Table of Contents

Project Overview

Project Title:

AgileAI: Cognitive Load-Aware Project Management

Project Overview:

- **Objective:**
  To develop an AI driven framework which aids in project management for software development based on Agile-scrum model practices, automating project breakdown, parts of sprint planning and user story allocation based on team availability, backlog priority while considering user fatigue, thereby enhancing efficiency, accuracy and adaptability in Agile project management.

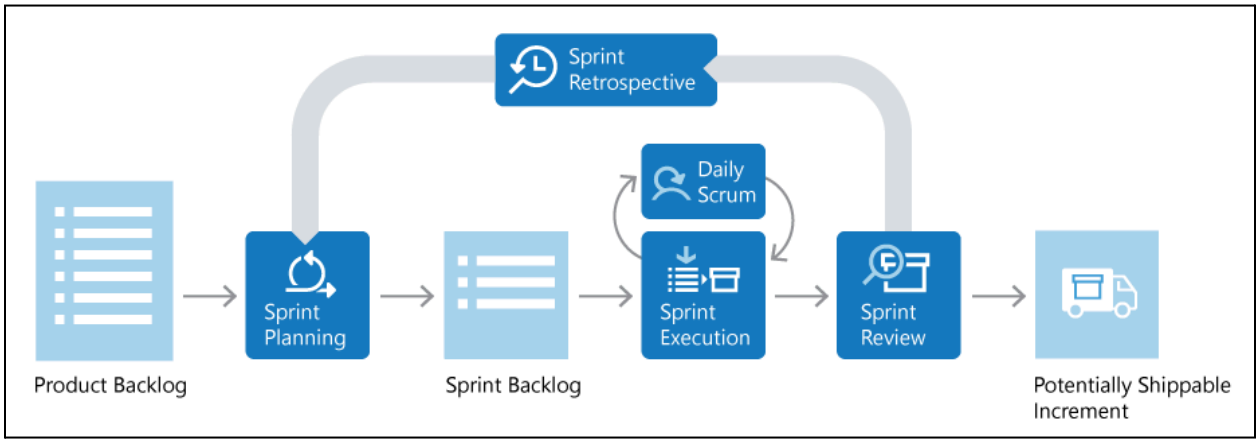
  Considering the limited timeframe for this semester, I aim to focus on building a system which would break down the project description into the Epics, Features and User Story format. For this, I would define a standard template which would have all the common

fields that are considered in project description. This would be the input to the model for it to understand the scope and define the user stories. Additionally, I also want to work on a basic model of dynamic story pointing, based on the limited data available in open source platforms.

- **Background:**
  For software development, most companies implement **Agile methodology** which is a defined approach of project management which ensures structure, flexibility and streamlined progress during software development. Azure DevOps defines *Agile as a term that describes approaches to software development that emphasize incremental delivery, team collaboration, continual planning, and continual learning.*

  Azure further defines **Scrum** as the *framework used by teams to manage work and solve problems collaboratively in short cycles. Scrum implements the principles of Agile as a concrete set of artifacts, practices, and roles.*

  Following are the steps in Agile Scrum Software development lifecycle:



Source: https://learn.microsoft.com/en-us/devops/plan/what-is-scrum

- **Scope:**
  To create a framework which aids in scrum ceremonies in following ways:
  - Analyze the project documentation to automate its representation into Epics, Features and User Stories with proper descriptions
  - Dynamically analyze each team members' based on their skillsets, expertise, complexity patterns in previous work, past performance based on velocity, completion rates etc to allocate them the available user stories and suggest adaptive story point estimation based on each individual's capacity instead of static poker estimates.

- Dynamic sprint planning which considers team availability based on factors like holidays, planned leaves, buffer etc to allocate stories to the sprint based on the current priority. It should also account for resource fatigue ie. if someone is consistently overworked or has spillovers, the system should reduce allocation in the upcoming sprint plannings.

For this project, I would be considering the structure of Azure DevOps as the agile framework. There are several other frameworks which follow agile such as JIRA, Rally etc, but we would be focussing only on Azure DevOps.

As mentioned in the objective, for this semester, I would be restricting the scope to the first two points.

- Epics/Feature/Story generation: The main focus for this would be the input data templates. Considering multiple project definitions available online, I want to focus on creating a standard template which would have all the basic headings which are present in all projects. This template would be the input to the LLM model, which would understand the context and thereby break the project into long term and short term goals represented as epics, features and user stories.
- Dynamic Story Pointing: Although there is limited open source data available about sprints and story points, we can still create a basic regression model for story point assignments.

- **AI Techniques and Tools:**
  This project can be divided into 3 major models:

  - Epic/Feature/Story generation: LLM model like GPT for natural language understanding and generation
  - Dynamic Story Pointing: Supervised Regression models trained on historic data to predict the story pointing, reinforcement learning to take in the feedback actual vs estimated. For this semester, I would only be focussing on the regression model because of limited availability of dataset.
  - Dynamic Sprint Planning: Optimization algorithms, reinforcement learning model. This would be a future enhancement.

Project Timeline:

- **Milestones:**
  This project can be divided into the following milestone:
    - Project initialization and research: Finding best fitting models and datasets, design the framework.
    - Project decomposition feature: Model that generates Epics/Features/User Stories based on project description. This feature is the main target for this project. It can be broken further in the following milestones:
        - Gather project description dataset. Identify important headings common in these descriptions.
        - Define a standard template format which would be the input to our LLM
        - LLM prompt engineering, isolated instance for data privacy, testing if the model is able to create a proper breakdown
        - Integrating Azure APIs to create the Epics/Features/User Stories and their hierarchy in backend
        - Create a basic frontend for input of project description
        - Integrating frontend, model and backend api calls.
    - Dynamic story pointing feature: Collect data from past sprints, train ML model for story point estimation based on user story text and past velocity. For this I plan to use https://data.mendeley.com/datasets/skk2wn9j86/1 dataset.

Stakeholders:

- **Project Team:** Shrimayee Umesh Deshpande
- **End Users:** Agile Practitioners like Project Managers, SDE, Business Analysts, Engineering Managers
- **Other Stakeholders:** Startups, Scrum Masters, Agile Coaches, Leadership team can also be the stakeholders to this project.
- **Enterprise Software Providers:** This feature can be integrated in platforms like Azure DevOps (Microsoft), JIRA(Atlassian) as a future scope

Computing Infrastructure

1. Project Needs Assessment

The AI system will aid in the scrum workflow by understanding the structured contents from the project description input text file and represent the project in the scrum structure which is widely used by the industry using a LLM. The system will produce a hierarchy of Epics → Features → User Stories, ensuring:

- Accurate representation of the project requirements.
- Properly defined parent-child links between Epics, Features, and User Stories.
- Each generated work item includes a detailed description and suggested acceptance criteria.

