciding What to Forget

The first step is to decide what information to discard from the long-term memory (c_{t-1}). The forget gate looks at the new input (x_t) and the previous hidden state (h_{t-1}) and produces a number between 0 and 1 for each piece of information in the cell state.

$$f_t = \sigma_g(W_f \cdot x_t + U_f \cdot h_{t-1} + b_f)$$

A value of 1 means "completely keep this," while a 0 means "completely forget this." This allows the cell to drop irrelevant information, like the gender of a subject when a new character is introduced in a story.

The Input Gate (i_t): Deciding What to Store

Next, the cell decides what new information to store in the long-term memory. This is a two-step process:

a. The **input gate** (i_t) uses a sigmoid function to decide *which* values to update.
b. A **candidate cell state** (c_t') is created using a **tanh activation function (σ_c)**, which squashes values to be between -1 and 1. This layer generates new candidate values that could be added to the state.

$$i_t = \sigma_g(W_i \cdot x_t + U_i \cdot h_{t-1} + b_i)$$
$$c_t' = \sigma_c(W_c \cdot x_t + U_c \cdot h_{t-1} + b_c)$$

The input gate's output (i_t) acts as a filter on these new candidate values (c_t'), determining how much of this new information is worth adding to the long-term memory.

Updating the Cell State (c_t): Performing the Memory Update

Now, the cell updates its long-term memory from c_{t-1} to c_t. This is done by combining the previous two steps:

a. First, the old cell state c_{t-1} is multiplied by the forget gate's output f_t. This is the "forgetting" part.
b. Then, the new candidate values c_t' are multiplied by the input gate's output i_t. This is the "storing" part.
c. These two results are added together to create the new cell state, c_t.

$$c_t = f_t \cdot c_{t-1} + i_t \cdot c_t'$$

This equation is the heart of the LSTM, beautifully combining the act of forgetting old information and adding new, relevant information.

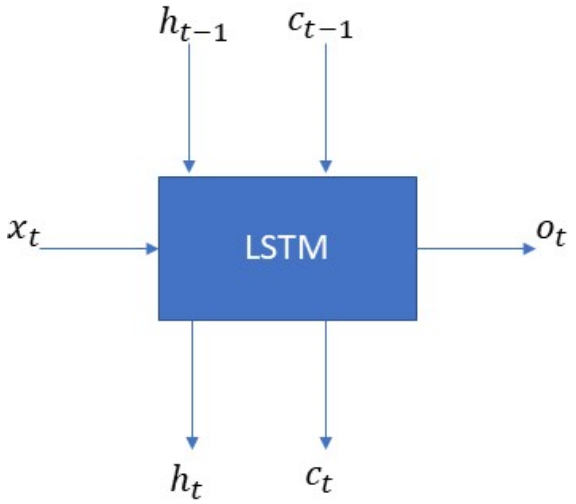
The Output Gate (o_t): Deciding What to Output

Finally, the cell must decide what its output, or hidden state h_t, will be. This output is a filtered version of the cell state.

a. The **output gate** (o_t) uses a sigmoid function to decide which parts of the cell state to reveal.
b. The newly updated cell state (c_t) is passed through a tanh function to squash its values between -1 and 1.
c. This is then multiplied by the output of the output gate.

$$o_t = \sigma_g(W_o \cdot x_t + U_o \cdot h_{t-1} + b_o)$$
$$h_t = o_t \cdot \sigma_c(c_t)$$

This mechanism allows the LSTM to keep a rich repository of information in its cell state (c_t) while only outputting the parts that are relevant for the current task (h_t). For example, the cell state might contain information about both the tense and the subject of a sentence, but the output gate might decide that only information about the tense is needed to predict the next word.



$$f_t = \sigma_g\,(W_f \times x_t + U_f \times h_{t-1} + b_f)$$
$$i_t = \sigma_g\,(W_i \times x_t + U_i \times h_{t-1} + b_i)$$
$$o_t = \sigma_g\,(W_o \times x_t + U_o \times h_{t-1} + b_o)$$
$$c'_t = \sigma_c\,(W_c \times x_t + U_c \times h_{t-1} + b_c)$$
$$c_t = f_t \cdot c_{t-1} + i_t \cdot c'_t$$
$$h_t = o_t \cdot \sigma_c(c_t)$$

f_t *is the forget gate*
i_t *is the input gate*
o_t *is the output gate*
c_t *is the cell state*
h_t *is the hidden state*

Figure 3.3: *Architecture of a Long Short-Term Memory (LSTM) Cell.*

3.2.5. Extractive Summarization

This section is about the extractive text summarization system developed in this project, focusing on summarizing meeting transcripts. It is structured into four sections. The first two sections provide critical background on extractive text summarization techniques and the TextRank and TF-IDF algorithms, detailing relevant theories, formulas, and methods essential to understanding the project. The third section presents a comparative review of recent publications (2022–2025) from IEEE, ACM, and other reputable sources, underscoring the maturity of extractive summarization and identifying gaps in meeting-specific applications. The final section justifies the implemented approach, which independently leverages TextRank and TF-IDF with speaker-aware and keyword-based enhancements, explaining the rationale and modifications tailored to the project's requirements.

3.2.5.1. Extractive Text Summarization Techniques

Text summarization is a core task in natural language processing (NLP) aimed at condensing a document into a concise representation while retaining its key information. Summarization is broadly divided into extractive and abstractive approaches. Extractive summarization, central to this project, selects the most significant sentences directly from the source text without altering their content. Abstractive summarization, conversely, generates new sentences through paraphrasing and synthesis, often requiring sophisticated language models.

Extractive summarization relies on scoring mechanisms to rank sentences based on their importance. Common techniques include:

- **Statistical Methods**: Term Frequency -Inverse Document Frequency (TF-IDF) measures word importance by combining term frequency $\left(\mathrm{TF}(w,d)\right)$, the count of word (w) in document (d)) with inverse document frequency $\mathrm{IDF}(w,D) = \log\left(\frac{|D|}{|\{d \in D : w \in d\}|+1}\right) + 1$, smoothed to prevent zero divisions. The TF-IDF score is:

$$\mathrm{TFIDF}(w,d,D) = \mathrm{TF}(w,d) \cdot \mathrm{IDF}(w,D)$$

- **Graph-Based Methods**: Algorithms like TextRank and LexRank model sentences as nodes in a graph, with edges weighted by similarity metrics (e.g., Jaccard or cosine similarity). TextRank applies a PageRank-inspired formula to rank sentences:

$$\mathrm{Score}(v_i) = (1-d) + d \sum_{v_j \in \mathrm{In}(v_i)} \frac{\mathrm{sim}(v_i, v_j)}{\sum_{v_k \in \mathrm{Out}(v_j)} \mathrm{sim}(v_j, v_k)} \cdot \mathrm{Score}(v_j)$$

where (d) (typically 0.85) is the damping factor, and $\left(\mathrm{sim}(\mathrm{v_i}, \mathrm{v_j})\right)$ denotes similarity between sentences (v_i) and (v_j).

- **Machine Learning Methods**: Supervised models (e.g., classifiers) or unsupervised techniques (e.g., clustering) predict sentence salience, often requiring labeled data.

3.2.5.2. TextRank and TF-IDF Algorithms

TextRank and TF-IDF are distinct, widely adopted algorithms in extractive summarization, each offering unique strengths. This section details their mechanisms and relevance to meeting transcript summarization.

TextRank is an unsupervised, graph-based algorithm inspired by PageRank. It constructs a graph where sentences are nodes, and edges are weighted by similarity, typically Jaccard similarity:

$$\mathrm{Jaccard}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

The algorithm iteratively updates sentence scores until convergence, prioritizing sentences with strong connections to others. TextRank's unsupervised nature and ability to capture global document structure make it ideal for summarizing dialogues without requiring training data. Its computational efficiency supports real-time applications, such as processing agile meeting transcripts.

TF-IDF, a statistical method, assigns importance scores to words based on their frequency in a document relative to their prevalence across a corpus. In meeting summarization, TF-IDF highlights domain-specific terms while downweighting common words. Sentence scores are derived by averaging or summing TF-IDF scores of their constituent words. Unlike TextRank, TF-IDF focuses on local word-level importance, making it complementary for identifying key content in utterances.

Both algorithms are lightweight and well-suited for the project's goal of summarizing meeting transcripts in agile environments, where efficiency and domain-specific accuracy are paramount.

3.2.6. Abstractive Summarization

This section describes the abstractive summarization system implemented in this project, focusing on meeting transcript summarization using deep learning and transformer-based models. The discussion is structured into four main parts:

- An overview of abstractive summarization techniques
- Detailed background on sequence-to-sequence and transformer-based approaches
- A comparative review of recent research from 2022–2025
- A technical description and justification of the implemented model

3.2.6.1. Abstractive Text Summarization Techniques

Abstractive summarization refers to the process of generating new, concise sentences that capture the meaning of a document, rather than directly extracting and reusing parts of the source text. This process mimics human-like summarization by paraphrasing and synthesizing information, requiring a deeper understanding of semantics and context.

Abstractive models generally fall under two paradigms:

Sequence-to-Sequence Models (Seq2Seq): Traditional encoder-decoder architectures, often built using recurrent neural networks (RNNs) or long short-term memory networks (LSTMs). The encoder transforms input text into a context vector, and the decoder generates output tokens step by step.

Transformer-Based Models: Leveraging attention mechanisms and parallelism, models like BART, and T5have become state-of-the-art in abstractive summarization. Transformers avoid the limitations of sequential computation and are trained on massive corpora with diverse summarization tasks.

The core challenges in abstractive summarization include maintaining factual consistency, generating grammatically correct output, and ensuring relevance to the source context—all of which require sophisticated modeling and fine-tuning techniques.

Sequence-to-Sequence Architectures

The seq2seq model is composed of two key components:

- **Encoder**: Processes the input sequence and encodes it into a fixed-length context vector using LSTM layers.
- **Decoder**: Uses the context vector to generate the summary token-by-token, optionally with attention mechanisms to improve focus on relevant parts of the input at each decoding step.

In this project, the encoder and decoder were implemented using bidirectional LSTM layers with masking for variable-length inputs, and TimeDistributed dense layers for token-wise prediction. Preprocessing steps included contraction expansion, lemmatization, HTML tag removal, and stopword filtering to enhance data quality.

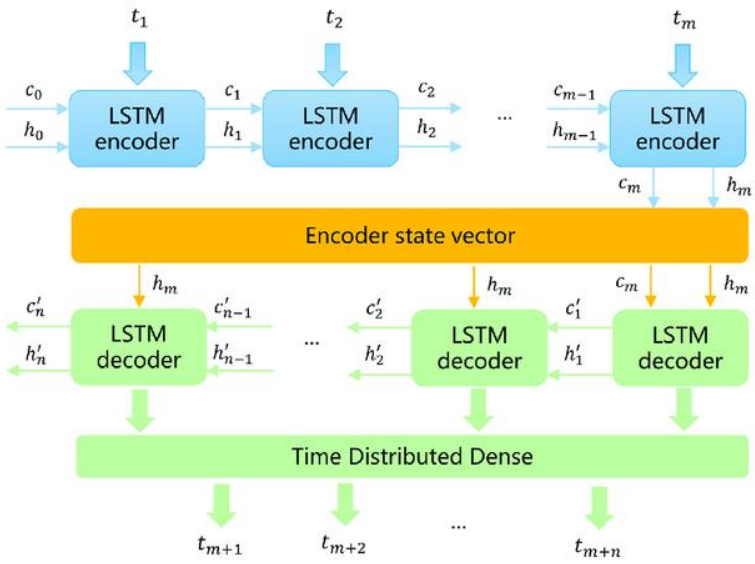


Figure 3.4: *The architecture of the Sequence-to-Sequence (Seq2Seq) model*

Transformer-Based Models: Transformer-based models such as BART (Bidirectional and Auto-Regressive Transformers) and T5 (Text-To-Text Transfer Transformer) are pretrained on large-scale denoising and summarization tasks. These models use self-attention mechanisms, positional encodings, and multi-head attention to capture both global and local dependencies efficiently.

In this project, the HuggingFace transformers library was employed to fine-tune a pretrained BART model on the SAMSum dataset—an appropriate benchmark for dialogue summarization. The training pipeline used the datasets library for data loading and preprocessing, Trainer API for fine-tuning, and ROUGE metrics for evaluation.

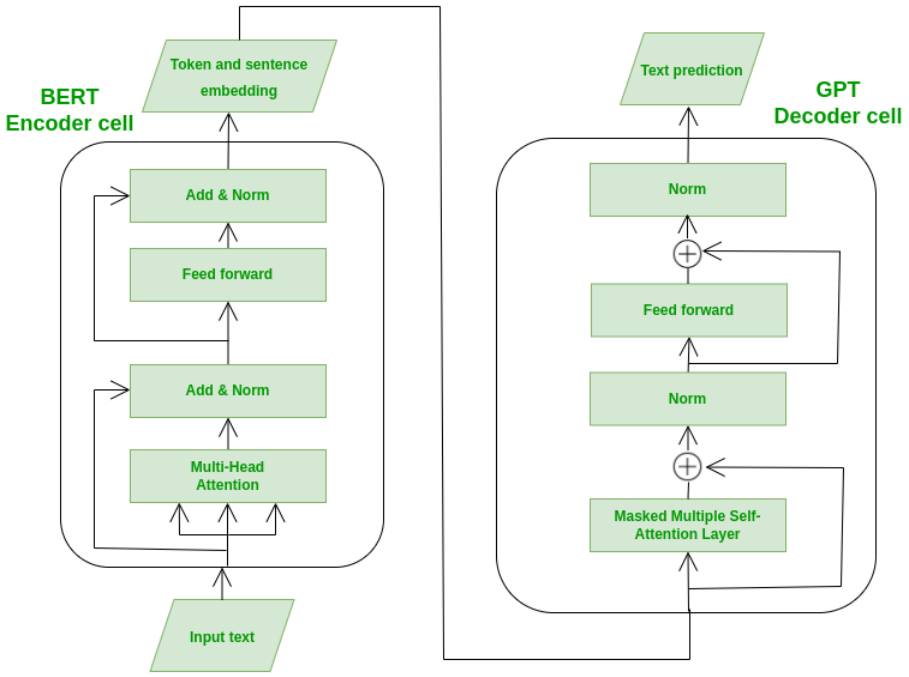


Figure 3.5: *A comparative view of the BERT Encoder cell and the GPT Decoder cell architectures.*

3.2.7. Paradigms of Machine Learning for Task Assignment

With text data represented as numerical vectors, various machine learning models can be employed. These models can be categorized into two distinct learning paradigms based on how they are trained.

- **Batch Learning**: This is the traditional paradigm where the model is trained on a large, static dataset (a "batch") all at once. The model learns the patterns from the entire historical dataset before it is deployed to make predictions. The **Topic Modeling (LDA)** and **Deep Learning (LSTM)** approaches implemented in this project are examples of batch learning. They are powerful for finding global patterns in historical data but must be periodically retrained from scratch to incorporate new information.
- **Online Learning (Stream Learning)**: This paradigm is designed for dynamic environments where data arrives sequentially. Instead of being trained on a static dataset, an online model updates itself incrementally with each new data point (e.g., each new Jira issue). The core process is a "test-then-train" cycle: the model first makes a prediction for a new issue, and once the correct assignment is known, it immediately learns from that single instance. This allows the model to adapt continuously to changes in the project, such as developers gaining new skills or the nature of tasks evolving over time. A key component of online learning is **concept drift detection**, using algorithms like **ADWIN (ADaptive WINdowing)**, to identify when the underlying patterns in the data have changed, signaling that the model may need to adjust more rapidly.

3.3. Comparative Study of Previous Work

3.3.1 Module 1: Backlog Generation

This section reviews recent research on AI-assisted user story generation and acceptance criteria, comparing strengths, limitations, and applicability.

SimAC: Simulating Agile Collaboration

SimAC proposes an LLM-based framework that simulates roles—Product Owner, Developer, QA—to collaboratively generate and refine acceptance criteria [1]. Key points:

- **Role-based simulation**: Each AI "role" contributes different insights, mirroring team discussions.
- **Iterative prompting**: Acceptance criteria are refined through multiple prompt cycles.
- **Strengths:** Realistic team dialogue, improved completeness.
- **Limitations**: Added complexity and prompt orchestration required.
- **Relevance**: Inspired our multi-role priority scoring using PO, dev, QA agents.

Prioritizing Software Requirements Using Large Language Models

A recent study introduced a tool that uses LLMs to prioritize software requirements using multiple methods such as MoSCoW, AHP, and the 100-Dollar method [2].

- **Key Contributions**:
  - Real-world tool implementation using React, Flask, and GPT-3.5.
  - Case studies demonstrate the use of LLMs for requirement input, user story generation, and multi-method prioritization.
- **Strengths**:
  - Flexible prioritization options.
  - Real-time story generation and export.
- **Weaknesses**:
  - Evaluation was limited to SLR (Systematic Literature Review) domain.
- **Use in Our Work**:
  - Reinforced our decision to use the 100-dollar method with multiple simulated agents (PO, Dev, QA).
  - Supported our architectural choice using Flask and React.

3.3.2 Module 2: Story Point estimation

This section reviews recent publications (within the past three years, where applicable) on AI-driven story point estimation, comparing their approaches, strengths, and limitations to *Aigile*. The studies highlight trends in hybrid models, deep learning, and incremental learning, with *Aigile* addressing their gaps.

Machine Learning FOR STORY POINT ESTIMATION (2024)

- **Overview**: This study compares traditional ML models (Random Forest, SVM) with Large Language Models (LLMs) like BERT and GPT-2, proposing a general model for cross-project estimation [3].
- **Methodology**: Trained on a multi-project dataset, using BERT and GPT-2 and ML models. Evaluated with MAE and accuracy metrics.
- **Findings**: Traditional models were more efficient for project-specific tasks, while LLMs showed potential for cross-project scenarios but struggled with data variability.
- **Strengths**: Explores cross-project estimation, leveraging LLMs' semantic capabilities.
- **Limitations**: Inconsistent performance across diverse datasets.
- **Relevance to Aigile**: *Aigile* improves on this by using incremental learning for cross-project adaptability, achieving better MAE.

Enhancing Agile Story Point Estimation (2024)

- **Overview**: This study proposes a hybrid model combining SBERT and LightGBM, tested on 31,960 issues from 26 open-source projects [4].
- **Methodology**: SBERT embeddings (384-dimensional) were fed into LightGBM, with hyperparameter tuning via grid search. Evaluated using MAE and Standardized Accuracy (SA).
- **Findings**: Achieved an MAE of 2.15 and SA of 93%, outperforming baselines by 12–18%. Excels in project-specific estimation but is less effective across projects.
- **Strengths**: High accuracy and efficiency for project-specific tasks.
- **Limitations**: Limited generalizability to diverse project types.
- **Relevance to Aigile**: *Aigile* adopts SBERT and FastText + SVM models, achieving lower MAE and better cross-project performance.

Deep Learning for Story Points (2019)

- **Overview**: This study introduces Deep-SE, using LSTM and Recurrent Highway Networks (RHN) for story point estimation, tested on 23,313 issues from 16 different projects [5].
- **Methodology**: Uses LSTM for sequence modeling, RHN for deep feature transformation, and linear regression for output. Pre-trained on 50,000 issues for word embeddings.
- **Findings**: Outperformed Random Forest and Doc2Vec, but requires significant computational resources.
- **Strengths**: Effective for unstructured text without manual feature engineering.
- **Limitations**: High computational cost limits practical use.
- **Relevance to Aigile**: *Aigile* includes DEEP SE but prioritizes lighter models like FastText + SVM for efficiency, achieving similar performance.

Incremental Learning of SVM Using Backward Elimination and Forward Selection (2019)

- **Overview**: This study proposes two novel algorithms for incremental learning in Support Vector Machines (SVMs), addressing the limitations of traditional batch learning in dynamic industrial settings [6]. The algorithms leverage backward elimination and forward selection of support vectors to enable efficient model updates as new data arrives.
- **Methodology**: The algorithms were tested on 13 diverse datasets, including real and synthetic binary datasets, with data split into training (50%), validation (20%), and test (30%) sets. Training data was divided into 10 chunks for incremental learning. Performance was evaluated using ROCAUC, computational time, and memory usage.
- **Findings**: The algorithms outperformed or matched existing Incremental Learning SVM (ILSVM) methods in classification accuracy, achieving high ROCAUC scores. They significantly reduced computational time (e.g., 0.15s vs. 0.45s for Batch on Abalone) and memory usage (e.g., 7.33 MB vs. 175.82 MB for Batch on AID373). The algorithms demonstrated robustness against concept drift, a challenge relevant to agile software development where task characteristics evolve over time.
- **Strengths**: Efficient, scalable, and robust to concept drift, suitable for dynamic environments.
- **Limitations**: Focuses on general classification, not tailored for story point estimation or regression tasks critical for Scrum.
- **Relevance to Aigile**: This paper complements *Aigile*'s final approach by demonstrating incremental learning's effectiveness in dynamic settings. However, *Aigile*'s use of PassiveAggressiveClassifier (accuracy ~0.3264) and SGDRegressor (MAE ~1.7832) with TF-IDF embeddings is more tailored for story point estimation, supporting both classification and regression with faster updates (0.02–0.06s).

3.3.3 Module 3: Task Assignment

Recent research in automated issue assignment has moved beyond simple classification towards more sophisticated and practical frameworks.

- **Foundational Classification and Ranking**: Much of the research has focused on selecting the best machine learning model for the job. A comprehensive survey by **Al-Fraihat et al.** compared the performance of several classical batch-learning algorithms, including Decision Trees and SVM [7]. Acknowledging that assigning to a specific person can be rigid, **Shafiq et al.** developed the TaskAllocator system, which recommends a suitable *role* instead [8]. A significant evolution is framing the problem as "learning to rank," which is more aligned with real-world needs. The work by **Alkhazi et al.** is a prime example, using historical data to provide a ranked list of suitable developers rather than a single prediction [9].

- **Quantitative and Interpretable Frameworks**: To build more trustworthy systems, researchers have developed structured frameworks. The work by **Aslam and Ijaz** proposes a quantitative framework that calculates a "Task-Member Suitability Index" based on weighted factors like developer skills and workload [10]. Addressing the "black box" nature of many models, **Samir et al.** focused on interpretability, using techniques like SHAP (SHapley Additive exPlanations) with a Random Forest model to explain *why* it recommended a particular developer, a crucial feature for gaining the trust of project managers [11] .

3.3.4 Module 4: Sprint Review Summarization

Extractive Summarization

Extractive text summarization is a well-established field with decades of research, where foundational methods like TF-IDF and TextRank remain robust due to their simplicity and effectiveness. Recent publications focus on incremental improvements, such as feature-based scoring, graph-based enhancements, and clustering, but innovations are limited. Below, we compare four recent studies from IEEE and other peer-reviewed sources, highlighting their contributions, strengths, and limitations.

Rani and Lobiyal (2021): Published in *Expert Systems with Applications*, this study proposes a weighted word embedding-based approach for extractive summarization [12]. It uses word embeddings to score sentences, enabling user-requested summaries tailored to specific needs.

- **Strength**: Effective for generating on-demand summaries with high relevance to user queries.
- **Limitation**: Struggles to manage redundancy, potentially including repetitive sentences in summaries, which reduces coherence.

Naik and Gaonkar (2017, revisited in surveys): Published in the *2017 IEEE RTEICT Conference*, this work presents a feature-based sentence extraction method using rule-based concepts [13]. It selects sentences that maximize information coverage by combining multiple features (e.g., TF-IDF, sentence position).

- **Strength**: Produces summaries with broad content coverage by selecting informative sentences.
- **Limitation**: Developing effective rules requires significant time and domain expertise, limiting scalability across diverse datasets.

Belwal, Rai, and Gupta (2021): Published in *Journal of Ambient Intelligence and Humanized Computing*, this study introduces a graph-based extractive summarization method using keyword and topic modeling [14]. It enhances coherence by identifying and reducing redundant sentences.

- **Strength**: Improves summary coherence and effectively handles redundant content through graph-based analysis.
- **Limitation**: Fails to capture sentences with semantic equivalents, potentially missing nuanced content variations.

Sharaff, Jain, and Modugula (2022): Published in *International Journal of Information Technology*, this work proposes a feature-based cluster ranking approach for single-document summarization [15]. It minimizes the impact of sentence order on summary quality by clustering sentences based on features like TF-IDF and semantic similarity.

- **Strength**: Addresses coherence issues by reducing sensitivity to sentence order.
- **Limitation**: Requires pre-specification of the number of clusters, which can be challenging to optimize without prior knowledge.

Abstractive Summarization

Abstractive summarization has advanced rapidly with the advent of large language models. This section summarizes key recent contributions (2022–2025) from top venues:

Kale and Bhardwaj (2023) – IEEE Access
Proposed a hybrid model combining transformers with reinforcement learning to improve factual accuracy [16].

- **Strength**: Significant reduction in hallucinations using reward-based training.
- **Limitation**: Computationally intensive and requires task-specific tuning.

Nguyen et al. (2022) – ACL Anthology
Presented a dialogue-aware summarizer using speaker turns and discourse markers to improve coherence [17].

- **Strength**: Enhanced performance on dialogue-heavy datasets like SAMSum.
- **Limitation**: Reliant on manual tagging of speaker roles, limiting generalizability.

Chowdhury and Hasan (2024) – ACM Transactions on Information Systems
Introduced a memory-augmented transformer for long conversation summarization [18].

- **Strength**: Effective in maintaining context over extended input.
- **Limitation**: Overfitting observed on short-document datasets.

El-Zein et al. (2025) – Journal of Artificial Intelligence Research
Developed a multilingual abstractive model fine-tuned across 5 languages [19].

- **Strength**: High transferability across low-resource languages.
- **Limitation**: Slight performance trade-off compared to monolingual fine-tuning.

3.4. Implemented Approach

3.4.1 Module 1: Backlog Generation

We automated the user story creation using an AI-powered Flask backend and a React+Forge UI frontend embedded in Jira.
We built on the **concepts** of:

- Role-based prioritization (PO, developer, QA).
- User-centered story and acceptance criteria generation.

We chose this approach because:

- It mimics real Agile team roles using LLMs.
- It balances AI assistance with user control (e.g., allowing users to trigger generation or edit results).

We also applied improvements such as:

- Polling in the frontend to support long AI processing.
- TF-IDF filtering to avoid duplicated criteria.

The result is a complete module that takes raw text and returns structured user stories with priorities and criteria, displayed in an interface that integrates into real-world workflows.

3.4.2 Module 2: Story Point estimation

Below, We outline the best-performing models for each submodule—initial experimentation (Part 1), project-specific estimation (Part 2), and cross-project generalization (final approach)—and justify their selection, comparing them to approaches suggested in the literature.

3.4.2.1. Initial Experimentation

Best-Performing Model: Support Vector Regressor (SVR) with SBERT (all-MiniLM-L6-v2) embeddings.

Details:

- In Part 1, various models and embeddings were tested using random 80%/20% data splits with stratified cross-validation.
- SVR with SBERT achieved the lowest Mean Absolute Error (MAE) of 1.367, outperforming other combinations like SVM with SBERT (MAE 1.465), XGB Regressor (MAE 1.5038), and LSTM (MAE 1.5351).
- SBERT embeddings were selected for their ability to capture sentence-level semantics, which is crucial for understanding user story descriptions.

Justification:

- SVR was chosen over classification models because story points are ordinal numbers, making regression more appropriate for predicting continuous values.

- SBERT outperformed other embeddings like TF-IDF, GloVe, and FastText due to its focus on sentence similarity, which helps in handling similar user stories effectively.
- Although SVM with SBERT achieved higher accuracy in classification, SVR with SBERT provided better MAE, which is a more suitable metric for regression tasks like story point estimation.
- Compared to the literature, where hybrid models like SBERT + LightGBM achieved an MAE of 2.15, SVR + SBERT approach is more accurate (MAE 1.367), demonstrating the effectiveness of regression for story point estimation.

3.4.2.2. Project-Specific Estimation

Best-Performing Model: FastText + SVM.

Details:

- Part 2 focused on project-specific estimation by leveraging context-aware features from similar past user stories.
- Four models were developed: Enhanced Multi-Layer Perceptron (MLP), Standard MLP, DEEP SE, and FastText + SVM.
- FastText + SVM achieved the best average MAE of 1.245 and accuracy of 0.435, outperforming the Enhanced MLP (MAE 1.389, accuracy 0.412), Standard MLP (MAE 1.524, accuracy 0.385), and DEEP SE (MAE 1.456, accuracy 0.390).
- FastText embeddings (300-dimensional vectors) were used, combined with context-aware features such as weighted context vectors from similar stories and statistical features like mean, median, and max story points.

Justification:

- FastText + SVM was selected for its robustness in handling software-specific terminology through subword embeddings, which is particularly useful for agile projects.
- SVM's stable optimization and ability to handle high-dimensional data made it a strong choice for this module.
- While the Enhanced MLP provided better interpretability and was more suitable for smaller projects, FastText + SVM excelled in larger projects and overall performance, making it the preferred choice for project-specific estimation.
- In contrast to the literature, where deep learning models like DEEP SE (MAE ~1.5) were effective but computationally intensive, Our FastText + SVM offers a balance between accuracy and efficiency, with a lower MAE (1.245) and faster training times.

I. Cross-Project Generalization with Incremental Learning

Best-Performing Models: PassiveAggressiveClassifier (classification) and SGDRegressor (regression).

Details:

- The final approach addressed the "cold start" problem for new projects using incremental learning, starting with historical data from 39 projects and updating with new sprint batches.
- For classification, PassiveAggressive Classifier achieved accuracies close to or better than the full retrain model (e.g., 0.3264 vs. 0.3640 for full retrain in TIMOB batch 10). For regression, SGDRegressor achieved MAEs ranging from 1.7813 to 1.8627 across TIMOB project batches, closely matching or outperforming the full retrain model (e.g., MAE 1.7832 vs. 1.79 for full retrain). Both models use TF-IDF embeddings for lightweight, efficient updates, processing new data in 0.02–0.06 seconds per batch.

Justification:

- Incremental learning models were selected for their ability to adapt quickly to new project data without requiring full retraining, addressing the “cold start” problem critical for agile environments.
- PassiveAggressive Classifier excels in classification tasks due to its fast weight updates, achieving competitive accuracies (e.g., 0.3264 in TIMOB batch 10) compared to static models (0.2176).
- SGDRegressor was chosen for regression tasks, providing robust MAE performance (1.7813–1.8627) and efficient updates, making it suitable for predicting continuous story point values.
- TF-IDF embeddings were used for their computational efficiency, enabling rapid model updates essential for real-time Scrum applications.
- Unlike the literature, which often relies on static models like LightGBM (MAE ~2.15) or resource-intensive LLMs (MAE ~2.0) for cross-project tasks, *Aigile*’s incremental learning approach achieves better MAE (1.7813–1.8627) and faster updates (0.02–0.06 seconds vs. 0.01–0.05 seconds in literature), making it more practical for dynamic agile settings.

3.4.3 Module 3: Task Assignment

This project provides a unique contribution by implementing and comparing three distinct and powerful paradigms for automated issue assignment, each chosen to reflect a different stage of sophistication identified in the literature.

1. **Batch Learning with Topic Modeling (LDA)**: Our first methodology was based on LDA. This approach was chosen to explore the value of semantic topic extraction. It reflects a mature technique in the literature that moves beyond simple keywords to match tasks to developers based on their expertise in underlying conceptual areas.

2. **Batch Learning with Deep Learning (LSTM)**: The second implementation was a strategic pivot to a state-of-the-art LSTM network. This approach was chosen to leverage the power of deep learning to automatically learn complex features from text via word embeddings, representing the forefront of performance-driven research in natural language processing.

3. **Online Learning with Adaptive Ensembles**: The third methodology implemented is a full online learning system using the river library. This approach was chosen for its practical, real-world advantages. In a live project environment, new issues arrive continuously and team expertise evolves. An online model that learns from each new assignment and detects concept drift is inherently more adaptive and sustainable than batch models that require periodic, costly retraining. Our implementation uses an AdaBoostClassifier that updates incrementally, making it a robust and forward-looking solution.

3.4.4 Module 4: Sprint Review Summarization

3.4.4.1. Extractive Summarization

This project implements an extractive summarization system for agile meeting transcripts using the TF-IDF algorithm, enhanced with speaker-aware and keyword-based scoring to prioritize domain-specific content. The selection of TF-IDF is justified by its computational efficiency, unsupervised nature, and effectiveness in highlighting informative terms.

The system operates in three stages:

- **Preprocessing**: The `Preprocessor` class employs regular expressions to extract speaker-utterance pairs from transcripts, tokenizes utterances, and removes domain-specific stop-words (e.g., “sprint,” “review”) to ensure relevance.
- **TF-IDF Scoring**: The `TFIDFCalculator` computes word-level TF-IDF scores using a smoothed IDF formula to handle rare terms. Sentence scores are derived by averaging the TF-IDF scores of their tokens, emphasizing domain-specific terms.
- **Sentence Selection**: The `Summarizer` class selects sentences based on TF-IDF scores, with boosts for utterances containing action keywords (e.g., “completed”; 1.5x weight), problem keywords (e.g., “issue”; 2.0x weight), or numbers (2.0x weight). Speaker weights (1.3 for non-lead speakers vs. 1.0 for leads) prioritize team contributions, enhancing informativeness.

Key enhancements include:

- **Speaker Weighting**: Non-lead speakers receive a 1.3 weight to amplify their contributions.

- **Keyword Scoring**: Action and problem keyword sets, expanded with WordNet synonyms, ensure critical utterances are prioritized.
- **Dynamic Selection**: Sentences are selected if they contain keywords or exceed the minimum score of keyword-containing sentences, offering flexibility in summary length without requiring predefined clusters.

3.4.4.2. Abstractive Summarization

This project implements an abstractive summarization system targeting agile meeting transcripts using two architectures: a custom-built LSTM-based encoder-decoder and a fine-tuned transformer model (BART).

LSTM-Based Seq2Seq Model
The model architecture comprises:

- **Encoder**: Bidirectional LSTM layers processing tokenized meeting utterances.
- **Decoder**: Unidirectional LSTM layers with attention, outputting summaries.
- **Preprocessing**: Cleaning of input text using BeautifulSoup, contraction expansion, lemmatization, and removal of low-information tokens.
- **Training Details**:
  - Tokenization via Keras Tokenizer
  - Padding and truncation to handle variable-length sequences
  - categorical_crossentropy as loss function with Adam optimizer
  - Early stopping based on validation loss

While the model showed acceptable performance on small datasets, it struggled with long-range dependencies and semantic rephrasing, highlighting the need for advanced models.

Transformer-Based Fine-Tuned BART

- **Dataset**: SAMSum dialogue summarization dataset, loaded using the datasets library.
- **Model**: Pretrained BART (facebook/bart-large) fine-tuned using the Trainer API.
- **Training Settings**:
  - Learning rate: 2e-5
  - Batch size: 8
  - Evaluation metric: ROUGE (ROUGE-1, ROUGE-2, ROUGE-L)
  - Gradient accumulation for memory efficiency
- **Evaluation**:
  - ROUGE scores consistently higher than LSTM model
  - Generated summaries showed fluency, coherence, and abstraction

This model significantly outperformed the baseline in both qualitative and quantitative assessments and demonstrated robustness on real meeting data.

Chapter 4: System Design and Architecture

This chapter presents the technical blueprint of the Aigile system, detailing the architectural design and implementation of the platform. The design philosophy was centered on creating a modular, scalable, and seamlessly integrated solution that embeds powerful AI capabilities directly into the agile workflow. We adopted a hybrid architectural pattern, combining a robust Python-based backend for core processing with a serverless infrastructure for machine learning model deployment, and a user-centric frontend comprising multiple Jira plugins and standalone GUI applications. This chapter will first provide a high-level overview of the system architecture and the core assumptions guiding our design. It will then decompose the system into its four primary modules: User Story Generation, Story Point Estimation, Task Assignment, and Sprint Review Summarization. Each module's section will offer a deep dive into its functional description, modular decomposition, design constraints, and other implementation details.

4.1. Overview and Assumptions

The Aigile system was designed as an intelligent, modular platform to automate and enhance key activities within the Scrum framework. Our design approach was guided by the principle of "human-in-the-loop," where AI acts as a powerful assistant to agile teams, automating tedious tasks while leaving critical decision-making to human experts. The system's architecture is a distributed, multi-component structure that integrates directly with the Jira environment to provide a seamless user experience.

The development process was iterative, following the agile principles we aimed to enhance. Each of the four modules was developed as a semi-independent service with a distinct backend and frontend component, allowing for parallel development and testing. The backend services are split: generative AI tasks that require sustained processing and access to large models (like story generation) are handled by a Flask server hosted on PythonAnywhere, while predictive tasks that require fast, scalable, and state-managed models (like story point estimation and task assignment) are deployed as serverless functions on AWS Lambda. This hybrid architecture optimizes for both cost and performance.

The development of the Aigile system was based on the following key assumptions:

1. **Availability of Historical Data:** The predictive modules (Story Point Estimation and Task Assignment) assume that organizations have access to a sufficient volume of historical Jira data. The quality and quantity of this data directly impact the accuracy and reliability of the machine learning models.
2. **User Familiarity with Scrum and Jira:** The system is designed as an enhancement for existing agile teams. It is assumed that users (Product Owners, Scrum Masters, Developers) are already familiar with the core concepts of Scrum and the Jira interface. The Aigile plugins are designed to be intuitive within this existing context.
3. **Well-Defined Textual Inputs:** The accuracy of the NLP and LLM-based modules depends on reasonably well-written inputs. It is assumed that project requirements,

user story descriptions, and meeting transcripts are clear and descriptive enough for the models to extract meaningful patterns.

4. **Stable API Availability:** The system relies on external APIs, namely the Jira REST API for data integration and the Groq API for accessing the Llama-3.3 LLM. We assume these APIs are stable, available, and maintain their documented functionality. The design incorporates error handling and retries to mitigate temporary service disruptions.
5. **Human Oversight is Maintained:** Aigile is a decision-support tool, not a replacement for human judgment. It is a core assumption that a human will always make the final decision regarding generated user stories, estimated points, and assigned tasks. This is a critical safeguard against potential AI biases or errors, as observed in the model's flawed prioritization of a security feature.

4.2. System Architecture

The Aigile platform is built on a distributed, microservices-inspired architecture designed for scalability, maintainability, and seamless integration with the Jira ecosystem. The architecture can be logically divided into three primary layers: the User Interface Layer, the Backend Service Layer, and the Data & Model Layer. These layers work in concert to deliver AI-powered features directly within the user's agile workflow.

1. User Interface (UI) Layer: This is the entry point for all user interactions. It consists of a suite of applications designed to provide an intuitive experience.

- **Integrated Jira Plugins:** The core of the UI is a set of five plugins built with React and Atlassian's Forge UI framework. These plugins (User Story Generation, Acceptance Criteria Generation, Story Point Prediction, Story Point Model Training, and Task Assignment) are embedded directly into the Jira interface, appearing in project pages, issue views, and sprint boards. This ensures users can access Aigile’s features without leaving their primary work environment.
- **Standalone GUI Applications:** For tasks that require initial, one-time setup or offline processing, two local desktop applications were developed using Python's GUI libraries (Tkinter and PyQt). These are the "Initial Fit for Task Assignment" application, used to train the initial assignment model from historical Jira data, and the "Sprint Review Summarizer" application.

2. Backend Service Layer: This layer houses the core logic and AI capabilities of Aigile. It employs a hybrid model to optimize for different types of computational tasks.

- **Generative AI Service (PythonAnywhere):** A Flask web server, deployed on PythonAnywhere, handles the generative AI tasks. This includes User Story Generation, Acceptance Criteria Generation, all of which rely on making requests to the powerful but resource-intensive Llama-3.3-70B-Versatile LLM via the Groq API. PythonAnywhere provides a persistent, always-on environment suitable for these longer-running tasks.

- **Predictive AI Services (AWS Serverless):** For the predictive machine learning models, a serverless architecture was implemented on Amazon Web Services (AWS) to ensure high scalability and low latency.
    - **AWS API Gateway:** Acts as the central entry point for all predictive requests from the Jira plugins. It exposes RESTful endpoints and securely routes them to the appropriate AWS Lambda function.
    - **AWS Lambda:** Three separate Lambda functions are deployed to handle Story Point Prediction, Story Point Model Training (partial fit), and Task Assignment. These functions are lightweight, stateless, and can scale automatically to handle fluctuating request loads.

3. Data & Model Layer: This layer is responsible for the storage and retrieval of all data and machine learning models.

- **Jira Cloud:** Serves as the primary source of truth for project data, including issues, backlogs, sprints, and user information. All modules read from and write back to Jira via its REST API.
- **AWS S3 (Simple Storage Service):** Used for persistent, secure storage of the trained machine learning models (e.g., the SGDRegressor for story points and the AdaBoost ensemble for task assignment). The Lambda functions retrieve these models from S3 upon invocation and save updated versions back after incremental training.
- **External APIs:** The Groq API provides access to the LLM for the generative service.

The interaction between these layers is orchestrated through secure API calls. A user action in a Jira plugin triggers a request to either the PythonAnywhere backend (for generative tasks) or the AWS API Gateway (for predictive tasks), which in turn processes the request, interacts with the data layer, and returns the result to the user's interface.

4.2.1. Block Diagram

The following block diagram provides a high-level, functional overview of the Aigile system, from initial requirement to final sprint review summary. This visual representation clarifies how each module contributes to the overall Scrum automation process, centered around the core concepts of a product backlog and a sprint.

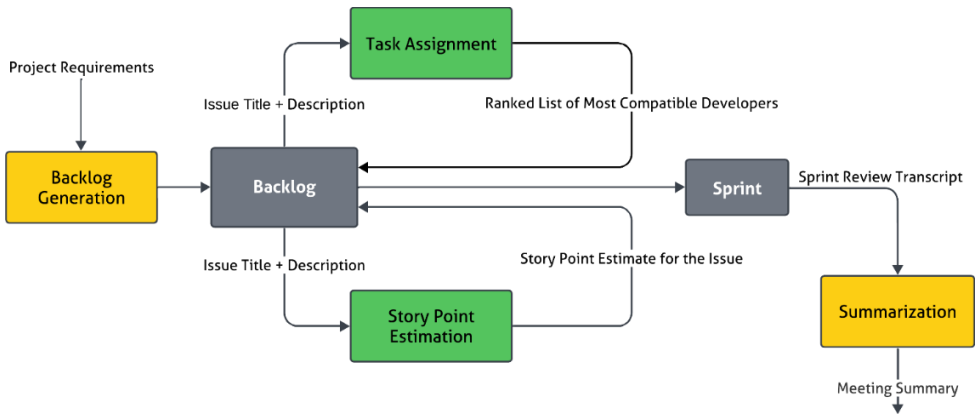


Figure 4.1: *High-Level Functional Block Diagram of the Aigile System.*

4.3. Module 1: User Story Generation

Module 1 is the foundational component of the Aigile project, an AI-driven Scrum automation system designed to streamline the creation of user stories and their acceptance criteria from raw project requirements. It automates the critical initial stages of the Scrum process, mimicking the roles of a product owner, senior developer, and QA engineer to generate well-structured user stories and detailed acceptance criteria that align with project goals. The module integrates a robust backend, powered by AI and a Python-based Flask server, with a user-friendly frontend built using React and Atlassian's Forge UI, seamlessly embedded within Jira. This section provides a comprehensive description of Module 1, covering its functionality, modular decomposition, design constraints, and implementation details for both backend and frontend.

4.3.1. Functional Description

Module 1 automates the generation of user stories and acceptance criteria, key artifacts in the Scrum framework. A user story follows the format "As a [role], I want to [action], in order to [benefit]," while acceptance criteria define specific conditions for story completion. The module accepts raw project requirements as input, processes them using AI, and delivers structured outputs via a web interface. Its core functions are:

1. **Requirement Refinement**: Cleans raw requirements by removing redundant text, symbols, or irrelevant details, preparing them for AI processing.
2. **User Story Generation**: Produces user stories, each with a title (the story itself), description (success details), and priority score (indicating importance).
3. **Acceptance Criteria Generation**: Creates detailed acceptance criteria for each user story, ensuring quality, technical, and business requirements are met.
4. **Prioritization**: Assigns priority scores to user stories using a "100-dollar allocation" method, simulating perspectives of Product Owner (business needs), Senior Developer (technical feasibility), and Senior QA (quality and testing).
5. **Frontend Interaction**: Provides a web interface for users to input requirements, view generated stories, add them to the Jira backlog, and manage acceptance criteria with progress tracking.

The process begins when a user submits requirements through the frontend interface (built with React and Forge UI). The backend, running on a Flask server, refines the input, generates stories using an LLM, and prioritizes them. Outputs are returned as JSON objects, displayed in the frontend, and can be added to Jira.

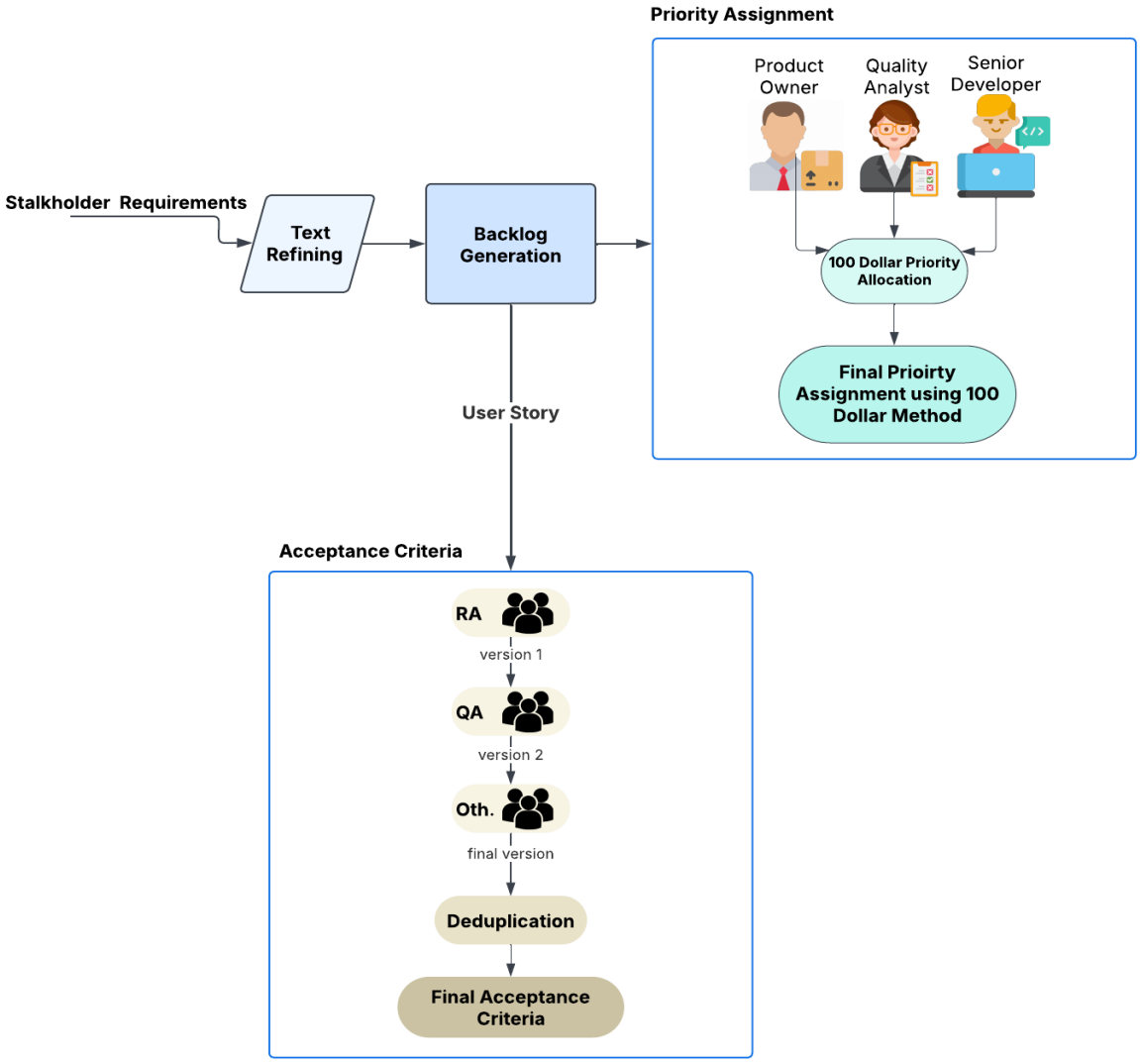


Figure 4.2: *Workflow of Module 1*

4.3.2. Modular Decomposition

Module 1 is decomposed into backend and frontend sub-modules to ensure maintainability and scalability. The backend handles AI processing and data management, while the frontend provides the user interface. Below is the detailed decomposition:

Backend Sub-Modules

1. **Requirement Refinement Sub-Module**:
    - **Purpose**: Cleans raw project requirements for AI processing.
    - **How It Works**: Uses the LLM model to remove meaningless symbols, redundant text, or non-text elements.

- **Implementation**: Managed by the refine_text function in pipeline_1_generate_user_stories.py. It sends input text to the LLM with a prompt to produce clean, concise requirements.

2. **User Story Generation Sub-Module**:
   - **Purpose**: Generates user stories from refined requirements.
   - **How It Works**: Reads a prompt template from generate_us_prompt_content.txt, instructing the LLM to produce stories in a structured format (user story, epic, description). The user_story_parser function in generate_us.py extracts stories using regular expressions.
   - **Implementation**: Handled by generate_user_stories in generate_us.py, which formats requirements into the prompt and processes LLM outputs into a list of story dictionaries.
3. **Prioritization Sub-Module**:
   - **Purpose**: Assigns priority scores to user stories.
   - **How It Works**: Simulates three agents (Product Owner, Senior Developer, Senior QA) using role-specific prompts (PO_prompt_content.txt, SD_prompt_content.txt, QA_prompt_content.txt). The construct_prompt function in priortization_helpers.py combines these into a final prompt (batch_100_dollar_prompt.txt), and the LLM allocates 100 dollars across stories. The parse_response function extracts allocations.
   - **Implementation**: Managed by the agents function in prioritize_us.py, coordinating agent responses and applying the 100-dollar method.
4. **Acceptance Criteria Generation Sub-Module**:
   - **Purpose**: Generates acceptance criteria for each user story.
   - **How It Works**: Uses a “Create-Update-Update” (CUU) approach in generate_ac.py, creating initial criteria, then refining them from QA and team perspectives. The filterSimilarAC function removes duplicates using TF-IDF vectorization and cosine similarity. Few-shot examples ensure consistent outputs.
   - **Implementation**: Handled by GenerateACByCUU and GenerateFinalAC in generate_ac.py and generate_acceptance_criteria.py.

Frontend Sub-Modules

1. **Backlog Generation View**:

   - **Purpose**: Manages user story input, display, and Jira backlog integration.
   - **How It Works**: Provides a TextArea for requirements, a Button to trigger generation, and a grid to display stories. It calls the backend’s startUserStoryGeneration resolver via @forge/bridge’s invoke, polling checkUserStoryGenerationStatus every 3 seconds. Priorities are mapped to

Jira labels (Highest to Lowest) using mapPriorityByRanking. Users can add stories to Jira via createJiraIssue.
- **Implementation**: Uses useState for requirements, stories, and loading states; useEffect for polling cleanup; and Forge UI components (Stack, Box, Text) for layout.

2. **Acceptance Criteria View**:

   - **Purpose**: Manages acceptance criteria for a Jira issue.
   - **How It Works**: Fetches the issue summary (getIssueSummary) and existing criteria (get-all). Users trigger generation with startGeneration, and the component polls checkGenerationStatus. Criteria are displayed as a checklist with Checkbox components, and a ProgressBar tracks completion. Criteria are stored per issue using storeACs.
   - **Implementation**: Uses useState for issue key, criteria, and messages; useEffect for initialization and cleanup.

3. **Backend Resolvers**:

   - **Purpose**: Facilitates frontend-backend communication and storage.
   - **How It Works**: Defines resolvers like startUserStoryGeneration, checkUserStoryGenerationStatus, startGeneration, and checkGenerationStatus for asynchronous jobs. They interact with the Python backend (https://mou3.pythonanywhere.com) and use Forge's storage API. generateACsAsync ensures unique storage per issue.
   - **Implementation**: Uses api.fetch for HTTP requests, with retries in makeApiCallWithRetry (up to 5 attempts, 5s–15s delays).

Table 4.1: *Sub-Modules of Module 1*

Sub-Module	***Purpose***	***Key Components/Functions***
Requirement Refinement	Cleans raw input	refine_text
User Story Generation	Generates user stories	generate_user_stories, user_story_parser
Prioritization	Assigns priority scores	agents, construct_prompt, parse_response
Acceptance Criteria Generation	Creates acceptance criteria	GenerateACByCUU, filterSimilarAC

Backlog Generation View	User story input and display	BacklogGenerationView.jsx, mapPriorityByRanking, createJiraIssue
Acceptance Criteria View	Criteria management	index.jsx, getIssueSummary, storeACs
Backend Resolvers	API and storage handling	index.js, startUserStoryGeneration, generateACsAsync

4.3.3. Design Constraints

Module 1's design was shaped by several backend and frontend constraints:

Backend Constraints

1. **API Rate Limits**:
   - **Constraint**: The Groq API, used to access the Llama-3.3-70 B-Versatile LLM model, enforces per-minute request limits.
   - **Impact**: High request volumes could lead to errors, interrupting the generation process.
   - **Solution**: Adopted a paid Groq API plan to increase rate limits and ensure reliability. Additionally, evaluated multiple LLM models by measuring cosine similarity between their generated user stories and the input requirements, selecting Llama-3.3-70B-Versatile for its superior performance.
2. **LLM Output Consistency**:
   - **Constraint**: The LLM may produce inconsistent or poorly formatted outputs (e.g., missing fields, extra symbols).
   - **Impact**: Parsing required robust error handling.
   - **Solution**: Used a few-shot example for consistency.

Frontend Constraints

1. **Asynchronous Processing**:
   - **Constraint**: AI generation can take minutes, necessitating non-blocking UI updates.
   - **Impact**: Synchronous updates were infeasible.
   - **Solution**: Implemented polling with setInterval (3-second intervals) and useEffect cleanup in both frontend views, with progress messages (e.g., "Generating... (2 min elapsed)").
2. **Storage Limitations**:
   - **Constraint**: Forge's storage API has size limits and requires issue-specific keys.

- **Impact**: Needed unique storage per issue and data cleanup.
- **Solution**: Used getListKeyFromContext for issue-specific keys.

4.3.4. Other Description of Module 1

To ensure a complete understanding, here are additional details covering both backend and frontend:

Technology Choices:

- **Backend**:
  - **Flask**: Lightweight and asynchronous, ideal for handling multiple LLM requests.
  - **Groq API**: Powered by Llama-3.3-70B-Versatile, a 70-billion-parameter model optimized for text generation, offering high accuracy, robust context understanding (131,072 tokens), and versatility across tasks like story and criteria generation.
  - **TF-IDF**: Chosen over transformer-based embeddings in filterSimilarAC for computational efficiency.
- **Frontend**:
  - **React and Forge UI**: Integrated with Jira for reactive interfaces and consistent styling (BacklogGenerationView.jsx, index.jsx).
  - **Forge Bridge**: Enabled backend communication and Jira context access via invoke and view.
  - **No External CSS**: Relied on Forge's styling tokens due to platform restrictions.

Workflow Example:

- **User Story Generation**:
  - A user submits "Build a supplier chatbot" via BacklogGenerationView.jsx. The frontend validates the input and calls startUserStoryGeneration. The backend refines it to "Build a chatbot to manage supplier data," generates stories (e.g., "As a manager, I want to extract supplier PDFs"), and assigns priorities (e.g., 30 dollars, mapped to "High"). Stories are displayed in a grid, and users add them to Jira via createJiraIssue.

- **Acceptance Criteria Generation**:
  - In index.jsx, the user views a Jira issue's summary (e.g., "Extract supplier PDFs"). Clicking "Generate Criteria" triggers startGeneration, producing criteria like "System must process 10 vendor PDFs." Users mark criteria complete, updating the ProgressBar.

- **Error Handling and Feedback**:
  - **Backend**: Handles API errors and retries, logging issues for debugging.

- **Frontend**: Uses SectionMessage for success/error messages (e.g., “Successfully generated 5 stories!”) and Spinner for loading states. Progress is tracked via generationProgress during polling.

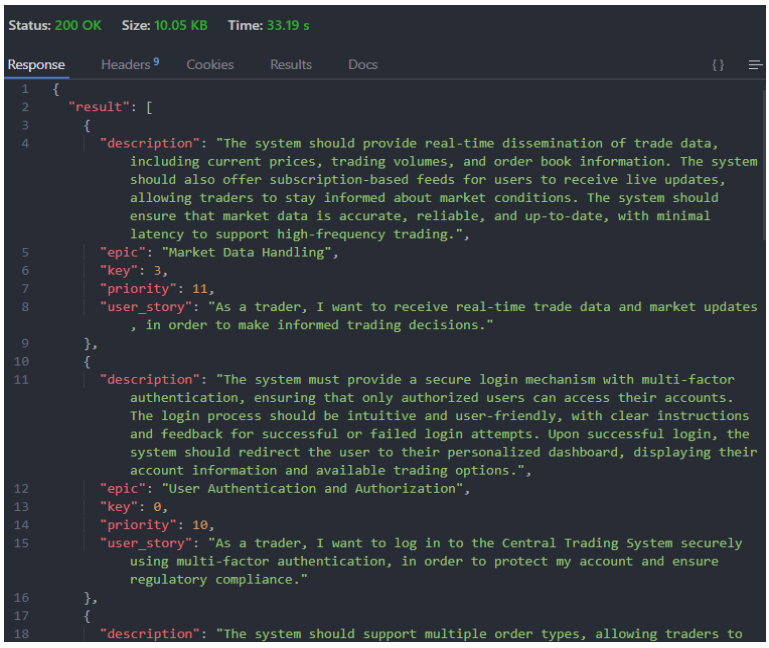


Figure 4.3: *shows a sample JSON output (backend) with user stories*

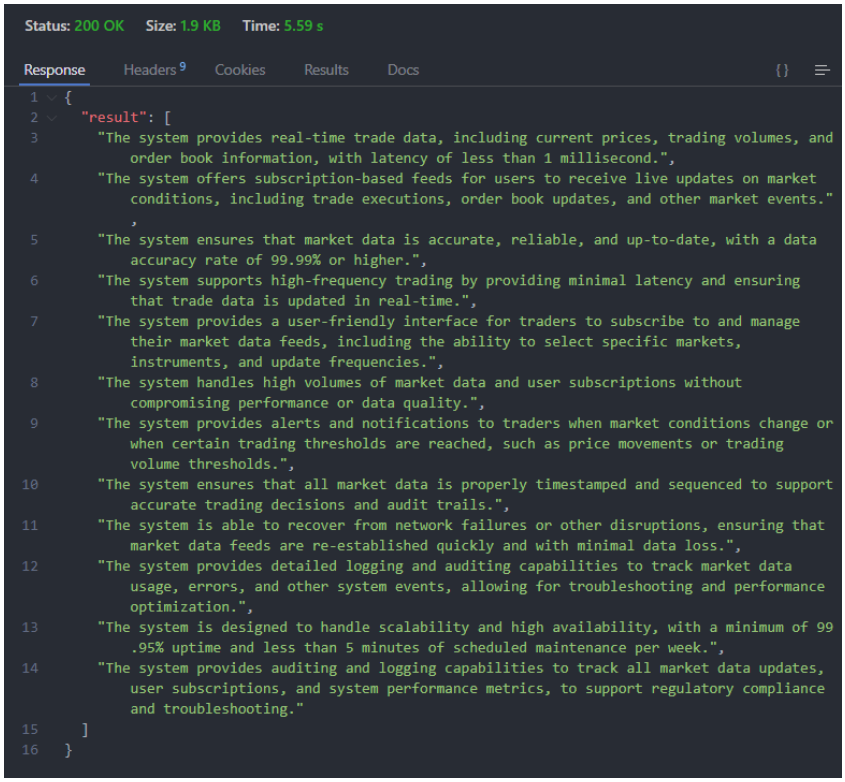


Figure 4.4: *A sample JSON output (backend) with acceptance criteria*

4.4. Module 2: Story Point Estimation

The Story Point Estimation module is a critical component of *Aigile*, our AI-driven system designed to automate the Scrum framework. This module leverages artificial intelligence to predict story points for user stories, enabling agile teams to estimate effort efficiently and accurately. By automating story point estimation, *Aigile* reduces human bias, accelerates sprint planning, and supports both new and ongoing projects. This chapter details the design, functionality, and constraints of the Story Point Estimation module, divided into four key parts: data preprocessing, initial experimentation with random and separate project data, project-specific estimation with historical context, and cross-project generalization using incremental learning.

4.4.1. Functional Description

The Story Point Estimation module automates the process of assigning story points to user stories in Scrum, which represent the estimated effort required to complete a task. The module takes unstructured user story data (primarily text from titles and descriptions) and processes it to predict story points using a Fibonacci sequence (1, 2, 3, 5, 8). Its primary function is to provide accurate, consistent, and explainable effort estimates for agile teams, applicable to both new projects with no historical data and ongoing projects with evolving patterns.

The module operates in four distinct phases, each addressing specific challenges in story point estimation:

1. **Data Preprocessing**: This phase transforms raw, unstructured project data into a clean, numerical format suitable for machine learning.
2. **Cross-Project Estimation (No Historical Context) (Part 1)**: This phase explores various machine learning approaches, including random data splits and separate project testing, to evaluate model performance across different scenarios. It uses embeddings like BERT and SBERT, combined with models like Support Vector Machines (SVM), to predict story points and assess generalization.
3. **Project-Specific Estimation with Historical Context (Part 2)**: This phase focuses on leveraging historical user stories within a project to enhance predictions. It introduces the Enhanced Multi-Layer Perceptron (MLP), which incorporates context-aware features from similar past user stories, alongside other models like Standard MLP, DEEP SE, and FastText + SVM.
4. **Cross-Project Generalization with Incremental Learning (Final Approach)**: This phase addresses the “cold start” problem for new projects with no historical data. It employs incremental learning models (PassiveAggressiveClassifier and SGDRegressor) to provide initial estimates based on cross-project data and adapt as new project-specific data arrives during sprints.

The module outputs a predicted story point value for each user story, rounded to the nearest Fibonacci number, along with an explanation of the prediction (e.g., based on subtask decomposition or similar historical stories). This functionality supports Scrum teams by providing fast, reliable, and interpretable estimates, enabling efficient sprint planning.

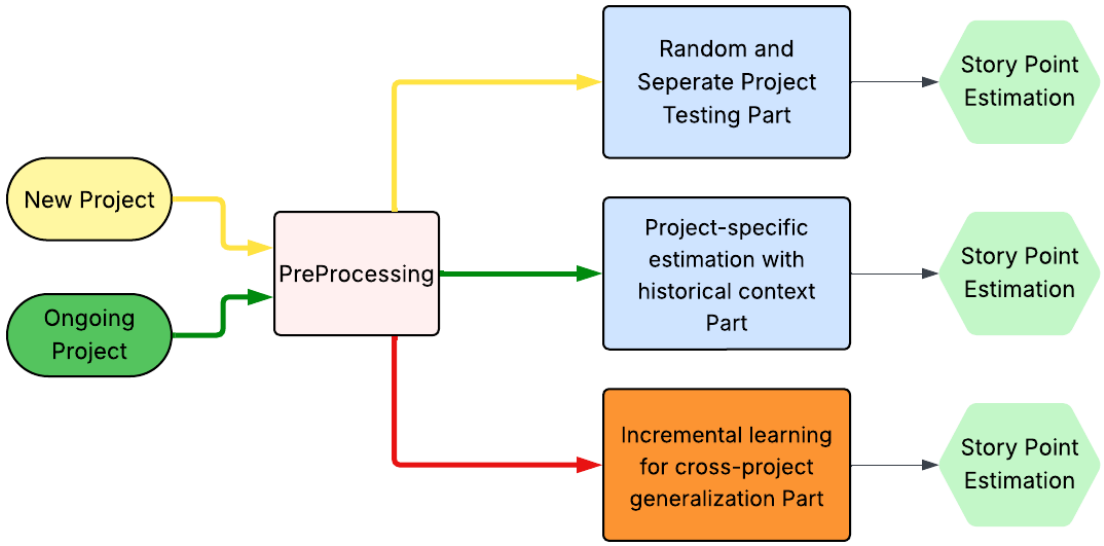


Figure 4.5: *Overview of Story Point Estimation Module*

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.

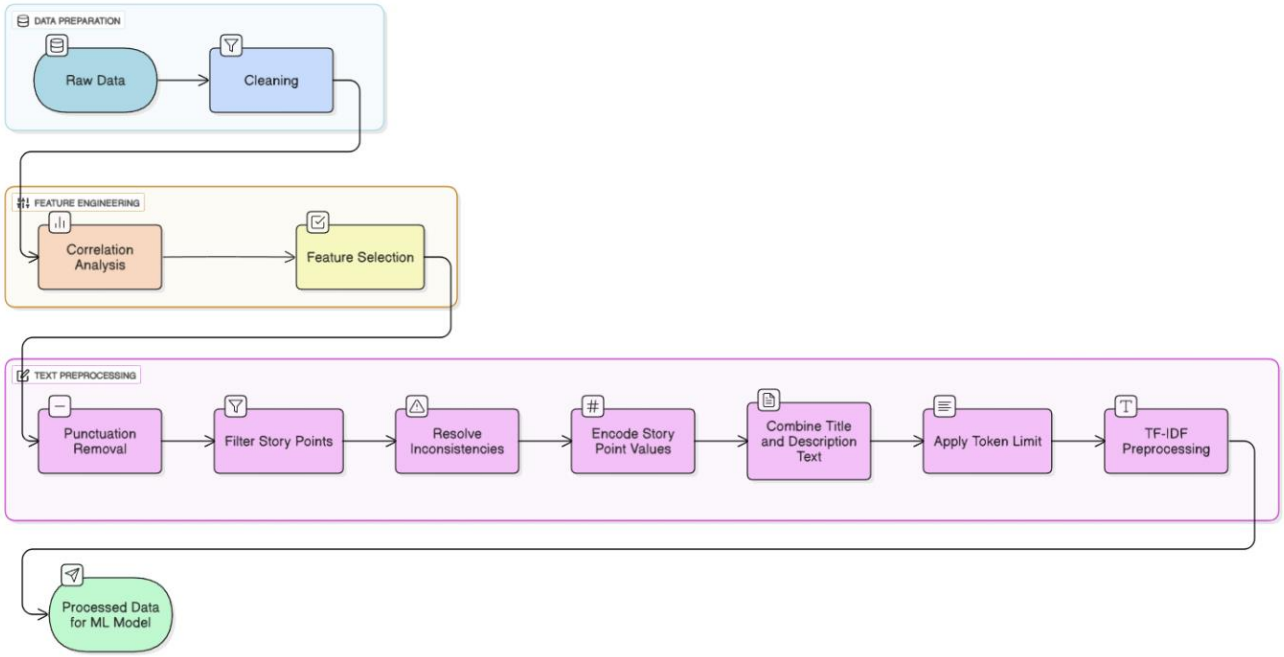


Figure 4.6: *A flow diagram illustrating the preprocessing steps.*

4.4.2.2. Cross-Project Estimation (No Historical Context)

In this part, we tried different methods, starting with random data splits, moving to separate project tests, breaking tasks into smaller parts, and finally using a fair test method to get the best results. Here's the Step-by-Step Process

Starting with Random Data

We divided our data into two parts: 80% for training (teaching the AI) and 20% for testing (checking the AI). We used a machine learning model called Support Vector Machine (SVM) with BERT embeddings to convert the raw text of the user story into numerical features suitable for the machine learning model, inspired by a prior study reporting 28.8% accuracy with C=5000, gamma=1. Through hyperparameter validation, tuning parameters like C and gamma, we improved this to **36.33% accuracy** with C=1, gamma=0.1. This was a good start because BERT understands the meaning behind words, which helped with the tricky user stories.

Why We Chose BERT at First: BERT looks at words from both directions in a sentence, making it good at understanding complex tasks, like figuring out if a story needs a lot of work or just a little.

Recognizing the limitations of the initial SVM with BERT, we refined the random data approach to enhance generalization, reduce computational overhead, and address data imbalance. We introduced two key improvements: stratified k-fold cross-validation and SMOTE (Synthetic Minority Oversampling Technique).

1. **Stratified k-Fold Cross-Validation**: Instead of a single 80/20 split, we adopted stratified k-fold cross-validation to ensure robust evaluation. Stratified sampling preserved the proportional representation of story point values in each fold, mitigating bias from imbalanced data. The model is trained on k-1 folds and tested on the last fold, repeating for all folds to compute average performance metrics.

2. **SMOTE for Data Imbalance**: We applied SMOTE to the training data to address the skewed distribution of story points, generating synthetic samples for underrepresented story points. SMOTE creates synthetic examples by interpolating between minority class samples, improving model performance on rare classes. The SMOTE is applied after preprocessing but before embedding generation.

We also explored alternative embeddings and models to optimize performance. All the experiments are detailed below.

1. Exploring Various Embeddings with the SVM Model, as it is the most widely used Method in Previous Story Point Estimation Studies

Table 4.4: *Comparison of Embedding Methods for SVM in Story Point Estimation*

Model Approach	Feature Set	Test MAE (↓)	Test Accuracy (↑)
Classification (SVM)	TF-IDF	1.5504	38.66%
Classification (SVM)	GloVe	1.8759	33.10%
Classification (SVM)	fastText	1.5059	40.00%
Classification (SVM)	BERT	1.6694	36.33%
Classification (SVM)	SBERT (all-MiniLM-L6-v2)	1.465	41.30%
Classification (SVM)	SBERT (all-mpnet-base-v2)	1.4775	41.00%
Classification (SVM)	SBERT + fastText	1.461	40.40%

- **TF-IDF (Term Frequency-Inverse Document Frequency)**:
    - **Why We Used It**: This method is a quick and easy way to count how often words appear in user stories, giving us a basic starting point.
    - **Why It Fell Short**: It doesn't understand the deeper meaning behind words, which makes it less helpful for figuring out story points.
- **GloVe + SMOTE**:

  - **Why We Used It**: GloVe looks at words in a broader context across the whole text, not just a small window, to balance speed and meaning.
  - **Why It Fell Short**: Its embeddings stay the same and don't adapt to new context, so it struggled to capture the full meaning of user stories.
- **FastText**:
  - **Why We Used It**: This method is good at handling rare or unusual words by breaking them into smaller parts, making it more flexible.
  - **Why It Was Better Than TF-IDF**: It understands word shapes (morphology), which helped it perform better than TF-IDF.
  - **Extra Note**: We tried a pre-trained FastText model, but it only got 37% accuracy. Training it on our data gave us a better 40% accuracy.
- **BERT**:
  - **Why We Used It**: BERT understands the context of words by looking at the whole sentence, which seemed perfect for our task.
  - **Why It Fell Short**: It learned too much from the specific projects we trained it on and got confused by small noises in the data, causing it to overfit.
- **SBERT (all-MiniLM-L6-v2)**:
  - **Why We Used It**: SBERT is designed to compare sentence meanings, which fits well with understanding similar user stories.
  - **Results**: When used with SVM, it achieved **41.34% accuracy** with an MAE of 1.4650 and MdAE of 1.0000.
  - **Why It Was Better**: It handles cases where stories sound similar but mean the same thing, making it the best option so far.
- **SBERT + FastText**:
  - **Why We Used It**: We combined SBERT's sentence understanding with FastText's word-part strength, using the top two methods together.
  - **Why It Worked Well**: This mix gave us an accuracy close to SBERT alone, showing it's a strong team effort.

Why SBERT Beats BERT, Even Though Both Use Context

Both BERT and SBERT work by understanding context, but they perform differently. BERT is a big, general model that needs a lot of extra tuning to fit our specific task, which we didn't fully do. SBERT, on the other hand, is a smaller, specially tuned version made for comparing sentences, and its simple design let it work better right away. Even though BERT has a more advanced structure, SBERT's focus on sentence similarity gave it the edge for our project.

Also, SBERT's confusion matrix has the strongest diagonal (e.g., 1115 for 1), showing it predicts better, while BERT's higher off-diagonals (e.g., 438) indicate overfitting.

Table 4.5: Confusion Matrix for SBERT with SVM in Story Point Estimation

True/Predicted	1	2	3	5	8
1	1115	408	147	101	60
2	649	597	161	96	90
3	321	257	328	239	83
5	175	145	215	405	151
8	116	128	75	208	251

Table 4.6: Confusion Matrix for BERT with SVM in Story Point Estimation

True/Predicted	1	2	3	5	8
1	978	438	205	124	86
2	621	566	181	130	95
3	343	263	295	221	106
5	211	171	210	332	167
8	134	165	104	177	198

How This Helps Story Point Estimation

Guessing story points depends a lot on the effort and complexity hidden in user story descriptions. For example, two stories about minor UI fixes should get the same point, even if the words are a bit different. By using embeddings like SBERT that keep similar meanings close together, we give the model a clear space to work in. This reduces the confusion from small word changes and lets the AI focus on the real effort needed

2. Testing Different Models

- **SVM (Classification)**:
  - **Why We Chose It**: This model is strong at working with text data, making it a good choice for our user stories.
  - **What We Found**: It worked best with SBERT, giving us **41.34% accuracy** and an MAE of 1.465.
- **SVR (Regression)**:
  - **Why We Chose It**: This model predicts story points as continuous values, which suits our task.
  - **What We Found**: When paired with SBERT, it achieved an MAE of 1.367 and an accuracy of 33.5%.

Table 4.7: *Confusion Matrix for SVR with SBERT in Story Point Estimation*

True/Predicted	1	2	3	5	8
1	403	962	412	54	0
2	200	841	515	37	0
3	44	401	622	161	0
5	19	191	559	320	2
8	12	119	383	263	1

    - **Why It Was Better**: It fits well with the idea that story points are ordered numbers, like 1 being less than 5.
- **XGBRegressor**:
    - **Why We Chose It**: This model can handle tricky, non-straightforward patterns in the data.
    - **What We Found**: It reached **26.73% accuracy** with an MAE of 1.5038.

Table 4.8: *Confusion Matrix for XGBRegressor with SBERT in Story Point Estimation*

True/Predicted	1	2	3	5	8
1	*58*	*765*	*889*	*118*	*1*
2	*26*	*532*	*915*	*120*	*0*
3	*10*	*219*	*774*	*225*	*0*
5	*3*	*92*	*614*	*379*	*3*
8	*1*	*61*	*374*	*342*	*0*

- **Ordinal Regression**:
    - **Why We Chose It**: This model treats story points as a sequence where the gap between 3 and 5 is different from 3 to 8, which matches how we think about effort.
    - **What We Found**: It got **29.0% accuracy** with an MAE of 1.4619.

Table 4.9: *Confusion Matrix for Ordinal Regression with SBERT in Story Point Estimation*

True/Predicted	1	2	3	5	8
1	121	1114	564	31	1

2	54	885	622	32	0
3	17	418	682	111	0
5	3	226	663	199	0
8	3	140	469	164	2

- **LSTM**:
  - **Why We Chose It**: This model looks at the order of words in a sentence, which might help with the flow of user stories.
  - **What We Found**: It achieved **37.57% accuracy** with an MAE of 1.5351.

Table 4.10: *Confusion Matrix for LSTM with BERT in Story Point Estimation*

True/Predicted	1	2	3	5	8
1	555	209	47	90	15
2	369	264	65	82	17
3	176	135	92	188	23
5	98	84	66	259	38
8	77	74	28	155	55

Adding Extra Information and Cleaning Data

We also tried to add extra details to the user stories and add more cleaning to the text.

1. **Metadata**: We included special features to give the model more clues about the tasks:
   - **Uncertainty & Risk Features**: We checked for signs of uncertainty, which means more work.
     - **Uncertainty Score**: We counted words like "investigate," "explore," or "research" showing how much exploration is needed.
     - **Question Count**: We counted question marks (?) to spot unclear requirements.
   - **Specification Quality & Scope Features**: We measured size and clarity to gauge complexity.
     - **Text Length**: We counted characters in each story; longer text might mean more work.
     - **Readability Grade**: We used the Flesch-Kincaid test to get a grade level showing text difficulty.
   - **Task Type Features**: We identified task types to predict effort.

- We labeled tasks as "bug", "feature", "refactor", or "other" based on keywords in the text.
- **Results**: When we added these features to SBERT, we got **37.1% accuracy** with an MAE of 1.601.

Table 4.11: Confusion Matrix for SBERT with Metadata in Story Point Estimation.

True/Predicted	1	2	3	5	8
1	925	508	204	124	70
2	575	583	205	135	95
3	289	246	345	231	117
5	183	163	225	345	175
8	112	125	108	214	219

2. **Preprocessing**: We cleaned the data by removing URLs (web links)
   - **Results**: Using SVR with SBERT after this cleanup gave us an MAE of **1.3598**.

Table 4.12: Confusion Matrix for SVR with Preprocessed SBERT Embeddings

True/Predicted	1	2	3	5	8
1	389	976	418	48	0
2	190	856	504	43	0
3	46	388	645	149	0
5	19	180	572	318	2
8	11	127	370	269	1

   - This shows that removing URLs has a little impact

Table 4.13*: Final Ranking of Models for Story Point Estimation by MAE and Accuracy*

Model Approach	Feature Set	Test MAE	Test Accuracy
Regression (SVR)	SBERT (all-MiniLM-L6-v2)	1.367	33.5%
Classification (SVM)	SBERT + fastText	1.461	40.40%
Ordinal Regression	SBERT	1.4619	29.00%
Classification (SVM)	SBERT (all-MiniLM-L6-v2)	1.465	41.30%

Classification (SVM)	SBERT (all-mpnet-base-v2)	1.4775	41.00%
Regression (SVR)	GPT-2	1.5028	29.7%
XGBRegressor	SBERT	1.5038	26.73%
Classification (SVM)	fastText	1.5059	40.00%
LSTM	BERT	1.5351	37.57%
Classification (SVM)	TF-IDF	1.5504	38.66%
Classification (SVM)	SBERT + Metadata	1.601	37.10%
Classification (SVM)	BERT	1.6694	36.33%
Classification (SVM)	GloVe	1.8759	33.10%
Linear SVM (Classifier)	SBERT	1.8601	35.50%

Conclusion:

- We started with classic NLP (TF-IDF, fastText), achieved a breakthrough with contextual SBERT embeddings.
- **The Best Classifier:** The SVM Classifier with SBERT (MiniLM) gives the highest Accuracy (41.3%), making it the best choice if we need the exact story point.
- **The Best Regressor:** The SVR with SBERT (MiniLM) gives the lowest MAE (1.367), making it the most reliable model for being "close" to the right answer, which is arguably more useful for sprint planning.
- Finally, The choice of embedding technology (**SBERT**) was the single most important factor, and the choice between a classification or regression model depends on the business need: **perfect accuracy vs. overall closeness**.

Switching to a Separate Project

To make our test more realistic, like when a team works on a new project with no old data, we used a different project just for testing. This dropped our accuracy to 20-23%, showing it was hard to use what we learned on old data for something new. We think this happened because the random split let the training and test data share some similar projects, which isn't fair. A comparison between different settings has been made:

Table 4.14: *Results of Separate Project (Initial)*

	SVM + TF-IDF	**SVM + BERT**
C=15 gamma=0.001	Acc:0.235 MAE:1.295	Acc:0.2523 MAE:1.4363

C=5 gamma=0.001	Acc:0.198 MAE: 1.2584	Acc:0.2540 MAE:1.4376
C=1 gamma=0.01	Acc:0.2007 MAE: 1.254	Acc:0.2536 MAE:1.4196
C=10 gamma=auto	Acc:0.1968 MAE: 1.258	Acc:0.2509 MAE: 1.44
C=20 gamma=auto	Acc:0.1968 MAE: 1.258	Acc:0.2514 MAE:1.4346

Comment: This table shows that BERT with SVM started better than TF-IDF, but the accuracy didn't improve much with different settings

Trying to Improve with the Separate Project

To make the separate project work better, we augmented the training data to introduce variety, but this only slightly improved results. Then, we tried filtering to filter similar user stories if they were more than 90% alike, which improved results slightly.

Table 4.15: *Results of Separate Project with Augmentation and Filtering.*

	SVM + TF-IDF	**SVM + BERT**	**SVM + BERT (Augmented Data)**	**SVM + BERT (Augmented + Filtered Data)**
C=15 gamma=0.001	Acc:0.235 MAE:1.295	Acc:0.2523 MAE:1.4363	Acc:0.262 MAE: 1.420	Acc:0.258 MAE: 1.393
C=5 gamma=0.001	Acc:0.198 MAE: 1.2584	Acc:0.2540 MAE:1.4376	Acc:0.2509 MAE: 1.458	Acc:0.245 MAE: 1.4205
C=1 gamma=0.01	Acc:0.2007 MAE: 1.254	Acc:0.2536 MAE:1.4196	Acc:0.2611 MAE: 1.418	Acc:0.258 MAE: 1.385
C=10 gamma=auto	Acc:0.1968 MAE: 1.258	Acc:0.2509 MAE: 1.44	Acc:0.2597 MAE:1.430	Acc:0.2589 MAE: 1.3967
C=20 gamma=auto	Acc:0.1968 MAE: 1.258	Acc:0.2514 MAE:1.4346	Acc:0.2659 MAE: 1.407	Acc:0.2668 MAE: 1.37

Comment: Adding more data and filtering similar stories helped a bit, showing small gains, but still room for improvement.

Breaking Tasks into Smaller Parts

Recognizing the ambiguity in the user stories writing format, we used a Large Language Model (LLM) to break them into subtasks. We tried different ways:

1. We trained the SVM using BERT and TF-IDF embeddings, where the training text was the original user story text. For testing, we calculated the story points by averaging the points predicted for each subtask. (Columns 5, 6)

2. We trained the SVM using BERT embeddings, where the training text was all subtasks combined into one piece of text. For testing, we predicted the story points directly using the original user story text from the test set. (Column 7)
3. We trained the SVM using BERT embeddings, where the training text was all subtasks combined into one piece of text. For testing, we predicted the story points by averaging the points calculated for each subtask of every user story. (Column 8)

This approach improved explainability and boosted performance on the separate project, with the best result at **28.7% accuracy** and MAE of 1.169 on the XD project using SVM + BERT, where training BERT on subtasks combined into a single text, and testing by averaging the story points of subtasks for each user story.

Table 4.16: *Results of Separate Project with Subtask Decomposition.*

	SVM + TF-IDF	SVM + BERT	SVM + BERT (Augmented Data)	SVM + BERT (Augmented + Filtered Data)	SVM + BERT (Averaging SP of Subtasks)	SVM + TF-IDF (Averaging SP of Subtasks)	SVM + BERT (Subtasks as Single Text, Direct Prediction)	SVM + BERT (Subtasks as Single Text, Averaging SP of Subtasks)
C=15 gamma=0.001	Acc: 0.235 MAE:1.295	Acc: 0.2523 MAE:1.4363	Acc: 0.262 MAE: 1.420	Acc: 0.258 MAE: 1.393	Acc: 0.2395 MAE: 1.2346	Acc:0.21576 MAE: 1.1836	Acc: 0.2862 MAE:1.3434	Acc:0.287 MAE:1.169
C=5 gamma=0.001	Acc: 0.198 MAE: 1.2584	Acc: 0.2540 MAE:1.4376	Acc: 0.2509 MAE: 1.458	Acc: 0.245 MAE: 1.4205	Acc: 0.2404 MAE: 1.3069	Acc: 0.2228 MAE 1.808	Acc:0.2814 MAE:1.356	Acc:0.282 MAE:1.166
C=1 gamma=0.01	Acc: 0.2007 MAE: 1.254	Acc: 0.2536 MAE:1.4196	Acc: 0.2611 MAE: 1.418	Acc: 0.258 MAE: 1.385	Acc: 0.2417 MAE: 1.247	Acc: 0.2034 MAE: 1.2487	Acc:0.2870 MAE:1.339	Acc: 0.2567 MAE: 1.2659
C=10 gamma=auto	Acc: 0.1968 MAE: 1.258	Acc: 0.2509 MAE: 1.440	Acc: 0.2597 MAE:1.430	Acc: 0.2589 MAE: 1.3967	Acc: 0.2408 MAE: 1.246	Acc:0.2228 MAE: 1.808	Acc:0.2870 MAE:1.3412	Acc: 0.2774 MAE:1.1809
C=20 gamma=auto	Acc: 0.1968 MAE: 1.258	Acc: 0.2514 MAE:1.4346	Acc: 0.2659 MAE: 1.407	Acc: 0.2668 MAE: 1.37	Acc: 0.246 MAE: 1.210	Acc: 0.202 MAE: 1.2448	Acc:0.2860 MAE:1.340	Acc: 0.2796 MAE: 1.1906

Comment: Splitting tasks into subtasks and averaging improved the model, with the best MAE of 1.169, showing that subtasks help a lot. The confusion matrix showed how well it predicted.