The system is designed to integrate with Azure DevOps (ADO) via its API for direct creation of work items and will eventually be deployable as a web-based application or cloud service, providing teams with an interactive, scalable, and efficient backlog generation tool.

- **Performance Requirements:**
  - **Latency**: Since I plan to use open-source models, the expected inference latency ranges between 3–7 seconds, while interactions with the ADO APIs can add approximately 30 seconds depending on the number of links generated. Since a project typically consists of multiple features and user stories, the number of API calls scales with project size, thereby contributing further to the overall delay. As a result, the end-to-end latency is estimated to be around 10 seconds or more, with larger projects experiencing higher latency. (Referred from KPIs for gen AI)
  - **Throughput**: Given the constraints of open source model, the system's throughput is expected to be around 5 projects per minute in a single threaded setup, primarily constrained by ADO API latency rather than model inference.
  - **Accuracy:** This will be defined by how well the model preserves the Epic → Feature → User Story hierarchy, ensures correct parent–child links, and produces clear, complete descriptions and acceptance criteria. We target ~95% structural accuracy and ~80% content adequacy, combining automated checks with human evaluation for validation.
- **Limitations and tradeoff**: Primary constraint is the use of ADO apis which can cause a latency of up to 30 seconds, which may affect the throughput. Hence, larger the project, higher the latency, smaller the throughput if we focus on accuracy.

2. **Hardware Requirements Planning**

   The project will leverage a combination of local and cloud hardware for development and model execution.

   - **MacBook Air (M4 chip, 24GB unified memory)** will be used for code development, prompt engineering, input preprocessing, and small-scale inference tests.This setup is sufficient for lightweight models and tasks like hierarchy validation and JSON formatting.
   - **Google Colab** will provide access to GPUs such as **T4/P100/A100**, which would enable running larger models, performing lightweight fine-tuning (LoRA/PEFT), and handling medium-sized project descriptions efficiently.
   - **Evidence:**
     - Apple M4 chip supports ML workloads up to ~7B parameters in 4–8-bit quantized form.
     - Colab GPUs (T4/P100/A100) are benchmarked to provide 2–3× faster inference for medium-sized LLMs than local hardware.
   - **Tradeoff:** Prioritizing accessibility and cost-efficiency while enabling the ability to experiment with larger models and slightly faster inference via cloud GPUs.
   - **Risk/Trigger:**
     - GPU availability limits in Colab may restrict inference speed or fine-tuning capability.
     - If local memory proves insufficient for intermediate processing, cloud execution will be used as fallback.

3. **Software Environment Planning**

   - **OS:** macOS Ventura (local development), Linux (Colab backend)
   - **Frameworks/Libraries:** Python 3.10+, PyTorch, Hugging Face Transformers, PEFT/LoRA
   - **Frontend:** Streamlit
   - **Containers:** Docker for portability and scaling
   - **Integration:** Azure DevOps via ADO APIs, GPT LLM for backlog generation

   Options Considered:

   - Frameworks: PyTorch vs TensorFlow: chose PyTorch for Hugging Face compatibility

Evidence:

- Hugging Face and PyTorch guides confirm compatibility with macOS and Linux, including 4–8 bit quantized LLM models.
- Streamlit documentation supports rapid Python-based UI development.

Tradeoff Statement:

- Gains: Easy development, LLaMA/GPT integration, streamlined UI, and scalable deployment via Docker.
- Harder: Full-scale deployment and advanced scaling may require additional orchestration (Kubernetes).

Risk/Trigger:

- Version mismatches between libraries (Transformers, PyTorch) could break model loading.
- Deprecation of packages or changes in Colab/Streamlit environments may require stack updates.

4. **Cloud Resources Planning**

Decision:

- **Compute & Services:** Google Colab for GPU-based model inference and optional fine-tuning.
- **Storage & Scaling:** Google Drive for persistent storage of project inputs, generated backlog files, and temporary outputs; Docker containers for scalable backend/frontend deployment.
- **Frontend Hosting:** Google Cloud Run to deploy the Dockerized Streamlit app, providing automatic scaling and easy integration with the backend.

Options Considered:

- Cloud providers: AWS SageMaker vs Azure ML vs Google Colab: I chose Colab for free GPU access, Python-native workflow, and ease of setup. AWS and AML are paid services.
- Frontend hosting: Google Cloud Run vs Hugging Face Spaces: I chose Cloud Run for seamless Docker deployment and Google ecosystem integration.

Evidence:

- Colab documentation confirms GPU availability (T4/P100/A100) and compatibility with PyTorch/Hugging Face.
- Google Drive integrates seamlessly with Colab for persistent storage.
- Cloud Run supports containerized Python apps and scales automatically based on traffic.

Tradeoff Statement:

- Gains: Free GPU resources, persistent storage, simple deployment, and scalable Docker containers.
- Harder: Colab session limits (~12 hours) and occasional GPU unavailability; free tier of Cloud Run has limited concurrent requests.
- Alternatively, I could also use Azure ML and Azure Kubernetes Services for this application since our end goal is to integrate with Azure DevOps, the only reason why I didn't select this was because AML is a paid service.

Risk/Trigger:

- GPU unavailability, session timeouts, or storage limits could require switching to paid tiers or alternative cloud providers.
- High traffic or large-scale usage may necessitate scaling to more robust Google Cloud services or managed ML platforms.

5. **Scalability, and Performance Planning**

- **Decision:** Use Docker containers on Google Cloud with auto-scaling. Apply quantization and pruning for faster inference. Monitor latency, accuracy, and resource use via Cloud Monitoring.
- **Options considered:** Horizontal vs. vertical scaling; optimized vs. unoptimized model.
- **Evidence:** Quantization/pruning cut latency and model size with minimal accuracy loss (Han et al., 2016). Google Cloud supports auto-scaling and monitoring best practices.
- **Tradeoff:** Prioritize responsiveness and scalability over slight accuracy drop.
- **Risk/trigger:** Reassess if inference >150ms or cloud costs exceed budget by 20%.

Security, Privacy, and Ethics (Trustworthiness)

1. Problem Definition

Goal: Define the ethical and societal impacts of an AI system that converts project descriptions into epics, features, and user stories, ensuring outputs are accurate, transparent, and bias-free.

- **Stakeholder Involvement:** Collect feedback from project managers, developers, and Scrum masters on usability and trust.
- **Ethical Impact Assessments:** Identify risks such as incorrect hierarchies, vague acceptance criteria, or biased task structures.
- **Risk Analysis Frameworks:** Prioritize transparency (explainable outputs), accountability (ownership of errors), and security (safeguarding project descriptions).

Tool/Approach: *Ethical Risk Checklist* – a lightweight internal checklist to evaluate risks around bias, explainability, and data misuse during development.

2. Data Collection

Goal: Ensure project description inputs used for training and fine-tuning the LLM are representative, unbiased, and securely handled.