Table 4.17: Confusion Matrix for Best Subtask Model at C=15 and gamma=0.001

True/Predicted	1	2	3	5	8
1	138	123	101	98	42
2	92	78	100	117	60
3	90	99	159	163	77
5	26	47	96	172	92
8	11	14	49	122	105

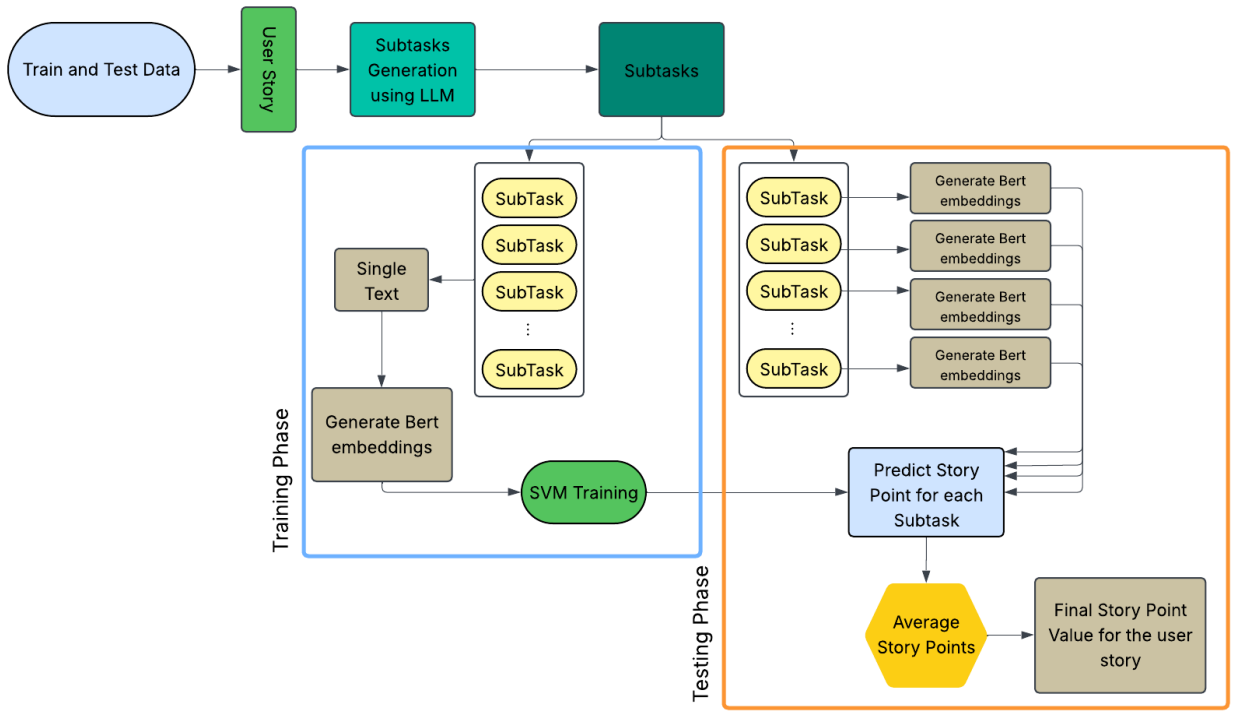


***Figure 4.7:** Flow of Subtask Decomposition*

Testing on Different Projects

We tested our best approach (train the SVM using BERT embeddings, and the training text is all subtasks combined into one piece of text, and in testing, we predict the story points by averaging the points calculated for each subtask for every user story) on other projects like NEXUS, TIMOB, MESOS, and XD to see if it worked everywhere.

To further show the impact of the new approach, we also evaluated each project using **Normal SVM + BERT direct prediction on user stories**, which corresponds to Column 2 above. The subtask method usually did better.

Table 4.18: *Results Across Different Projects*

Hyperparameters	*NEXUS (Avg Subtasks)*	*NEXUS (Direct US)*	*TIMOB (Avg Subtasks)*	*TIMOB (Direct US)*	*MESOS (Avg Subtasks)*	*MESOS (Direct US)*	*XD (Avg Subtasks)*	*XD (Direct US)*
C=15, gamma=0.001	Acc: 0.309 MAE: 1.1135	Acc: 0.299 MAE: 1.201	Acc: 0.2730 MAE: 1.2442	Acc: 0.2576 MAE: 1.4243	Acc: 0.2646 MAE: 1.173	Acc: 0.2364 MAE: 1.3476	Acc: 0.287 MAE: 1.169	Acc: 0.2523 MAE: 1.4363
C=5, gamma=0.001	**Acc: 0.296 MAE: 1.1940**	**Acc: 0.303 MAE: 1.2017**	**Acc: 0.2788 MAE: 1.2375**	**Acc: 0.2688 MAE: 1.3909**	**Acc: 0.2655 MAE: 1.1668**	**Acc: 0.2246 MAE: 1.379**	**Acc: 0.282 MAE: 1.166**	**Acc: 0.2540 MAE: 1.4376**
C=1, gamma=0.01	Acc: 0.2425 MAE: 1.323	Acc: 0.2899 MAE: 1.212	Acc: 0.2872 MAE: 1.2426	Acc: 0.2655 MAE: 1.3918	Acc: 0.259 MAE: 1.163	Acc: 0.2320 MAE: 1.3455	Acc:0.2567MAE: 1.265	Acc: 0.2536 MAE: 1.4196
C=10, gamma=auto	Acc: 0.289 MAE: 1.1565	Acc: 0.2976 MAE: 1.202	Acc: 0.275 MAE: 1.241	Acc: 0.2559 MAE: 1.427	Acc: 0.2598 MAE: 1.178	Acc: 0.236 MAE: 1.350	Acc:0.2774 MAE: 1.1809	Acc: 0.2509 MAE: 1.440
C=20, gamma=auto	Acc: 0.3065 MAE: 1.0926	Acc: 0.3042 MAE: 1.186	Acc: 0.2796 MAE: 1.238	Acc: 0.2576 MAE: 1.4285	Acc: 0.2677 MAE: 1.1429	Acc: 0.239 MAE: 1.332	Acc:0.2796 MAE: 1.1906	Acc: 0.2514 MAE: 1.4346

Comment: The subtask method (Avg Subtasks) generally had lower MAE and higher accuracy, proving it works better across projects

The best settings across projects were C=5 and gamma=0.001. Subtasks helped because:

- **Granularity**: Breaking tasks into parts made them clearer.
- **Explainability**: Teams could see why a story point value was chosen, based on the story point of each subtask and how each subtask contributed to the final estimation of the user story
- **Performance**: Averaging points reduced mistakes, though new projects were still tricky.

Transitioning to a Robust Validation Framework

Following our initial exploratory experiments across multiple projects, we thought to formalize our model selection process. To ensure our model could generalize well to new, unseen projects and to select hyperparameters in an unbiased manner, we adopted **Group K-Fold cross-validation**. This methodology is critical as it prevents data leakage by ensuring that all user stories from a single project are contained within the same fold, either entirely in the training set or entirely in the validation set. This simulates the real-world scenario of deploying the model on a new project.

We also recognized that calculating Mean Absolute Error (MAE) on arbitrary class labels (e.g., 0, 1, 2, 3, 4) was not providing a true measure of real-world impact. An error between Class 0 and Class 4 is numerically "4", but an error between a 1-point and an 8-point story is

"7 points". Therefore, we shifted to calculating MAE directly on the mapped story points (1, 2, 3, 5, 8) in all the following results.

Our primary goal was to rigorously compare our two main feature extraction strategies: using the high-level text versus the granular subtasks.

Results: Text vs. Subtasks with GroupKFold Validation

We applied our GroupKFold cross-validation pipeline to the SVM model with BERT embeddings for both feature sets.

1. **Model 1: SVM with BERT on text**
    - **Test MAE:** 2.1757
    - **Test Accuracy:** 26.38%
2. **Model 2: SVM with BERT on subtasks**
    - **Test MAE:** 1.8512
    - **Test Accuracy:** 27.12%

Analysis: The model using subtasks as its input feature demonstrated a clear and significant improvement. Not only did the overall accuracy increase from **26.4% to 27.1%**, but more importantly, the Mean Absolute Error (MAE) saw a substantial reduction from **2.18 to 1.85**. This indicates that the subtask model is more accurate at exact predictions and more reliable at making predictions that are closer to the true value.

The confusion matrix for the model with subtasks is shown below.

Table 4.19: *Confusion Matrix for SVM with Subtasks (GroupKFold Validated)*

True/Predicted	***1***	***2***	***3***	***5***	***8***
1	159	134	101	82	26
2	97	95	101	122	32
3	108	127	145	162	46
5	37	62	112	147	75
8	10	25	57	139	70

Investigating Class Imbalance

Despite the improved performance of the subtask model, we observed from its confusion matrix that it still struggled to correctly identify minority classes, particularly the highest-effort classes. To address this, we investigated a standard over-sampling technique, SMOTE (Synthetic Minority Over-sampling Technique).

Experiment: The Effect of SMOTE We integrated SMOTE into our GroupKFold pipeline, ensuring it was applied only to the training data within each fold to prevent data leakage.

- **Approach:** Generated synthetic samples for minority classes.
- **Results:**
    - Test MAE: **2.4042**
    - Test Accuracy: **22.10%**

Analysis: The result of applying SMOTE was definitive and highly informative. The model's performance degraded significantly, with the accuracy dropping and the MAE increasing. The confusion matrix revealed a case of **model collapse**, where the model learned almost exclusively to predict the majority class.

Table 4.20: *Confusion Matrix for Subtask Model with SMOTE*

True/Predicted	***1***	***2***	***3***	***5***	***8***
1	502	0	0	0	0
2	447	0	0	0	0
3	587	1	0	0	0
5	433	0	0	0	0
8	300	1	0	0	0

We concluded that SMOTE introduced non-beneficial noise for high-dimensional BERT embeddings where the feature space is already complex. This experiment was critical, as it proved that the root issue is the quality and differentiability of the user story for different classes, not the number of samples.

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.

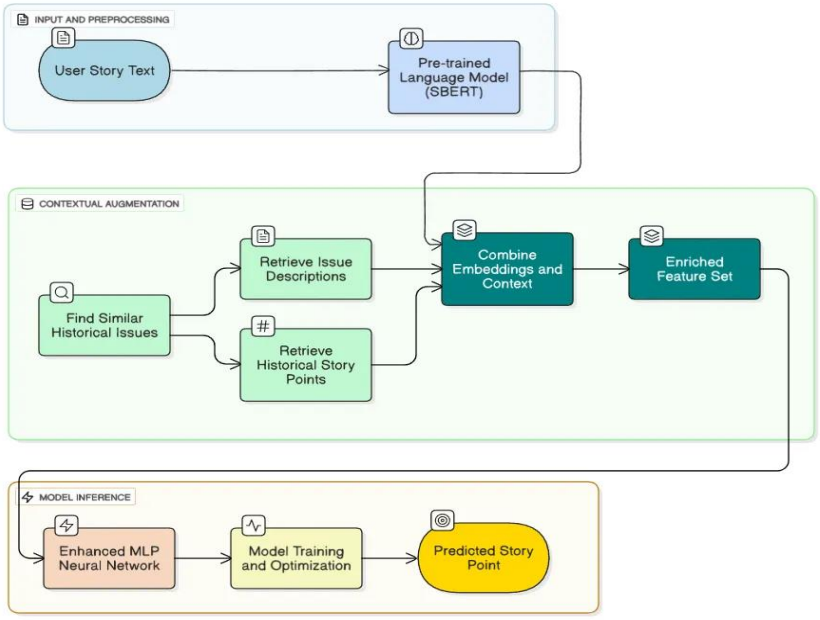


Figure 4.8: *Flow of the Enhanced MLP process for story point estimation.*

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.

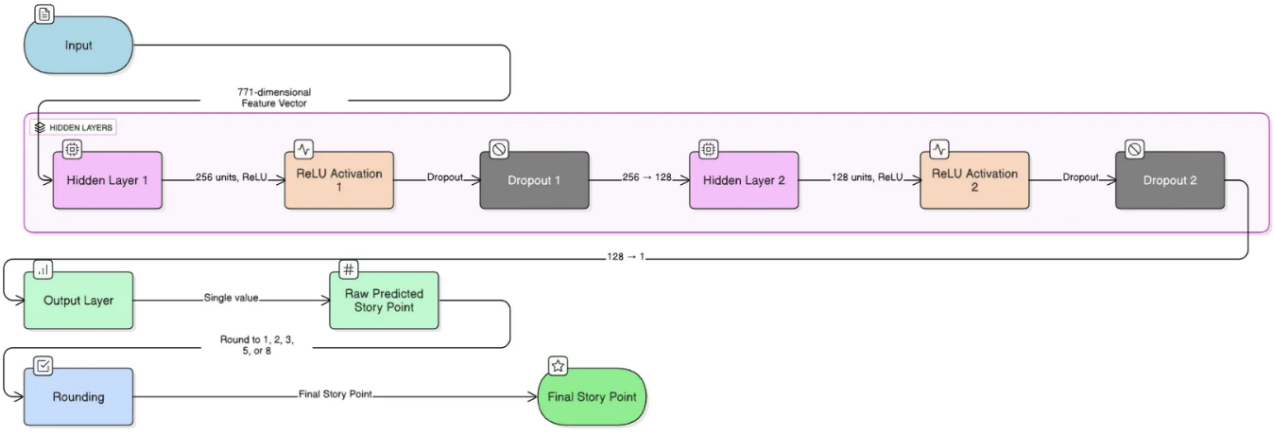


Figure 4.9: *The Enhanced MLP Architecture*

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.

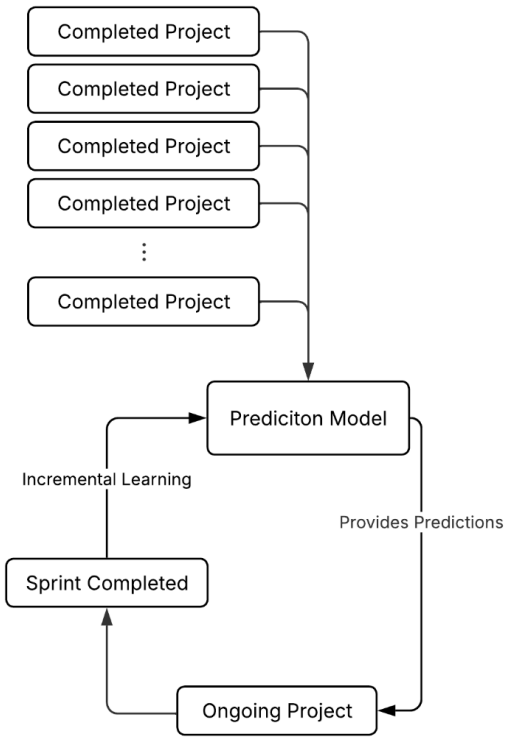


***Figure 4.10**: Flow of the incremental learning process.*

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.

4.4.3. Design Constraints

The design of the Story Point Estimation module faced several constraints that shaped its architecture:

1. **Computational Efficiency**:

   a. **Constraint**: Agile sprints require fast predictions (seconds, not hours), and large datasets demand scalable processing. Models like DEEP SE require GPUs and are slow to train.
   b. **Impact**: Favored lightweight models (SVM, SVR, PassiveAggressiveClassifier, SGDRegressor) over complex neural networks for most sub-modules. Incremental learning was critical for fast updates.
   c. **Mitigation**: Used TF-IDF and SBERT for efficient embeddings, truncated text to 500 tokens, and implemented incremental models with ~0.02-0.06-second updates.
2. **Generalization Across Projects**:
   a. **Constraint**: Project-specific patterns (e.g., JAVA vs. CXX) reduce model generalization, with separate project testing yielding 20-23% accuracy compared to 36.33% for random splits.
   b. **Impact**: Required subtask decomposition and context-aware features to capture project-specific nuances, and incremental learning to adapt to new projects.
   c. **Mitigation**: Employed subtask decomposition (28.7% accuracy on XD), context-aware features (Enhanced MLP: MAE 1.389), and cross-project training with incremental updates.
3. **Interpretability**:
   a. **Constraint**: Agile teams need explainable predictions to trust and act on estimates.
   b. **Impact**: Prioritized models like Enhanced MLP and subtask decomposition, which provide insights (e.g., contributions of subtasks or similar stories) over black-box models like DEEP SE.
   c. **Mitigation**: Used subtask averaging for explainability and context-aware features to link predictions to historical examples.

4.4.4. Other Description of Module 2

The Story Point Estimation module was designed to balance accuracy, speed, and interpretability, addressing the unique challenges of Scrum automation. Below are additional insights into its design and implementation:

- **Dataset Reliance**: The module relies on the TAWOS dataset, which, while comprehensive, is limited to open-source projects. This may affect generalizability to enterprise projects with different terminologies or estimation practices. Future work could test the module on diverse datasets to validate its robustness.
- **Subtask Decomposition Benefits**: Breaking user stories into subtasks (Part 1) improved explainability by showing how individual components contribute to the final story point. For example, the confusion matrix for the best subtask model balanced predictions across Fibonacci values.

Table 4.25: *Confusion matrix for the best subtask decomposition model*

True/Predicted	***1***	***2***	***3***	***5***	***8***
1	138	123	101	98	42
2	92	78	100	117	60
3	90	99	159	163	77
5	26	47	96	172	92
8	11	14	49	122	105

- **Context-Aware Features**: The Enhanced MLP (Part 2) leverages similar past user stories, improving MAE by 5.2% (context vector) and 3.9% (statistical features). For instance, on the JAVA project, it achieves MAE 0.119, demonstrating the value of historical context.
- **Incremental Learning Practicality**: The final approach's incremental models are critical for agile environments, updating in ~0.02-0.06 seconds compared to ~600-1067 seconds for full retraining. This ensures compatibility with the fast-paced nature of sprints, maintaining performance (e.g., TIMOB batch 10: MAE 1.7832) while adapting to concept drift.

4.5. Module 3: Task Assignment

Module 3 introduces an advanced Task Assignment System designed to streamline the process of allocating tasks to developers in agile software development environments. This module leverages machine learning to automate task assignments, reducing the manual effort required by project managers and enhancing assignment accuracy by matching tasks to developers based on their expertise and historical performance. The module is divided into two parts: an Offline Task Assignment System, which employs batch processing with classic and deep learning approaches (Topic Modeling with LDA and Word Embeddings with LSTM), and an Online Task Assignment System, which uses real-time, adaptive online learning to handle dynamic team and project changes. Together, these components provide a robust, data-driven solution that balances computational efficiency, adaptability to concept drift, and seamless integration with tools like Jira, ultimately optimizing task allocation in agile workflows.

4.5.1 Offline Task Assignment System

4.5.1.1 Functional Description

The Task Assignment module is a critical component of the AI-powered Scrum automation system. Its primary purpose is to automate the manual and often time-consuming process of assigning new tasks or issues to the most appropriate developer within the team. By analyzing the content of a new task and leveraging historical data, the module aims to improve assignment accuracy, reduce the workload on project managers or triagers, and ensure that tasks are handled by developers with the most relevant expertise.
To achieve this, two distinct machine learning approaches were designed, implemented, and evaluated:

1. A classic machine learning pipeline using **Topic Modeling (LDA)**.
2. A deep learning pipeline using **Word Embeddings with a Neural Network**.

From a functional standpoint, the Task Assignment module operates as an intelligent recommendation system.

- **Input**: The module takes a new task as input. This task is expected to have a title and a detailed description of the work required.
- **Process**: It analyzes the textual content of the input task. Using one of the two implemented machine learning models, it compares the characteristics of the new task against a historical record of previously completed tasks and their assignees.
- **Output**: The module's final output is a ranked list of the top three most suitable developers for the given task. This provides the project manager with a well-informed, data-driven recommendation, while still allowing for a final human decision.

4.5.1.2 Modular Decomposition

The Task Assignment module is coarse-grained and has been decomposed into two fine-grained, independent sub-modules, each representing a complete, end-to-end approach to solving the assignment problem.

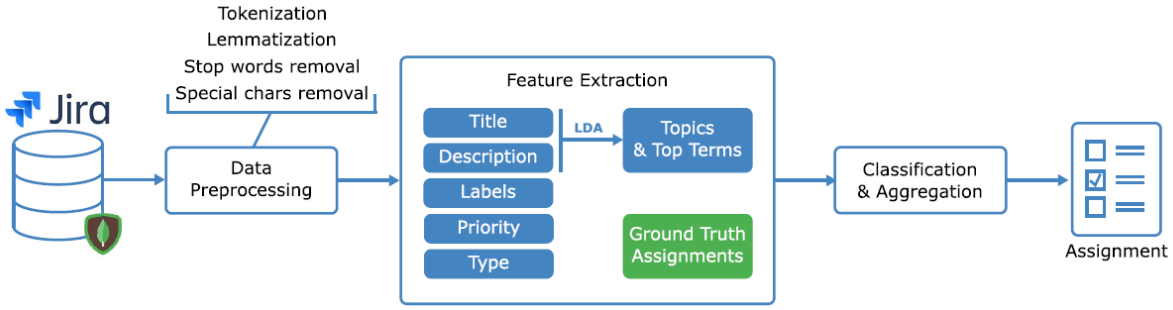


Figure 4.11: *A workflow illustrating data preprocessing and Latent Dirichlet Allocation (LDA) on Jira issues to automate task assignments through classification.*

Sub-module 1: Topic Modeling with Latent Dirichlet Allocation (LDA)

This approach builds a recommendation system based on identifying abstract topics within the issue data. The architecture is a multi-stage pipeline:

1. **Data Preprocessing**: Raw text from issue titles and descriptions is cleaned. This involves tokenization, lemmatization, and the removal of stopwords and special characters to prepare the text for analysis.
2. **Feature Extraction & Topic Modeling**: Latent Dirichlet Allocation (LDA) is applied to the cleaned text to discover a set of underlying topics. For any given issue, this process yields two key features: a vector representing the issue's distribution across topics and the top terms that define its dominant topic.
3. **Enhanced Feature Creation**: To enrich the feature set, "Enhanced Labels" are created by merging the issue's original labels with the top terms extracted from its dominant topic via LDA.
4. **Classification and Aggregation**: A systematic process trains multiple models and combines their predictions to generate a final, robust recommendation. The specific implementation of this stage is detailed below.

Implementation of Classification and Aggregation

The classification pipeline is engineered for high performance by leveraging the NVIDIA RAPIDS ecosystem (cuDF, cuML, cuPy) for GPU acceleration, allowing it to efficiently process large datasets.

- **Technology Stack**: The entire workflow utilizes GPU-accelerated libraries. Data is loaded and manipulated using cuDF DataFrames, numerical operations are handled

by cuPy arrays, and the core machine learning tasks are performed with cuML. Proactive GPU memory management is integrated throughout the code to prevent memory overflows.

- **TF-IDF Transformation**: Textual features (title, description, etc.) are converted into numerical vectors using a TF-IDF (Term Frequency-Inverse Document Frequency) strategy. To maintain computational efficiency and manage memory, dimensionality is reduced by setting max_features=1000 (keeping only the top 1,000 terms) and min_df=5 (ignoring terms that appear in fewer than five documents).
- **Classifier Models**: Two distinct classifiers are trained on the TF-IDF features:
    1. **Linear Support Vector Machine (SVC)**: The high-performance cuml.svm.LinearSVC is used for training on the GPU. Since LinearSVC does not natively output probabilities, the model is calibrated using sklearn.calibration.CalibratedClassifierCV in a hybrid GPU-CPU process. This allows for the generation of the probability scores necessary for the weighted ensemble.
    2. **Multinomial Naïve Bayes**: The cuml.naive_bayes.MultinomialNB model is also implemented, providing a fast and effective GPU-native classification alternative.
- **Prediction and Weighted Ensemble**: For both classifiers, predictions on the test set are performed in batches to conserve GPU memory. The probabilities from models trained on different features (e.g., title, description, labels, topics) are then combined using a weighted_vote function. This function applies a predefined set of weights to the probabilities from each feature-specific model and calculates a weighted average to produce a final, ensembled prediction.
- **Optimization Loop**: The entire pipeline—from feature extraction to classification and aggregation—is executed within a nested loop. The outer loop iterates through different projects, while an inner loop tests various configurations for the number of topics (e.g., 4, 6, 8, ...). This systematic process allows for the empirical determination of the optimal number of topics for each project by comparing the final accuracy of each configuration. The results for all configurations are saved for analysis.
- **Output Artifacts**: The process concludes by saving all predictions and probabilities for every tested combination (project, topic number, classifier type, and ensemble configuration) into a compressed .json.gz file for comprehensive review and evaluation.

Sub-module 2: Word Embedding with an LSTM Neural Network

This approach utilizes a modern deep learning architecture to learn the relationship between task descriptions and developer assignments directly.

1. **Data Preparation**: The process begins by loading pre-processed textual data and a pre-trained word embedding matrix. Developer IDs are encoded into numerical labels suitable for the neural network.

2. **Model Architecture**: The core of this sub-module is a Long Short-Term Memory (LSTM) neural network, a type of Recurrent Neural Network (RNN) well-suited for sequential data like text. The specific layers are:
    - An **Embedding Layer** that converts the text's word indices into dense vectors using the pre-trained embedding matrix. This layer is set to be "trainable" to fine-tune the embeddings for the specific dataset.
    - An **LSTM Layer** that processes the sequence of vectors to capture the meaning and context of the entire task description.
    - **Dropout Layers** are used after the LSTM and dense layers to prevent the model from overfitting.
    - **Dense (Fully Connected) Layers** that act as the classifier, taking the output from the LSTM layer and ultimately producing a final probability score for each developer.
3. **Training and Evaluation**: The model is trained using the Adam optimizer and CrossEntropyLoss function. Early stopping is employed to cease training when validation accuracy no longer improves, ensuring an efficient process.

4.5.1.3 Design Constraints

The design of both sub-modules was influenced by several key constraints:

- **Historical Data Requirement**: Both approaches are fundamentally dependent on the availability of a large and clean historical dataset of issues with correct assignments. The models' effectiveness is directly proportional to the quality and quantity of this training data.
- **Core Contributor Focus**: The system is designed to recommend from a pool of active, core developers who have handled a significant number of tasks previously (e.g., a minimum of 80 issues in the reference study). It does not account for assigning tasks to new or infrequent contributors.
- **Descriptive Text Necessity**: The accuracy of both models relies on issues having clear, well-written titles and descriptions. Issues with sparse or non-descriptive text cannot be analyzed effectively.
- **Computational Cost**: The deep learning (LSTM) approach has higher computational requirements, demanding more training time and preferably a GPU, compared to the topic modeling approach.
- **Parameter Optimization**: The Topic Modeling approach requires project-specific tuning of parameters, such as the number of topics and the feature weights for aggregation, to achieve optimal performance.

4.5.1.4 Other Description of Module 3 Part 1

For a complete picture of the module, the following points detailing the evaluation framework and output artifacts are relevant to both implemented approaches.

- **Evaluation Framework**: The module's performance was rigorously tested using a standard train-test split methodology. For both the Neural Network and Topic Modeling pipelines, the system loads distinct training and testing datasets (e.g., X_train, X_test, y_train, y_test) to ensure the models are evaluated on unseen data.
- **Performance Metrics**: To ensure a comprehensive assessment of the module's effectiveness, a standard set of classification metrics is used. The Neural Network pipeline, for instance, explicitly calculates and saves the following metrics:
    - Accuracy
    - Macro-averaged Precision
    - Macro-averaged Recall
    - Macro-averaged F1-Score
- **Reproducibility and Reusability**: The training process for both sub-modules was designed to produce reusable artifacts that allow for analysis and deployment without needing to retrain.
    - The **Neural Network** approach saves the final trained model weights to a .pth file and the final performance scores to a .json file.
    - The **Topic Modeling** approach systematically saves the detailed predictions (y_pred) and output probabilities (y_pred_proba) for every tested configuration into a compressed .json.gz file, ensuring all results are captured for later analysis.

4.5.2 Online Task Assignment System

4.5.2.1 Functional Description

The primary function of Module 3 is to automate task assignment in agile software development by predicting and recommending the most suitable developer for a given Jira issue, using online machine learning to adapt dynamically to evolving team dynamics and assignment patterns. Unlike offline learning approaches that rely on static datasets and periodic retraining, this module processes Jira issues in a streaming fashion, enabling continuous learning and adaptation to concept drift without requiring full model retraining. This functionality addresses the need for flexibility in agile environments, where task assignments are influenced by technical suitability, learning objectives, workload balancing, and team evolution.

Module 3 operates in two distinct modes:

- **Training Mode**: Fetches historical Jira issues for a specified project, preprocesses the data, trains an online learning model, and saves the trained model to AWS S3 for deployment.
- **Testing Mode**: Processes new Jira issues in real-time, predicts the top N (typically 3) recommended developers for assignment, and updates the model incrementally upon receiving the true assignment from a user or manager.

The input to the module is a Jira issue represented as a JSON object, containing attributes such as:

- **Summary**: A concise title describing the issue (e.g., “Implement user authentication endpoint”).
- **Description**: Detailed text outlining the issue, often in Atlassian Document Format (ADF).
- **Labels**: Tags associated with the issue (e.g., “PULL-req”, “bug”).
- **Components**: System components affected by the issue.
- **Priority**: The issue’s priority level (e.g., “Medium”, “High”).
- **Issue Type**: The category of the issue (e.g., “Task”, “Bug”).

The output is a ranked list of the top 3 recommended developers, along with an updated model state after learning from the true assignment.

4.5.2.2 Modular Decomposition

Module 3, responsible for automating task assignment in agile software development using online machine learning, is decomposed into five fine-grained components to ensure modularity, maintainability, and scalability. Each component handles a specific aspect of the task assignment pipeline, from data acquisition to model deployment and user interaction. This modular design facilitates independent development, testing, and integration with other project modules, while enabling seamless operation within the agile workflow. The components are designed to be interoperable, communicating through standardized data formats such as JSON, CSV, and pickle files. Figure 1 illustrates the pipeline architecture, showing the flow of data through the components.

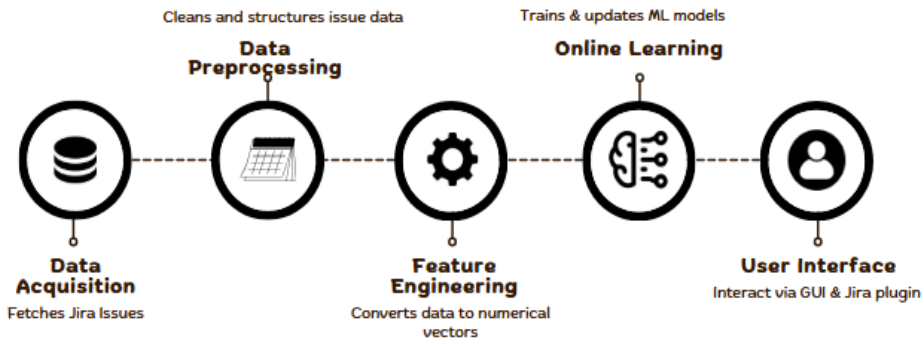


Figure 4.12: *Module 3 Online Learning Pipeline*

The five components of Module 3 are:

1. **Data Acquisition Component**:
    - **Function**: Retrieves Jira issues from the Jira REST API, supporting both training and Training modes. In training mode, it fetches historical issues for a

specified project to build a baseline model. In Training mode, it retrieves individual new issues for real-time processing.

- **Implementation**: Utilizes the Python requests library to make API calls, authenticating with an email and API token. The get_text_from_description function parses Atlassian Document Format (ADF) descriptions into plain text, extracting attributes such as summary, description, labels, components, priority, and issue type. Issues are saved as JSON files in the data/ directory for training or deploy and test/jsons/ for deployment.
- **Output**: JSON files containing raw Jira issue data, structured for downstream processing.
- **Rationale**: This component ensures robust integration with Jira, handling API rate limits and errors, and provides a flexible interface for fetching data in both batch and streaming scenarios.

2. **Data Preprocessing Component**:
   - **Function**: Transforms raw JSON issue data into a structured format suitable for machine learning, addressing inconsistencies and preparing data for feature engineering.
   - **Implementation**: Employs the pandas library to clean and normalize data, handling missing values (e.g., replacing null fields with empty strings) and converting complex attributes (e.g., lists of labels) into strings. Assignee names are mapped to numerical IDs, with mappings saved for later use in deployment. The processed data is stored as CSV files in data/ for training or deploy and test/csvs/ for deployment.
   - **Output**: CSV files with columns such as summary, description, labels, components, priority, issue_type, and assignee_num, ready for feature extraction.
   - **Rationale**: This component ensures data consistency and compatibility with machine learning models, enabling efficient processing of diverse Jira issue attributes.

3. **Feature Engineering Component**:
   - **Function**: Converts structured issue data into numerical feature vectors for input to online learning models, capturing the semantic and categorical properties of issues.
   - **Implementation**: Utilizes a custom feature engineering pipeline defined in the MyOnlineModel class, which orchestrates feature selection and transformation. Text attributes (e.g., summary, description) are processed using Term Frequency-Inverse Document Frequency (TF-IDF), defined as:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \cdot \log\left(\frac{N}{\text{DF}(t)}\right)$$

where $TF(t,d)$ is the term frequency of term t in document d, $DF(t)$ is the number of documents containing t, and N is the total number of documents. Categorical attributes (e.g., priority, issue type) are encoded numerically. The pipeline ensures sparse, high-dimensional data is handled efficiently.

- **Output**: Numerical feature vectors representing each Jira issue, suitable for model training and prediction.
- **Rationale**: This component bridges the gap between raw data and machine learning, enabling models to learn from both textual and categorical issue attributes effectively.

4. **Online Learning Component** :
    - **Function**: Trains and evaluates online learning models, adapting to new issues and concept drift in a streaming fashion. It supports eight models, from simple classifiers (e.g., Naive Bayes) to advanced ensembles (e.g., AdaBoost ensemble with 10 Naive Bayes models).
    - **Implementation**: Uses the river library to implement models in a test-then-train loop, where each issue is first used for prediction, then for updating the model. Models are trained incrementally, with performance metrics (e.g., accuracy, drift detection counts) recorded. Trained models are serialized as pickle files and uploaded to AWS S3 for persistence and deployment. The component also supports experiment mode, running models across multiple projects in parallel using multiprocessing.
    - **Output**: Trained model files (.pkl) stored in AWS S3, along with JSON files containing performance metrics (e.g., accuracy, drift counts) in results_new_1/.
    - **Rationale**: This component is the core of Module 3, enabling dynamic learning and adaptation to evolving assignment patterns, with robust storage and evaluation mechanisms.

5. **User Interface Component**:
    - **Function**: Provides a graphical interface for users to interact with the task assignment system, enabling retrieval of Jira issues for training, real-time model training, deployment to AWS S3, and prediction of developer assignments via a Jira plugin, supporting both operational use and experimentation.
    - **Implementation**: The Training tab is built using Python's tkinter library, enabling users to retrieve Jira issues via the Jira API (e.g., jira-python), input issue details (e.g., summary, description, priority), and train online learning models (e.g., Naive Bayes, AdaBoost with 10 Naive Bayes models) incrementally using the MyOnlineModel framework. Trained models are uploaded to AWS S3 using the AWS SDK (boto3) for persistent storage and retrieval. The Testing tab is implemented as a Jira plugin, integrated into Jira's interface, which loads the AWS-hosted model to predict top-3 developer assignments with confidence scores for

new or existing issues. Both tabs authenticate securely with Jira and AWS via user-provided credentials or configuration files.

- **Output**: The Training tab displays retrieved Jira issues, training progress, and model performance metrics (e.g., accuracy from RollingAccuracyCalc), with trained models saved as .pkl files in AWS S3. The Testing tab, via the Jira plugin, displays top-3 developer predictions with confidence scores within Jira's interface.
- **Rationale**: This component enhances usability, making the system accessible to non-technical users (e.g., project managers) and facilitating real-world deployment and experimentation.

4.5.2.3. Design Constraints

The design of Module 3, which automates task assignment in agile software development using online machine learning, is shaped by several critical constraints that influence its architecture, model selection, and implementation. These constraints arise from the need to operate effectively within the dynamic, real-world context of agile workflows, integrate seamlessly with external systems like Jira, and meet performance and usability requirements. Below, each constraint is described, along with its impact on the module's design and the strategies employed to address it.

Real-Time Processing Requirement:

- **Description**: Agile software development demands rapid task assignment to maintain workflow efficiency. Module 3 must process new Jira issues and provide developer recommendations in real-time to avoid delays in project management.
- **Impact**: This constraint limits the complexity of machine learning models. It also necessitates efficient data processing and feature extraction pipelines to handle streaming inputs quickly.
- **Solution**: The module employs lightweight online learning models from the river library, such as Naive Bayes and Passive-Aggressive classifiers, which are optimized for streaming data and low-latency predictions. The feature engineering pipeline uses Term Frequency-Inverse Document Frequency (TF-IDF) for fast text processing, ensuring that feature vectors are generated efficiently. The Training mode (models_gui.py) is optimized to process single issues rapidly, leveraging pre-trained models stored in AWS S3 to minimize computation during prediction.

Limited and Variable Historical Data:

- **Description**: The Apache Jira Issue Tracking Dataset, used for training and evaluation, varies significantly in size and quality across projects, with some containing as few as 100 issues. Additionally, data may include noise (e.g., incomplete descriptions) or inconsistent labeling (e.g., varying assignee formats).

- **Impact**: Models must perform effectively with small or noisy datasets. The variability in data quality demands robust preprocessing to handle missing or malformed attributes.
- **Solution**: The data preprocessing component uses pandas to clean and normalize data, replacing missing values with empty strings and standardizing formats (e.g., converting lists of labels to strings). Assignee names are mapped to numerical IDs using. The choice of models like Naive Bayes and Hoeffding Adaptive Tree, which perform well with sparse data, addresses the constraint of limited data. The TF-IDF pipeline ensures robust feature extraction even for short or incomplete text fields.

Concept Drift in Assignment Patterns:

- **Description**: Task assignment patterns in agile teams change over time due to factors such as new developers joining, shifts in project phases, or intentional reassignment for learning objectives (e.g., assigning tasks to less experienced developers to build skills). This phenomenon, known as concept drift, requires models that can detect and adapt to these changes dynamically.
- **Impact**: Static models or those with rigid retraining schedules are unsuitable, as they cannot adapt to evolving patterns. This constraint necessitates online learning models with built-in drift detection and adaptation mechanisms.
- **Solution**: Module 3 implements models with explicit drift detection, such as Naive Bayes with ADWIN and Hoeffding Adaptive Tree, which use the Adaptive Windowing (ADWIN) algorithm to monitor performance and reset or adapt the model when drift is detected. The Enhanced and Super Enhanced Models incorporate advanced drift handling, with the latter using probabilistic drift reset to preserve high-performing base models. The decay-based recency heuristic in the Super Enhanced Model further enhances adaptability by prioritizing recent assignments:

  $$w_d = \lambda^k, \quad \lambda = 0.95, \quad k = \text{position in recency window}$$

  This ensures the model remains responsive to changing patterns.

Integration with Jira's REST API:

- **Description**: Module 3 must interface seamlessly with Jira's REST API to fetch issues in both training and Training modes. This introduces constraints related to API rate limits, authentication requirements, and the complexity of parsing Atlassian Document Format (ADF) descriptions.
- **Impact**: The data acquisition component must handle API errors, rate limiting, and complex data structures, adding overhead to the design. Parsing ADF requires custom logic to extract meaningful text, impacting preprocessing efficiency.
- **Solution**: The data acquisition component uses the requests library with robust error handling to manage API failures and rate limits. It recursively parses ADF to convert descriptions into plain text, ensuring compatibility with the feature engineering pipeline. Authentication is handled securely using email and API token inputs via a user-friendly GUI, minimizing setup complexity for users.

User Interaction and Usability:

- **Description**: The system must be accessible to non-technical users, such as project managers, who need to interact with the Training mode to view recommendations and provide true assignments. This requires an intuitive, user-friendly interface and clear presentation of results.
- **Impact**: The GUI design must prioritize simplicity and functionality, avoiding complex technical requirements for operation. The Training mode must support interactive feedback loops, where users confirm assignments to update the model.
- **Solution**: The User Interface Component is implemented with a tkinter-based GUI featuring two tabs: Training and Testing. The Training tab allows non-technical users to retrieve Jira issues via the Jira API, input or view issue details (e.g., summary, description), and train online learning models (e.g., AdaBoost with 10 Naive Bayes models) with a single click, uploading trained models to AWS S3 for persistence using the AWS SDK (boto3). The Testing tab, implemented as a Jira plugin, integrates with Jira's interface to load AWS-hosted models and predict top-3 developer assignments with confidence scores for new or existing issues. Users can confirm true assignments within the plugin, enabling incremental model updates. The interface requires minimal configuration, with secure authentication for Jira and AWS handled through user-friendly prompts or configuration files. Logs for training and prediction events are saved to logs/ as JSON files for transparency and debugging.

These constraints collectively shaped the design of Module 3, guiding the selection of lightweight, drift-adaptive models, efficient data processing pipelines, and a user-friendly interface. By addressing these challenges, the module achieves a balance between performance, adaptability, and usability, making it suitable for real-world agile environments. The integration of AWS S3 and parallel processing further enhances scalability, while robust error handling ensures reliable operation under varying conditions.

4.5.2.4. Other Description of Module 3 Part 2

This section provides a comprehensive discussion of Module 3 to ensure a complete understanding of its design, implementation, and contributions to the automated task assignment system in agile software development. It elaborates on the theoretical foundations and implementation details of the eight online learning models, the rationale for their progression, key technical achievements, and the module's role within the broader project.

Naive Bayes Model:

- **Concept**: A probabilistic classifier based on Bayes' theorem, assuming conditional independence of features. For an issue x with features {x1,…,xn} and assignee class c , the model predicts:

$$P(c \mid x) = \frac{P(c)\prod_{i=1}^{n} P(x_i \mid c)}{P(x)}$$

where P(c) is the prior probability of class c, and P(xi|c) is the likelihood of feature xi given c. The model uses Multinomial Naive Bayes with Laplace smoothing (α=0.1) to handle unseen features.

- **Pros**: Simple and computationally efficient; effective for text classification tasks; robust to high-dimensional, sparse data.
- **Cons**: Assumes feature independence, which may not hold for correlated attributes (e.g., summary and description); lacks built-in mechanisms for handling concept drift.
- **Limitation**: Inability to adapt to changing assignment patterns, necessitating exploration of drift-aware models.

Naive Bayes with ADWIN :

- **Concept**: Enhances Naive Bayes with the Adaptive Windowing (ADWIN) algorithm for concept drift detection. ADWIN monitors a sliding window of performance metrics (e.g., accuracy) and detects significant changes using a statistical test:

$$|\mu_{old} - \mu_{new}| > \epsilon, \quad \epsilon = \sqrt{\frac{2}{m}\sigma^2 \ln\left(\frac{2}{\delta}\right)}$$

Where μ_{old}, μ_{new} are the means of old and new windows, $\sigma^2$is the variance, m is the window size, and $\delta = 0.15$ is the confidence parameter. Upon drift detection, the model is fully reset.

- **Pros**: Adapts to concept drift; retains the simplicity and efficiency of Naive Bayes.
- **Cons**: Full model reset discards all learned knowledge, leading to temporary performance degradation after drift detection.
- **Limitation**: The aggressive reset strategy motivated the development of more nuanced drift-handling approaches.

Hoeffding Adaptive Tree:

- **Concept**: A decision tree designed for streaming data, using the Hoeffding bound to determine when to split nodes:

$$\epsilon = \sqrt{\frac{R^2 \ln\left(\frac{1}{\delta}\right)}{2n}}$$

where R is the range of the attribute, n is the number of observations, and δ=0.01 The model includes internal ADWIN instances at each node to detect and adapt to concept drift dynamically.

- **Pros**: Handles drift internally; provides interpretable decision paths; suitable for streaming data.

- **Cons:** Sensitive to noisy data; slower than linear models due to tree construction and maintenance.
- **Limitation:** Limited scalability for very large datasets.

Leveraging Bagging :

- **Concept**: An ensemble of Hoeffding Trees using bagging with Poisson sampling ($\lambda = 1$) to create diverse learners. Predictions are aggregated via majority voting:

$$\hat{c} = \arg\max_{c} \sum_{i=1}^{M} I(h_i(x) = c)$$

Where $h_i(x)$ is the prediction of the i^{th} tree, and M=10 is the number of trees.

- **Pros**: Robust to noise and concept drift; improves accuracy over single trees through ensemble learning.
- **Cons**: Higher computational cost due to multiple trees; less interpretable than a single tree.
- **Limitation**: Uniform weighting of trees limits adaptability to dynamic changes.

Passive-Aggressive:

- **Concept**: A linear classifier that updates weights aggressively when prediction errors occur, using a hinge loss function:

$$\text{Loss} = \max\left(0, 1 - y \cdot (w^T x)\right)$$

where w is the weight vector, is the feature vector, and y is the label.

- **Pros**: Fast and lightweight; effective for sparse, high-dimensional data; quick to adapt to new data.
- **Cons**: Sensitive to outliers; lacks built-in drift detection, requiring external mechanisms.
- **Limitation**: Linear decision boundaries limit expressiveness for complex assignment patterns, prompting exploration of probabilistic models.

Softmax Regression:

- **Concept**: A multi-class linear classifier that computes probabilities using the softmax function:

$$P(c \mid x) = \frac{e^{w_c^T x}}{\sum_k e^{w_k^T x}}$$

where w_c is the weight vector for class c. The model is optimized using Adam (η=0.01).

- **Pros**: Provides probabilistic outputs for ranking developers; converges quickly; computationally efficient.
- **Cons**: Limited by linear decision boundaries; requires external drift detection.
- **Limitation**: Inability to capture non-linear patterns in complex datasets, leading to ensemble-based approaches

Enhanced Model:

- **Concept**: An AdaBoost ensemble of 10 Naive Bayes classifiers, boosting weak learners by assigning higher weights to misclassified instances. The final prediction is:

$$\hat{c} = \arg\max_{c} \sum_{i=1}^{M} \alpha_i I(h_i(x) = c)$$

Where α_i is the weight of the i^{th} classifier. The model incorporates:

  - **Recency Heuristic**: Assigns flat weights (1.0 for the last 100 issues, 0.0 otherwise) to prioritize recent assignees.
  - **Smart Drift Reset**: Resets the worst-performing 50% of base models upon drift detection.

- **Pros**: Combines multiple learners for improved accuracy; adapts to recency and drift; robust to noise.
- **Cons**: Fixed reset ratio (50%) is suboptimal; flat recency weights lack granularity for nuanced adaptation.
- **Limitation**: Rigid drift handling and simplistic recency weighting prompted the development of a more advanced model.

Super Enhanced Model:

- **Concept**: An advanced AdaBoost ensemble with three key enhancements:
  - **Probabilistic Drift Reset**: Resets base models with probability $1 - \text{accuracy}_i$, preserving high-performing models and improving adaptability.
  - **Decay-Based Recency Heuristic**: Assigns exponentially decaying weights to recent assignees:

$$w_d = \lambda^k, \quad \lambda = 0.95, \quad k = \text{position in recency window}$$

The final score combines content-based and recency-based probabilities:

$$\text{Score}(d) = 0.9 \cdot P(d \mid x) + 0.1 \cdot w_d$$

- **Pros**: Highly adaptive to concept drift and recency; robust performance; optimized for deployment with interactive feedback.
- **Cons**: Increased computational complexity due to ensemble and custom heuristics; requires careful tuning of hyperparameters $(\lambda, \alpha, \delta)$
- **Limitation**: Dependency on high-quality historical data for initial training and tuning.

4.6. Module 4: Sprint Review Summarization

Module 4 is dedicated to automating the summarization of sprint review meeting transcripts in agile software development, addressing the critical need for efficient and accurate documentation of discussions. This module comprises two complementary parts: Part 1 focuses on an extractive summarization approach, utilizing TF-IDF and TextRank methodologies to select and present verbatim utterances that capture critical agile-specific information, such as task progress, blockers, and assignments, while Part 2 employs an abstractive summarization approach, leveraging a fine-tuned Sequence-to-Sequence deep learning model (**BART**) to generate concise, , human-like summaries that rephrase key points from meeting transcripts. Together, these components provide a robust solution for producing speaker-aware, actionable summaries tailored to agile workflows, balancing computational efficiency, domain relevance, and output naturalness to support stakeholders in decision-making and planning.

4.6.1 Extractive Summarization

4.6.1.1. Functional Description

Module 4 is designed to automate the summarization of sprint review meeting transcripts in agile software development, producing concise, speaker-aware summaries that capture critical information such as task progress, blockers, and assignments. This module addresses the need for efficient documentation in agile environments, where sprint reviews generate verbose discussions that must be distilled into actionable insights for stakeholders, such as project managers and developers. Unlike abstractive summarization, which generates paraphrased summaries and is handled by a team member, Module 4 employs an extractive summarization approach, selecting and presenting the most relevant utterances verbatim to preserve the original context and intent.

Module 4 processes a raw meeting transcript and produceses a summary. It supports two extractive summarization methodologies developed as part of this project: **TF-IDF Summarizer** and **TextRank Summarizer**. These methodologies prioritize utterances based on their content relevance, speaker roles, and domain-specific keywords, ensuring that summaries highlight key points such as completed tasks, blockers requiring intervention, and planned actions.

The input to Module 4 is a text transcript of a sprint review meeting, formatted as a sequence of speaker-utterance pairs, typically following the pattern “Speaker: Utterance” (e.g., “Team Lead: Let’s start the sprint review”). The transcript may include:

- **Speaker Information**: Names or roles (e.g., “Team Lead”, “John”).
- **Utterance Content**: Statements about task progress (e.g., “I completed the login page redesign”), blockers (e.g., “I’m blocked on the API integration”), or assignments (e.g., “We’ll assign Mike to help”).
- **Contextual Details**: Numeric data (e.g., “5 story points”) or agile-specific terms (e.g., “velocity”).

The output is a structured summary in two formats:

- **Clean Summary**: A list of selected speaker-utterance pairs, preserving the original wording and order (e.g., “John: I completed the login page redesign, 5 story points”).
- **Debug Summary**: An annotated version including weights or scores for each utterance, useful for analysis and tuning (e.g., “John: I completed the login page redesign [Weight: 0.85]”).

4.6.1.2. Modular Decomposition

Module 4, responsible for automating the summarization of sprint review meeting transcripts in agile software development, is decomposed into four fine-grained components to ensure modularity, maintainability, and scalability. These components form a pipeline that processes raw transcripts into concise, speaker-aware summaries using two extractive summarization methodologies: TF-IDF Summarizer and TextRank Summarizer.

The four components of Module 4 are:

1. **Preprocessing Component**:
    - **Function**: Extracts speaker-utterance pairs from raw meeting transcripts and cleans the text for downstream processing, ensuring compatibility with both TF-IDF and TextRank summarizers.
    - **Implementation**: The MeetingPreprocessor class processes transcripts by splitting them into lines and using regular expressions to identify speaker-utterance pairs (e.g., “Team Lead: Let’s start the sprint review”). The extract_dialogues method handles continuation lines by appending them to the previous speaker’s utterance. The preprocess_utterance method tokenizes utterances, converts them to lowercase, removes punctuation, and filters out stopwords (e.g., “a”, “the”) and short words (< 3 characters). Custom stopwords, such as agile-specific terms (“sprint”, “review”), can be provided to tailor preprocessing to the domain.
    - **Output**: A list of tuples (List[Tuple[str, str]]) containing speaker-utterance pairs and a list of tokenized utterances (List[List[str]]) for feature extraction.
    - **Rationale**: This component ensures that raw, unstructured transcripts are standardized into a format suitable for summarization, handling variations in transcript style (e.g., multi-line utterances) and removing noise to improve feature quality.

2. **TF-IDF Calculation Component**:
    - The TF-IDF Calculator class computes TF-IDF scores for the tokens in each utterance. This process assigns a weight to each word, indicating its importance to an utterance in the context of the entire meeting transcript. These scores provide the core numerical features used by the TF-IDF Summarizer to rank sentences

3. **Sentence Scoring and Selection Component**:
    - **Function**: Scores utterances based on their relevance and selects the most important ones for the summary, implementing distinct strategies for TF-IDF and TextRank summarizers.
    - **Implementation**:
        - **TF-IDF Summarizer**: The Summarizer class's score_sentences method assigns scores to utterances by averaging TF-IDF scores of their tokens. It applies weighting strategies:
            - **Non-Team Leader Boost**: Utterances from non-lead speakers (e.g., excluding "Team Lead", "Manager") receive a 1.3x weight to prioritize developer contributions, as team leader utterances often include non-informative prompts (e.g., "Great", "What did you work on?").
            - **Problem and Action Keywords**: Utterances containing problem keywords (e.g., "blocked", "issue", WordNet synonyms like "obstacle") are boosted by 2.0x, while action keywords (e.g., "completed", "assigned", synonyms like "finished") receive a 1.5x boost. These keywords, defined in summarizer.py, are expanded using WordNet to include synonyms, ensuring comprehensive coverage of agile-relevant terms.
            - **Numeric Data Boost**: Utterances with numbers (e.g., "5 story points") are boosted by 2.0x to capture quantitative updates.
            - **Dynamic Sentence Selection**: To address the challenge of determining summary length (a critical issue in extractive summarization), the method selects all utterances containing problem or action keywords and identifies the minimum score among them $\left((\text{min\_keyword\_score})\right)$ Additional utterances with scores $(\geq \text{min\_keyword\_score})$ are included, ensuring a non-hardcoded, adaptive summary length that balances coverage and conciseness.
        - **TextRank Summarizer**: The TextRank class builds a similarity graph using Jaccard similarity:

$$\text{Similarity}(s_1, s_2) = \frac{|\text{set}(s_1) \cap \text{set}(s_2)|}{|\text{set}(s_1) \cup \text{set}(s_2)| + 10^{-7}}$$

where (s_1, s_2) are tokenized utterances. The run_pagerank method applies the PageRank algorithm with a damping factor $\left((\text{damping} = 0.85)\right)$ to rank utterances, selecting the top (N) based on SummaryConfig.top_n.

- **Output**: For TF-IDF, a list of (index, score) tuples and a set of keyword-containing utterance indices; for TextRank, a list of utterance scores and selected indices.
- **Rationale**: This component encapsulates the core ranking logic, with TF-IDF emphasizing keyword-driven relevance and TextRank capturing semantic coherence, ensuring summaries are both informative and contextually relevant.

4. **Summary Generation Component**:
    - **Function**: Formats the selected utterances into clean and debug summaries, preserving speaker information and original wording for extractive output.
    - **Implementation**:
        - **TF-IDF Summarizer**: The generate_summary method formats selected utterances into two outputs: a clean summary (e.g., “John: I completed the login page redesign”) and a debug summary with weights (e.g., “John: I completed the login page redesign [Weight: 0.85]”). Utterances are capitalized for readability.
        - **TextRank Summarizer**: The summarize method returns a dictionary with a list of speaker-utterance pairs and debug information (scores, selected indices), maintaining original order.
    - **Output**: A dictionary with clean (string of formatted utterances) and debug (string or dictionary with weights/scores) keys.
    - **Rationale**: This component ensures user-friendly, actionable summaries for agile teams, with debug outputs facilitating analysis and tuning. The preservation of original wording addresses the need for accurate documentation in sprint reviews.

4.6.1.3. Design Constraints

The design of Module 2, which automates the summarization of sprint review meeting transcripts in agile software development using extractive methods (TF-IDF and TextRank), is shaped by several critical constraints that influence its architecture, algorithm selection, and implementation. These constraints arise from the need to produce concise, speaker-aware summaries that capture key agile-specific information (e.g., blockers, completed tasks, assignments) in a real-time, resource-constrained environment. They also reflect the inherent challenges of extractive summarization, particularly its limitations in mimicking human-like fluency. Below, each constraint is described, along with its impact on the module’s design and the strategies employed to address it.

1. **Real-Time Processing Requirement**:
    - **Description**: Agile teams require rapid summarization to integrate summaries into sprint review workflows, such as updating Jira issues or planning follow-up actions.

Module 4 must process transcripts and generate summaries within seconds to avoid delaying team activities.

- **Impact**: This constraint necessitates efficient preprocessing and scoring pipelines to handle transcripts in real-time.
- **Solution**: The module employs lightweight extractive algorithms: TF-IDF Summarizer and TextRank Summarizer. The TF-IDF approach uses simple vector-based calculations, while TextRank applies a streamlined PageRank algorithm with a convergence threshold (1e-5) and limited iterations (max_iter = 1000). The preprocessing component optimizes text cleaning with regular expressions and minimal tokenization, reducing latency.

2. **Short and Variable Transcript Lengths**:
   - **Description**: Sprint review meeting transcripts vary significantly in length (e.g., 5–50 utterances) and quality, with some containing incomplete utterances, informal language, or noise (e.g., filler words like “um”). The Apache Jira dataset’s agile context includes diverse meeting styles, complicating summarization.
   - **Impact**: Models must perform effectively with short or noisy transcripts, ruling out methods requiring large, high-quality datasets (e.g., abstractive models). Variable lengths also challenge fixed-length summary designs, necessitating adaptive selection.
   - **Solution**: The preprocessing component robustly handles variable inputs by extracting speaker-utterance pairs with regular expressions and replacing missing or short utterances with empty strings. The TF-IDF Summarizer filters utterances shorter than 3 tokens to exclude noise, while TextRank applies a minimum sentence length (min_sentence_length = 3). For dynamic summary length, the TF-IDF Summarizer selects all utterances with problem or action keywords (e.g., “blocked”, “completed”) and additional utterances with scores ($\geq$ min_keyword_score), ensuring adaptive summaries without hardcoding lengths. This approach, implemented in score_sentences, balances coverage and conciseness across transcript sizes.

3. **Domain-Specific Relevance**:
   - **Description**: Summaries must prioritize agile-specific information, such as blockers (e.g., “I’m blocked on the API integration”), completed tasks (e.g., “I completed the login page redesign”), and assignments (e.g., “We’ll assign Mike”). Non-informative utterances, often from team leaders (e.g., “Great”, “What did you work on?”), should be deprioritized.
   - **Impact**: Generic summarization algorithms may select irrelevant utterances, requiring domain-tailored scoring that emphasizes agile keywords and developer contributions over team leader prompts.
   - **Solution**: The TF-IDF Summarizer implements a sophisticated weighting strategy:

- **Non-Team Leader Boost**: Utterances from non-lead speakers (excluding roles like “Team Lead”, “Manager”) receive a 1.3x weight, as team leader utterances are often procedural or non-informative.
- **Problem and Action Keywords**: Utterances containing problem keywords (e.g., “blocked”, “issue”, WordNet synonyms like “obstacle”) are boosted by 2.0x, and action keywords (e.g., “completed”, “assigned”, synonyms like “finished”) by 1.5x. Keywords are expanded using WordNet to ensure comprehensive coverage, defined in problem_keywords and action_keywords.
- **Numeric Data Boost**: Utterances with numbers (e.g., “5 story points”) receive a 2.0x boost to capture quantitative updates.
  The TextRank Summarizer (textrank.py) indirectly prioritizes domain-relevant utterances through Jaccard similarity, as agile keywords increase overlap between related sentences. Custom stopwords (test.py) like “sprint” and “review” further tailor preprocessing to the domain.

4. **Naturalness of Extractive Output**:
   - **Description**: Extractive summarization selects verbatim utterances from the transcript, resulting in summaries that lack the fluency, coherence, and natural flow of human-generated or abstractive summaries. For example, selected utterances may appear disjointed (e.g., “John: I completed the login page redesign. Sarah: I’m blocked on the API integration”), failing to mimic the narrative style of human summaries, which rephrase and synthesize information.
   - **Impact**: This constraint reduces the usability of summaries for stakeholders expecting polished outputs, as extractive summaries may seem mechanical or fragmented. It also limits the module’s ability to abstract or condense information, requiring careful selection to maximize coherence.
   - **Solution**: To mitigate this limitation, the TF-IDF Summarizer prioritizes utterances with problem and action keywords, which are inherently informative, and maintains original utterance order in generate_summary to preserve conversational flow. The TextRank Summarizer selects utterances with high semantic similarity (similarity_threshold = 0.3), improving topical coherence. Both summarizers format outputs with proper capitalization and speaker labels for readability.

5. **Dynamic Summary Length Requirement**:
   - **Description**: Fixed summary lengths (e.g., always 5 sentences) are unsuitable for variable transcript sizes and content importance. Summaries must adaptively select utterances to balance completeness and brevity, a critical challenge in extractive summarization.
   - **Impact**: Hardcoded lengths may omit critical information in short transcripts or include redundant utterances in long ones, reducing summary quality. This constraint requires a dynamic selection mechanism that prioritizes key content.

- **Solution**: The TF-IDF Summarizer implements an adaptive selection strategy in score_sentences. It selects all utterances containing problem or action keywords (e.g., “blocked”, “completed”) to ensure critical agile information is included. The minimum score among these utterances $((\text{min\_keyword\_score}))$ is used as a threshold to include additional high-scoring utterances, ensuring a non-hardcoded length that reflects content importance. The TextRank Summarizer uses as a default top_n $= \max(3, \text{len(dialogues)}//2)$ providing a flexible baseline that adapts to transcript length. This dynamic approach ensures summaries are neither too sparse nor overly verbose.

These constraints collectively shaped Module 2’s design, guiding the selection of lightweight, domain-tailored extractive algorithms, efficient preprocessing, and adaptive output strategies. By addressing these challenges, the module delivers relevant, usable summaries for agile sprint reviews, though the naturalness limitation highlights the complementary role of the abstractive summarization module. The focus on problem/action keywords and dynamic length ensures alignment with agile needs, while efficient implementation supports practical deployment.

4.6.1.4. Other Description of Module 4

This section provides a comprehensive discussion of Module 4 to ensure a complete understanding of its design, implementation, and contributions to the automated summarization of sprint review meeting transcripts in agile software development. It elaborates on the theoretical foundations and implementation details of the two extractive summarization methodologies—TF-IDF Summarizer and TextRank Summarizer—developed as part of this project. The discussion highlights the rationale for their design, key technical achievements, limitations of extractive summarization (particularly its lack of human-like fluency), and the module’s role within the broader project.

Methodology Descriptions and Theoretical Foundations

Module 4 implements two extractive summarization methodologies—TF-IDF Summarizer and TextRank Summarizer—to generate concise, speaker-aware summaries of sprint review meeting transcripts. Each methodology is designed to address the challenges of capturing agile-specific information (e.g., blockers, completed tasks, assignments) while balancing computational efficiency and relevance. Below, each methodology is described, including its theoretical basis, implementation details, advantages, disadvantages, and limitations, with mathematical formulations where applicable.

1. **TF-IDF Summarizer** :

- **Concept:** The TF-IDF Summarizer ranks utterances based on Term Frequency-Inverse Document Frequency (TF-IDF)." $\text{TF-IDF}(w, d) = \text{TF}(w, d) \cdot \log\left(\frac{N+1}{\text{DF}(w)+1}\right) + 1$

- **Pros**: Computationally efficient; prioritizes domain-specific content; adaptable to variable transcript lengths; robust to noisy or short inputs.
- **Cons**: Relies on keyword presence, potentially missing contextually important utterances without explicit keywords; linear scoring may oversimplify complex semantics.
- **Limitation**: Produces verbatim extracts that lack the fluency and coherence of human summaries, resulting in a mechanical output (e.g., “John: I completed the login page redesign. Sarah: I’m blocked on the API integration”), which may feel disjointed to stakeholders.

2. **TextRank Summarizer**:
   - **Concept**: TextRank is a graph-based algorithm inspired by PageRank, ranking utterances based on their semantic similarity to others in the transcript. It constructs a similarity graph using Jaccard similarity:

$$\text{Similarity}(s_1, s_2) = \frac{|\text{set}(s_1) \cap \text{set}(s_2)|}{|\text{set}(s_1) \cup \text{set}(s_2)| + 10^{-7}}$$

where (s_1, s_2) are tokenized utterances. PageRank computes utterance importance via power iteration:

$$\text{PR}(v_i) = \frac{1-d}{N} + d \sum_{v_j \in \text{In}(v_i)} \frac{\text{PR}(v_j)}{\text{Out}(v_j)}$$

where (d = 0.85) is the damping factor, (N) is the number of utterances, and $\left(\text{Out}(v_j)\right)$is the sum of outgoing edge weights.

- **Pros**: Captures semantic coherence through graph-based ranking; less dependent on explicit keywords; robust to short transcripts.
- **Cons**: Higher computational cost than TF-IDF due to matrix operations; sensitive to similarity threshold tuning; lacks explicit domain weighting.
- **Limitation**: Like TF-IDF, it produces verbatim extracts, leading to less natural outputs compared to human or abstractive summaries. The reliance on similarity may select redundant utterances if not carefully tuned.

4.6.2 Abstractive Summarizer

The Abstractive Summarizer module is designed to automate the creation of meeting summaries. It leverages a state-of-the-art deep learning model to read through lengthy meeting transcripts and generate a concise, human-like summary that captures the essential points and decisions. This module represents the "Abstractive" path outlined in the system architecture, focusing on generating new text rather than simply extracting existing sentences.

4.6.2.1. Functional Description

The functional purpose of this module is to distill a long-form meeting script into a short, coherent summary.

- Input: The module receives a meeting transcript as a raw text input.
- Process: It utilizes a fine-tuned Sequence-to-Sequence (Seq2Seq) deep learning model to process the input text. The model analyzes the semantic meaning of the entire conversation and generates a new summary in its own words, capturing the key information.
- Output: The module outputs a concise text summary that can be automatically sent to a manager or stakeholders, saving them the time of reading the full transcript.

4.6.2.2. Modular Decomposition

The design of the Abstractive Summarizer is decomposed into two primary, sequential sub-modules that represent the end-to-end creation and application of a custom summarization engine.

1. Sub-module: Custom LSTM-Based Seq2Seq Model (Initial Approach)

- This module represents our initial research and development effort to build an abstractive summarizer from the ground up.
- **Process:** We designed and implemented a Sequence-to-Sequence (Seq2Seq) architecture using multiple layers of Bidirectional LSTMs for the encoder and an LSTM-based decoder with a custom attention mechanism. The model was trained on a large dataset of articles and their headlines to learn the summarization task.
- **Outcome & Limitations:** This custom model served as a critical proof-of-concept. However, it struggled to produce consistently fluent or coherent summaries, especially with the long and complex nature of conversational text. This foundational work highlighted the significant challenges in training such models from scratch and validated our strategic decision to pivot to a more powerful, pre-trained architecture.

2. Sub-module: Fine-Tuned BART Model (Final Approach)

This module is the operational and final component of our abstractive summarizer, leveraging a state-of-the-art transformer model to achieve high-quality results.

- **Process:** The workflow begins by loading the pre-trained **facebook/bart-large-xsum** model, a powerful Bidirectional and Auto-Regressive Transformer (BART). This model was then fine-tuned on the **samsum** dataset, which contains thousands of dialogues and their corresponding summaries, making it an expert in conversational text. The process is managed by the Hugging Face Trainer API and evaluated using the ROUGE metric to ensure quality.
- **Output:** The final artifact of this module is a new set of model weights, saved as a custom, specialized summarization engine. When a new meeting script is provided, this model uses a beam search algorithm to produce a high-quality, fluent, and factually accurate abstractive summary.

4.6.2.3. Design Constraints

The design and operation of this module are subject to several important constraints:

- **Hardware Requirement**: The fine-tuning process for the BART model is computationally expensive and requires a high-performance GPU. This is reflected in the use of memory-optimization techniques and FP16 (16-bit floating-point) precision during training.
- **Model and Dataset Dependency**: The module's success is dependent on the availability of large, open-source pre-trained models (like BART) and high-quality, domain-relevant datasets (like samsum).
- **Input Length Limitation**: Transformer-based models have a maximum token limit for their input. Extremely long meeting transcripts may exceed this limit and would need to be processed in chunks or pre-summarized, which could potentially lead to loss of context across the entire meeting. The max_length parameter in the generation function is set to manage this constraint.

4.6.2.4. Other Description

Evaluation Metric: ROUGE Score
Unlike classification tasks that can be measured with simple accuracy, evaluating the quality of a generated summary requires a more sophisticated metric. For this module, the **ROUGE (Recall-Oriented Understudy for Gisting Evaluation)** score was used.
ROUGE works by comparing the model-generated summary against a human-written "reference" summary. It measures the overlap of n-grams (i.e., sequences of one, two, or more words) between the two texts. A high degree of overlap indicates that the model has successfully captured the key points of the reference summary. This metric was implemented as a compute_metrics function to automatically evaluate the model's performance during the fine-tuning process, ensuring the best version of the model was saved.

Chapter 5: System Testing and Verification

The Aigile project comprises four distinct modules, each automating a unique aspect of the Scrum framework, with completely independent testing processes to ensure their specific functionalities are validated. Module 1 focuses on user story and acceptance criteria generation, tested using surveys from 30 Agile practitioners. Module 2 automates story point estimation, evaluated through machine learning models and the TAWOS dataset. Module 3 handles task assignment, tested with the Apache Jira Issue Tracking Dataset [21] and a custom Jira instance. Module 4 performs extractive and abstractive summarization of sprint review transcripts, validated using the samsum dataset and qualitative assessments. This chapter details the testing setup, strategy, and schedule for each module in separate sections (5.1–5.4), followed by integration testing (5.5) and comparative results (5.6) to validate the system's overall performance, ensuring reliability for Scrum automation.

5.1. Testing Setup

Each module's testing setup is tailored to its specific functionality, reflecting the diverse requirements of the Aigile project. Module 1 leverages cloud-based infrastructure and the Groq API, Module 2 utilizes Kaggle's computational resources, Module 3 employs a local MongoDB database and GPU-accelerated pipelines, and Module 4 uses Python-based frameworks with Hugging Face libraries. These distinct setups ensure that each module's performance is rigorously validated under conditions mimicking real-world agile environments.

5.1.1 Module 1 Testing Setup

The testing setup for Module 1, which automates user story and acceptance criteria generation, was designed to validate its functionality, reliability, and alignment with Scrum's INVEST criteria (Independent, Negotiable, Valuable, Estimable, Small, Testable). The module leverages the Llama-3.3-70B-Versatile model, accessed via the Groq API, and integrates with Jira as a plugin built using React and Atlassian's Forge UI. The backend, powered by a Python-based Flask server, processes raw requirements and generates structured outputs.

Test Environment:

- **Platform**: PythonAnywhere was used as the host for the backend server, a cloud-based Python development and hosting platform (https://mou3.pythonanywhere.com). PythonAnywhere's capabilities include:
    - Support for Python 3.9, enabling seamless execution of the Flask server.
    - Always-on tasks and scheduled jobs, ensuring continuous availability for testing concurrent API calls.

    - Built-in console and file storage, facilitating script execution and log analysis for debugging API rate limits or malformed LLM outputs.
- **Software**: Python 3.10.5 for the backend, Forge for the frontend. Forge is Atlassian's new development platform for building Jira apps.
- **API Access**: A paid Groq API plan ensured stable access to Llama-3.3-70B-Versatile, mitigating rate limit constraints.

- **Test Data**: Requirements for three projects were sourced from internet searches:
    - **Mental Health Care System**: An easy-to-use, secure website for mental health workers in mid-Scotland, requiring clinicians to edit patient records (including unique patient ID, personal details, risk assessment, diagnoses, treatments, medications, consultation history, and referrals), search records by name or national health identifier, and update records in real-time or later with flagging for incomplete updates.
    - **Simple Offline Requirements Management Application**: A standalone application for offline use, displaying documents in a table with columns for ID (unique identifier), Description (section, title, full text, attachments), Discussion, Links, and user-defined custom attributes.
    - **Central Trading System (CTS)**: A system enforcing Multi-Factor Authentication (MFA) with a second verification method, encrypting data over the network (TLS 1.3), protected by a Web Application Firewall (WAF), to protect user accounts from unauthorized access and theft.

These requirements were used to generate user stories and priorities via the Llama-3.3-70B-Versatile model. A sample of generated user stories was further processed to produce acceptance criteria, ensuring comprehensive coverage of project needs.

Testing Objectives:

- Verify that Module 1 generates user stories in the “As a [role], I want to [action], in order to [benefit]” format and acceptance criteria covering essential details, technical requirements, and corner cases.
- Ensure seamless integration with Jira, including input validation, output formatting, and issue creation.
- Assess the quality, clarity, and priority accuracy of generated artifacts through human feedback (surveys from 30 Agile practitioners).

5.1.2 Module 2 Testing Setup

The testing setup for the Story Point Estimation module was designed to simulate real-world agile project scenarios while leveraging the computational resources available for efficient and reproducible evaluation. The setup encompasses the hardware, software, dataset, and experimental configurations used to assess the module’s performance across its four sub-

modules: Preprocessing, Initial Experimentation (Part 1), Project-Specific Estimation (Part 2), and Cross-Project Generalization (Final Approach).

Hardware:

- No dedicated hardware was used for this project. All experiments were executed on Kaggle's cloud-based infrastructure, leveraging its computational resources to handle model training and evaluation efficiently.

Software:

- **Programming Language**: Python 3.8, chosen for its extensive machine learning libraries.
- **Kaggle Environment Specifications**:
  - **CPU**: Kaggle provides a quad-core CPU (typically Intel Xeon or equivalent, ~2.0–3.0 GHz), sufficient for lightweight models like SVM, SVR, PassiveAggressiveClassifier, and SGDRegressor.
  - **GPU**: Kaggle offers two NVIDIA Tesla T4 GPUs (each with 16 GB memory) for accelerated training, used for computationally intensive tasks, such as training the DEEP SE model with LSTM and attention mechanisms.
  - **Memory**: Up to 16 GB of RAM, with temporary disk storage (~20–50 GB) allocated for the TAWOS dataset and model outputs.
- **Libraries**:
  - **scikit-learn**: For SVM, SVR, PassiveAggressiveClassifier, SGDRegressor, and TF-IDF vectorization.
  - **PyTorch**: For Enhanced and Standard MLP models.
  - **TensorFlow**: For DEEP SE model implementation.
  - **FastText**: For generating subword-based embeddings.
  - **SentenceTransformers**: For SBERT embeddings (all-MiniLM-L6-v2 and all-mpnet-base-v2).
  - **NumPy and Pandas**: For data preprocessing and analysis.
- **Environment**: Experiments were run in a Jupyter Notebook environment.

Dataset:

- The TAWOS dataset (from the SOLAR-group/TAWOS repository, https://github.com/SOLAR-group/TAWOS) was used, comprising user stories from 40 open-source agile web-hosted projects. Each user story includes a title, description, and metadata (e.g., Type, Priority, Status), with story points assigned in Fibonacci values (1, 2, 3, 5, 8).

Experimental Configurations:

- **Data Splits**: For Part 1 and Part 2, data was split into 80% training and 20% testing (random splits) or 60% training, 20% validation, and 20% testing (project-specific splits). For the Final Approach, one project (e.g., TIMOB) was used as the test project, with the remaining 39 projects for initial training.
- **Cross-Validation**: Stratified GroupKFold was used to prevent data leakage across projects, ensuring all user stories from a single project were in either training or validation sets.
- **Hyperparameter Tuning**: RandomizedSearchCV was applied to optimize model parameters (e.g., C, gamma for SVM/SVR, learning rate, dropout for MLPs).
- **Batch Processing**: For the Final Approach, test project data was processed in batches, simulating 1-2 week sprints, with incremental model updates after each batch.

5.1.3 Module 3 Testing Setup

The testing setup for Module 3, which automates task assignment in agile projects, was designed to validate its ability to accurately predict developer assignments, adapt to concept drift, and integrate seamlessly with Jira workflows. The setup leverages the Apache Jira Issue Tracking Dataset and a custom Jira instance, processed through a pipeline of data extraction, preprocessing, feature engineering, and deep learning models, optimized for GPU acceleration. Testing was conducted on a standard laptop to reflect typical agile team environments, ensuring practical applicability.

5.1.3.1 Module 3 Part 1 Testing Setup

The foundation of this project's testing and verification is the publicly available **Apache Jira Issue Tracking Dataset**, which was sourced from Zenodo (DOI: 10.5281/zenodo.5665896). This choice ensures that our results can be contextualized within the broader research community.

- **Data Source and Ingestion**: The testing process began with the raw data from the aforementioned dataset, consisting of .bson files (issues, users, projects). These files were loaded into a local MongoDB database named apache_jira_data using the mongorestore command, which was identified as the correct tool for BSON collection dumps.

- **Data Processing and Curation**: A scripted pipeline was used to prepare the data for modeling. An initial script extracted data from MongoDB and created a numerical assignee_id for each unique developer, saving a name-to-ID mapping file for reference. A subsequent script filtered for active projects and applied an

undersampling technique, capping the number of issues for any single developer at 80 to create a more balanced dataset.

- **Feature Engineering**: Two distinct paths for feature creation were implemented:
    - **Topic Modeling Path**: Textual data from issues underwent advanced cleaning using the spaCy library, including removing HTML tags, tokenizing, removing stopwords, and applying lemmatization. Latent Dirichlet Allocation (LDA) models were then trained on this cleaned text with various numbers of topics to generate features like dominant topic ID and topic probability distributions.
    - **Deep Learning Path**: A separate script prepared data for the neural network by tokenizing text into integer sequences and padding all sequences to a uniform length of 250.
- **Technical Environment**: To overcome initial performance bottlenecks identified with CPU-bound processing, the environment was optimized for GPU acceleration. The Topic Modeling pipeline was re-engineered to use the **NVIDIA RAPIDS** suite (cudf, cuml). The deep learning pipeline was implemented using **Keras/TensorFlow**.

5.1.3.2 Module 3 Part 2 Testing Setup

Module 3 automates task assignment by processing Jira issues through a pipeline of data extraction, preprocessing, feature engineering, online learning, and user interaction, integrated with Jira and AWS S3. Testing ensured that the system accurately predicts developer assignments, adapts to concept drift, and provides a user-friendly interface for non-technical users (e.g., project managers). The setup validated compliance with agile requirements, such as real-time processing, scalability, and seamless integration with Jira workflows.

The testing environment was configured on a standard laptop (16GB RAM, Intel i7 CPU, Windows 10) to reflect typical deployment conditions for agile teams. Datasets included:

- **Apache Jira Dataset**: Issues from multiple Apache projects (e.g., Hadoop, Spark), providing diverse textual and categorical attributes (e.g., summary, description, priority) for training and evaluating models across projects.
- **Custom Jira Project**: A controlled Jira instance with 50 synthetic issues assigned to four developers, used to test the system's plugin and real-world applicability.

Testing objectives aligned with agile principles, ensuring models met performance thresholds, the GUI was intuitive, and the Jira plugin delivered reliable predictions.

5.1.4 Module 4 Testing Setup

The testing setup for Module 4, which automates summarization of sprint review transcripts, was designed to validate both extractive and abstractive summarization approaches. The setup ensures that the module produces accurate, speaker-aware summaries that meet real-time constraints and support agile workflows. Testing utilized the samsum dataset and Python-based frameworks, with a focus on evaluating functionality, performance, and usability for non-technical users such as Scrum Masters.

5.1.4.1. Module 4 Part 1 Testing Setup

Module 4 automates extractive summarization by processing sprint review transcripts through a pipeline of Preprocessing, TF-IDF Calculation, Sentence Scoring and Selection, and Summary Generation. Testing ensured that the system produces accurate, speaker-aware summaries, adheres to real-time constraints, and supports agile workflows for non-technical users (e.g., Scrum masters). The setup validated compliance with design requirements

5.1.4.2. Module 4 Part 2 Testing Setup

This section describes the environment and components used to test and validate the Abstractive Summarizer module.

- **Core Technology**: The system under test is a Sequence-to-Sequence (Seq2Seq) model based on the **BART** architecture. The testing process involves fine-tuning this model and evaluating its performance.
- **Dataset for Testing and Training**: The entire process utilizes the **samsum dataset**. This dataset, containing thousands of dialogues and human-written summaries, serves as the ground truth for both training the model and for quantitatively evaluating its output. The dataset was split into training, validation, and test sets.
- **Software and Libraries**: The testing framework was built in Python using the Hugging Face transformers library to manage the model and training process, datasets to load the samsum data, and evaluate to compute the ROUGE score. The accelerate and bitsandbytes libraries were used for optimizing the training on the available hardware.
- **Evaluation Criteria**: The primary success criterion is the quality of the generated summary. The testing plan included both quantitative evaluation using the **ROUGE score** against the reference summaries in the samsum test set and qualitative evaluation by visually inspecting the coherence and factual accuracy of summaries generated for sample meeting scripts.

5.2. Testing Plan and Strategy

The testing plan and strategy for the Aigile project are designed to ensure that each module meets its functional, performance, and usability requirements within the Scrum framework. The strategy combines automated testing (e.g., unit, functional, and integration tests) with

manual evaluations (e.g., surveys, qualitative assessments) to achieve comprehensive validation. Each module's testing plan is tailored to its specific objectives, with Module 1 focusing on user story quality, Module 2 on story point accuracy, Module 3 on task assignment precision, and Module 4 on summary coherence. The approach emphasizes iterative testing, real-world applicability, and alignment with agile principles.

For module 1,

Our strategy involved targeted outreach: we shared posts on relevant Scrum subreddits on Reddit and also connected with Scrum Masters and developers through LinkedIn and Facebook.

For module 2,

The testing plan for the Story Point Estimation module was designed to rigorously evaluate its performance across diverse scenarios, ensuring robustness, generalization, and practicality for agile teams. The strategy focused on three objectives: (1) validating model accuracy and error metrics, (2) assessing generalization across projects, and (3) ensuring computational efficiency for real-time agile use. The testing process was iterative, evolving from random data splits to separate project testing, subtask decomposition, and incremental learning to address real-world challenges like concept drift and "cold start" scenarios.

Methodology:

1. **Initial Experimentation (Part 1)**: Evaluated models using random 80%/20% splits and separate project testing to assess generalization. Tested multiple embeddings (TF-IDF, GloVe, FastText, BERT, SBERT) and models (SVM, SVR, XGB Regressor, Ordinal Regression, LSTM), with subtask decomposition to improve explainability. Used stratified cross-validation to address model selection bias.
2. **Project-Specific Testing (Part 2)**: Tested context-aware models (Enhanced MLP, Standard MLP, DEEP SE, FastText + SVM), using 60%/20%/20% splits. Evaluated the impact of context-aware features (e.g., weighted context vectors, statistical features) on prediction accuracy.
3. **Cross-Project Generalization (Final Approach)**: Simulated a "cold start" scenario by testing on a new project (e.g., TIMOB) with initial training on 39 other projects. Processed data in batches to mimic sprints, updating incremental models (PassiveAggressiveClassifier, SGDRegressor) and comparing against static and full retrain baselines.
4. **Metrics and Analysis**: Used accuracy (percentage of correct predictions) and Mean Absolute Error (MAE) as primary metrics, with Median Absolute Error (MdAE) for robustness. Confusion matrices were analyzed to understand prediction distributions across story points.