- **Data Augmentation:** Use synthetic variations of project descriptions to capture different writing styles (formal, informal, concise, detailed).
- **Data Anonymization:** Remove or mask sensitive details (e.g., client names, internal project codes) to preserve privacy.
- **Bias Detection and Correction:** Check for overrepresentation of certain industries or domains (e.g., IT vs. healthcare) and balance accordingly.

Tool/Approach: *Snorkel* – for generating synthetic project descriptions that simulate varied Scrum workflows and reduce bias from limited training samples.

3. AI Model Development

Goal: Develop fair, interpretable, and robust AI models that reliably generate epics, features, and user stories from project descriptions.

- **Algorithmic Fairness:** Apply reweighting or debiasing techniques during fine-tuning to prevent the model from overrepresenting certain project types or industries.

- **Explainability Tools:** Integrate tools to provide insights into model outputs, ensuring generated stories and features are interpretable and trustworthy.
- **Robustness and Stress Testing:** Test the model with unusual, complex, or adversarial project descriptions to verify consistent performance.

Tool/Approach: *SHAP (SHapley Additive exPlanations)* – to understand which parts of the input description influence specific epics, features, or story generation, improving transparency and trustworthiness.

4. AI Deployment

Goal: Deploy the AI model securely, ensuring reliable performance and accountability in real-world use.

- **Secure Model Serving:** Protect model APIs to prevent unauthorized access and ensure data privacy.
- **CI/CD Pipelines:** Automate deployment, updates, and rollbacks to maintain consistency and minimize downtime.
- **Feedback Loops:** Enable mechanisms for user feedback and anomaly reporting to detect unexpected or incorrect outputs.

Tool/Approach: *BentoML* – for serving the LLM model with secure APIs, scalable deployment, and integrated monitoring, allowing robust feedback handling and accountability.

5. Monitoring and Maintenance

Goal: Continuously monitor the LLM-based backlog generator to ensure accurate generation of epics, features, and user stories, detect biases in project outputs, and address drift to maintain reliability and trustworthiness.

- **Performance and Drift Monitoring:** Track latency, accuracy, and changes in input project description patterns to detect model drift.
- **Bias Detection:** Regularly evaluate generated epics, features, and user stories for fairness across different project types or domains.
- **Retraining Pipelines:** Automate retraining and updates when drift or bias is detected to keep the model relevant and accurate.

Tool/Approach: *NannyML* – for detecting data and concept drift in the LLM model, enabling proactive retraining and ensuring consistent performance and fairness.

Human-Computer Interaction (HCI)

The goal of this section is to ensure that the AI backlog generator aligns with user needs, is intuitive, and supports effective decision-making in real-world workflows. Human-Computer Interaction (HCI) principles guide the design of interfaces, the presentation of AI outputs, and the mechanisms for user feedback, ultimately improving usability, trust, and adoption of the system.

The following questions were designed to gather insights on system usefulness, potential risks, desired features, performance expectations, and maintenance considerations:

1. Do you think a system that automatically generates epics, features, and user stories from project descriptions would be useful in your workflow?
2. How much manual verification would you anticipate needing before using the AI outputs?
3. Would you trust the AI's hierarchy linking between epics, features, and stories?
4. What risks do you see with using an AI-generated backlog (e.g., incorrect priorities, misleading story definitions, bias toward certain project types)?
5. Are there scenarios where this system could cause more confusion than benefit?
6. What features would make this system more practical for real teams (e.g., tagging, sprint planning suggestions, dashboards)?
7. Would explanations of why the AI made certain decisions (explainability) be helpful?
8. What interface elements or visualization methods would help you interpret generated stories quickly?
9. How fast would you expect the AI system to generate a backlog for a medium-sized project?
10. What is your expectation for system response time (latency) when generating stories?
11. What mechanisms would you like to see for reporting errors or providing feedback on AI outputs?
12. How frequently should the model be updated or retrained to reflect evolving project practices?

Interview 1 Context:
I conducted an unstructured interview with **Alex Karpman**, who works as a director at BlackRock as Global Lead of Implementation Services team, to gather qualitative insights. He is the lead of a team which has 3 products, each product has multiple projects. The discussion focused on practical use, potential risks, HCI considerations, and technical expectations for AgileAI.

1. Yes, very useful, will save a lot of manual work. Considering we have previous sprint data, it would have a good reference.

2. As long as the previous sprints are representative of actual work done and somewhat consistent, we should be good. But that's the challenge here, since there are several things which might not be represented in the sprint but are done because they are emergencies

3. This links to how well the previous stories are defined because that's what the model will learn on. Challenge identified here:

- how do you generalize it across all teams? because every team has their own way of describing a task, how detailed they want the stories, how broken down the stories should be etc.. how will the model capture that?

4. All of these are risks to be considered, and are heavily dependent on model accuracy, should be well prompt engineered to avoid the LLM from hallucination

5. If it starts generating out of scope tickets, or biased/repetitive tasks, it'll just double the work of scrum master rather than helping them.

6. Ticket generation should be interactive. Take feedback before generating final tickets. This way there is more transparency.

7. Yes

8. They will be finally visualized on the ADO team dashboard, but before that also we should have some interface which will show us the rough structure because cleanup in the dashboard is difficult and if the model fails, bulk cleaning will be even more difficult.

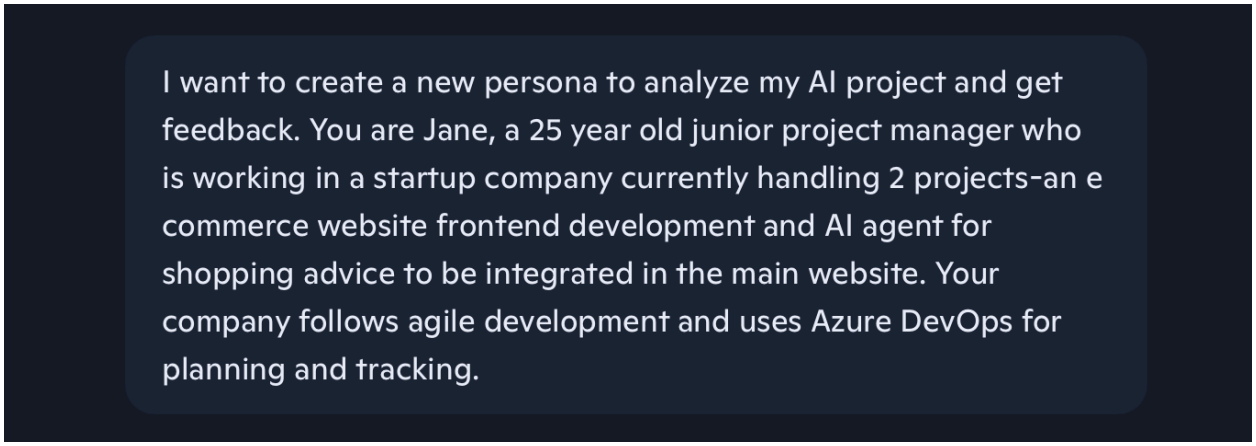
9. Ticket creation is done over months, it's not necessarily a speed dependent activity, so even if it takes a few minutes, it's still fine. It doesn't have to be in seconds.

10. It doesn't have to be quick, it can take several minutes.

11. There should be a feedback loop in place to validate if the generated stories are accurate.

12. Ideally, every new project can incorporate the feedbacks of the previous projects, but training and retraining can be done per team basis.

Interview 2 Context:
In addition to human feedback, we conducted an interview with an AI persona, Jane, using Microsoft Copilot. The goal was to gather insights from an AI-assisted perspective on potential system risks, usability, and feature suggestions.



Interview details: https://copilot.microsoft.com/shares/UrnZ9jo3MAX2qAk18oaCy

Key Insights from Combined Interviews

1. The AI backlog generator is highly useful, reducing manual effort while leveraging previous project data.
2. Trust and accuracy depend on data quality, team-specific conventions, and careful prompt engineering.
3. Main risks include LLM hallucination, out-of-scope tasks, bias, and inconsistent outputs across teams.
4. Enhancements suggested: interactive ticket generation, preliminary hierarchy visualization, and explainability features.
5. Continuous learning through feedback loops and team-specific retraining is essential to maintain reliability and relevance.

Github link: https://github.com/Shrimayee30/AgileAI/tree/main

---

Planned HCI Enhancement and Evaluation Metrics

To enhance usability and ensure transparency, the system will include a verification step before final ticket generation. After the AI produces the Epic → Feature → User Story hierarchy, users will be prompted to review and mark it as “Satisfied” or “Unsatisfied” before proceeding. This step allows users to validate the output, reducing confusion and preventing incorrect or redundant tickets on the ADO dashboard.

Usability Metrics:

- User Satisfaction Rate: Percentage of users marking the generated hierarchy as "Satisfied" on the first attempt.
- Task Completion Time: Average time taken from generation to confirmation, indicating the efficiency of the user interaction flow.

Flow Diagram

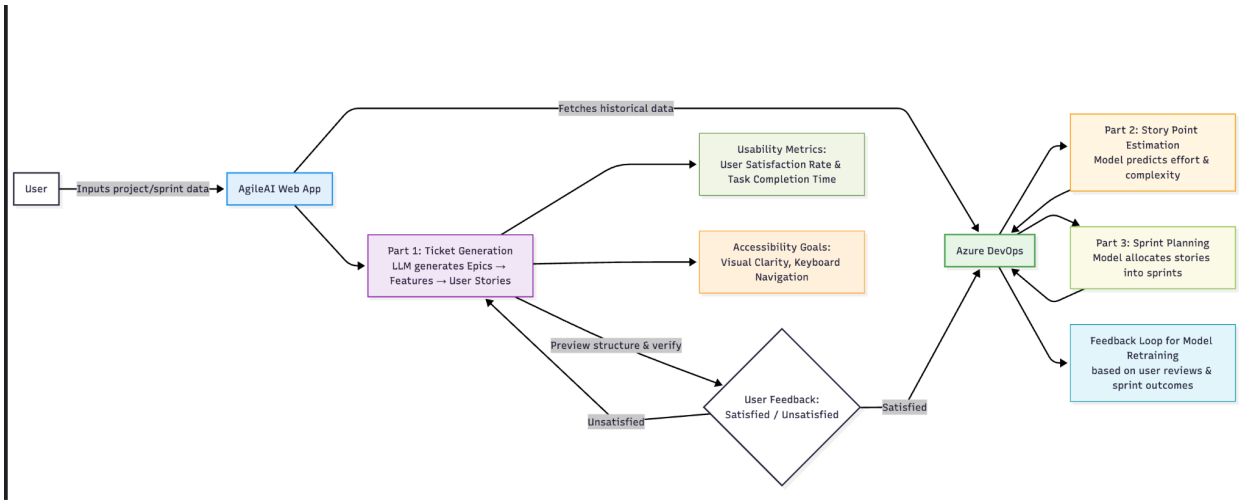


Fig 2: AgileAI flow diagram: https://www.mermaidchart.com/d/7ff04d39-573c-4e67-bfbf-bfb3d1267e94

Usability Testing Plan

Objective: Assess AgileAI's usability by comparing AI-assisted project ticket generation with manual creation.

Participants: 5-7 members from one Agile team (Scrum Master, Product Owner, and Developers).

Procedure:

- Use a new project description to create tickets manually and via AgileAI.
- Measure task completion time and collect satisfaction ratings after each run.
- Observe workflow clarity, ease of use, and accuracy of generated outputs.

Metrics:

- User Satisfaction Score (1–5 scale)
- Task Completion Time (manual vs. AI-assisted)
- Error Rate: Number of incorrect or redundant tickets generated

Analysis:
Use Affinity Diagramming to group qualitative feedback (ease, transparency, accuracy) and identify key pain points and improvements for future iterations.

Risk Management Strategy

1. Problem Definition
In AgileAI, an unclear problem definition could result in generating irrelevant or low-quality user stories, leading to planning inefficiencies and mistrust among Scrum teams. The goal is to ensure that the AI-generated stories, points, and sprints align closely with how teams actually plan and execute work.

Key Risks:

- Misalignment of generated backlog with actual project priorities
- Bias from team-specific sprint data affecting generalizability
- LLM hallucinations leading to false or redundant tasks

Mitigation Strategies:

- Conduct early stakeholder interviews and establish feedback loops for alignment
- Use iterative prototyping and structured requirement mapping (e.g., Lucidchart) to validate outputs

2. Data Collection

Data quality and representativeness are critical for accurate AI-generated backlogs. Poor preprocessing or biased inputs could lead to irrelevant or inconsistent stories.

Key Risks:

- **Data Quality:** Noisy or incomplete text from PDF project documents
- **Bias in Data:** Overrepresentation of certain project types or team styles
  Data Privacy: Risk of mishandling sensitive project information

Mitigation Strategies:

- **Non-technical:** Define the project description template with relevant fields only
- **Technical:** Normalize text for prompt consistency (consistent casing, module separation, clean sentences) and apply automated data cleaning/augmentation techniques

3. AI Model Development

During model development, risks like bias, overfitting, or lack of transparency can reduce the reliability and fairness of AI-generated backlog items.

Key Risks:

- **Algorithmic Bias:** Model may favor patterns from certain project types or team styles
- **Overfitting/Underfitting:** Poor generalization to new projects
- **Explainability:** LLM outputs may be hard for stakeholders to interpret

Mitigation Strategies:

- **Non-technical:** Provide stakeholder visibility through interactive ticket review before final generation
- **Technical:** Apply prompt engineering, cross-validation, and hyperparameter tuning; use explainability tools (e.g., SHAP/LIME) to clarify model decisions

4. AI Deployment

Deploying AgileAI in real-world environments may introduce risks like integration challenges or security vulnerabilities, affecting system reliability and trust.