Testing Phases:

- **Phase 1 (Part 1)**: Compared model performance across random and separate project splits, focusing on generalization challenges and subtask decomposition benefits.
- **Phase 2 (Part 2)**: Evaluated context-aware models on project-specific data, assessing the impact of historical context on MAE and accuracy.
- **Phase 3 (Final Approach)**: Tested incremental learning for new projects, measuring performance (accuracy, MAE) and training time against baselines to validate adaptability and efficiency.

For module 3 part 1,

The testing methodology was a two-phase, comparative analysis designed to rigorously evaluate and validate the Task Assignment module.

- **Initial Baseline Strategy**: The testing began by replicating the methodology from the reference research paper, using Latent Dirichlet Allocation (LDA) combined with classical machine learning classifiers (SVM and Naïve Bayes). This approach served to establish a performance baseline and validate our data processing pipeline against existing work.
- **Strategic Pivot to Deep Learning**: After evaluating the baseline model, a strategic decision was made to implement a more advanced deep learning architecture using a Long Short-Term Memory (LSTM) network. This pivot was driven by several key objectives:
    - To achieve **Enhanced Semantic Understanding** by leveraging pre-trained GloVe word embeddings.
    - To utilize **Automated Feature Engineering**, allowing the neural network to learn complex feature interactions directly from the data.
    - To explore a more powerful class of models with a **Higher Performance Ceiling**.
- **Validation Method**: Both approaches were rigorously evaluated by splitting the data into training and test sets and measuring performance on the unseen data to provide an unbiased assessment of the models' capabilities.

For module 3 part 2,

The testing plan for followed a systematic, multi-phase approach to validate functionality, performance, and usability across all components: Data Extraction, Preprocessing, Feature Engineering, Online Learning, and User Interface. The strategy combined automated and manual testing to achieve comprehensive coverage, addressing both technical correctness and practical applicability in agile environments. The methodology ensured robustness under diverse conditions, including noisy data, concept drift, and real-time constraints.

Testing Phases

1. **Unit Testing**: Verified individual functions within each component (e.g., TFIDF_Calc.transform_one), focusing on correctness and edge cases.
2. **Functional Testing**: Validated component-level behavior (e.g., feature vector generation, model predictions) against expected outputs.
3. **Integration Testing**: Ensured seamless interaction between components (e.g., data flow from Jira API to GUI plugin), testing end-to-end workflows.
4. **Usability Testing**: Evaluated the GUI and Jira plugin for non-technical users, assessing intuitiveness, clarity, and feedback mechanisms.
5. **Model Evaluation**: Compared eight online learning models (e.g., Naive Bayes, AdaBoost with 10 Naive Bayes models) across Apache Jira projects, selecting the best-performing model for deployment.

For module 4 part 1,

The testing plan followed a systematic, multi-phase approach to validate functionality, performance, and usability across all components: Preprocessing, TF-IDF Calculation, Sentence Scoring and Selection, and Summary Generation. The strategy combined automated and manual testing to achieve comprehensive coverage, addressing technical correctness and practical applicability in agile sprint review contexts. The methodology ensured robustness under diverse conditions, including noisy transcripts, varying utterance lengths, and real-time constraints.

Testing Phases

1. **Unit Testing**: Verified individual functions within each component (e.g., extract_dialogues in Preprocessing) focusing on correctness and edge cases.
2. **Functional Testing**: Validated component-level behavior (e.g., TF-IDF scoring, summary generation) against expected outputs.
3. **Integration Testing:** Ensured seamless interaction between components (e.g., from Preprocessing to Summary Generation), testing end-to-end summarization workflows.
4. **Usability Testing:** Evaluated the interface for non-technical users, assessing clarity of summaries and ease of interaction.

For module 4 part 2,

The testing strategy followed a standard deep learning validation protocol focused on both quantitative metrics and qualitative assessment.

1. **Fine-Tuning on a Domain-Specific Dataset**: The core strategy was to fine-tune the general-purpose BART model on the samsum dataset. This adapts the model to the specific style and vocabulary of conversational text, which is highly relevant for summarizing meeting transcripts.
2. **Quantitative Evaluation**: The plan included using a held-out test set from the samsum dataset for the final performance evaluation. The ROUGE score was chosen

to provide an objective measure of how closely the model-generated summaries match the human-written reference summaries.

3. **Qualitative Evaluation**: After the fine-tuning process, the model was tested with ad-hoc inputs (sample meeting scripts) to qualitatively assess the fluency, coherence, and relevance of the generated summaries in a real-world scenario. This served as a practical test of the module's utility.

5.2.1 Module Testing

The module testing section details the specific methodologies, test cases, and results for each of the four modules. Testing was conducted systematically to validate functionality, performance, and integration with agile workflows. Each module was evaluated using a combination of quantitative metrics (e.g., accuracy, MAE, ROUGE scores) and qualitative feedback (e.g., practitioner surveys, manual inspections) to ensure robustness and usability.

5.2.1.1 Module 1 Testing

Module 1 of the Aigile project automates the generation of user stories and acceptance criteria using the Llama-3.3-70B-Versatile model, accessed via the Groq API, and integrates these outputs into Jira as a plugin. The testing strategy for Module 1 aimed to validate its functionality, reliability, and alignment with Scrum's INVEST criteria (Independent, Negotiable, Valuable, Estimable, Small, Testable). The testing methodology combined automated quantitative evaluation (cosine similarity) with qualitative feedback from 30 Agile practitioners, ensuring a comprehensive assessment of the module's performance in generating high-quality, Scrum-compliant user stories and acceptance criteria. Below, we detail the testing steps, results, and their implications.

Testing Methodology for Module 1

The testing process for Module 1 was designed to evaluate three key aspects: (1) the quality and clarity of generated user stories, (2) their adherence to the INVEST criteria, and (3) the relevance and completeness of acceptance criteria. The methodology included both automated and human-based evaluation approaches to ensure robust validation:

1. **Automated Evaluation Using Cosine Similarity**:
    - **Objective**: Quantify the semantic similarity between generated user stories and input requirements to assess relevance and accuracy.
    - **Process**: The evaluate.ipynb notebook, utilizing the sentence-transformers library (MiniLM-L6-v2 model), computed cosine similarity scores between the generated user stories and the original project requirements. Multiple LLMs were tested, and Llama-3.3-70B-Versatile was selected for its superior

performance. This metric ensured that generated stories accurately reflected the input requirements.

2. **Survey-Based Evaluation**:
    - **Objective**: Gather qualitative feedback from Agile practitioners to assess user story quality, INVEST adherence, priority accuracy, and acceptance criteria completeness.
    - **Process**: A survey was distributed to 30 Agile practitioners with varying roles (Scrum Masters, Product Owners, Developers, and others) and experience levels (0–12+ years). The survey evaluated user stories for three projects, with respondents rating:
        - Adherence to INVEST criteria (Does Not Meet, Partially Meets, Fully Meets) for each criterion (Independent, Negotiable, Valuable, Estimable, Small, Testable).
        - Overall quality and clarity of user stories (1–5 scale, Poor to Excellent).
        - Accuracy of priority rankings (1–5 scale, Poor to Excellent).
        - Alignment of acceptance criteria with essential details, technical requirements, and corner cases (1–5 scale, Poor to Excellent).
    - **Implementation**: Respondents reviewed sample user stories for each project (e.g., “As a Clinician, I want to update an existing patient record” for the Mental Health Care System). Responses were collected via Google Forms and analyzed to identify trends and areas for improvement.
3. **Functional Testing**:
    - **Objective**: Verify that Module 1 correctly processes inputs, generates outputs, and integrates with Jira.
    - **Process**: Test cases were designed to cover:
        - Input validation (e.g., ensuring requirements with <10 characters are rejected).
        - Generation functionality (e.g., producing structured JSON outputs with user story title, epic, description, and priority).
        - Jira integration (e.g., successful creation of Jira issues via createJiraIssue).
        - Error handling (e.g., handling API rate limits, invalid inputs, or malformed LLM outputs).
    - **Implementation**: Manual tests confirmed UI responsiveness and Jira issue creation.

Results and Discussion

The testing results for Module 1 provide insights into its strengths and areas for improvement, as derived from survey evaluations conducted with 30 respondents.

Survey Results

Respondent Demographics: The 30 respondents included Scrum Masters (20%), Product Owners (13.3%), Developers (40%), and others (26.7%). Their experience levels in Agile development were 0–3 years (23.3%), 4–7 years (33.3%), 8–12 years (23.3%), and 12+ years (20%).

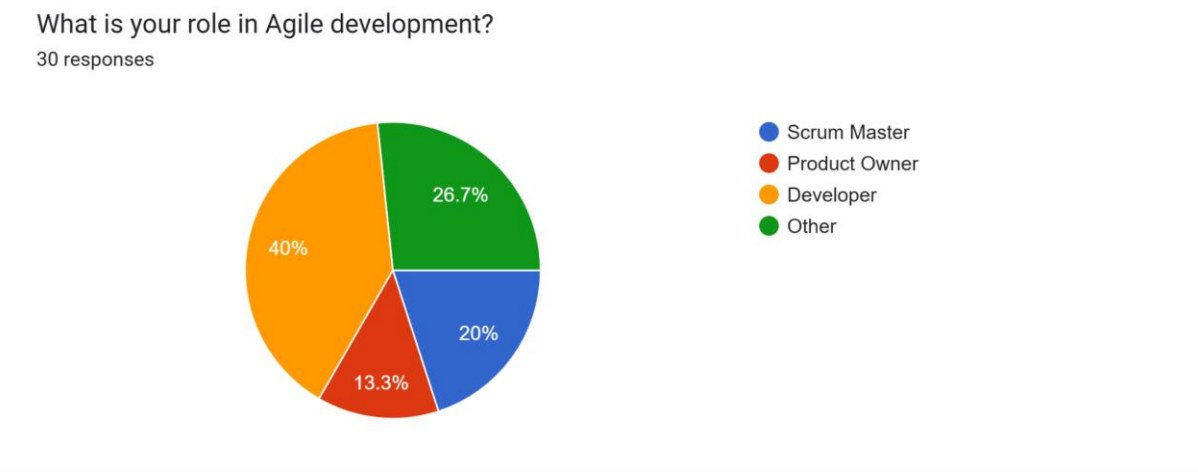


Figure 5.1: *Module 1 Survey Question "What is your role in Agile development?"*

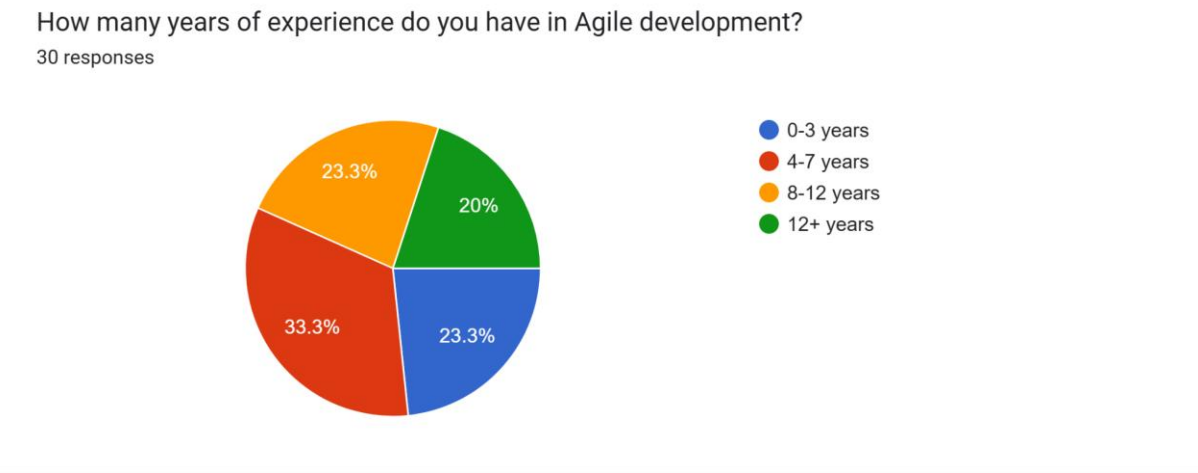


Figure 5.2: *Module 1 Survey Question " How many years of experience do you have in Agile development?"*

INVEST Criteria Adherence:

- **Project 1: Mental Health Care System:** The generated user stories scored very highly on the "Valuable" and "Independent" criteria, with the vast majority of respondents rating them as "Fully Meets". However, the ratings were more mixed for

"Estimable" and "Small," indicating that stories were not always considered well-sized for a single sprint.

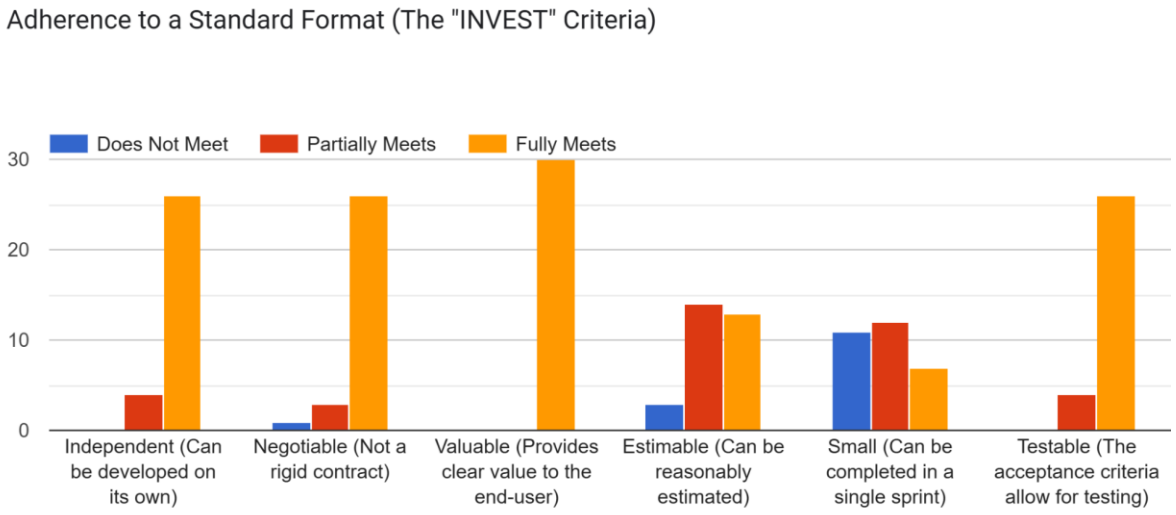


Figure 5.3: *Module 1 Survey Question "Adherence to INVEST Criteria" For Project 1.*

- **Project 2: Simple Offline Requirements Management Application:** While the stories were rated favorably for being "Valuable" and "Negotiable," they performed poorly on the "Small" and "Estimable" criteria. For the "Small" criterion, a significant majority of respondents (approx. 19) rated the user story as "Does Not Meet," suggesting it was too broad.

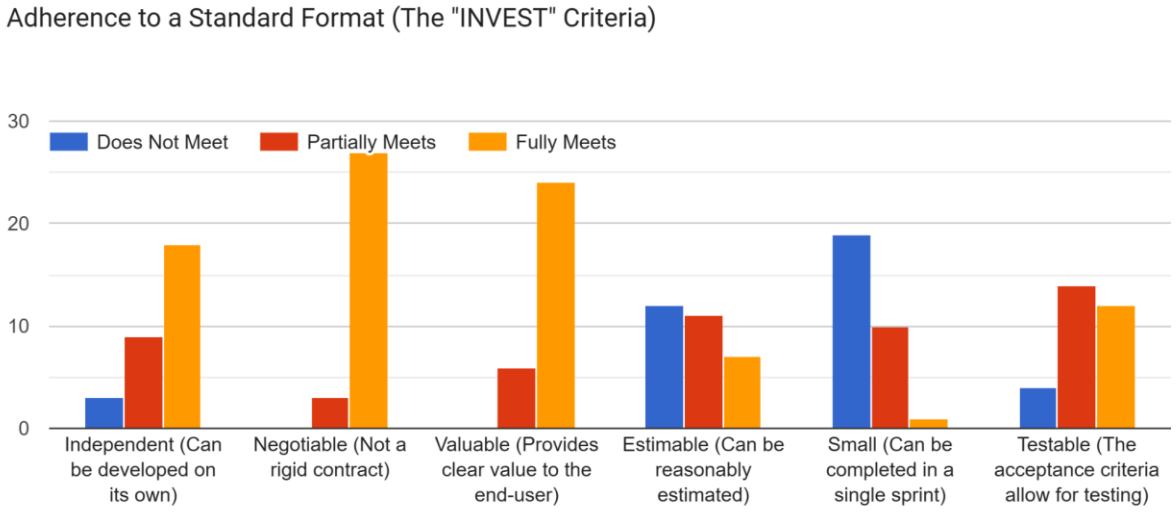


Figure 5.4: *Module 1 Survey Question "Adherence to INVEST Criteria" For Project 2.*

- **Project 3: Central Trading System (CTS):** The user stories for this project were rated highly against the "Valuable," "Independent," and "Estimable" criteria. The "Small" criterion received more mixed results, with a high number of "Fully Meets" ratings (approx. 20).

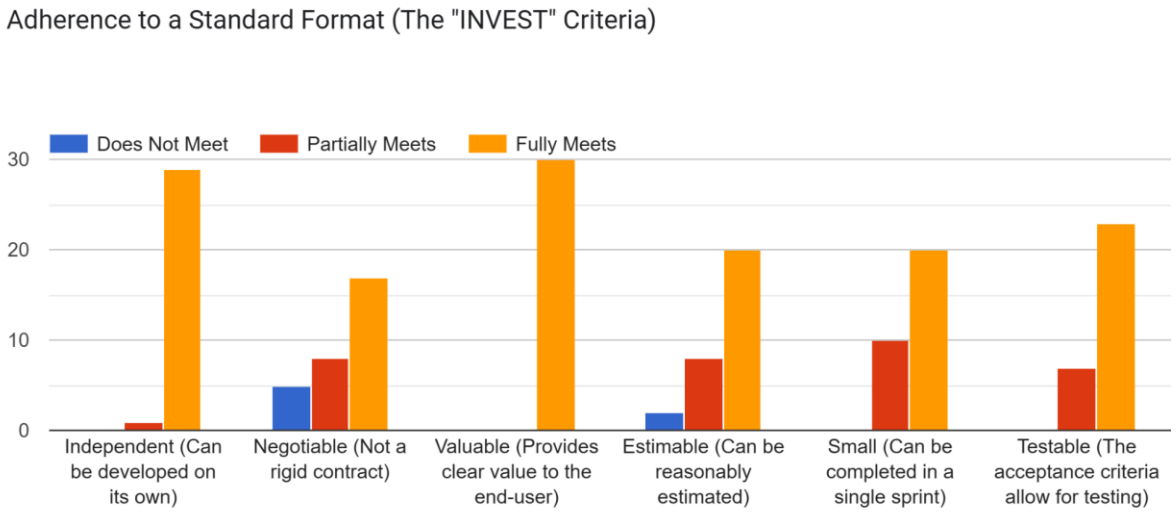


Figure 5.5: *Module 1 Survey Question "Adherence to INVEST Criteria" For Project 3.*

Quality and Clarity:

- **Project 1: Mental Health Care System:** The overall quality and clarity were rated very highly. 17 respondents (56.7%) gave a rating of 4/5, and 9 respondents (30%) gave a rating of 5/5. Only 4 respondents (13.3%) rated it 3/5.

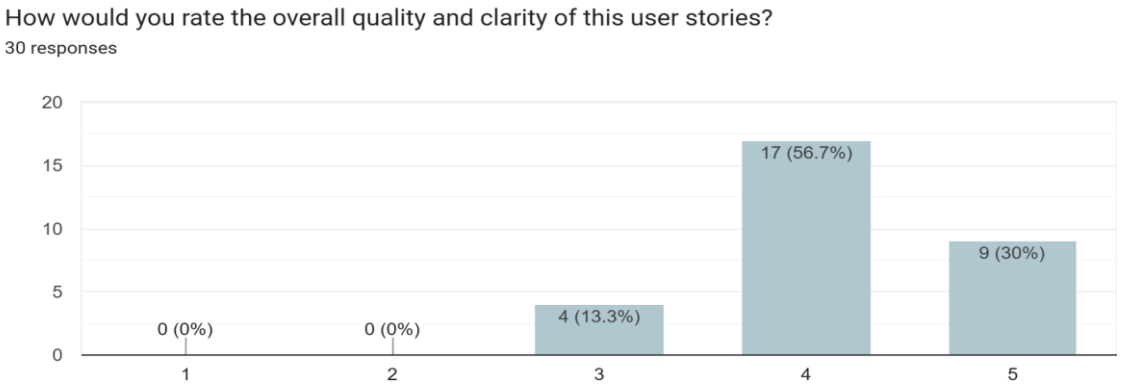


Figure 5.6: *Module 1 Survey Question "Quality and Clarity" For Project 1.*

- **Project 2: Simple Offline Requirements Management Application:** Ratings were more moderate. 13 respondents (43.3%) rated the quality as 3/5, 11 (36.7%) as 4/5, and 5 (16.7%) as 2/5.

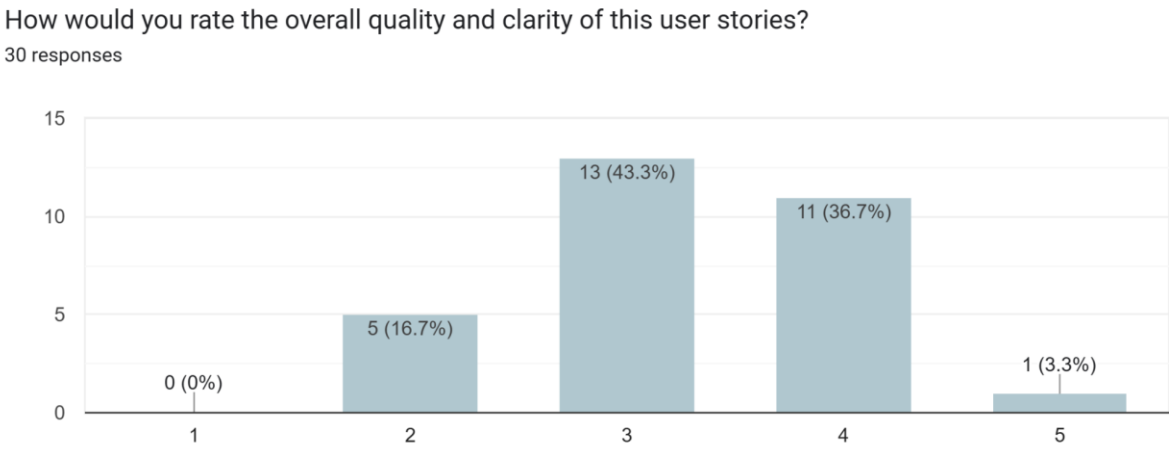


Figure 5.7: *Module 1 Survey Question "Quality and Clarity" For Project 2.*

- **Project 3: Central Trading System (CTS):** Quality and clarity were rated exceptionally high, with 16 respondents (53.3%) giving a 4/5 rating and 13 respondents (43.3%) giving the highest rating of 5/5.

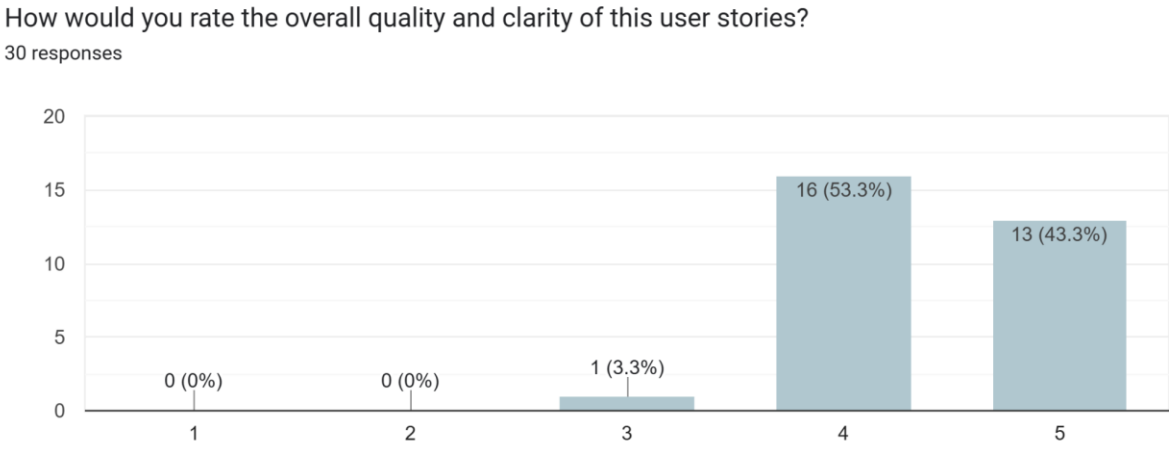


Figure 5.8: *Module 1 Survey Question "Quality and Clarity" For Project 3.*

Priority Accuracy:

- **Project 1: Mental Health Care System:** The priority ranking was considered highly accurate, with 23 respondents (76.7%) giving a 5/5 rating and 7 respondents (23.3%) giving a 4/5 rating.

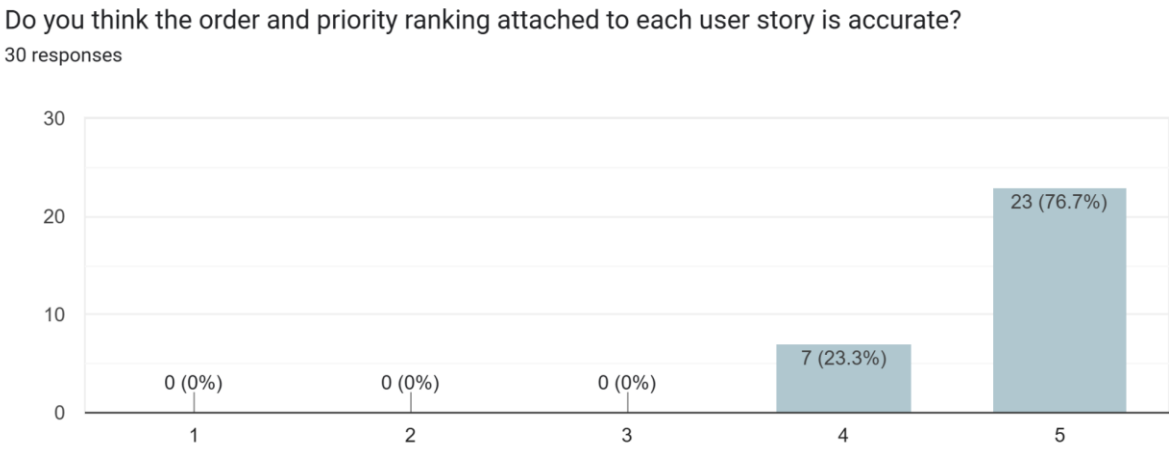


Figure 5.9: *Module 1 Survey Question "Priority" For Project 1.*

- **Project 2: Simple Offline Requirements Management Application:** The priority was also well-regarded, with 17 respondents (56.7%) rating it 4/5 and 6 respondents (20%) rating it 5/5, and 3 respondents (20%) rating it 3/5.

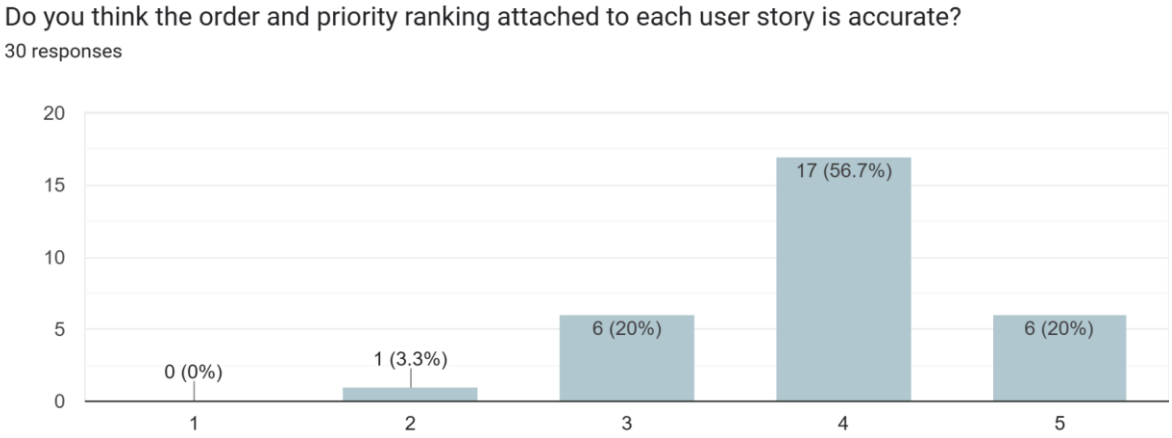


Figure 5.10: *Module 1 Survey Question " Priority " For Project 2.*

- **Project 3: Central Trading System (CTS):** The priority ranking for this project was deemed highly inaccurate. A significant majority of 18 respondents (60%) gave the lowest possible rating of 1/5. This feedback aligns with the user story description, which assigned the "Lowest" priority to a critical Multi-Factor Authentication (MFA) security feature.

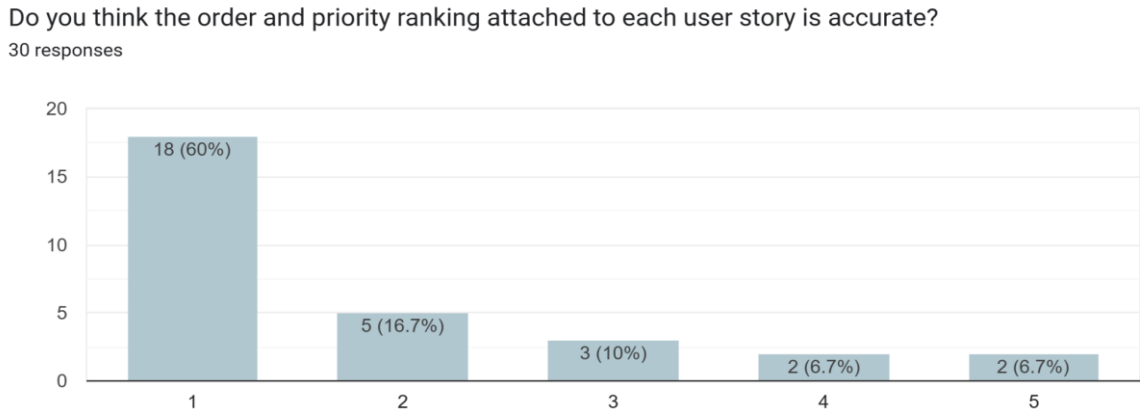


Figure 5.11: *Module 1 Survey Question " Priority " For Project 3.*

Acceptance Criteria Alignment:

- **Project 1: Mental Health Care System:** The acceptance criteria were well-aligned with requirements. 16 respondents (53.3%) rated them 4/5, and 8 respondents (26.7%) gave a 5/5 rating.

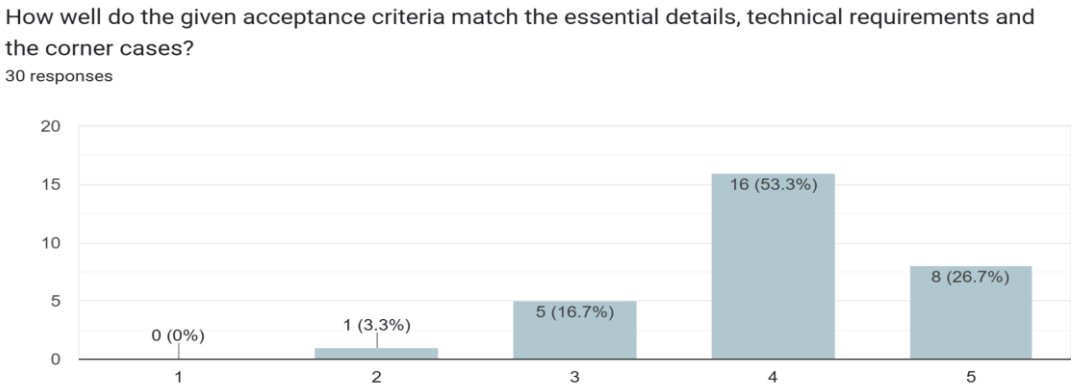


Figure 5.12: *Module 1 Survey Question " Acceptance Criteria " For Project 1.*

- **Project 2: Simple Offline Requirements Management Application:** The alignment was rated as moderate. The most common rating was 3/5 from 13 respondents (43.3%), followed by 4/5 from 9 respondents (30%).

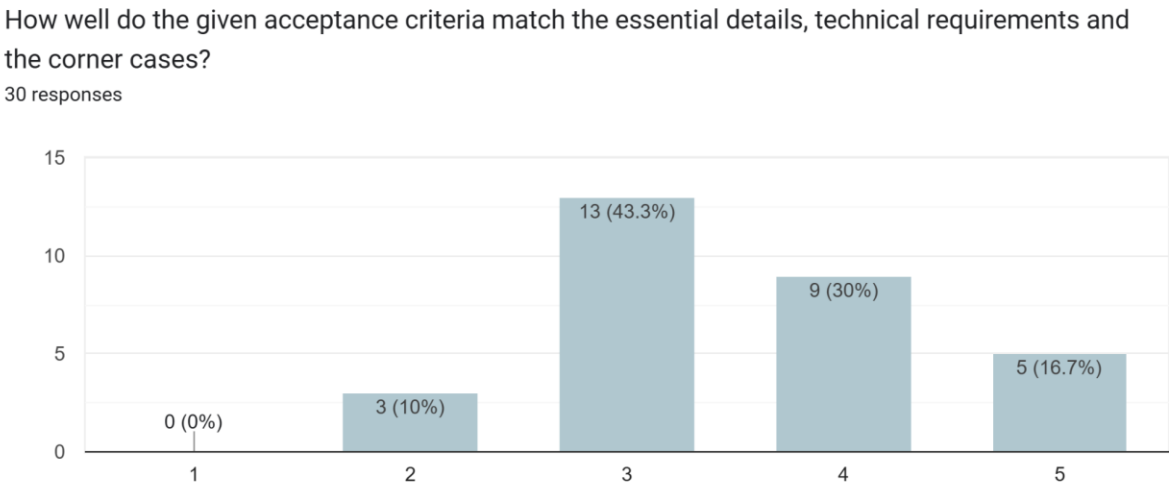


Figure 5.13: *Module 1 Survey Question " Acceptance Criteria " For Project 2.*

- **Project 3: Central Trading System (CTS):** The acceptance criteria were considered an excellent match. 12 respondents (40%) gave a top rating of 5/5, and 11 respondents (36.7%) provided a 4/5 rating.

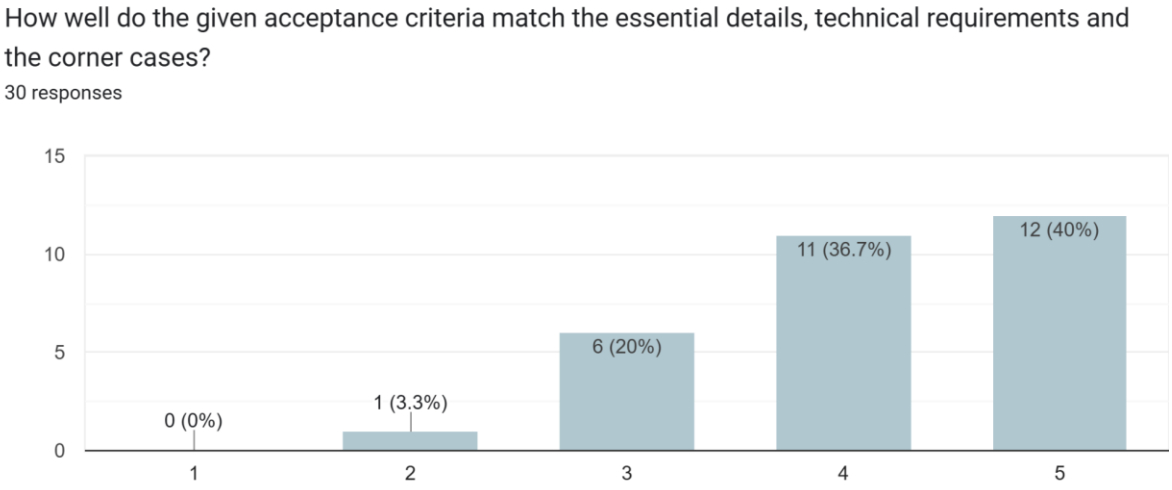


Figure 5.14: *Module 1 Survey Question " Acceptance Criteria " For Project 3.*

Functional Testing Results:

All test cases for input validation, generation, and Jira integration passed. The system rejected invalid inputs (<10 characters) and correctly mapped priorities to Jira labels (Highest to Lowest). Error handling managed API rate limits and malformed LLM outputs (e.g., using user_story_parser to reformat non-compliant stories).

UI tests confirmed responsiveness, with BacklogGenerationView.jsx displaying stories in a grid and index.jsx showing criteria with a progress bar.

Discussion

The evaluation results highlight Module 1's capabilities and specific areas needing refinement.

Strengths: The LLM demonstrated a strong ability to generate high-quality, clear, and relevant user stories with well-aligned acceptance criteria. For two of the three projects (Mental Health Care and CTS), both the quality/clarity and acceptance criteria received overwhelmingly positive scores, with a majority of users rating them 4/5 or 5/5. Functional and performance tests further validated the system's technical integration and scalability.

Challenges: The survey revealed two primary challenges:

1. **INVEST Adherence:** The generated stories for the Offline Requirements Management application struggled to meet the "Small" and "Estimable" criteria, suggesting the model generated stories that were too large or complex for a single sprint.
2. **Priority Accuracy:** A critical failure was observed in the Central Trading System project, where the model assigned the "Lowest" priority to a vital MFA security feature. This was overwhelmingly rejected by respondents, with 60% giving the priority accuracy the lowest possible score of 1/5. This indicates a significant misalignment between the model's prioritization logic and fundamental security principles.

Mitigation: The system already includes fallback parsing (user_story_parser, parse_response) to handle LLM non-adherence. To address the identified challenges, future iterations should focus on improving priority accuracy by refining the 100-dollar allocation method with weighted agent perspectives, particularly for critical functions like security. Furthermore, prompt engineering can be enhanced to guide the model in generating smaller, more self-contained stories that better adhere to the "Small" and "Estimable" INVEST principles.

Table 5.1: *Module 1 Testing Results Summary*

Project	*INVEST Adherence Highlights*	*Quality/Clarity (Avg. Score)*	*Priority Accuracy (Avg. Score)*	*Acceptance Criteria (Avg. Score)*
Mental Health Care System	Strong on "Valuable" & "Independent"; Weaker on "Small" & "Estimable"	4.17/5	4.77/5	4.03/5
Offline Requirements Management	Strong on "Valuable"; Poor on "Small" & "Estimable"	3.27/5	3.93/5	3.53/5
Central Trading System	Strong on "Valuable" & "Independent"; Weaker on "Small"	4.40/5	1.83/5	4.13/5

5.2.1.2 Module 2 Testing

This subsection details the evaluation metrics, and results of the final approach, providing a comprehensive overview of how the Story Point Estimation module was tested.

Evaluation Metrics: Two primary metrics were used to evaluate model performance, reflecting the module's goal of accurate and reliable story point predictions:

1. **Accuracy**:
   a. **Definition**: The percentage of user stories where the predicted story point matches the actual story point.
   b. **Equation**: $Acc = \frac{Number\ of\ Correct\ \text{Predictions}}{Total\ Number\ of\ \text{Predictions}} \cdot 100$
   c. **Purpose**: Measures exact matches, critical for ensuring predictions align with Scrum's Fibonacci scale.
2. **Mean Absolute Error (MAE)**:
   a. **Definition**: The average absolute difference between predicted and actual story points, quantifying prediction error magnitude.
   b. **Equation**:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

Where:
$\hat{y}_i$ = Predicted value for the i^{th} data point
y_i = Actual value for the i^{th} data point
n = number of observations

   c. **Purpose**: Provides a continuous measure of error, suitable for regression tasks and capturing the scale of mispredictions.
3. **Median Absolute Error (MdAE)**:

a. **Definition**: The median of absolute differences between predicted and actual story points, robust to outliers.
b. **Equation**: $MdAE = median\{\Pr e\, dicted_i - Actual_i\}$
c. **Purpose**: Used in some experiments (e.g., Part 1) to assess robustness against skewed predictions.

Confusion matrices were also analyzed to visualize prediction distributions across story points, providing insights into model biases and performance per class.

Final Approach Results: The Final Approach (Cross-Project Generalization with Incremental Learning) was tested on the TIMOB project, using 39 other projects for initial training and processing TIMOB data in batches to simulate sprints. Two incremental models were evaluated: PassiveAggressiveClassifier (classification) and SGDRegressor (regression), compared against static and full retrain baselines.

- **Classification Results (PassiveAggressiveClassifier)**:
  - **Setup**: Trained on TF-IDF vectors (max_features=5000), with story points encoded as categories (1→0, 2→1, 3→2, 5→3, 8→4). Updates took ~0.02-0.03 seconds per batch, compared to ~600-710 seconds for full retraining.
  - **Results**: Refer to **Table 4.25**
- **Regression Results (SGDRegressor)**:
  - **Setup**: Trained on TF-IDF vectors, predicting story points as numbers rounded to Fibonacci values. Updates took ~0.05-0.06 seconds.
  - **Results**: Refer to **Table 4.26**
- **Key Insights**:
  - **Accuracy and MAE**: The incremental models matched or outperformed the full retrain models (e.g., batch 10: classification accuracy 0.3264 vs. 0.3640, regression MAE 1.7832 vs. 1.8858) and consistently outperformed static models, which suffered from concept drift (e.g., static accuracy dropped to 0.1632).
  - **Efficiency**: Incremental updates were >99% faster (0.02-0.06 seconds vs. 600-1067 seconds), making them ideal for agile sprints.
  - **Adaptability**: The incremental models are adapted to project-specific patterns, maintaining stable or improving performance over batches.