Key Risks:

- **Integration Issues:** AI-generated stories may not sync properly with Azure DevOps
- **Security Breaches:** Potential exposure of sensitive project data during deployment

Mitigation Strategies:

- **Non-technical:** Monitor performance post-deployment and gather user feedback for smooth integration
- **Technical:** Use Docker containerization and CI/CD pipelines to ensure secure, consistent, and maintainable deployments

5. Monitoring and Maintenance

After deployment, AgileAI must be continuously monitored to maintain accuracy, fairness, and reliability as project data and workflows evolve.

Key Risks:

- **Model Drift:** AI-generated stories may lose relevance over time
- **Emerging Security Threats:** New vulnerabilities or unauthorized access

Mitigation Strategies:

- **Non-technical:** Regularly review system outputs and gather user feedback
- **Technical:** Implement drift detection and automated retraining pipelines; perform periodic security audits

6. Residual Risk Assessment

Risk	Likelihood	Impact	Residual Risk Level	Action
Misalignment of generated backlog	Medium	High	High	Continuous stakeholder feedback, iterative validation
Bias from team-specific sprint data	Medium	Medium	Medium	Prompt engineering, cross-team training, manual review
LLM hallucinations / false tasks	Low	High	Medium	Interactive review before final generation
Integration issues with Azure DevOps	Low	Medium	Low	Test in staging environment, monitor outputs

Security breaches	Low	High	Medium	CI/CD pipeline security, periodic audits
Model drift over time	Medium	Medium	Medium	Automated monitoring, scheduled retraining

---

Data Collection Management and Report

1. Data Type

- **Part 1 (Epic-Feature-Story Generation):**
  - **Type:** Unstructured data – project description PDFs
  - **Challenge:** Extracting relevant sections and normalizing text for consistent prompts

- **Part 2 (Story Point Estimation):**
  - **Type:** Structured data – historical sprint data, task durations, team performance metrics
  - **Challenge:** Ensuring completeness and accuracy across multiple sprints

- **Part 3 (Sprint Planning):**
  - **Type:** Structured or semi-structured data – backlog items, resource availability, and task dependencies from various sources
  - **Challenge:** Aggregating data from multiple systems with varying formats

- **Data Granularity:** Mix of raw (PDF text, raw sprint logs) and processed (normalized prompts, cleaned tables) data

2. Data Collection Methods

Part 1 – Epic/Feature/Story Generation

- **Source of Data:**
  Open-source project description documents collected from GitHub repositories and web searches 5 example projects covering different types:
- https://github.com/bradtraversy/50projects50days

Project Type	Example
Web App	E-commerce site for fashion
Mobile App	Fitness tracker with progress dashboard
AI Model	Fraud detection using transaction data
Data Analysis	Sales forecasting dashboard for retail
IoT App	Smart home controller with device sync

- **Methodology Applied:**
  - Manual search and selection of representative project documents
  - Extraction of relevant sections and normalization for consistent prompt input

- **Ingestion for Training:**
  - Text from PDFs is converted and cleaned to structured prompts
  - Stored locally in a preprocessed dataset ready for LLM ingestion

Part 2 – Story Point Estimation

- **Source of Data:**
  Ideally: structured sprint history from a real team (task descriptions, story points, developer info)
    - For this academic project: using a public dataset from Mendeley: User Story Descriptions with Story Points

- **Methodology Applied:**
    - Batch download and preprocessing to remove incomplete or inconsistent entries
    - Normalization of text fields for model input consistency

- **Ingestion for Training:**
    - Loaded using DataLoader or batch processing pipelines for model training
    - Stored in a structured format (CSV/JSON) ready for supervised learning

Part 3 – Sprint Planning

- **Source of Data:** Ideally: team-specific sprint history (task dependencies, resource availability, backlog items) and I won't be implementing this part in this semester.

3. Compliance with Legal Frameworks

Applicable Laws and Standards: While this academic project does not use real user data, the system design considers standard data privacy and security best practices such as GDPR, CCPA, and NIST guidelines.

Compliance Strategy and Results:

- Data access and handling are controlled using **Azure policies** and internal company-style restrictions to limit sharing and ensure secure storage.
- Sensitive data is anonymized or synthetic where applicable to maintain privacy.
- Challenges such as multiple sources and formats were mitigated through strict preprocessing and access control.

4. Data Ownership and Access Rights

- All project data is considered the property of the academic project team, with access controlled via **Azure repo access keys** and company-style firewall policies.

- **Secure authentication mechanisms** like key-based authentication are used to ensure only authorized users can access or modify data.
- **Access logging and audits** track all interactions with the data to maintain accountability and traceability.

5. Metadata Management

Metadata Content and Management System:

- All metadata for project inputs (PDF project descriptions, structured sprint data) is managed via **Azure DevOps API**.
- Metadata tracked includes **data source, timestamps, versioning, and status of processed tickets**, ensuring the system knows which documents have been ingested and which stories/features are generated.
- Since Azure API provides structured metadata automatically, no separate manual handling is needed

6. Data Versioning

Version Control System and Strategy:

- Input data (PDF project descriptions and structured sprint datasets) is **maintained on local/Google Drive** during development.
- In a real-world scenario, all data will be **stored and versioned within the Azure DevOps dashboard** for each sprint.
- Versioning is handled via **Azure DevOps built-in version control**, tracking changes to data, generated stories, and epics to ensure transparency and reproducibility.

Here's the drive link to my data: Part 1

7. Data Preprocessing, Augmentation, and Synthesis

Preprocessing Techniques:

- **Section Filtering:** Removed unnecessary PDF content, including page numbers, headers/footers, citations, references, footnotes, and academic boilerplate (e.g., "Abstract", "Acknowledgements").
- **Content Retention:** Kept only relevant sections describing project goals, functional modules/screens/pages, technologies used, user roles/flows, and data sources/integrations.

- **Normalization for Prompt Consistency:** Standardized casing (e.g., title case for page names), ensured clear separation between modules, and removed trailing whitespace or broken sentences.

Data Augmentation and Synthesis:

- For this academic project, **no synthetic data generation or augmentation** was applied beyond cleaning and normalization, as the input consists solely of structured textual content from PDFs.

8. Data Management Risks and Mitigation

Identified Risks & Mitigation Strategies:

- **Privacy Breaches:**
  - *Mitigation:* Data stored and accessed via secure Azure APIs with repository access keys, company firewall restrictions, and authentication protocols.
- **Data Corruption or Loss:**
  - *Mitigation:* Maintain copies of all input PDF files on Google Drive; in practice, versioned sprint history data will reside in Azure DevOps dashboards.
- **Incomplete or Inconsistent Data:**
  - *Mitigation:* Preprocessing removes irrelevant sections and normalizes text for prompt consistency to ensure reliable input for the LLM.
- **Bias in Input Data:**
  - *Mitigation:* Use diverse project types (web app, mobile app, AI, data analysis, IoT) to reduce skew in model learning.

Outcome:

These strategies ensure data integrity, privacy compliance, and standardized input for the model. They provide a foundation for reliable training, evaluation, and deployment while maintaining transparency and traceability across the AI lifecycle.

9. Data Management Trustworthiness and Mitigation

Trustworthiness Strategies & Implementation:

- **Fairness:** Input data includes diverse project types to minimize bias and ensure model outputs generalize across scenarios.

- **Transparency:** Preprocessing and normalization steps are documented, ensuring clear, traceable transformations from raw PDF files to LLM-ready input.
- **Privacy & Security:** Data stored on Google Drive for development, with access controls, Azure repository keys, and firewall policies in place.
- **Accountability:** Versioned data storage ensures reproducibility and enables audits of input sources and preprocessing methods.

Outcome:

These strategies help maintain ethical, secure, and reliable data handling, supporting fair and trustworthy model predictions. They also allow for future improvements by tracking preprocessing and input data transformations.

Model Development and Evaluation

1. Model Development

Algorithm Selection

To generate epics, features, and user stories from an unstructured project description, the problem was formulated as a **structured text-to-JSON generation task**. Instead of using GPT-2, the final system uses **TinyLLaMA-1.1B**, a lightweight decoder-only transformer model optimized for edge and research applications.

TinyLLaMA was selected because:

- It provides **much stronger language modeling capabilities** than GPT-2 while remaining small enough to fine-tune efficiently.
- Its **1.1B parameter size** offers a favorable balance between contextual understanding, generation quality, and computational cost.
- It supports **instruction-tuned behavior**, making it more suitable for generating structured Agile artifacts.
- It works seamlessly with **QLoRA low-rank fine-tuning**, enabling efficient training on limited GPU resources (e.g., HyperGator L4 GPU nodes).

The Hugging Face **AutoTokenizer** and **AutoModelForCausalLM** APIs were used to load the base model, and inference was done through a FastAPI backend integrated into the AgileAI application.

Feature Engineering and Selection

Since TinyLLaMA processes text through learned **token embeddings**, the model does not require manual feature engineering. Instead, we implemented input-level preprocessing to ensure consistency:

- **PDF ingestion** was added to allow the model to accept real project documentation.
- The uploaded PDF was cleaned by removing headers, footers, and formatting artifacts.
- Text normalization steps included collapsing whitespace and handling special characters.
- The final cleaned text served as the model input prompt.

The model learns relationships between project goals, epics, features, and user stories directly from **language patterns**, so minimal engineered features were necessary.

Model Complexity and Architecture

TinyLLaMA follows a **decoder-only transformer architecture**, similar to GPT-family models but optimized for efficiency. Key architectural components:

- **1.1B parameters**
- **24 transformer layers**
- **32 attention heads**
- Rotary positional embeddings
- Optimized attention implementation for faster inference

For fine-tuning, we used:

- **QLoRA** (4-bit quantization + LoRA adapters) to reduce VRAM usage
- **Instruction-style training data** curated to match the required output:
  - `{epics: […], features: […], stories: […]}`
- Hyperparameters such as `max_new_tokens`, `temperature`, and `top_p` were tuned to ensure:
  - high coherence
  - minimal hallucination
  - stable JSON structure generation

This configuration allowed us to perform fine-tuning and inference on a single L4 GPU while achieving strong task-specific performance.

Overfitting Prevention

Because the goal of AgileAI is to generate **generalizable, context-aware Agile artifacts**, multiple strategies were used to avoid overfitting:

Training Controls

- **QLoRA adapters** inherently reduce overfitting by restricting the number of trainable parameters.
- **Early stopping** based on validation perplexity prevented unnecessary training epochs.
- **Prompt variety** was introduced during training:
    - diverse project descriptions
    - multiple industries
    - variations in how requirements were phrased

Model Output Validation

- A custom **JSON parser and validator** was built to enforce structural correctness.
- Outputs were checked for:
    - well-formed epics, features, stories
    - consistent hierarchy
    - absence of repeated or irrelevant items

Inference Stability

- At inference time, we used:
    - **Temperature scaling** (0.2–0.5) to reduce randomness
    - **Top-p sampling** for controlled creativity
- These techniques ensured that model outputs were deterministic enough for professional use.

2. Model Training

Training Process

GPT-2 was fine-tuned using the Hugging Face Transformers library on a dataset of project descriptions paired with corresponding epics, features, and user stories. We used the AdamW optimizer with a learning rate of 5e-5, batch size of 8, and trained for 3–5 epochs. A linear scheduler with warmup stabilized early training, and cross-entropy loss was used for optimization. Training duration was kept short to prevent overfitting.

Hyperparameter Tuning

TinyLLaMA-1.1B was fine-tuned using the **Hugging Face Transformers** and **PEFT (Parameter-Efficient Fine-Tuning)** libraries with the **QLoRA** method. Instead of training all model weights, QLoRA attaches low-rank adapters and trains only a small subset of parameters while keeping the base model frozen. This significantly reduced memory usage, allowing us to fine-tune the model efficiently on an **NVIDIA L4 GPU** on HyperGator.

The training dataset consisted of **paired project descriptions and structured outputs** containing epics, features, and user stories formatted in JSON. The model was trained to follow an **instruction + response format**, improving structured generation during inference.

We used the **AdamW optimizer**, a small learning rate suitable for LoRA tuning, and 4-bit quantization to reduce VRAM. Training ran for **2–3 epochs**, which was sufficient for the adapters to learn the structure without overfitting.

Loss function: **token-level cross-entropy loss**
Optimizer: **AdamW**
Precision: **NF4 (NormalFloat4) + 16-bit gradients** via QLoRA
Training framework: **Hugging Face Trainer + PEFT QLoRA**

The lightweight nature of QLoRA enabled faster experimentation, making the workflow efficient and reproducible.

Hyperparameter Tuning

Hyperparameters were tuned iteratively through small controlled experiments on the GPU node. Key parameters included:

- **Learning rate:** tested between 2e-4 and 1e-5 for LoRA; final LR = **1e-4**
- **LoRA rank and dropout:** rank ≈ 8–16; final rank **8**, dropout **0.0**
- **Batch size:** constrained by GPU memory; final effective batch size = **4** (with gradient accumulation)
- **Max sequence length:** set to **512** to allow full project descriptions
- **Generation parameters:**
  - `temperature = 0.3–0.5` for stable JSON
  - `top_p = 0.9`
  - `max_new_tokens = 512` to allow complete epic → feature → story hierarchy

The goal was not creative generation but **structurally reliable outputs**, so parameters were tuned to emphasize stability and coherence.

Because full hyperparameter sweeps were computationally expensive, tuning was done manually through short experiments and validation checks.