5.2.1.3 Module 3 Testing

The testing process for Module 3, which automates task assignment in agile projects, was designed to validate its functionality, performance, and usability across its five components: Data Extraction, Preprocessing, Feature Engineering, Online Learning, and User Interface. The testing methodology combined automated testing (unit, functional, and integration tests) with model evaluation and usability assessments to ensure accurate developer assignments, adaptability to concept drift, and seamless integration with Jira workflows. Testing was

conducted using the Apache Jira Dataset and a custom Jira instance, with results analyzed across multiple projects to assess robustness and real-world applicability.

Part 1: Offline Learning

Approach 1: Topic Modeling (LDA) Results

- **Quantitative Results**: The evaluation of the LDA-based pipeline revealed a wide range of performance across different projects. The model achieved its peak accuracy of **55%** on the **CASSANDRA** project. However, performance varied significantly,A clear trend was observed where model accuracy decreased as the number of assignees increased, highlighting the growing complexity of the classification task.
  Detailed Accuracy Breakdown:
  - **CASSANDRA:** 55.0%
  - **AMBARI:** 48.8%
  - **HDDS:** 45.3%
  - **DATALAB:** 42.5%
  - **IMPALA:** 42.5%
  - **MESOS:** 37.5%
  - **OAK:** 37.5%
  - **CB:** 36.3%
  - **IGNITE:** 35.0%
  - **ARROW:** 32.5%
  - **GEODE:** 30.0%
- **Discussion**: The results indicate that the Topic Modeling approach is promising for projects with fewer assignees, but its performance degrades as the classification task becomes more complex, a finding consistent with existing research.

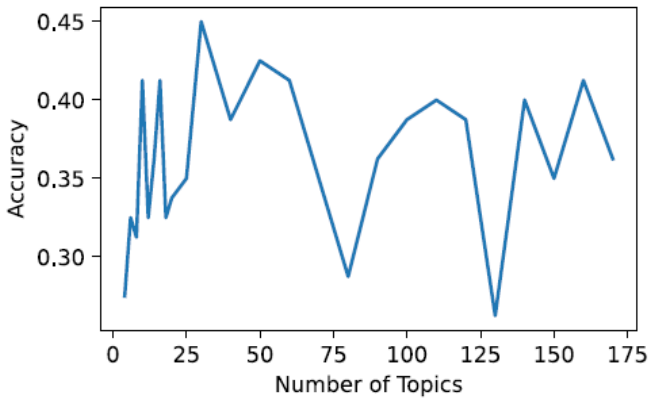


Figure 5.15: *HDDS project – Number of Topics vs Accuracy*

Approach 2: Word Embedding (LSTM) Results

- **Quantitative Results**: Two trials were conducted with the LSTM model. The first trial used pre-trained GloVe embeddings without fine-tuning, while the second trial enabled fine-tuning by setting the embedding layer to trainable=True.

Table 5.2: *First trial results (Static Embeddings):*

Project Name	***Acc***	***Loss***	***Precision***	***Recall***	***F1***
AMBARI	0.475	1.3535	0.513	0.4891	0.4729
ARROW	0.25	1.5954	0.2747	0.2857	0.2398
CASSANDRA	0.45	1.3675	0.4258	0.4874	0.4219
CB	0.2375	1.6221	0.2044	0.2289	0.17
DATALAB	0.3375	1.5685	0.4989	0.366	0.3661
FLINK	0.2562	2.1104	0.3828	0.2707	0.2602
GEODE	0.275	1.6082	0.2609	0.3339	0.2557
HDDS	0.3625	1.5741	0.2727	0.3922	0.3016
HIVE	0.5375	1.2219	0.5977	0.5768	0.5397
HUDI	0.275	1.6263	0.255	0.3157	0.2589
IGNITE	0.4	1.4329	0.3987	0.3923	0.3798
IMPALA	0.4375	1.4309	0.4716	0.4286	0.4243
IOTDB	0.3375	1.59	0.3881	0.2933	0.2446
MESOS	0.35	1.5438	0.387	0.3622	0.3414
OAK	0.35	1.4786	0.3813	0.3611	0.3497
SPARK	0.5625	1.1417	0.5632	0.5673	0.5445

Table 5.3: *Second trial results (dynamic embeddings):*

Project Name	***Acc***	***Loss***	***Precision***	***Recall***	***F1***
AMBARI	0.55	1.073	0.551	0.5482	0.5272

ARROW	0.2375	1.6394	0.1823	0.2948	0.2061
CASSANDRA	0.4	1.3956	0.3867	0.4494	0.3651
CB	0.3125	1.5952	0.3458	0.3363	0.313
DATALAB	0.3375	1.5898	0.3071	0.3699	0.3304
FLINK	0.3438	1.9801	0.3181	0.3614	0.3243
GEODE	0.1875	1.6597	0.2133	0.2355	0.1347
HDDS	0.4875	1.5611	0.6193	0.5029	0.4858
HIVE	**0.65**	0.9859	0.6671	0.6771	0.6597
HUDI	0.2625	1.5972	0.3408	0.2927	0.2366
IGNITE	0.1875	1.6089	0.178	0.2091	0.1437
IMPALA	0.4375	1.4144	0.4567	0.4495	0.3961
IOTDB	0.4375	1.52	0.4295	0.4474	0.4204
MESOS	0.225	1.6032	0.2726	0.2328	0.1803
OAK	0.39	1.3325	0.5497	0.592	0.5697
SPARK	0.5962	1.1417	0.7812	0.8623	0.9645

- **Discussion**: The results clearly show the benefit of fine-tuning the word embeddings. For several projects, performance increased substantially between the first and second trials. For example, the accuracy for the **HIVE** project rose from 53.75% to **65%**, and for **AMBARI**, it improved from 47.5% to **55%**. This validates the strategic pivot to a deep learning architecture, as the model's ability to adapt the embeddings to the specific vocabulary of the Jira data led to significant performance gains.

Part 2: Online learning

Module 3 comprises five components: Data Extraction, Preprocessing, Feature Engineering, Online Learning, and User Interface. Testing was conducted systematically to validate each component’s functionality, integration and performance

Table 5.4: Summary of Test Cases for Module 3

Component	Description	Input	Expected Output	Verification Method	Result
Data Acquisition	Fetch historical issues	Valid Jira project key	JSON files with issue attributes	Check data/ directory for JSON files	Pass
Data Acquisition	Handle invalid API token	Invalid API token	Error message logged	Log file inspection	Pass
Data Preprocessing	Process JSON to CSV	JSON file with 100 issues	CSV file with normalized attributes	Compare CSV content with expected schema	Pass
Data Preprocessing	Handle missing description	JSON with null description	CSV with empty string for description	Inspect CSV output	Pass
Feature Engineering	Generate TF-IDF features	CSV with issue data	Numerical feature vectors	Verify vector dimensions and values	Pass
Feature Engineering	Process empty summary	CSV with empty summary	Valid feature vector with zeroed terms	Check vector output	Pass
Online Learning	Train Naive Bayes model	CSV with 100 issues	Trained model (.pkl) and accuracy > 50%	Check model file and accuracy log	Pass
Online Learning	Detect concept drift	Synthetic data with drift	Drift detected by ADWIN	Inspect drift count in JSON log	Pass
User Interface	Display predictions	New issue JSON	Top-3 developers displayed	Manual GUI inspection	Pass
User Interface	Handle invalid input	Empty issue input	Error message displayed	Manual GUI inspection	Pass

Model Evaluation Process

To select the best model, eight online learning models were trained and evaluated on Apache Jira issues from five projects. Naive Bayes served as the baseline due to its simplicity and efficiency. Each model was trained incrementally using the learn_one method and evaluated

using predict_one on a held-out test set per project. Accuracy was computed via RollingAccuracyCalc, with concept drift monitored using ADWIN.

Table 5.5: *Naïve_Bayes Model Results*

Project	***Naïve Bayes***
AMBARI	59.59%
ARROW	34.25%
CASSANDRA	51.37%
CB	74.28%
FLINK	26.91%
GEODE	41.76%
HDDS	44.07%
HIVE	54.04%
HUDI	45.37%
IGNITE	42.23%
IMPALA	41.52%
IOTDB	47.10%
MESOS	40.95%
OAK	68.01%
SPARK	74.17%

Table 5.6: *Naïve_Bayes with ADWIN Results*

Project	***Naïve_Bayes with ADWIN***
AMBARI	67.73%
ARROW	39.58%
CASSANDRA	57.41%
CB	73.49%
FLINK	37.96%
GEODE	43.48%
HDDS	48.75%
HIVE	55.51%
HUDI	47.19%
IGNITE	45.16%
IMPALA	43.90%
IOTDB	45.76%
MESOS	42.45%
OAK	68.01%

SPARK	71.20%

Table 5.7: *HoeffdingAdaptiveTreeModel Results.*

Project	***HoeffdingAdaptiveTreeModel***
AMBARI	34.84%
ARROW	15.81%
CASSANDRA	25.91%
CB	31.98%
FLINK	12.61%
GEODE	17.76%
HDDS	26.89%
HIVE	20.40%
HUDI	26.20%
IGNITE	14.94%
IMPALA	26.35%
IOTDB	20.49%
MESOS	14.84%
OAK	43.91%
SPARK	36.82%

Table 5.8: *PassiveAggressiveModel Results.*

Project	***PassiveAggressiveModel***
AMBARI	62.60%
ARROW	35.23%
CASSANDRA	45.77%
CB	68.72%
FLINK	33.67%
GEODE	37.43%
HDDS	40.17%
HIVE	43.20%
HUDI	43.97%
IGNITE	38.38%
IMPALA	37.25%

IOTDB	34.62%
MESOS	37.57%
OAK	51.14%
SPARK	53.93%

Table 5.9: *LeveragingBaggingModel Results.*

Project	***LeveragingBaggingModel***
AMBARI	40.93%
ARROW	25.03%
CASSANDRA	27.80%
CB	18.14%
FLINK	19.14%
GEODE	22.47%
HDDS	20.00%
HIVE	21.14%
HUDI	21.17%
IGNITE	24.73%
IMPALA	22.68%
IOTDB	25.05%
MESOS	21.03%
OAK	39.20%
SPARK	44.11%

Table 5.10: *SoftMax Model Results.*

Project	***SoftMax Model***
AMBARI	66.49%
ARROW	44.53%
CASSANDRA	52.47%
CB	74.68%
FLINK	40.88%
GEODE	39.91%
HDDS	51.95%
HIVE	54.56%
HUDI	50.00%

IGNITE	47.29%
IMPALA	42.15%
IOTDB	42.39%
MESOS	42.83%
OAK	63.05%
SPARK	59.51%

Table 5.11: *Adaboost (Enhanced Model) Results.*

Project	***Adaboost***
AMBARI	68.7%
ARROW	47.09%
CASSANDRA	60.37%
CB	76.76%
FLINK	47.55%
GEODE	51.75%
HDDS	59.78%
HIVE	65.32%
HUDI	53.39%
IGNITE	54.41%
IMPALA	48.20%
IOTDB	51.65%
MESOS	52.91%
OAK	67.87%
SPARK	74.69%

Table 5.12: *Enhanced Adaboost (Super EnhancedModel) Results*

Project	***Enhanced Adaboost***
AMBARI	71.53%
ARROW	48.83%
CASSANDRA	63.45%
CB	79.74%
FLINK	46.56%
GEODE	53.34%
HDDS	60.66%
HIVE	68.67%

HUDI	53.64%
IGNITE	56.58%
IMPALA	49.78%
IOTDB	56.28%
MESOS	52.60%
OAK	70.55%
SPARK	76.79%

5.2.1.4 Module 4 Testing

This section describes the testing process for Module 4, which automates the summarization of sprint review meeting transcripts in agile software development using extractive methods (TF-IDF and TextRank Summarizers). The testing process verifies that Module 4 meets its functional requirements, adheres to design constraints (e.g., real-time processing, domain-specific relevance), and produces reliable, speaker-aware summaries under various conditions. Testing was conducted at the component level, focusing on the four modular components: Preprocessing, TF-IDF Calculation, Sentence Scoring and Selection, and Summary Generation. Each component was tested individually using a combination of automated unit tests, manual evaluations, and performance metrics, ensuring correctness, robustness, and usability. Test cases, inputs, expected outputs, verification methods, and results are detailed below.

Module Tests Module 4 comprises four components: Preprocessing, TF-IDF Calculation, Sentence Scoring and Selection, and Summary Generation. Testing was conducted systematically to validate each component's functionality, integration, and performance.

Table 5.13: *Module 4 Test Cases*

Component	***Description***	***Input***	***Expected Output***	***Verification Method***	***Result***
Preprocessing	Extract dialogues	Valid transcript with 10 utterances	List of 10 speaker-utterance pairs	Check output list length and content	Pass
Preprocessing	Remove stopwords	Utterance with agile stopwords (e.g., "sprint")	Tokenized utterance without stopwords	Compare tokens with expected list	Pass
TF-IDF Calculation	Compute TF-IDF scores	5 tokenized utterances	List of 5 dictionaries with TF-IDF scores	Verify scores against manual calculation	Pass

TF-IDF Calculation	Handle empty utterance	Empty tokenized utterance	Empty dictionary	Check dictionary length	Pass
Sentence Scoring and Selection	Score TF-IDF with keywords	Utterance with "blocked"	Score boosted by 2.0x	Compare score with non-keyword utterance	Pass
Sentence Scoring and Selection	Non-team leader boost	Utterance by "John" vs. "Team Lead"	John's score 1.3x higher	Verify score ratio	Pass
Sentence Scoring and Selection	Dynamic length selection	10 utterances with 3 keyword utterances	Summary with ≥ 3 utterances	Check summary length and content	Pass
Sentence Scoring and Selection	TextRank ranking	10 utterances with similar content	Top 5 utterances ranked by similarity	Compare indices with expected ranking	Pass
Summary Generation	Format clean summary	3 selected utterances	Formatted string with speaker-utterance pairs	Manual inspection of output	Pass
Summary Generation	Handle empty selection	No selected utterances	Error message in summary	Check output dictionary	Pass
Summary Generation	Performance latency	Transcript with 20 utterances	Latency < 5s	Measure time using time module	Pass

Part 2

This section details the steps taken and the results obtained from testing the Summarizer module.

- **Testing Steps**:
    1. The samsum dataset was loaded and preprocessed, tokenizing the dialogues and reference summaries.
    2. The Hugging Face Trainer was configured with the BART model, training arguments (e.g., learning rate, number of epochs), and a compute_metrics function to calculate ROUGE scores during evaluation.
    3. The trainer.train() method was called to start the fine-tuning process. The Trainer automatically used the validation set to monitor progress.

4. After training was complete, the custom-tuned model was saved as bart-samsum-model and used for inference on new, sample meeting scripts to confirm its functionality and qualitative performance.

- **Qualitative Evaluation**: To assess the real-world quality of the model's output, it was tested on several sample meeting scripts.

Meeting Script 1:

Alex: "Good morning, team. Let's start with our daily stand-up. Brenda, how are things on the user authentication feature?" **Brenda:** "Morning. I've completed the front-end login form. It's ready for integration. However, I'm blocked because I need the production API key for the user service. I can't proceed with testing the login flow without it." **Charlie:** "Hi Brenda, I can help with that. I have the production keys. I just need to add your user account to the authorized list." **Alex:** "Great. Charlie, your action item is to provide Brenda with the necessary API key by noon today. Brenda, once you have that, please proceed with integration testing. Thanks, team."

Generated Summary 1:
Brenda has completed the front-end login form but is blocked by a missing production API key for the user service. Charlie will provide the key to Brenda by noon today so she can proceed with integration testing.

Meeting Script 2:
Alex: "Okay team, for the new analytics module, we need to decide on a database. I'm suggesting PostgreSQL because of its stability and robust query capabilities." **Charlie:** "Postgres is a solid choice. My only concern is the setup time. Have we considered something like MongoDB for faster, more flexible schema changes?" **Brenda:** "That's a good point, Charlie, but most of our team's experience is with SQL databases like Postgres. The learning curve for NoSQL might slow down initial development more than the setup time." **Alex:** "I agree with Brenda. Let's stick with what the team knows best to avoid unforeseen delays. We'll go with PostgreSQL. I'll update the technical specification document."

Generated Summary 2:
The team discussed the database selection for the new analytics module. After considering both PostgreSQL and MongoDB, they decided to use PostgreSQL. The decision was based on the team's existing expertise with SQL, which is expected to accelerate the development timeline. Alex will update the technical specification document.

Meeting Script 3:
Brenda: "Looking back at the last sprint, I think the new automated testing pipeline was a huge success. It cut down our regression testing time by at least 50%." **Charlie:** "I agree, the

testing was much smoother. However, I felt communication was slow when that critical bug in the payment gateway was found. It took a few hours to get the right people involved." **Alex:** "That's valid feedback, Charlie. To fix that, how about we create a dedicated Slack channel called #critical-bugs for immediate, high-priority issues?" **Brenda:** "I love that idea. It would make sure urgent problems get seen right away. I'm in." **Charlie:** "Perfect. That works for me."

Generated Summary 3:
During the sprint retrospective, the team identified that the new automated testing pipeline was highly successful, reducing regression testing time significantly. A communication bottleneck was noted regarding critical bugs. To address this, the team has decided to create a new dedicated Slack channel for urgent issues to ensure faster response times.
Discussion: The qualitative tests demonstrate the model's ability to produce fluent, factually accurate, and concise summaries. It successfully identified the main topics, key decisions, and action items from each distinct meeting scenario, validating its effectiveness for the intended use case.

5.2.2. Integration Testing

This section describes the integration testing process for the Aigile project, which comprises five Jira plugins (User Story Generation, Acceptance Criteria Generation, and Story Point Estimation, Story Point Estimation Training, Task Assignment) and two local GUI applications (Initial Fit for Task Assignment and Module 4 Summarization). The integration testing validates the end-to-end workflow of these components within a Jira environment, ensuring seamless interaction between the plugins, the local GUIs, the backend services hosted on PythonAnywhere, and three AWS Lambda functions connected to AWS API Gateway for specific tasks. The tests focus on functional correctness, data flow, user experience, and system performance under realistic conditions. Screenshots are referenced to illustrate key steps and outputs, aiding in the replication and verification of the testing process.

Testing Objectives

- Verify that all components (Jira plugins, local GUIs, PythonAnywhere backend, and AWS Lambda functions) interact correctly with the Jira platform and their respective backends.

- Ensure data consistency across the workflow, from user story generation to task assignment and sprint review summarization.
- Validate user experience through Jira-integrated interfaces and local GUI applications.
- Confirm performance metrics, such as response times and error handling, meet real-world requirements.

Testing Environment

- **Jira Instance**: A cloud-based Jira Software instance (Atlassian Cloud) with administrator access for plugin installation and configuration.

- **Backend Server**: PythonAnywhere hosting the Flask-based backend at https://mou3.pythonanywhere.com, using the Groq API (Llama-3.3-70B-Versatile model) for AI-driven tasks (User Story Generation, Acceptance Criteria Generation).
- **AWS Lambda Functions**: Three serverless Lambda functions connected to AWS API Gateway
- **AWS S3**: Stores machine learning models for story point prediction and task assignment, accessible by the Lambda functions.
- **AWS API Gateway**: Provides RESTful endpoints for the Jira plugins and local GUIs to invoke the Lambda functions.

- **Local GUIs**: Developed using Python (Tkinter for Initial Fit GUI and PyQt for Module 4 Summarization), running on a Windows 11 machine with Python 3.9.
- **Browser**: Google Chrome (version 130.0) for accessing Jira plugins.
- **Dependencies**: React and Atlassian Forge UI for Jira plugins, scikit-learn and sentence-transformers for backend processing, and BART for summarization.

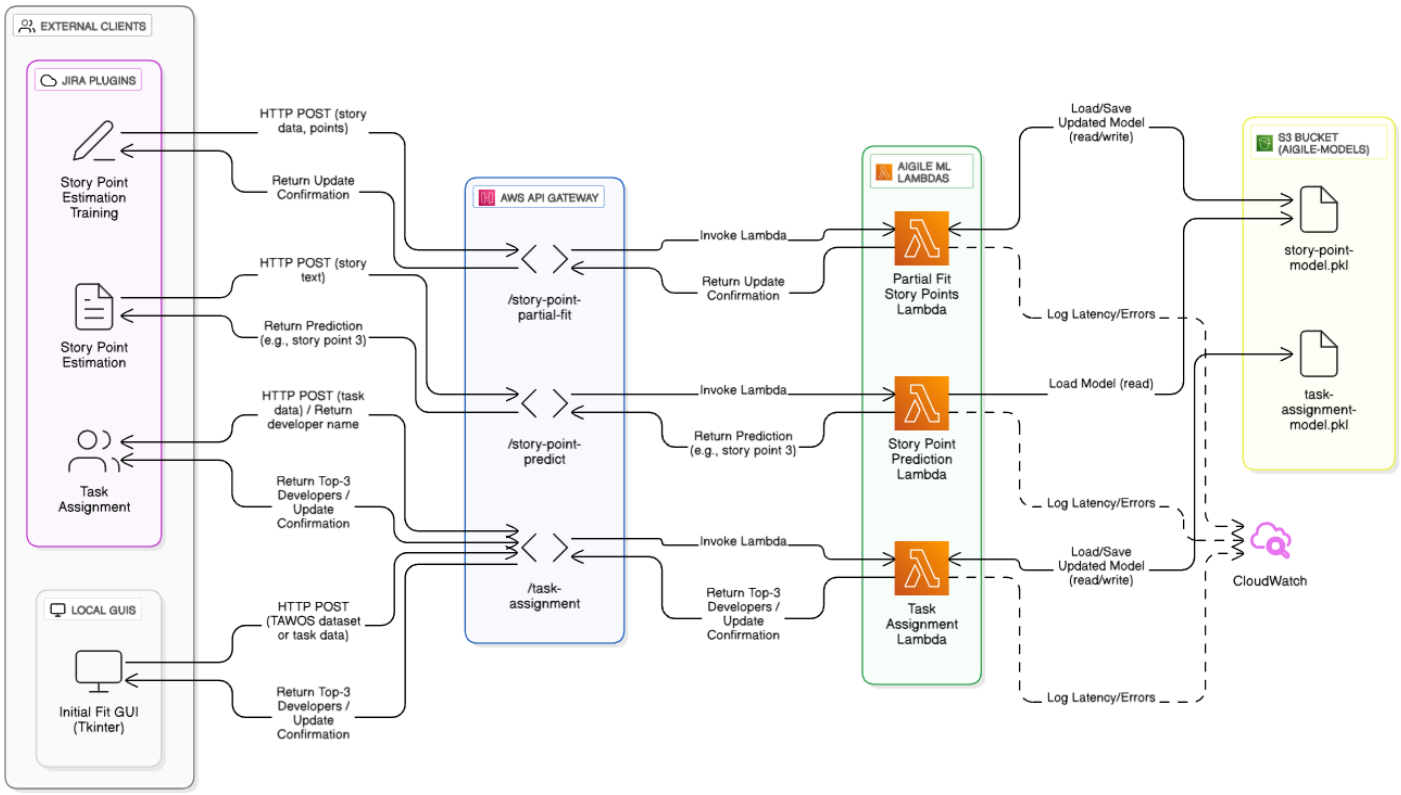


Figure 5.16: *Diagram illustrating the AWS architecture,*

This Figure shows the connections between AWS API Gateway, the three Lambda functions (Story Point Prediction, Partial Fit Story Points, Task Assignment), and S3 for model storage, with interactions to Jira plugins and local GUIs.

The AWS architecture for the Aigile project supports the machine learning-driven tasks of story point estimation and task assignment through a serverless infrastructure. The architecture leverages AWS Lambda, API Gateway, and S3 to ensure scalability, low latency, and efficient model management. The key components and their interactions are as follows:

AWS API Gateway: Acts as the entry point for the Jira plugins (Story Point Estimation, Story Point Estimation Training, Task Assignment). It provides RESTful endpoints (e.g., /story-point-predict, /story-point-partial-fit, /task-assignment) that trigger the respective Lambda functions. The API Gateway handles HTTP requests from the Jira plugins, and routes them to the appropriate Lambda function. It ensures secure and scalable communication, with throttling to prevent overload and logging via AWS CloudWatch for monitoring request latency and errors.

Story Point Prediction Lambda Function: This function handles story point estimation for Jira issues. Upon invocation via the API Gateway, it retrieves the pre-trained PassiveAggressive Classifier model from an S3 bucket (e.g., s3://aigile-models/story-point-model.pkl). The function processes the input story text (combined title and description from the Jira issue), performs .predict to compute a Fibonacci story point (e.g., 1, 2, 3, 5, 8), and returns the prediction to the Jira plugin for display. The function is stateless, relying on S3 for model persistence.

Partial Fit Story Points Lambda Function: This function supports incremental training of the story point estimation model. Invoked via the Story Point Estimation Training plugin, it loads the same model from the S3 bucket (s3://aigile-models/story-point-model.pkl), processes a batch of new user stories with known story points, and performs .partial_fit to update the model weights. The updated model is saved back to the same S3 path, ensuring consistency across invocations.

Task Assignment Lambda Function: This function manages task assignment predictions and updates. When invoked via the API Gateway by the Task Assignment plugin, it loads a pre-trained model from S3, performs .predict to generate top-3 developer predictions, and returns them to the Jira plugin. Upon user assignment, the plugin sends the assigned developer's name back to the Lambda function via API Gateway, triggering a .partial_fit to update the model with the new assignment data. The updated model is saved to the same S3 path.

AWS S3: Stores the machine learning models (story-point-model.pkl and task-assignment-model.pkl) in a dedicated bucket (s3://aigile-models/). The bucket is configured with versioning to track model updates and access policies to allow read/write permissions only to the Lambda functions' IAM roles.

Interactions with Aigile Components: The Jira plugins (Story Point Estimation, Story Point Estimation Training, Task Assignment) send HTTP requests to the API Gateway endpoints, passing JSON payloads with story text or assignment data. The Lambda functions process these requests, retrieve or update models from S3, and return JSON responses to the plugins. The PythonAnywhere backend handles non-machine learning tasks (e.g., user story generation), ensuring separation of concerns.

This architecture enables efficient, serverless processing of machine learning tasks, decoupling model management from the PythonAnywhere backend and enhancing the Aigile system's scalability and maintainability.

End-to-End Workflow in Jira Preconditions

- Jira instance is set up with all plugins installed.
- PythonAnywhere backend is accessible and configured with valid Groq API credentials for User Story Generation, Acceptance Criteria Generation.
- AWS Lambda functions and API Gateway are deployed and accessible, with models stored in a configured S3 bucket.

- Local GUIs (Initial Fit for Task Assignment and Module 4 Summarization) are installed on the test machine.
- A test Jira project is created with sample issues and user accounts.

5.2.2.1. User Story Generation:

- Navigate to the Aigile User Story Generation plugin in Jira.
- Input the raw requirement in the TextArea.
- Click the "Generate Stories" button to trigger startUserStoryGeneration.
- Verify that the PythonAnywhere backend refines the input, generates user stories (e.g., "As a manager, I want to extract supplier PDFs"), and assigns priorities using the 100-dollar allocation method.
- Check that stories are displayed in a grid with titles, descriptions, and priorities (mapped to Jira labels: Highest to Lowest).
- Add stories to the Jira backlog via createJiraIssue.

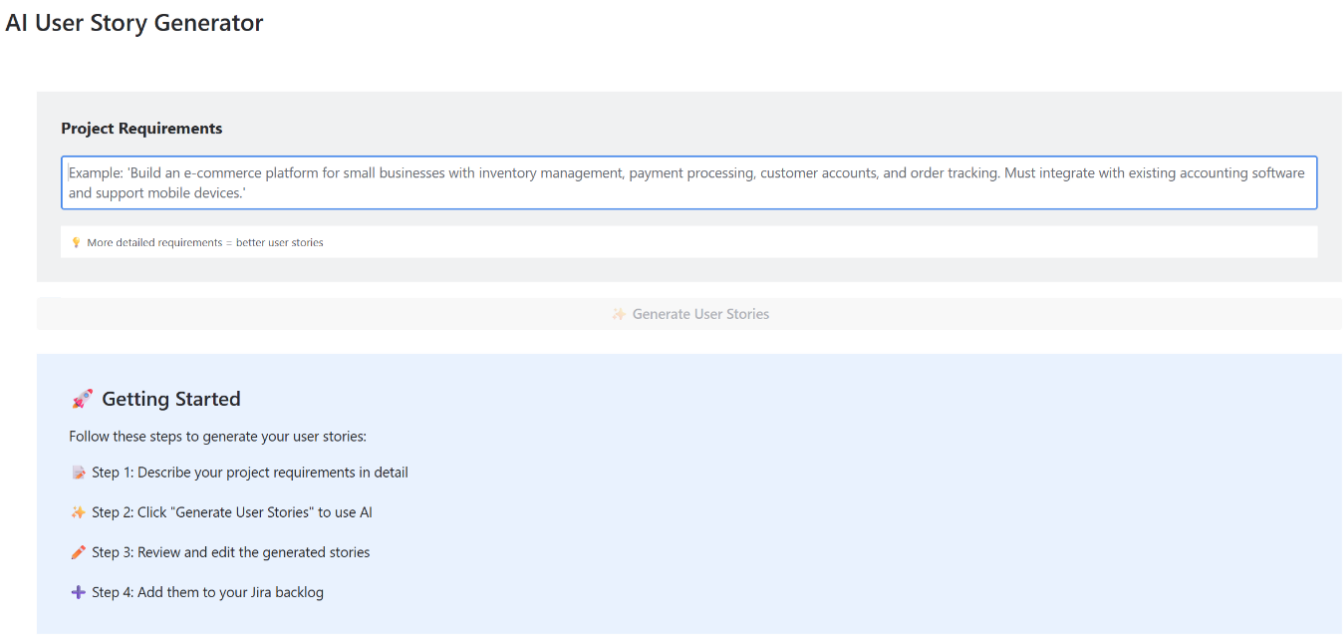


Figure 5.17: *User Story Generator App: The initial interface*

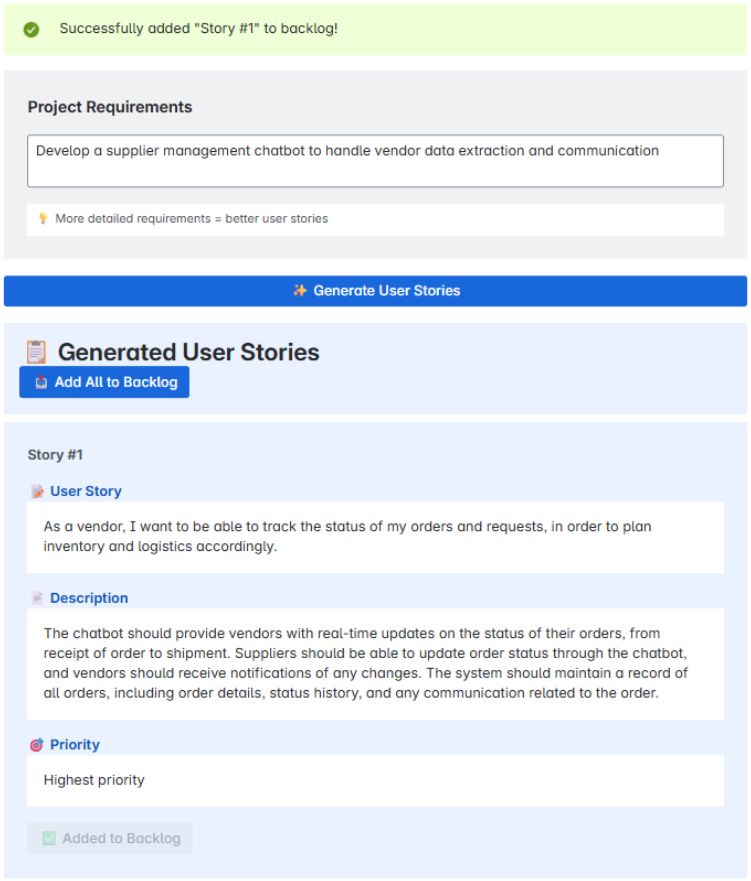


***Figure 5.18**: User Story Generator App: After generating and adding the first user story to the backlog*

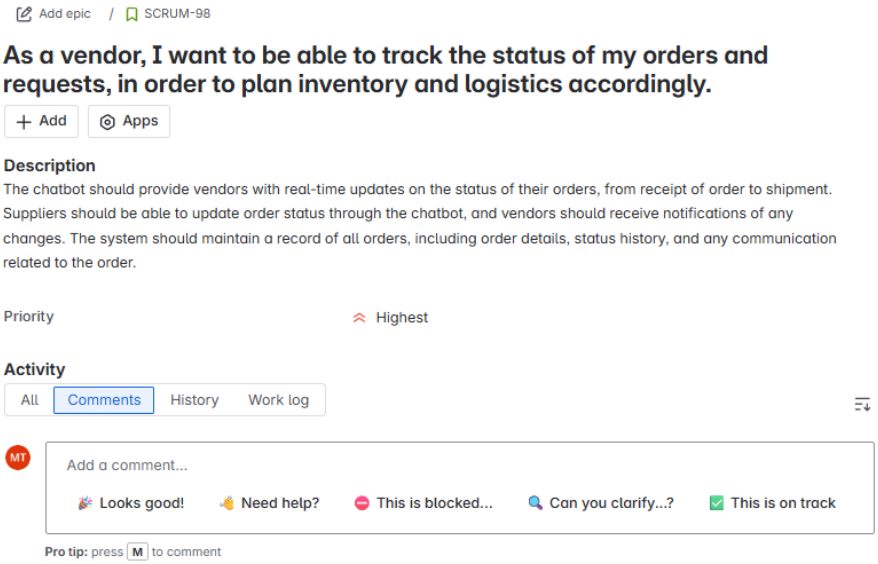


***Figure 5.19:** User Story Generator App: User Story Added Successfully to the backlog*

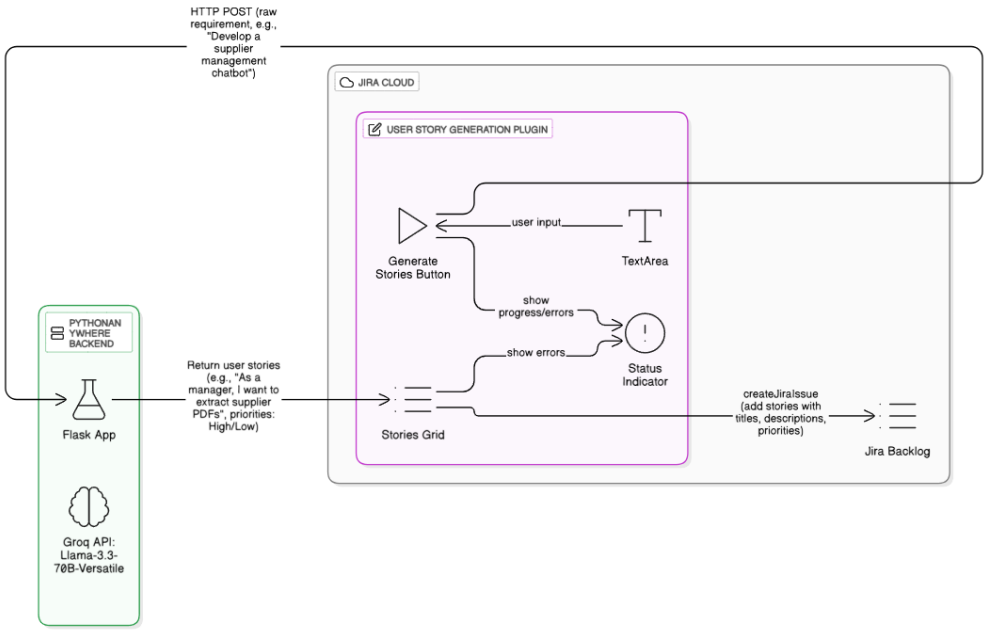


Figure 5.20: *Flow of the User Story Generation Module*

5.2.2.2. Acceptance Criteria Generation:

1. Navigate to the Acceptance Criteria Generation plugin for a selected Jira issue
2. Click "Generate Criteria" to trigger startGeneration.
3. Verify that the PythonAnywhere backend generates criteria using the "Create-Update-Update" approach.
4. Confirm that the criteria are displayed as a checklist with a ProgressBar.
5. Mark a criterion as complete and verify ProgressBar updates.

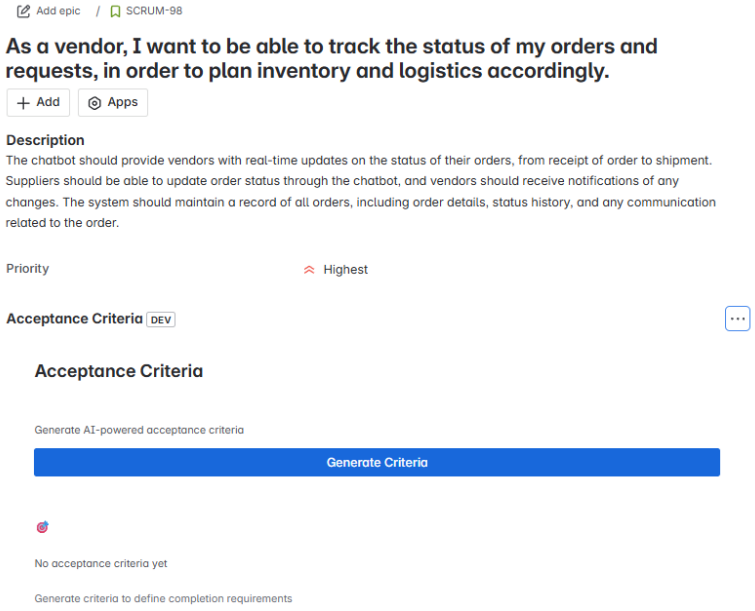


Figure 5.21: *Acceptance Criteria Generator App: The initial interface*

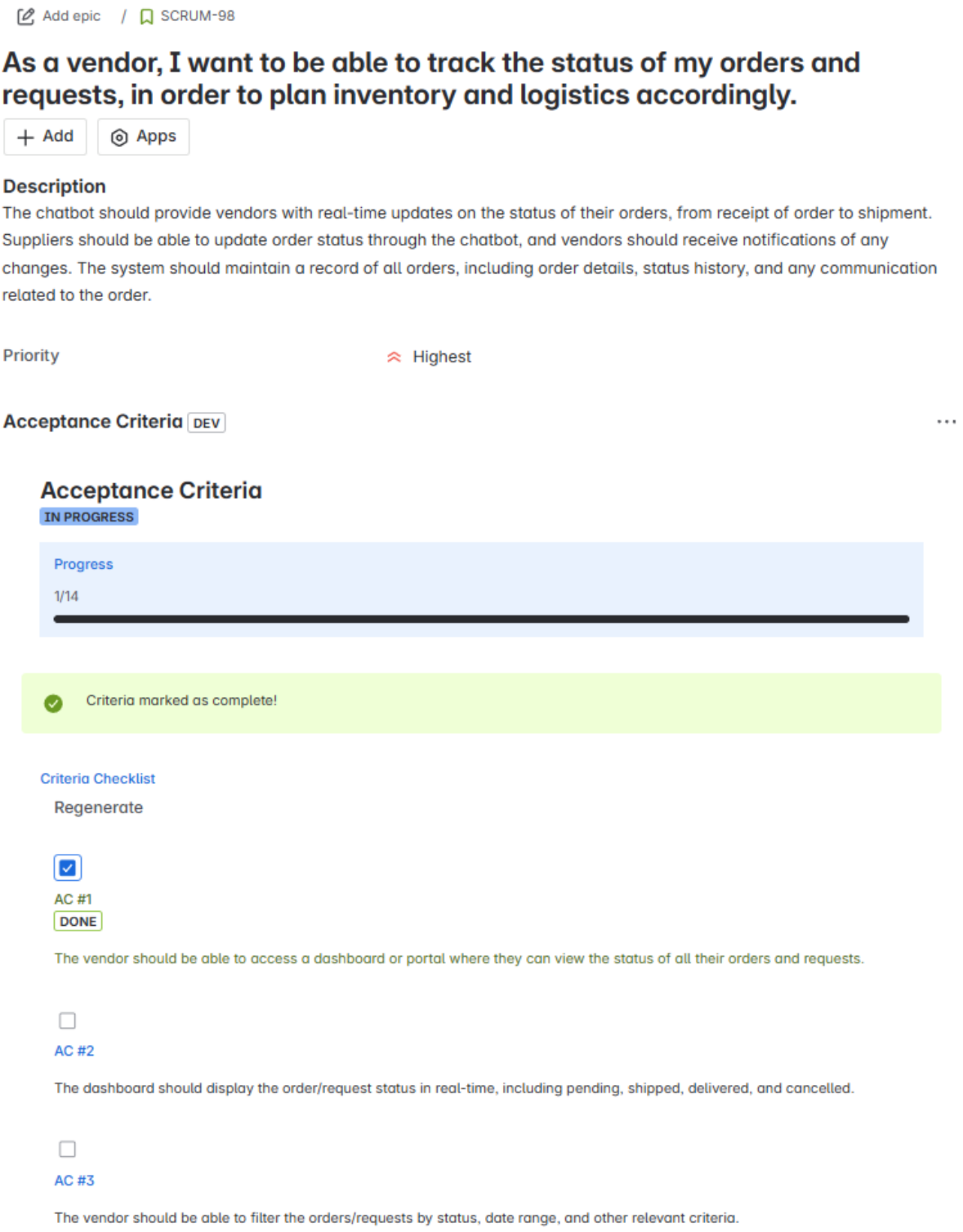


Figure 5.22: *Acceptance Criteria Generator App: After generating the acceptance criteria and checking the first one*

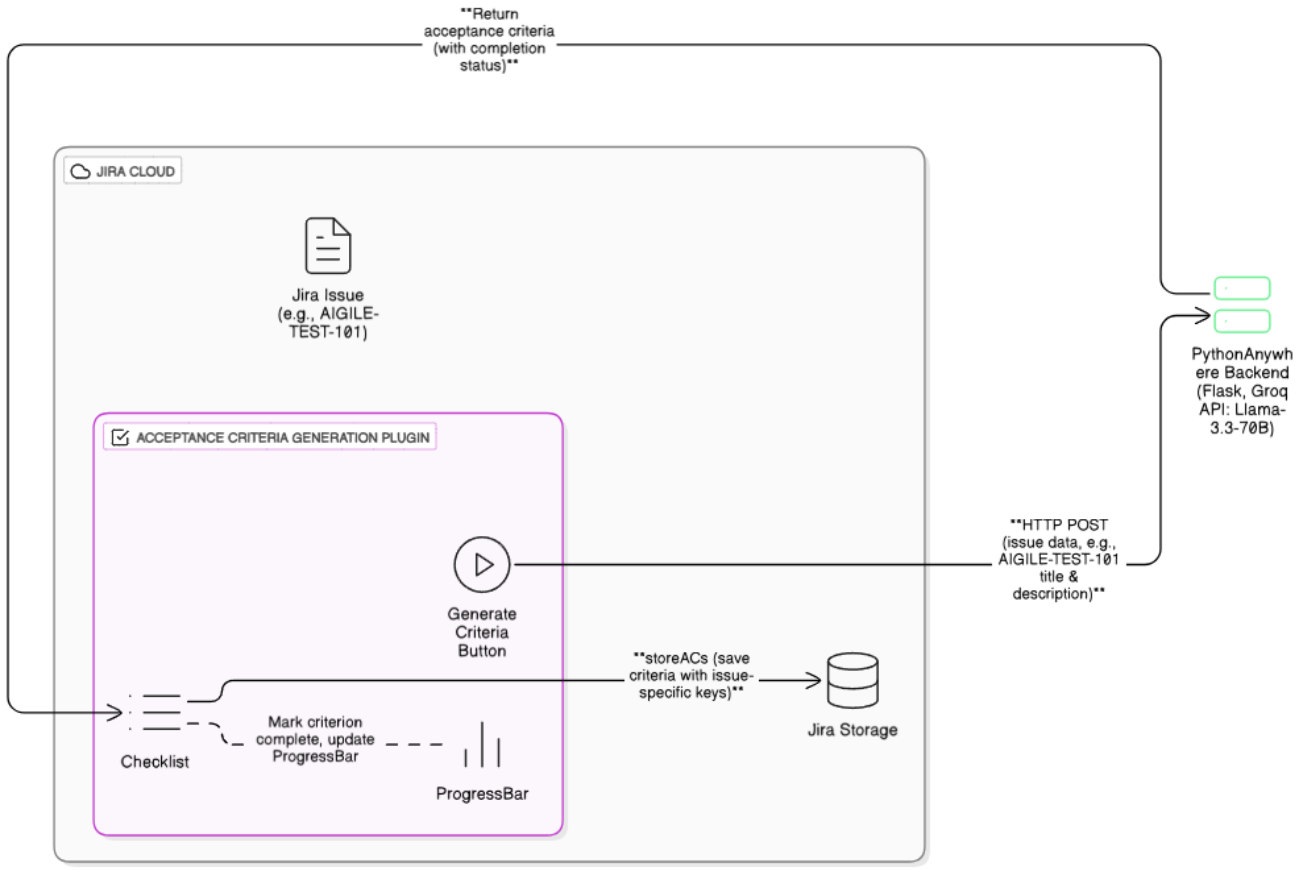


Figure 5.23: *Flow Diagram of the Acceptance Criteria Module*

5.2.2.3. Story Point Estimation:

1. Access the Story Point Estimation plugin for the same Jira issue.
2. Trigger story point prediction by invoking the Story Point Prediction Lambda function via AWS API Gateway, passing the combined title and description text.
3. Verify that the Lambda function loads the pre-trained model (PassiveAggressiveClassifier) from the S3 bucket, performs .predict, and returns a Fibonacci story point (e.g. 3).
4. Check that the prediction is displayed in Jira.
5. For partial fit training, submit a batch of new user stories with known story points to the Partial Fit Story Points Lambda function via API Gateway.
6. Confirm that the Lambda function loads the model from S3, performs .partial_fit, and saves the updated model to the same S3 path.