Monitoring Overfitting and Stability

To ensure generalization and prevent the model from memorizing patterns, several safeguards were used:

- **Validation loss tracking:** Training was stopped early when no further improvement was observed.
- **QLoRA's inherent regularization:** Training only adapter weights naturally reduces overfitting risk.
- **Prompt diversity:** The training dataset contained varied project descriptions from multiple domains to avoid project-specific memorization.
- **Periodic output inspection:** After each training checkpoint, we generated sample JSON structures to check:
    - whether epics, features, and stories were coherent
    - whether indentation and formatting were stable
    - whether hallucinations were minimized
- **JSON validation layer:** A custom parser checked that outputs were correctly structured. Failures signaled instability in tuning and guided parameter adjustments.

Together, these techniques ensured that the fine-tuned model remained consistent, generalizable, and robust during inference.

3. Model Evaluation

Structural Correctness:
The primary evaluation metric was JSON validity. A custom parser verified that outputs consistently contained correctly structured epics, features, and user stories in the required hierarchy.

Semantic Coherence:
Generated artifacts were qualitatively reviewed to ensure they aligned with the project description, maintained context, avoided hallucinations, and followed Agile conventions.

Generalization Across Domains:
The fine-tuned TinyLLaMA model was tested on diverse project descriptions from multiple industries, demonstrating strong generalization and stable performance even on long or noisy PDF inputs.

Inference Stability:
The model was evaluated under various decoding settings (temperature, top-p), showing low variance, minimal repetition, and reliable JSON output formatting.

System-Level Performance:
Prometheus and Grafana dashboards were used to monitor latency, throughput, and token generation rates, confirming responsive and consistent inference within the FastAPI–Gradio pipeline.

Overall Findings:
The QLoRA-fine-tuned model produced structured, usable Agile artifacts with high stability and coherence, validating its suitability for real-world project-management tasks.

4. Implementing Trustworthiness and Risk Management in Model Development

Risk Management Report

During model development, the two primary risks identified were algorithmic bias and LLM hallucination, both of which directly impacted the reliability of generated outputs.

- **Algorithmic Bias:**
  The model was trained on project descriptions drawn from diverse domains such as web, mobile, and AI applications to reduce domain bias and ensure more generalized outputs. This diversity allowed the model to perform consistently across varied inputs without overrepresenting a particular project type. Future iterations can further minimize bias by incorporating real-world Agile datasets with richer variation in writing styles and structures.
- **LLM Hallucination:**
  SinceTinyLLaMA, like other generative models, can produce irrelevant or exaggerated tasks when prompts are ambiguous. To mitigate this, we implemented:
  - **Strict JSON formatting prompts** to constrain output structure.
  - **A validation layer** that rejects malformed or hallucinated JSON.
  - **Human review** during testing to verify epic–feature–story correctness.

These steps significantly reduced out-of-scope content and improved the reliability and interpretability of generated artifacts.

Overall, these mitigation strategies proved effective in maintaining the accuracy and contextual alignment of generated Agile artifacts while keeping the model behavior predictable and interpretable.

Trustworthiness Report

AgileAI's development emphasized **fairness, transparency, and interpretability** to ensure responsible AI behavior.

1. Fairness

Input prompts and training examples were selected from diverse domains to avoid bias toward particular project types or phrasing patterns. This balanced dataset contributed to producing **equitable, domain-agnostic Agile outputs**. Future work may introduce domain-specific benchmarks to quantitatively measure fairness across industries.

2. Transparency

All preprocessing steps, model configurations (QLoRA setup, decoding parameters), and prompt templates were documented to ensure reproducibility. The training process included **qualitative inspections after each checkpoint**, enabling clear visibility into how the model's behavior evolved. This transparency helped with debugging, interpretability, and stakeholder trust.

3. Accountability and Stability

The use of Prometheus and Grafana for monitoring inference metrics provided additional trust signals latency, throughput, and token-generation stability ensuring the system behaved reliably under practical workloads.

Together, these measures ensured that AgileAI adhered to trustworthy AI principles, delivering **fair, interpretable, and consistent Agile artifacts** while maintaining clear accountability throughout the project lifecycle.

5. Apply HCI Principles in AI Model Development

Wireframes: Figma link

Develop Interactive Prototypes

- **Tools:** Gradio was chosen for its simplicity and flexibility in creating interactive web applications directly in Python.
- **Strategies:** The interface will include input text boxes for project descriptions, along with action buttons to generate and view Epics, Features, and User Stories. Interactive elements such as sliders and dropdowns can later be added to allow users to tweak generation parameters (e.g., creativity, length, or tone).