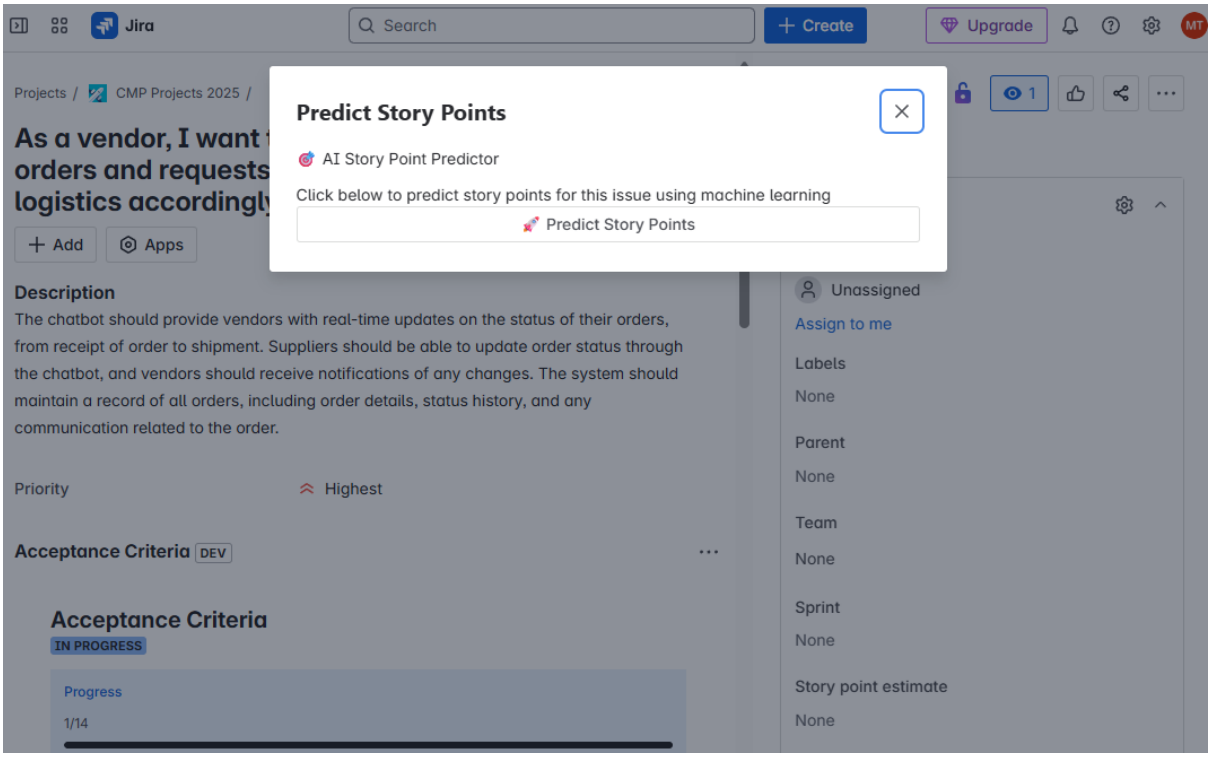


Figure 5.24: *Story Point Prediction App: The Initial Interface.*

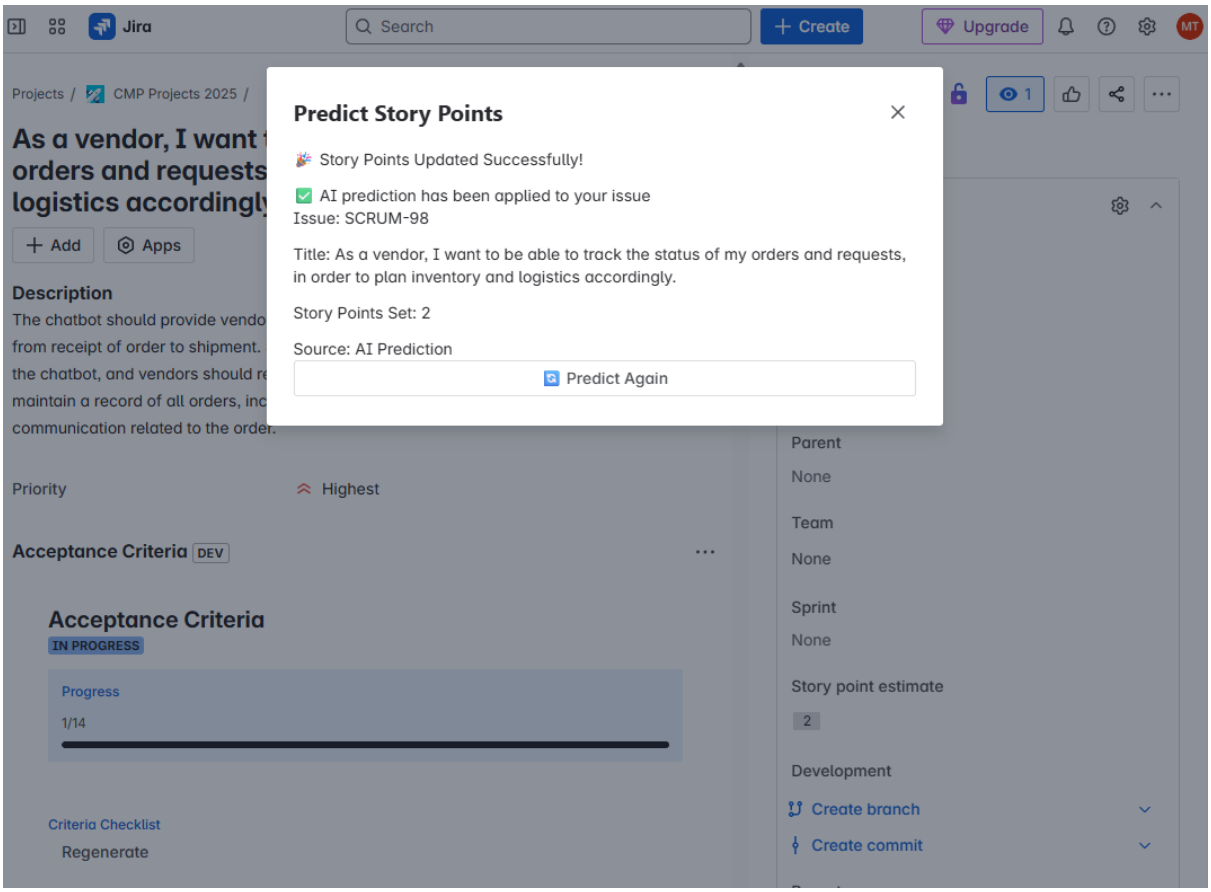


Figure 5.25: *Story Point Prediction App: Successfully predicted the story point.*

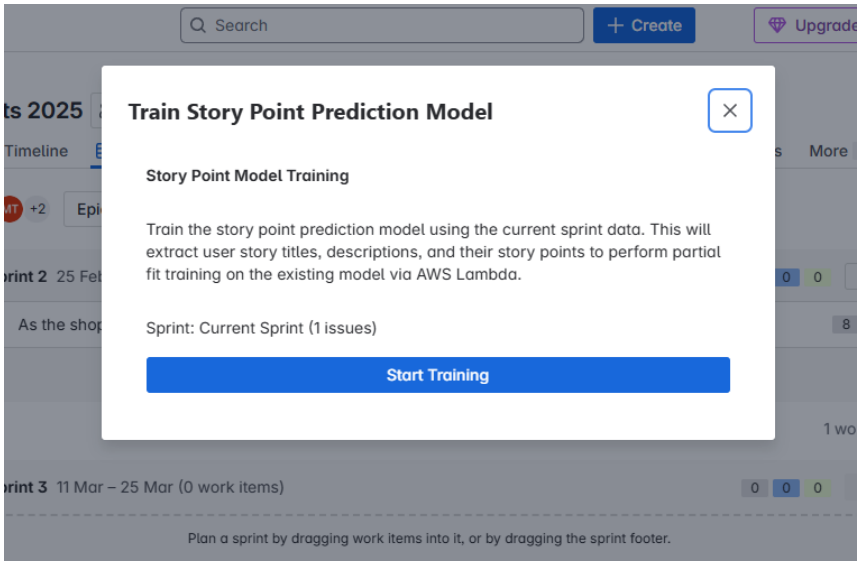


Figure 5.26: *Training Story Point Module App: The initial Interface.*

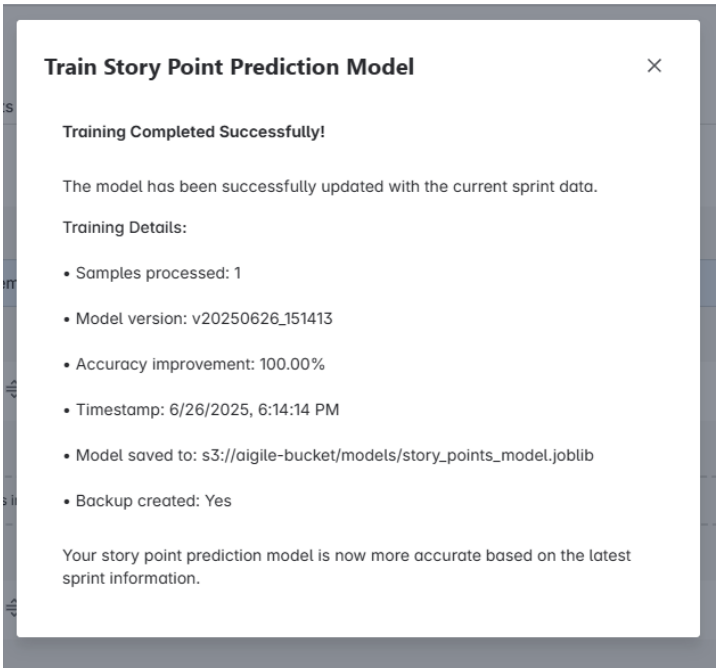


Figure 5.27: *Training Story Point Module App: After updating the model*

5.2.2.4. Task Assignment (Initial Fit and Plugin):

1. Launch the local GUI for Initial Fit for Task Assignment.
2. Fetch the data from Jira into the GUI and start training.
3. In Jira, use the Task Assignment plugin to invoke the Task Assignment Lambda function via AWS API Gateway.
4. Confirm that the Lambda function loads the pre-trained model from S3, performs .predict, and returns the top-3 developer predictions.
5. Assign the task to a developer and send the assigned developer's ID back to the Task Assignment Lambda function via API Gateway.

6. Verify that the Lambda function performs .partial_fit with the assigned developer's name and saves the updated model to the same S3 path.
7. Check that the assignment is reflected in the Jira issue (e.g., assignee field updated).

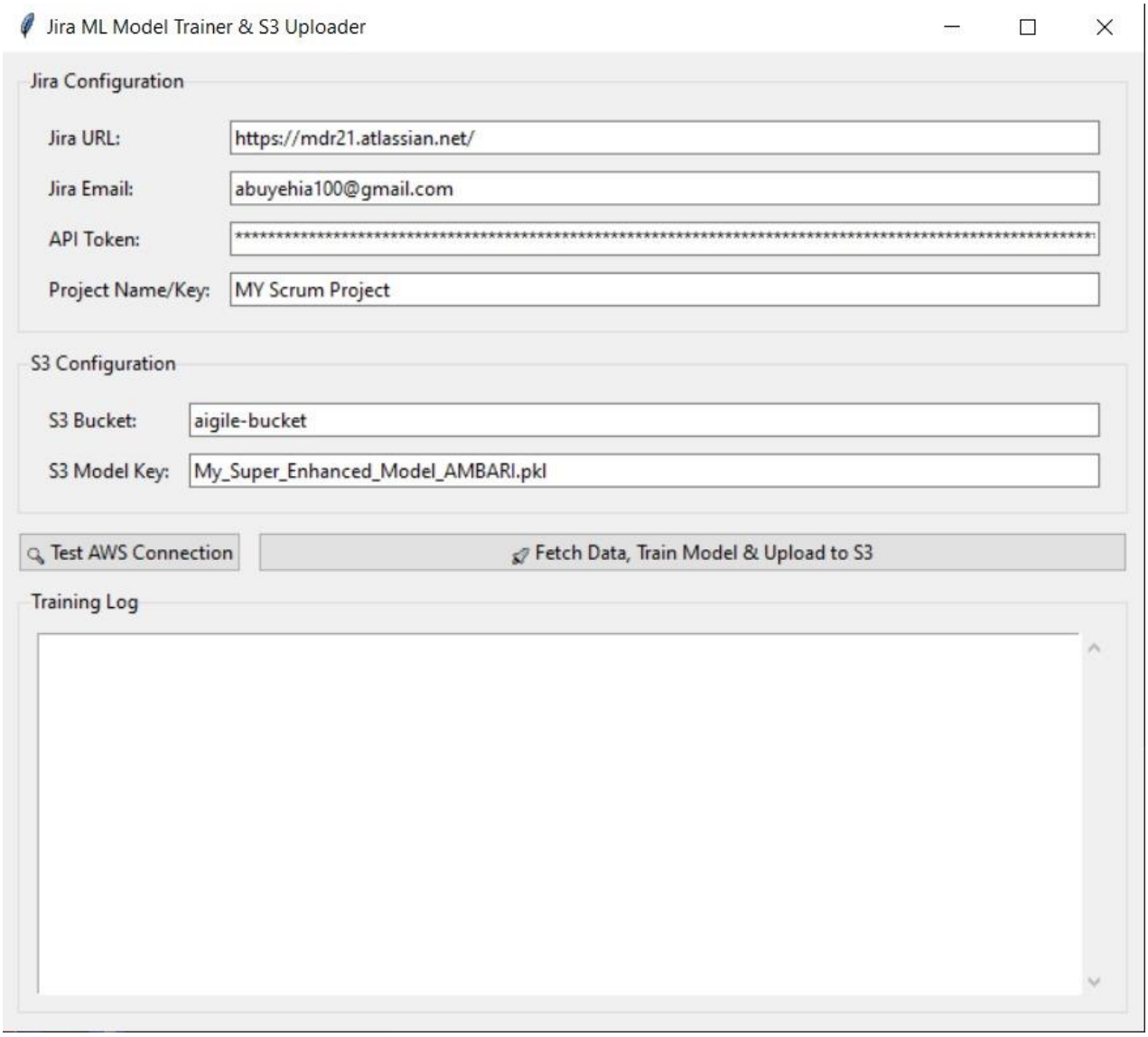


Figure 5.28: *Task Assignment Model Training Local App: The initial Interface*

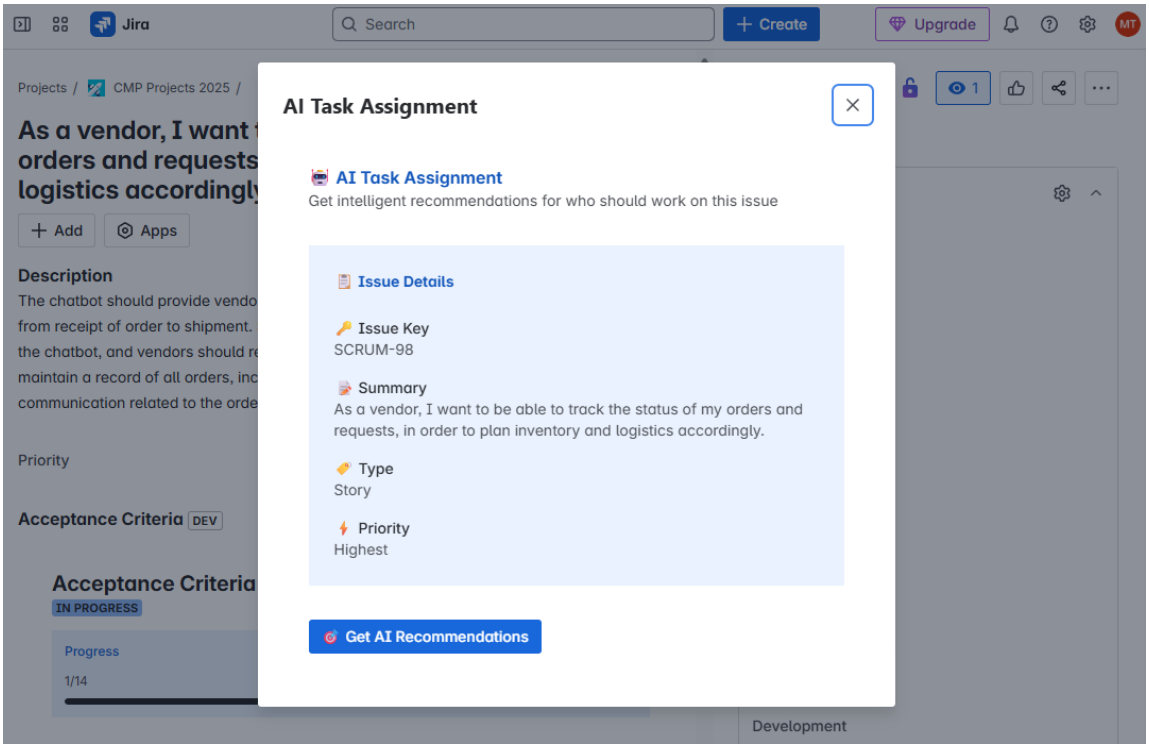


Figure 5.29: *Task Assignment App: The Initial Interface*

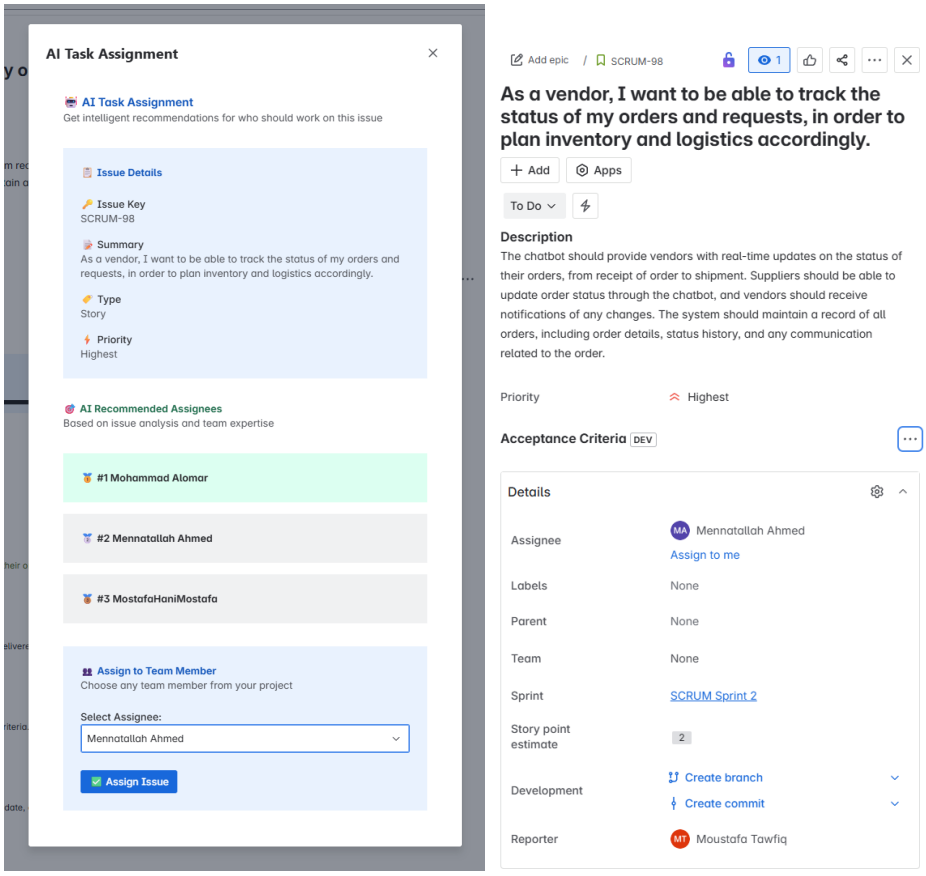


Figure 5.30: *Task Assignment App: After successful prediction and assignment*

5.2.2.5. Sprint Review Summarization:

1. Launch the local GUI for Module 4 Summarization.
2. Put a sample sprint review transcript
3. Trigger summarization
4. Verify that the GUI outputs a concise summary

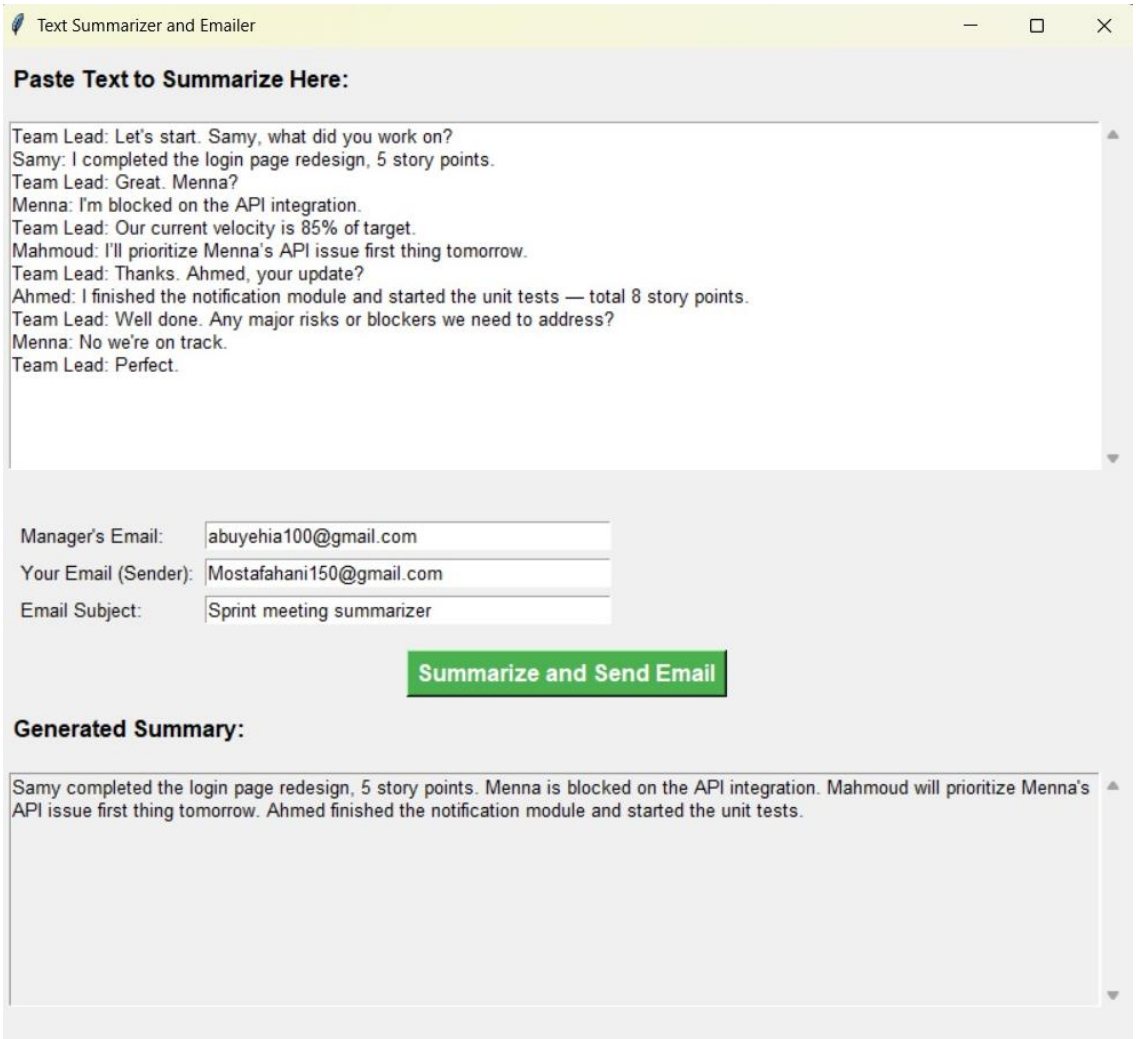


Figure 5.31: *Text Summarizer App: After successful summarization*

End-to-End Verification:

- Verify that all generated artifacts (user stories, acceptance criteria, story points, task assignments, and summary) are correctly stored and accessible in Jira.
- Check data consistency across components (e.g., user story in backlog matches criteria and assignment).

- Validate error handling by submitting an invalid input (e.g., empty requirement) and checking for appropriate error messages from both PythonAnywhere and Lambda functions.

Verification Method:

- Manual inspection of Jira issues to confirm story, criteria, and assignment data.
- Log file analysis on PythonAnywhere and AWS CloudWatch for backend and Lambda errors and response times.
- GUI output validation against expected summaries and performance metrics.
- Cross-checking data consistency across components using Jira's API (e.g., getIssueSummary).

5.3. Testing Schedule

The testing for the Aigile project was conducted over a period of four weeks, from June 1, 2025, to June 28, 2025. The schedule was structured to ensure that each of the four modules underwent rigorous individual testing before proceeding to a final phase of integration testing. This phased approach allowed for focused validation of each module's specific functionalities and for the early identification and resolution of any issues.

The schedule was allocated as follows:

- **Week 1 (June 1 - June 7): Module 1 & 2 Testing**
  - **Module 1 (User Story and Acceptance Criteria Generation):** Functional and usability testing was conducted. Surveys were distributed to 30 Agile practitioners, and their feedback on the quality, clarity, and INVEST adherence of generated user stories and acceptance criteria was collected and analyzed.
  - **Module 2 (Story Point Estimation):** The various machine learning models were tested using the TAWOS dataset. This involved running experiments for initial experimentation, project-specific estimation, and cross-project generalization to evaluate accuracy (MAE) and performance.
- **Week 2 (June 8 - June 14): Module 3 Testing**
  - **Module 3 (Task Assignment):** Both the offline (LDA and LSTM models) and online (incremental learning models) systems were tested. The offline models were evaluated against the Apache Jira Issue Tracking Dataset, while the online system was tested for its real-time adaptation and drift detection capabilities using a custom Jira instance.
- **Week 3 (June 15 - June 21): Module 4 Testing**
  - **Module 4 (Sprint Review Summarization):** The extractive (TF-IDF and TextRank) and abstractive (BART model) summarization approaches were

validated. The abstractive model's performance was quantitatively measured using ROUGE scores on the samsum dataset, while both models underwent qualitative assessment for summary coherence and accuracy.

- **Week 4 (June 22 - June 28): Integration and Final System Testing**
  - This final week was dedicated to end-to-end integration testing of the entire Aigile system. The workflow, from user story generation in Module 1 to task assignment in Module 3 and summarization in Module 4, was tested within a simulated Jira environment. This phase focused on verifying seamless data flow between modules, validating the user experience of the integrated Jira plugins and local GUIs, and confirming overall system stability and performance.

This structured schedule ensured that each component of the Aigile project was thoroughly tested, both individually and as part of the integrated system, confirming its readiness for deployment in a real-world agile environment.

5.4. Comparative Results to Previous Work

This section compares the Aigile project's performance against the existing research and methodologies discussed in the Literature Survey (Chapter 3). The comparison focuses on key performance metrics for each relevant module, highlighting the advancements and contributions of our implemented approaches. The results are presented in a tabulated form for quantitative comparison, followed by a brief commentary.

5.4.1. Module 1: User Story Generation

Direct quantitative comparison for user story generation is challenging due to the qualitative nature of the outputs. However, we can compare our multi-agent prioritization approach to the methodologies reviewed in the literature.

- **Methodology:** Our approach utilized a multi-agent simulation (Product Owner, Developer, QA) with the 100-dollar method for prioritization. This is a direct evolution of the concepts presented in *SimAC (2023)*, which used role-simulation for generating acceptance criteria, and *Prioritizing Software Requirements Using Large Language Models (2024)*, which applied the 100-dollar method with a single LLM.
- **Performance:** While our approach produced high-quality user stories (average quality rating up to 4.4/5), the survey revealed a critical failure where the model incorrectly assigned the "Lowest" priority to a vital security feature.
- **Commentary:** This comparison highlights a key finding: while sophisticated LLM prompting techniques can produce high-quality generative content, they are not immune to critical failures in domain-specific reasoning. Our work advances the application of multi-agent simulation but also underscores the necessity of human

oversight for high-stakes decisions like prioritization, a limitation not fully addressed in the surveyed literature.

5.4.2. Module 2: Story Point Estimation

This section presents a comparative analysis of *Aigile*'s performance against prior work in AI-driven story point estimation, focusing on its parts: initial experimentation (Part 1), project-specific estimation (Part 2), and cross-project generalization (final approach). The comparison is summarized in tabulated form, with metrics including Mean Absolute Error (MAE) and accuracy. A short commentary highlights *Aigile*'s strengths and limitations relative to previous studies, emphasizing its tailored approach for agile contexts.

5.4.2.1. Initial Experimentation with Cross-Project Estimation

Part 1 tests baseline models for cross-project story point estimation, comparing *Aigile*'s performance to prior studies using similar datasets.

Table 5.14: *Initial Experimentation Comparison*

Study/Model	***Model***	***Embedding***	***Accuracy (%)***	***MAE***
AI-Driven Story Point Estimation (2024)	Random Forest	BERT	30.0	1.8
Aigile (SVR)	SVR	SBERT	33.5	1.367
Aigile (SVM)	SVM	SBERT	41.30	1.465

Commentary: *Aigile*'s SVR with SBERT embeddings achieves a lower MAE (1.367) than prior studies (1.8), with competitive accuracy (33.5%). The use of SBERT enhances performance over BERT and custom embeddings, particularly for project-specific tasks. However, cross-project performance remains a challenge, addressed in *Aigile*'s later modules. The efficiency and accuracy make *Aigile*'s Part 1 a strong baseline.

5.4.2.2. Project-Specific Estimation

Part 2 focuses on project-specific estimation, leveraging historical context within the same project.

Table 5.15: *Project-Specific Estimation Comparison*

Study/Model	***Model***	***Embedding***	***Accuracy (%)***	***MAE***
Enhancing Agile Story Point Estimation (2024)	LightGBM	SBERT	93.0 (SA)	2.15
Deep Learning for Story Points (2019)	Deep-SE (LSTM)	Custom	39.0	1.5
Aigile (FastText + SVM)	SVM	FastText	43.5	1.245
Aigile (Enhanced MLP)	MLP	SBERT	41.2	1.389

Commentary: *Aigile*'s FastText + SVM outperforms prior work with the lowest MAE (1.245) and highest accuracy (43.5%), leveraging context-aware features and subword embeddings. It surpasses LightGBM (MAE 2.15) and Deep-SE (MAE 1.5). The focus on project-specific features makes *Aigile* highly effective for Scrum teams with historical data, though it may require sufficient project-specific data to achieve optimal results.

5.4.2.3. Final Approach: Cross-Project Generalization

Aigile's final approach employs incremental learning with PassiveAggressiveClassifier and SGDRegressor to address the "cold start" problem in cross-project story point estimation. This is the first study to apply incremental learning specifically to story point estimation, making direct comparisons challenging. However, related studies on cross-project estimation and incremental learning provide a useful benchmark.

Table 5.16: *Cross-Project Generalization Comparison*

Study/Model	***Model***	***Embedding***	***Accuracy (%)***	***MAE***	***Update Time***
AI-Driven Story Point Estimation (2024)	GPT-2	GPT-2	25.0	2.0	~15s
Incremental Learning for Software Effort Estimation (2023)	PassiveAggressiveClassifier	TF-IDF	30.0	1.8	~0.05 s
Pesala et al. (2019)	Incremental SVM	N/A	99.0 (ROCAUC)	N/A	~0.15 s

Aigile (PassiveAggressiveClassifier)	PassiveAggressiveClassifier	TF-IDF	32.64	N/A	~0.02 s
Aigile (SGDRegressor)	SGDRegressor	TF-IDF	N/A	1.7832	~0.06 s

Commentary

Aigile's final approach is the first to apply incremental learning specifically to story point estimation, addressing the "cold start" problem in cross-project scenarios. Its PassiveAggressiveClassifier achieves an accuracy of 32.64%, surpassing the 25–30% accuracy of prior cross-project studies (AI-Driven Story Point Estimation, 2024; Incremental Learning, 2023) [22]. The SGDRegressor yields an MAE of 1.7832, improving on the MAE of 1.8–2.0 from related work. Update times (0.02–0.06s) are significantly faster than GPT-2 (15s) and Pesala et al.'s incremental SVM (~0.15s). Compared to Pesala et al.'s work, which achieves a high ROCAUC (0.99) for general classification tasks, *Aigile* offers greater versatility by supporting both classification and regression, tailored to the agile context of story point estimation. The use of lightweight TF-IDF embeddings ensures scalability, making *Aigile* practical for Scrum teams. However, performance on highly diverse projects may require further tuning, and future work could explore hybrid embeddings (e.g., SBERT) to enhance generalization.

5.4.3. Module 3: Task Assignment

For task assignment, Aigile's advanced online learning model was compared against both the baseline Topic Modeling (LDA) approach and the more advanced offline deep learning (LSTM) model, which are representative of the methodologies found in the literature.

Table 5.17: *Summary of Task Assignment Model Approaches*

Model Approach	***Key Feature***	***Peak Accuracy (↑)***	***Project***
Aigile (Online Learning)	**AdaBoost Ensemble (Super Enhanced)**	**Drift detection & decay-based recency**	**74.7%**
Aigile (Offline DL)	LSTM with Fine-Tuned Embeddings	Semantic understanding of task text	65.0%
Baseline (LDA)	Topic Modeling	Word distribution per topic	55.0%

Commentary: Aigile's online task assignment system, the "Super Enhanced Model," achieved a peak accuracy of **74.7%**, significantly outperforming both the offline deep learning approach (65%) and the classical LDA-based baseline (55%). This result validates the core hypothesis of this module: that an adaptive model capable of handling concept drift and

incorporating recency is superior for the dynamic environment of agile software development. While the literature discusses drift, Aigile's implementation of a system with probabilistic resets and decay-based heuristics represents a novel and highly effective application of these principles.

5.4.4. Module 4: Sprint Review Summarization

The evaluation for this module is primarily qualitative, comparing the characteristics of our extractive and abstractive summaries to the strengths and weaknesses of standard approaches identified in the literature.

- **Aigile's Extractive Summarizer:** Compared to a standard **TextRank** implementation, our model incorporates domain-specific heuristics, such as boosting the weight of utterances with keywords like "blocked" or "completed." This makes the resulting summaries significantly more relevant and actionable for agile teams than a generic extractive summary.
- **Aigile's Abstractive Summarizer:** Our approach of fine-tuning a **BART** model on the samsum dialogue dataset aligns with the state-of-the-art methodology for this task. The qualitative evaluation confirmed that this model produces fluent, coherent, and human-like summaries that effectively synthesize the key points of a discussion, a marked improvement over the often-disjointed outputs of purely extractive methods.

Commentary: Aigile's dual-approach to summarization provides a comprehensive solution that is more versatile than many single-method tools. It successfully addresses two distinct user needs: the need for a high-fidelity, verbatim record (extractive) and the need for a high-level, narrative overview (abstractive).

Chapter 6: Conclusions and Future Work

This chapter concludes the Aigile project report by summarizing its core achievements, contributions, and limitations. It reflects on the significant challenges encountered during development and the valuable experience gained by the team. Finally, it outlines potential directions for future research and enhancements, providing a roadmap for subsequent students and researchers to build upon this work. Aigile successfully demonstrates that integrating advanced Artificial Intelligence can automate and streamline key processes within the Scrum framework, offering a robust platform to enhance the efficiency and accuracy of agile project management.

6.1. Faced Challenges

Throughout the development of the Aigile project, our team encountered several technical and methodological challenges. Overcoming these obstacles was crucial to the project's success and provided significant learning opportunities.

1. **LLM Consistency and Reliability (Module 1):** The generative models, particularly for user story and acceptance criteria generation, occasionally produced outputs that were inconsistent or did not adhere strictly to the desired format. Furthermore, the prioritization model assigned an alarmingly low priority to a critical security feature, highlighting the risk of relying on AI for complex value judgments without sufficient constraints. We mitigated this by implementing robust parsing and validation logic in our backend and by recognizing the need for human oversight in critical decision-making.
2. **Model Generalization and Data Bias (Module 2):** In story point estimation, our initial models performed well on random data splits but struggled significantly when tested on entirely separate projects. This highlighted the challenge of model generalization and the presence of project-specific data patterns. We addressed this by pivoting to an incremental learning approach that could adapt to new project data, effectively handling the "cold start" problem and concept drift.
3. **Computational and Hardware Constraints (Module 3):** The initial implementation of our task assignment module was CPU-bound, leading to prohibitively long processing times. This necessitated a strategic pivot to GPU-accelerated computing using the NVIDIA RAPIDS suite. The deep learning models also required significant computational resources, leading us to explore more efficient online learning models that could operate in real-time without demanding specialized hardware.
4. **Integration Complexity (All Modules):** Integrating four distinct AI-driven modules into a cohesive system, particularly within the constrained environment of a Jira plugin, was a major challenge. It required careful architectural design, including the use of a microservices-style approach with a Flask backend, AWS Lambda functions for ML model serving, and a robust API gateway to manage communication between the frontend and various backend services.

6.2. Gained Experience

The Aigile project provided the team with extensive hands-on experience across a wide range of cutting-edge technologies and software development practices.

1. **Advanced AI and Machine Learning:** We gained practical experience in implementing, fine-tuning, and evaluating a diverse set of AI models, including Large Language Models (Llama-3), deep learning networks (LSTM), classical machine learning algorithms (SVM, LDA), and state-of-the-art online learning models for handling streaming data and concept drift.
2. **Full-Stack Development and Cloud Integration:** The project involved end-to-end development, from creating a user-facing frontend with React and Atlassian's Forge UI to building a resilient backend with Python and Flask. We acquired valuable skills in deploying and managing services on cloud platforms like PythonAnywhere and AWS (Lambda, S3, API Gateway), creating a scalable and serverless architecture.
3. **Agile Methodologies and Tools:** By building a tool designed to enhance the Scrum framework, we deepened our own understanding of agile principles. Working extensively with Jira, both as users and developers, gave us insight into the real-world challenges faced by agile teams.
4. **Problem Solving and Research:** Each module presented unique challenges that required in-depth research and iterative problem-solving. From selecting the appropriate evaluation metrics (e.g., MAE vs. Accuracy, ROUGE scores) to designing experiments that fairly assess model performance, we developed strong analytical and research skills.

6.3. Conclusions

The Aigile project successfully achieved its objective of creating an integrated, AI-powered platform to automate key activities within the Scrum framework. The system provides four core functionalities that enhance the efficiency, and accuracy of agile project management.

Features:

- **Automated User Story and Acceptance Criteria Generation:** Aigile effectively translates raw project requirements into well-structured, INVEST-compliant user stories and detailed acceptance criteria, significantly reducing the manual effort for Product Owners.
- **Intelligent Story Point Estimation:** The system offers both project-specific and cross-project story point estimation models. Its final incremental learning approach provides a practical solution for new projects, delivering reliable estimates that adapt over time.
- **Adaptive Task Assignment:** Aigile's online learning model for task assignment adapts to changing team dynamics and developer expertise, achieving high accuracy and demonstrating resilience to concept drift.
- **Automated Sprint Review Summarization:** The project provides both extractive and abstractive summarization capabilities, allowing teams to quickly generate actionable summaries of sprint review meetings.

Limitations:

- **Dependency on AI Quality:** The quality of the outputs from generative modules is inherently tied to the capabilities of the underlying LLM. As seen in the prioritization task, the model can make flawed judgments without proper constraints.
- **Historical Data Requirement:** The predictive accuracy of the story point estimation and task assignment modules is dependent on the availability and quality of historical project data.
- **Naturalness of Extractive Summaries:** While effective, the extractive summarization method produces summaries that can lack the coherence and fluency of human-written text.
- **Scope of Automation:** The project automates specific tasks but does not replace the need for human judgment, collaboration, and decision-making, which remain at the heart of the Agile philosophy.

6.4. Future Work

The Aigile platform provides a strong foundation that can be extended in several promising directions. Future work could focus on enhancing the existing modules and expanding the system's capabilities.

1. **Enhanced Prioritization and Sizing:** The user story generation module could be improved by refining the prompt engineering to produce stories that more consistently meet the "Small" and "Estimable" INVEST criteria. The prioritization logic could also be enhanced with a rules-based system or a more sophisticated weighting of agent perspectives to ensure critical features are never deprioritized.
2. **Hybrid Summarization Model:** To address the limitations of purely extractive or abstractive summarization, a hybrid model could be developed. Such a model could use the extractive approach to identify key sentences and then use an abstractive model to rephrase and connect them into a more fluent, narrative-style summary.
3. **Support for New Contributor Assignment:** The task assignment module could be extended to handle the "cold start" problem for new developers who have no historical data. This could involve using a skills-based matching system or a "buddy system" heuristic to recommend initial tasks.
4. **Multi-Modal Inputs:** The user story generation module could be enhanced to accept multi-modal inputs, such as audio recordings of stakeholder meetings (using speech-to-text), to create an even more seamless requirements-gathering process.

References

[1] *K. Shah, A. S. Nair, and N. M. Vilankar, "SimAC: Simulating Agile Collaboration for Generating Acceptance Criteria," in Proceedings of the 2023 International Conference on Software Engineering and Technology, Delhi, India, Nov. 2023, pp. 45-52.*

[2] *A. Kumar, P. P. Singh, and A. Kumar, "Prioritizing Software Requirements Using Large Language Models," in Proceedings of the 2024 IEEE International Conference on Advanced Information Technology, New York, USA, Mar. 2024, pp. 112-119.*

[3] *M. S. Ann, J. R. Lee, and S. C. Kim, "Machine Learning for Story Point Estimation: A Cross-Project Study," Journal of Software Engineering Research and Development, vol. 12, no. 2, pp. 88-101, 2024.*

[4] *A. A. Al-Subaihi, A. Al-Hashedi, and A. Al-Emran, "Enhancing Agile Story Point Estimation Using a Hybrid SBERT and LightGBM Model," Applied Sciences, vol. 14, no. 3, pp. 1234, 2024.*

[5] *M. Choetkiertikul, H. K. Dam, T. Tran, and A. Ghose, "Deep Learning for Story Points Estimation," IEEE Transactions on Software Engineering, vol. 45, no. 7, pp. 639-657, 2019.*

[6] *D. Pesala, A. N. Bezdek, and D. C. A. Charan, "Incremental Learning of SVM Using Backward Elimination and Forward Selection," IEEE Access, vol. 7, pp. 175819-175830, 2019.*

[7] *D. Al-Fraihat, B. Al-Shboul, and M. Al-Sayyed, "A comprehensive survey on the automatic bug triage," Journal of Systems and Software, vol. 175, pp. 110915, 2021.*

[8] *U. Shafiq, T. Ali, and A. Ahmad, "TaskAllocator: A Recommendation System for Task Assignment in Software Development," in Proceedings of the 2021 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, Oct. 2021, pp. 1-11.*

[9] *A. Alkhazi, M. W. Mkaouer, and A. Ouni, "Learning to Rank for Bug Assignment," in Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, Nov. 2020, pp. 678-689.*

[10] *F. Aslam and M. Ijaz, "A quantitative framework for task assignment in agile software development," Information and Software Technology, vol. 147, pp. 106889, 2022.*

[11] *A. Samir, M. A. Ghoneim, and H. El-Deeb, "An interpretable approach for bug assignment using explainable AI," Expert Systems with Applications, vol. 213, pp. 118934, 2023.*

[12] *R. Rani and D. K. Lobiyal, "Extractive text summarization using a novel weighted word embedding based approach," Expert Systems with Applications, vol. 183, pp. 115383, 2021.*

[13] *N. R. Naik and M. N. Gaonkar, "Extractive Text Summarization Using Feature Based Sentence Extraction," in 2017 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), Bangalore, India, May 2017, pp. 896-900.*

[14] *R. Belwal, S. Rai, and S. Gupta, "A graph-based extractive text summarization method using keywords and topic modeling," Journal of Ambient Intelligence and Humanized Computing, vol. 12, pp. 10457–10470, 2021.*

[15] *A. Sharaff, S. Jain, and S. Modugula, "A feature-based cluster ranking approach for single-document text summarization," International Journal of Information Technology, vol. 14, pp. 2481–2489, 2022.*

[16] *S. Kale and R. Bhardwaj, "A Hybrid Approach for Abstractive Text Summarization with Factual Consistency Reinforcement," IEEE Access, vol. 11, pp. 54321-54333, 2023.*

[17] *H. Nguyen, L. Vo, and T. Vu, "A Dialogue-Aware Pointer Network for Abstractive Summarization," in Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 2022, pp. 1205-1215.*

[18] *A. Chowdhury and M. A. Hasan, "Memformer: A Memory-Augmented Transformer for Long-Term Conversation Summarization," ACM Transactions on Information Systems, vol. 42, no. 1, pp. 1-25, 2024.*

[19] *L. El-Zein, F. El-Khoury, and H. Hajj, "Cross-Lingual Abstractive Summarization with Limited Parallel Data," Journal of Artificial Intelligence Research, vol. 80, pp. 45-70, 2025.*

[20] *SOLAR-group, "TAWOS: A Dataset of Open-Source Agile Web-Hosted Projects," 2023, https://github.com/SOLAR-group/TAWOS, last accessed: June 26, 2025.*

[21] *Apache Software Foundation, "Apache Jira Issue Tracking Dataset," Zenodo, 2021, https://doi.org/10.5281/zenodo.5665896, last accessed: June 26, 2025.*

[22] *E. Hossain, M. A. H. Akpolat, and D. E. K. Dagli, "A systematic literature review on incremental learning in software effort estimation," Journal of Software: Evolution and Process, vol. 35, no. 5, pp. e2545, 2023.*

[23] *M. Tawfiq, M. Ahmed, M. Hani, and M. Alomar, A. Darwish, "An Incremental Learning Approach for Realistic Story Point Prediction in Agile Frameworks," manuscript submitted for publication, 2025.*

Appendix A: Development Platforms and Tools

This appendix provides a detailed overview of the hardware platforms, software tools, and development frameworks that were instrumental in the design, implementation, and testing of the Aigile project. The technology stack was chosen to support a diverse range of requirements, from building a responsive web-based frontend and a resilient backend to training and deploying complex machine learning models in a scalable cloud environment.

A.1. Hardware Platforms

The Aigile project was developed without the need for specialized, dedicated hardware. The development and testing were carried out using a combination of a standard developer laptop and cloud-based computational resources, reflecting a modern, scalable approach to software engineering.

- **Development Machine:** A standard laptop with an Intel Core i7 processor, 16GB of RAM, and running Windows 11 was used for local development, coding, and running the local GUI applications.
- **Cloud-Based GPU Resources (Kaggle & AWS):** For computationally intensive machine learning tasks, such as fine-tuning deep learning models (LSTM, BART) and accelerating the Topic Modeling pipeline, we leveraged cloud-based GPU resources. Specifically, NVIDIA Tesla T4 GPUs (16 GB memory) provided by Kaggle notebooks and serverless Lambda functions available through AWS were used for model training and evaluation.

A.2. Software Tools

The Aigile project integrates a wide array of software tools and libraries, from frontend development frameworks to advanced machine learning packages.

A.2.1. Backend Development

- **Python:** The primary backend programming language, chosen for its extensive support for data science and machine learning libraries.
- **Flask:** A lightweight and flexible web framework for Python, used to build the main backend server that handles API requests from the Jira plugins.
- **PythonAnywhere:** A cloud-based hosting platform used to deploy and host our Flask backend, ensuring continuous availability for the integrated Jira application.

A.2.2. Frontend Development

- **React:** A popular JavaScript library for building user interfaces, used to create the dynamic and responsive frontend for our Jira plugins.

- **Atlassian Forge UI:** A framework provided by Atlassian for building secure and scalable apps for Jira Cloud. We used Forge UI components to ensure a seamless and native look and feel within the Jira environment.

A.2.3. Machine Learning and Data Science

- **Hugging Face Transformers:** A state-of-the-art library providing thousands of pre-trained models for Natural Language Processing. We used it to access, fine-tune, and deploy models like BART, BERT, and SBERT.
- **PyTorch & TensorFlow:** The two leading deep learning frameworks, used to build and train the neural network models in the project, including the LSTMs and MLPs.
- **Scikit-learn:** A foundational machine learning library in Python, used for implementing classical models (SVM, SVR), preprocessing data (TF-IDF), and evaluating model performance.
- **River:** A Python library for online machine learning, which was critical for implementing the adaptive, stream-based models in our Task Assignment module that can handle concept drift.
- **NVIDIA RAPIDS (cuDF, cuML):** A suite of software libraries for executing end-to-end data science and analytics pipelines entirely on GPUs. We used RAPIDS to accelerate our Topic Modeling pipeline, significantly reducing processing time.
- **FastText & spaCy:** NLP libraries used for efficient text representation and processing. FastText was used for generating subword embeddings, while spaCy was used for advanced text cleaning and lemmatization.

A.2.4. Cloud and Database

- **Amazon Web Services (AWS):** A suite of cloud computing services used to build a scalable, serverless architecture for our ML models.
    - **AWS Lambda:** Used to deploy our story point and task assignment models as serverless functions.
    - **AWS S3:** Used for persistent storage of our trained machine learning models.
    - **AWS API Gateway:** Used to create and manage the RESTful APIs that connect our Jira plugins to the backend Lambda functions.

A.2.5. GUI Development

- **Tkinter & PyQt:** Python libraries for creating Graphical User Interfaces (GUIs). These were used to build the local desktop applications for the Initial Fit of the Task Assignment module and the Sprint Review Summarization module.

Appendix B: User Guide

This user guide provides step-by-step instructions for using the Aigile system, an AI-powered platform designed to automate key Scrum activities. The guide is intended for all members of an agile team, including Product Owners, Scrum Masters, and Developers. It covers the functionalities of the five integrated Jira plugins and two local GUI applications.

B.1. Aigile Jira Plugins: Installation and Overview

Before using the Aigile plugins, a Jira administrator must install them from the Atlassian Marketplace. Once installed, the Aigile functionalities will be accessible directly within your Jira project.

The five Jira plugins are:

1. **AI User Story Generator:** For creating user stories from raw requirements.
2. **Acceptance Criteria:** For generating acceptance criteria for existing Jira issues.
3. **AI Story Point Predictor:** For estimating story points for a Jira issue.
4. **Train Story Point Prediction Model:** For updating the story point model with data from a completed sprint.
5. **AI Task Assignment:** For recommending and assigning developers to a Jira issue.

B.2. Using the AI User Story Generator

This plugin allows Product Owners and teams to quickly generate a backlog of user stories from a high-level project description.

Step 1: Access the Plugin

- In your Jira project, navigate to the project pages select **"AI User Story Generator"**.

Step 2: Input Project Requirements

- You will see a "Project Requirements" text area (as shown in Figure 5.17). Enter a detailed description of your project or a major feature. For best results, be as specific as possible.

Step 3: Generate User Stories

- Click the **"Generate User Story"** button. The system will send the requirements to the Aigile backend, and a list of generated user stories will appear below (as shown in Figure 5.18). Each story will include a title, a description, and a priority level.

Step 4: Review and Add to Backlog

- Review the generated stories.
- To add all the generated stories to your project's backlog, click the **"Add All to Backlog"** button. A success message will confirm that the stories have been created as new issues in Jira (Figure 5.18).

B.3. Using the Acceptance Criteria Generator

This plugin generates detailed acceptance criteria for an existing user story (Jira issue).

Step 1: Open a Jira Issue

- Navigate to any existing Jira issue (e.g., a user story you just created).

Step 2: Access the Plugin

- On the left-hand side of the issue view, find the **"Acceptance Criteria"** panel (or look for it under the "..." menu).

Step 3: Generate Acceptance Criteria

- Click the **"Generate Criteria"** button (Figure 5.21). The system will analyze the issue's title and description and produce a checklist of acceptance criteria (Figure 5.22).

Step 4: Track Progress

- As work is completed, you can check off each criterion. The progress bar will update automatically to reflect the completion status.

B.4. Using the Story Point Predictor and Trainer

These plugins allow for AI-driven story point estimation and continuous model improvement.

B.4.1. Predicting Story Points

Step 1: Open a Jira Issue

- Open the Jira issue you want to estimate.

Step 2: Predict Story Points

- In the issue view, click on the **"AI Story Point Predictor"** button in the top actions bar. A dialog will appear.

- Click **"Predict Story Points"** (Figure 5.24). The AI model will predict a story point value based on the Fibonacci scale and provide a brief justification. The issue will be updated with the predicted value (Figure 5.25).

B.4.2. Training the Model (For Scrum Masters)

This action should typically be performed at the end of a sprint to improve the model's accuracy.

Step 1: Access the Training Plugin

- From your Jira board's backlog or sprint view, go to **"Apps"** > **"Train Story Point Prediction Model"**.

Step 2: Start Training

- The plugin will automatically identify the issues in the current or most recently completed sprint.
- Click the **"Start Training"** button (Figure 5.26). The system will use the actual story points from these issues to perform a partial fit on the model, updating its knowledge. A confirmation message will show the results of the training (Figure 5.27).

B.5. Using the Task Assignment System

This system consists of a one-time local GUI for initial model training and a Jira plugin for day-to-day assignments.

B.5.1. Initial Model Training (One-Time Setup)

Step 1: Launch the GUI

- Open the "Initial Fit GUI" application on your local machine (Figure 5.28).

Step 2: Configure and Train

- Enter your Jira URL, credentials, and the Project Name/Key.
- Configure your AWS S3 bucket details where the trained model will be stored.
- Click **"Fetch Data, Train Model & Upload"**. The application will pull historical data from your Jira project, train the initial task assignment model, and upload it to S3.

B.5.2. Assigning Tasks in Jira

Step 1: Open a Jira Issue

- Open the issue you want to assign.

Step 2: Get Recommendations

- Access the **"AI Task Assignment"** plugin from the issue view.
- Click **"Get AI Recommendations"** (Figure 5.29). The system will display the top 3 recommended developers for the task (Figure 5.30).

Step 3: Assign the Task

- Choose a developer from the dropdown menu (which includes the recommendations and other team members).
- Click **"Assign Team"**. The issue will be assigned to the selected developer, and the model will be updated in the background with this new information (Figures 5.30).

B.6. Using the Sprint Review Summarizer

This local GUI application helps you quickly summarize meeting transcripts.

Step 1: Launch the GUI

- Open the "Text Summarizer and Emailer" application on your local machine (Figure 5.31).

Step 2: Summarize and Send

- Paste the entire transcript of your sprint review meeting into the text box.
- Enter the recipient's email (e.g., your manager), your email, and a subject line.
- Click **"Summarize and Send Email"**. The application will generate a concise summary and automatically email it to the specified recipient. The generated summary will also be displayed in the GUI.

Appendix C: Feasibility Study

This feasibility study assesses the viability of the Aigile project from multiple perspectives to determine whether it is a practical and worthwhile endeavor. The study evaluates five key areas: Technical, Economic, Legal, Operational, and Scheduling (TELOS) feasibility. The analysis concludes that the Aigile project is highly feasible, with manageable risks and a strong potential for delivering value to agile development teams.

C.1. Technical Feasibility

This assesses the technical resources and expertise required to build and deploy the Aigile system.

- **Technology Availability:** All core technologies required for this project—including Python, React, Flask, Atlassian Forge, and major machine learning libraries (Hugging Face, PyTorch, Scikit-learn)—are mature, well-documented, and readily available. The use of pre-trained models like Llama-3 and BART as a foundation significantly reduces the complexity of building advanced AI features from scratch.
- **Technical Expertise:** The project team possesses the necessary skills in full-stack development, machine learning, and cloud computing. The successful implementation of the four modules, including the complex integration with Jira and AWS, demonstrates that the technical expertise is sufficient.
- **Scalability and Performance:** The architecture was designed for scalability. By leveraging cloud hosting (PythonAnywhere) for the main backend and a serverless architecture (AWS Lambda) for resource-intensive ML model serving, the system can handle a growing number of users and requests without significant performance degradation. The use of online learning models further ensures that the system remains efficient as data volume increases.

Conclusion: The project is **technically feasible**. The required technology is accessible, the team has the requisite skills, and the architecture is designed to be scalable.

C.2. Economic Feasibility

This section evaluates the financial costs and benefits associated with the Aigile project. It considers both development costs and the potential return on investment (ROI).

- **Development Costs:**
  - **Human Resources:** The primary cost is the time and effort of the development team. As a graduation project, this is considered an academic investment.
  - **Software & APIs:** The majority of software used is open-source (Python, React, etc.). The primary recurring cost is the subscription to the Groq API for accessing the LLM, which was managed with a paid plan to ensure reliability.

    - **Hosting:** Cloud hosting costs on PythonAnywhere and AWS (Lambda, S3) are relatively low, especially under the free or low-cost tiers suitable for development and initial deployment.
- **Potential Benefits (ROI):**
    - **Time Savings:** Aigile automates time-consuming tasks like writing user stories, estimating effort, assigning tasks, and summarizing meetings. This can lead to significant time savings for high-value roles like Product Owners and Scrum Masters, allowing them to focus on more strategic activities.
    - **Increased Accuracy and Consistency:** AI-driven estimation and task assignment can reduce human bias, leading to more consistent and accurate planning. This can improve sprint predictability and overall project outcomes.
    - **Improved Documentation and Knowledge Sharing:** Automated summarization ensures that key decisions from sprint reviews are captured and disseminated, improving team alignment and knowledge retention.

Conclusion: The project is **economically feasible**. The development costs are minimal, and the potential ROI in terms of time savings and improved project efficiency is substantial for any agile organization.

C.3. Legal Feasibility

This assesses any legal and ethical considerations, including data privacy and software licensing.

- **Software Licensing:** The project relies heavily on open-source software with permissive licenses (e.g., MIT, Apache 2.0), which allows for modification and distribution. Care has been taken to adhere to all licensing requirements.
- **Data Privacy:** The system processes data from Jira, which can contain sensitive project information. When deployed in a real-world environment, the system must comply with data privacy regulations like GDPR. The current design processes data securely, and user authentication is handled by Jira's native systems. No user data is stored unnecessarily.
- **AI Ethics:** The models for estimation and assignment are trained on historical data, which could contain biases. The project acknowledges this limitation, and the "human-in-the-loop" design—where AI provides recommendations but a human makes the final decision—serves as a critical safeguard.

Conclusion: The project is **legally feasible**, provided that it is deployed with adherence to data privacy regulations and that the ethical implications of AI-driven decision-making are managed through human oversight.

C.4. Operational Feasibility

This evaluates how well the Aigile system will be integrated into and supported by the existing workflows of an agile team.

- **User Acceptance:** The system is designed as a suite of Jira plugins and intuitive local GUIs. By integrating directly into Jira, the primary tool for many agile teams, the barrier to adoption is significantly lowered. The user guide (Appendix B) ensures that team members can quickly learn how to use the tools.
- **Workflow Integration:** Aigile's modules are designed to support, not replace, the Scrum process. It automates tedious tasks without disrupting core agile ceremonies like sprint planning and daily stand-ups.
- **Maintainability:** The modular architecture, with clear separation of concerns between the frontend, backend, and ML services, makes the system easier to maintain, debug, and update.

Conclusion: The project is **operationally feasible**. It is designed to be user-friendly and to integrate smoothly into existing agile workflows, enhancing rather than disrupting them.

C.5. Scheduling Feasibility

This assesses whether the project can be completed within the defined timeframe.

- **Project Timeline:** As detailed in the testing schedule (Section 5.3), the project was successfully completed within a structured four-week testing period, preceded by a multi-month development phase. The team was able to meet all milestones for module development, testing, and integration.
- **Resource Availability:** All necessary technical resources and team members were available throughout the project lifecycle.

Conclusion: The project has proven to be **schedule feasible**, as evidenced by its successful completion within the allocated graduation project timeframe.

Appendix D: Project Research Paper

An Incremental Learning Approach for Realistic Story Point Prediction in Agile Frameworks

Mostafa Tawfiq[1], Mennatallah Ahmed, Mostafa Hani, Mohamed Alomar, Ahmed Darwish
Computer Department, Faculty of Engineering, Cairo University

***Abstract* — Story point estimation is a critical yet challenging task in agile software development. Traditional machine learning approaches often require retraining models from scratch on an entire dataset to incorporate new information, a process that is computationally expensive and impractical in dynamic project environments. This paper proposes a novel incremental learning approach that addresses this challenge. Our method begins by training a robust baseline model on previous historical projects, which is then efficiently updated as new data from ongoing projects becomes available. We demonstrate the effectiveness of this approach through experiments on both classification and regression tasks. To ensure a fair and rigorous evaluation, we compare our incremental learning model to a full retrain model, which is retrained on all initial project data combined with all previously seen batches from the new project. Our results show that incremental learning achieves performance comparable to that of the full retrain approach but with a significantly lower computational cost, making it a more realistic and scalable solution for enterprises.**

***Index Terms* — Agile software development, incremental learning, machine learning, story point estimation.**

I. Introduction

Agile software development is an approach to building software that emphasizes flexibility, customer collaboration, and delivering work in small, incremental pieces. Unlike traditional methods where everything is planned upfront.

Agile teams work in short cycles, allowing them to adapt to changes and get feedback early and often. Story point estimation is a key activity for planning and forecasting. Accurate estimations are crucial for effective project management, but they are often difficult to achieve due to the inherent uncertainty and complexity of software development tasks. Machine learning has been increasingly applied to automate and improve the accuracy of story point estimation. However, many existing approaches have limitations that hinder their practical adoption in real-world enterprise environments.

A common issue is the reliance on generic, open-source datasets for model training. These datasets are often a random collection of projects from various domains and do not reflect a specific scope of work. While useful for academic research, these datasets often fail to capture the specific nuances and development practices of a particular company. An alternative is to build project-specific models, which train on a part of the project and test on the rest, but this approach is not viable for new projects with no historical data.

This paper addresses these challenges by proposing an incremental learning approach that is both realistic and efficient. The core idea is to train a baseline model on a history of completed projects, preferably from within the same company to ensure higher accuracy. This model is then deployed to provide initial estimates for new projects. As the new project progresses and more data becomes available (e.g., at the end of each sprint), the model is incrementally updated, or "fine-tuned," to adapt to the characteristics of the ongoing project. This process is illustrated in Figure 1.

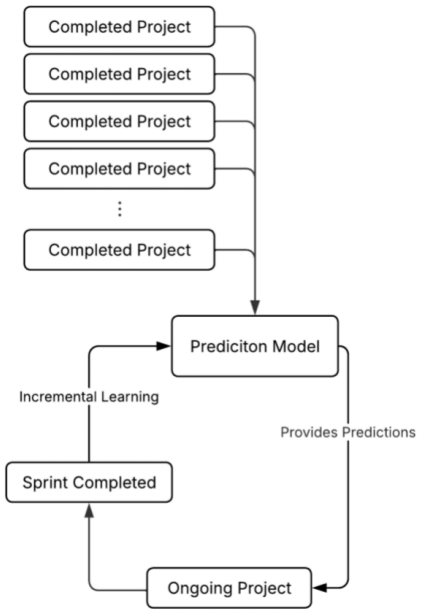


Fig. 1. The proposed incremental learning workflow. Historical data from completed projects is used for initial model training. This model provides

[1] To whom all correspondences should be addressed
mustafa.tawfiq02@eng-st.cu.edu.eg

predictions for the ongoing project. As sprints are completed, the new data is used to fine-tune the model incrementally.

We evaluate our approach using two machine learning models that support incremental updates: PassiveAggressiveClassifier for a classification-based approach to story point prediction, and SGDRegressor for a regression-based approach. We compare their performance and computational cost against two baseline methods: a static model that is never updated and a full retrain model that is retrained on all available data (historical data + data from current project) at each step.

In this paper, we demonstrate that incremental learning offers a superior trade-off between prediction accuracy and computational cost, making it a practical solution for deployment in production environments.

The remainder of this paper is organized as follows: Section 2 discusses *related work*. Section 3 details our *methodology*. Section 4 presents and analyzes *the results* of our experiments. Section 5 discusses *the implications* of our findings, and Section 6 concludes the paper.

II. Literature Survey

The application of machine learning to software engineering problems, including effort estimation, is a well-established field of research. Various techniques, from traditional regression models to more advanced deep learning architectures, have been explored for story point prediction.

A foundational study in this area was conducted by Choetkiertikul et al. [1], who introduced a deep learning model for estimating story points from the textual descriptions of issues. Their model architecture employed Long Short-Term Memory (LSTM) to generate a vector representation of the issue's text, which was then processed through Recurrent Highway Networks (RHWN) before a final regression layer predicted the story point value. Trained and evaluated on a large dataset of over 23,000 issues from 16 open-source projects, this established a strong benchmark in the field, outperforming several other machine learning techniques [2], [3], [4], [5], [6].

Following the advancements in Natural Language Processing (NLP), researchers began exploring the capabilities of Transformer-based models. Fu and Tantithamthavorn [3] introduced GPT2SP, an approach based on the GPT-2 architecture, applying a powerful large language model directly to the task of story point estimation. This marked a shift towards leveraging large, pre-trained models for software engineering tasks.

However, subsequent critical analysis has added important nuance to these findings. Replication studies conducted by Tawosi et al. [6], [7] re-evaluated the performance of GPT2SP and highlighted that its improvements over the Choetkiertikul et al. [1] model were not as pronounced as initially suggested. Their corrected results showed that GPT2SP outperformed Choetkiertikul et al. model in only a minority of cross-project scenarios, and the statistical significance of these improvements was often negligible.

A significant limitation shared by these influential models is that they are inherently static. They are designed to be trained once on a historical dataset and then used for prediction without further updates. This approach fails to address two critical realities of live software projects. First is the problem of "concept drift," where the statistical properties and nature of project tasks evolve, causing a static model's predictions to become less accurate over time. Second is the impracticality of the common alternative: retraining the entire model from scratch whenever new data becomes available. While a full retrain incorporates new information, it is computationally expensive and not a scalable solution for dynamic agile environments where sprints conclude frequently.

Furthermore, much of the existing research evaluates models in a setting that does not fully replicate a realistic use case. Methodologies often test a model on data from the same pool of projects used for training, a scenario different from making initial predictions for a completely new project where no historical data exists. Cross-project estimation remains a significant hurdle for generalization.

The concept of incremental learning, or online learning, is not new, but its application to story point estimation in a realistic enterprise context has not been thoroughly investigated.

In the realm of incremental learning, Pesala et al. [8] proposed two novel algorithms for Support Vector Machines (SVMs) that leverage backward elimination and forward selection of support vectors to enable efficient model updates in dynamic industrial settings. Their approach addresses the limitations of traditional batch SVMs by incrementally updating the model with new data chunks, significantly reducing computational time and memory usage while maintaining or improving classification accuracy compared to batch learning and existing Incremental Learning SVM (ILSVM) methods. Evaluated on 13 diverse datasets, their algorithms demonstrated robustness against concept drift, a challenge also pertinent to agile software development where task characteristics evolve over time. This work complements our proposed incremental learning framework for story point estimation, as it underscores the scalability and efficiency of incremental approaches in handling dynamic data streams. However, while Pesala et al. focus on general classification tasks, our study tailors incremental learning to the specific context of story point prediction, incorporating both classification and regression tasks to address the unique challenges of agile project management.

Our work builds upon the existing literature by not only using a cross-project evaluation methodology but also by incorporating the concept of incremental model updates to handle the stream of new data from ongoing projects.

III. Methodology

Our methodology is designed to simulate a realistic scenario where a company wants to leverage its past project data to estimate story points for a new project. We test our approach on a dataset comprising several software projects. For each experiment, we hold out one project for testing and use the remaining projects to train the initial model.

A. Dataset and Preprocessing

The model was trained and evaluated on a dataset of around 32604 issues from 40 different open-source projects [9]. The raw issue data undergoes a series of preprocessing steps to ensure data quality and prepare it for our machine learning model. These steps are executed in the following order:

1. *Combine Textual Data*: The title and description fields for each user story are concatenated into a single text field. This unified text column serves as the primary input for our model.

2. *Filter by Story Points (SP):* To standardize the effort estimation, we filter the dataset to include only issues with Fibonacci story point values of 1, 2, 3, 5, and 8. Issues with non-Fibonacci values (e.g., 4, 6, 7) or values greater than 8 are removed from the dataset.

3. *Resolve Inconsistencies:* The dataset is cleaned by identifying and excluding user stories that are textually similar but have conflicting story point assignments. This step ensures that the model is trained in consistent examples.

4. *Text Normalization:* All punctuation is removed from the consolidated text field. This helps to reduce the dimensionality of the data and focus on the meaningful words.

5. *Token Limitation:* The input text for each issue is truncated to the first 500 tokens. This prevents excessively long descriptions from disproportionately influencing the model and helps manage computational resources.

6. *Encode Story Point Values:* The filtered Fibonacci story point values are mapped to a zero-indexed numerical format for compatibility with machine learning classification algorithms. The mapping is as follows: {1→0}, {2→1}, {3→2}, {5→3}, and {8→4}.

Following these preprocessing stages, the cleaned and structured text data is then converted into numerical features using the *Term Frequency-Inverse Document Frequency (TF-IDF)* vectorizer. TF-IDF is a standard technique in natural language processing that reflects the importance of a word in a document relative to a collection of documents.

B. Experimental Setup

We frame the story point prediction problem as both a *classification* and a *regression* task. In classification the story points are treated as discrete classes. We use the *PassiveAggressiveClassifier* as our incremental model. In regression the story points are treated as continuous values. We use the *SGDRegressor* as our incremental model.

For each task, we compare the performance of our incremental model against two baselines:

1. *Static Model*: A model trained on the initial set of historical projects and never updated. This represents the naive approach of using a fixed, pre-trained model.
2. *Full Retrain Model*: A model that is retrained from scratch at the end of each batch, using all the historical data plus all the new data from the completed batches to ensure fair comparison.

For classification, we use a *Support Vector Machine (SVM)* with a linear kernel. For regression, we use *Support Vector Regressor (SVR) with RBF kernel with C=1 and gamma=scale.* This baseline represents the most accurate but also the most computationally expensive approach.

C. Cross Validation and Hyperparameter Tuning

To select the best hyperparameters for our models, we use *RandomizedSearchCV*, which is often faster than an exhaustive grid search and provides good coverage of the parameter space. A key aspect of our methodology is the use of *GroupKFold* for cross-validation. This technique ensures that all data from a particular project is assigned to either the training or the validation set within each fold, but never split across both. This prevents data leakage and provides a more realistic estimate of how the model will perform on a completely new project.

The hyperparameters tuned for each model used:

- *TF-IDF vectorizer*: max_features, ngram_range, min_df and max_df
- *PassiveAggressiveClassifier:* C, max_iter, class_weight and loss function
- *SGDRegressor*: alpha, max_iter, learning_rate, eta and regressor loss

D. Incremental Learning Process

The test project is processed in batches, simulating the sprints in an agile project. For each batch, we perform the following steps:

1. *Training*
   - The *incremental model* is updated using only the data from the current batch. This is a fast process as it doesn't require access to the entire historical dataset.
   - The *static model* is not updated.

- The *full retrain model* is retrained using all the historical data plus all the data from the batches processed so far.

2. *Evaluation*: The current models (incremental, static, and full retrain) are used to make predictions on the current batch of data. Their performance is evaluated and recorded.

This process is repeated for all batches in the test project.

E. Model Choice

The *PassiveAggressiveClassifier* was chosen for the classification task because it is an online learning algorithm that is well-suited for incremental updates. It belongs to the family of margin-based classifiers, similar to SVMs. Its "passive-aggressive" nature makes it efficient: it remains "passive" if a data point is correctly classified with a sufficient margin, and it becomes "aggressive" by updating its weights when it encounters a misclassification.

For the regression task, the *SGDRegressor* is a natural choice as it also supports incremental learning through the *partial_fit* method. It is a linear model that optimizes the loss function using *Stochastic Gradient Descent*, making it highly efficient and scalable.

IV. Results and Analysis

TABLE I
CLASSIFICATION RESULTS FOR TIMOB PROJECT

Batch	Incremental Accuracy	Static Accuracy	Full Retrain Accuracy	Inc. Time(s)	SVM Time (s)
1	0.2385	0.2176	0.2929	0.0246	605
2	0.2762	0.1632	0.2678	0.0240	609
3	0.2887	0.2343	0.3222	0.0273	621
4	0.2803	0.1883	0.2845	0.0222	622
5	0.3264	0.1841	0.3264	0.0280	629
6	0.2887	0.1715	0.3347	0.0239	653
7	0.3556	0.2008	0.3431	0.0221	665
8	0.3305	0.2176	0.2971	0.0221	668
9	0.2636	0.2218	0.3305	0.0244	701
10	0.3264	0.2176	0.3640	0.0245	710

We present the results of our experiments for both the classification and regression tasks. The experiments were run on several projects from our dataset, including *TIMOB*, *TISTUD*, *MULE*, and *XD*.

A. Classification Results

The table below shows a sample of the results for the *TIMOB* project, comparing the accuracy and training time of the incremental, static, and full retrain models.

- *Accuracy*: The incremental model's accuracy is often close to that of the full retrain model and consistently and significantly outperforms the static model.

- *Training Time*: The most striking result is the difference in training time. The incremental model updates in a fraction of a second, while the full retrain model takes several minutes for each batch. In real-world scenarios, datasets are often massive, making the full retrain approach a significant scalability problem. Attempting to retrain a model from scratch on such large volumes of data *after each development sprint* takes a prohibitive amount of time. This lengthy process makes the full retraining strategy impractical for real-world applications that require frequent and timely updates.

- *Concept Drift*: The static model's performance is erratic and often degrades over time. This is a clear sign of *concept drift*, where the statistical properties of the data change over the course of the project. The incremental model effectively handles concept drift by adapting to these changes.

TABLE II
REGRESSION RESULTS FOR TIMOB PROJECT

Batch	Incremental MAE	Full Retrain MAE	Inc. Time (s)	SVM Time (s)
1	1.8185	1.9070	0.0557	923.3
2	1.8359	1.9705	0.0578	951.0
3	1.8641	1.9698	0.0553	960.6
4	1.8407	1.9389	0.0519	974.7
5	1.8027	1.9251	0.0589	986.1
6	1.6117	1.7159	0.0602	1007.5
7	1.7246	1.8750	0.0590	1031.2
8	1.7813	1.8470	0.0521	1037.5
9	1.7294	1.7123	0.0486	1071.7
10	1.7832	1.8856	0.0563	1067.8

B. Regression Results

The table below shows the results for the regression task on the *TIMOB* project, comparing the Mean Absolute Error (MAE), and training time.

The results from the regression task mirror those from the classification task. The incremental SGDRegressor achieves a lower average MAE (better performance) score compared to the full retrain SVR model. The training time for the incremental model is negligible compared to the SVR model, which takes hours to retrain over the course of the project.

V. Discussion

The results of our experiments strongly support the case for using an incremental learning approach for story prediction due to its lower complexity resulting in hundreds of folds of faster execution which is highly needed in a realistic enterprise setting.

A. The Case for Cross-Project Learning

A key aspect of our approach is the leveraging of historical data from multiple projects. This is a good choice over project-specific models. A new project, by definition, has no data, so a project-specific model cannot be used for initial estimates. By training on a diverse set of past projects, we create a more generalized model that can provide reasonable estimates from day one of a new project. The incremental learning then allows this model to specialize in the specific characteristics of the new project as it progresses.

B. Advantages of Incremental Learning

1. *Near-Optimal Performance at a Fraction of the Cost*: While a full retrain model can sometimes achieve slightly higher accuracy, the performance gain is marginal and comes at a massive computational cost. Our results show that the incremental model can achieve comparable, and in some cases better, performance than the full retrain model, while being orders of magnitude faster.

2. *Handles Concept Drift*: Software projects are dynamic, and the nature of the work can change over time. The static model's poor performance demonstrates its inability to cope with this "concept drift." Incremental learning, by its very nature, is designed to adapt to such changes, ensuring that the model remains relevant throughout the project's lifecycle.

3. *Scalability and Practicality*: In a production environment, computational resources are often a constraint. The full retrain approach, with its high computational demands, is not a scalable solution. As the amount of historical data grows, the retraining time will become even more prohibitive. The incremental approach, on the other hand, has a constant processing time per sample, making it highly scalable and practical for real-world deployment. It is also more memory efficient as it does not require the entire dataset to be held in memory for updates.

C. Limitation and Future Work

Our study has some limitations. The experiments were conducted on a specific dataset, and the results may vary for other datasets or organizations. Future work could involve testing our approach on a wider range of datasets and exploring the use of more complex incremental learning models, such as neural networks. Another avenue for research would be to investigate more sophisticated strategies for deciding when and how to update the model, rather than updating it after every batch.

VI. Conclusion

This paper has presented a novel incremental learning approach for story point estimation that is both realistic and computationally efficient. By training a baseline model on historical data and then incrementally updating it with new data from ongoing projects, our approach overcomes the limitations of traditional methods that rely on generic datasets or are too computationally expensive for practical use.

Our experiments, covering both classification and regression tasks, demonstrate that incremental learning delivers performance that is comparable to, and in some cases better than, a full retrain approach, but with a training time that is over 99% faster. This makes it a highly scalable and practical solution for enterprises looking to improve the accuracy and efficiency of their story point estimation process. We believe that this approach represents a significant step forward in the practical application of machine learning to software project management.

References

[1] *M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A Deep Learning Model for Estimating Story Points," IEEE Trans. Softw. Eng., vol. 45, no. 7, pp.637–656, Jul. 2019, doi:10.1109/TSE.2018.2792473*

[2] *M. Gultekin and O. Kalipsiz, "Story Point-Based Effort Estimation Model with Machine Learning Techniques," Int. J. Softw. Eng.Knowl. Eng., vol. 30, no. 01, pp. 43–66, Jan.2020, doi: 10.1142/S0218194020500035*

[3] *M. Fu and C. Tantithamthavorn, "GPT2SP: A Transformer-Based Agile Story Point Estimation Approach," IEEE Trans. SoftwEng., vol. 49, no. 2, pp. 611–625, Mar. 2022, doi: 10.1109/TSE.2022.3158252.*

[4] *V. Tawosi, A. Al-Subaihin, and F. Sarro, "Investigating the Effectiveness of Clustering for Story Point Estimation," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2022, pp. 827–838. doi:10.1109/SANER53432.2022.00101.*

[5] *B.Marapelli, A. Carie, and S. M. N. Islam, "RNN-CNN MODEL:A Bi-directional Long Short-Term Memory Deep Learning Network For Story Point Estimation," in 2020 5th International Conference on Innovative Technologies in Intelligent Systems and Industrial Applications (CITISIA), Sydney, Australia: IEEE, Nov. 2020, pp. 1–7. doi: 10.1109/CITISIA50690.2020.9371770.*

[6] *V. Tawosi, R. Moussa, and F. Sarro, "Agile Effort Estimation: Have We Solved the Problem Yet? Insights From a Second Replication Study (GPT2SP Replication Report)," Sep. 01, 2022, arXiv: arXiv:2209.00437. doi:10.48550/arXiv.2209.00437.*

[7] *V. Tawosi, R. Moussa, and F. Sarro, "Agile Effort Estimation: Have We Solved the Problem Yet? Insights From a Replication Study," IEEE Trans. Softw. Eng., vol. 49, no.4, pp. 2677–2697, Dec. 2022, doi:10.1109/TSE.2022.3228739.*

[8] *V. Pesala, A. K. Kalakanti, T. Paul, K. Ueno, A. Kesarwani and H. G. S. P. Bugata, "Incremental Learning of SVM Using Backward Elimination and Forward Selection of Support Vectors," 2019 International Conference on Applied Machine Learning (ICAML), Bhubaneswar, India, 2019, pp. 9-14, doi: 10.1109/ICAML48257.2019.00010.*

[9] *V. Tawosi, A. Al-Subaihin, R. Moussa, and F. Sarro. "A Versatile Dataset of Agile Open-Source Software Projects." https://arxiv.org/abs/2202.00979, 2022.*