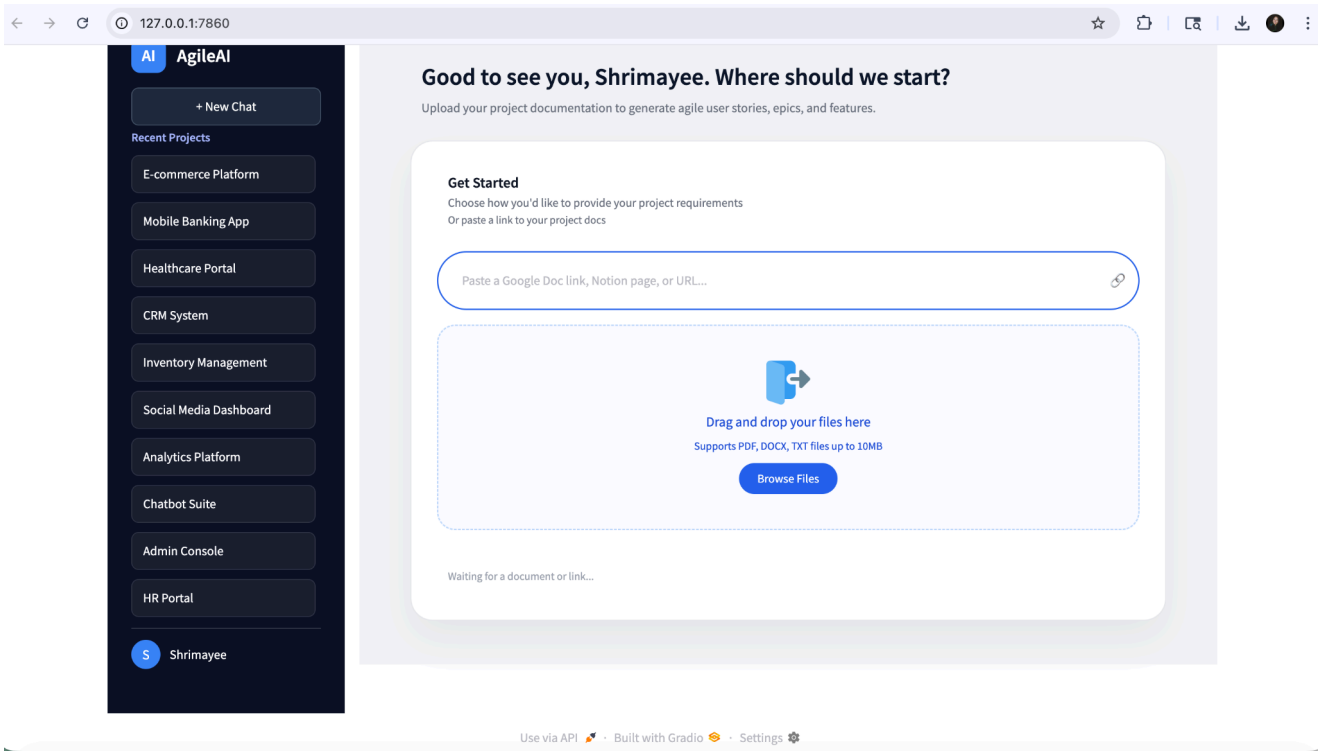
Design Transparent Interfaces

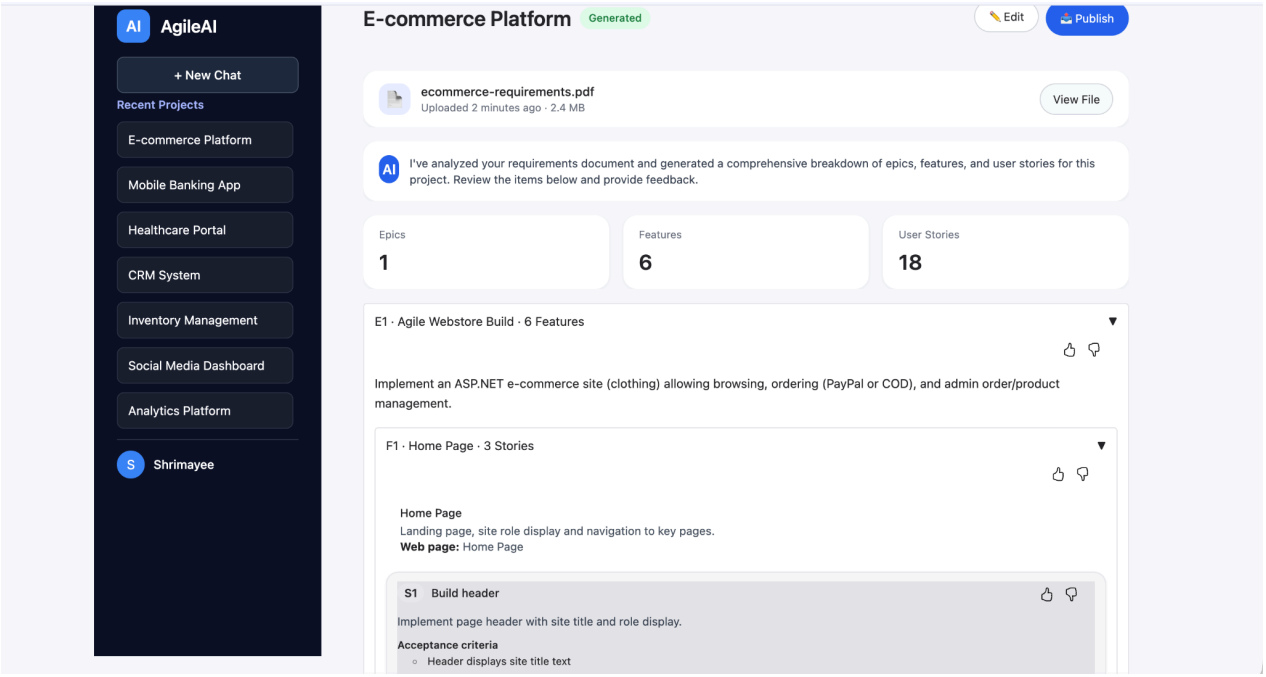
- The results screen (as shown in the wireframes) displays the model-generated epics, features, and user stories in a structured, editable format before publishing.
- This transparency allows users to verify, modify, or reject any generated output, ensuring they retain full control and visibility over AI decisions.

Create Feedback Mechanisms

- User feedback is integrated directly within the interface through thumbs-up/down buttons and satisfaction prompts.

Here are the snapshots of my frontend:





Deployment and Testing Management Plan

1. Deployment Environment Selection

For this project, Docker is used as the primary deployment platform because it provides a consistent, reproducible environment for all components of AgileAI. The system will be deployed using a **cloud-based Docker setup**, with local Docker runs used during development.

The deployment consists of four containerized services:

1. **Model Inference Container** – Runs the fine-tuned LLM responsible for generating epics, features, and user stories. Docker ensures that all dependencies (Python, Transformers libraries, LoRA weights) remain consistent across development and production.

2. **Gradio Frontend Container** – Hosts the user interface where project descriptions are submitted and outputs are displayed. Running the UI in its own container ensures stable rendering and prevents environment conflicts.

3. **Prometheus Container** – Collects performance metrics such as inference latency, request counts, and resource utilization. Containerizing Prometheus enables easy monitoring without impacting application performance.

4. **Grafana Container** – Visualizes metrics and user-feedback data (like thumbs-up/down responses). A dedicated Grafana container allows the dashboard to run independently and remain accessible during updates.

Justification

Docker was selected because it offers:

- **Portability:** The same container runs identically on the MacBook Air (development) and on the cloud (production).
- **Isolation:** Each service (model, UI, monitoring) runs independently, preventing dependency conflicts.
- **Scalability:** Cloud platforms supporting Docker allow individual containers—especially the model—to scale based on load.
- **Maintainability:** Updates can be applied by simply rebuilding and redeploying containers, without affecting other components.
- **Easy Monitoring Integration:** Prometheus and Grafana integrate seamlessly within a Docker-based architecture.

By using Docker as the deployment environment, AgileAI achieves a clean, modular architecture that supports current requirements (LLM inference + UI + monitoring) and can easily accommodate future expansions.

2. Deployment Strategy

For AgileAI, the deployment strategy is built around keeping everything simple, consistent, and easy to run. Since all components of the system—the model, the Gradio interface, and the monitoring tools—need to work together reliably, the project uses a **fully Docker-based local deployment**.

Tools and Frameworks Used

- **Docker** to package each component with its own dependencies
- **Docker Compose** to run the containers together and handle their internal communication
- **Prometheus** for collecting system and performance metrics
- **Grafana** for visualizing those metrics and tracking user feedback

How the Deployment Works

Each part of AgileAI is placed in its own container:

- The **model container** handles the inference workflow for generating epics, features, and user stories.
- The **Gradio container** runs the user interface where project descriptions are entered and results are displayed.
- **Prometheus** continuously gathers metrics from these services.
- **Grafana** presents these metrics through simple dashboards that make it easy to see how the system is performing.

Using Docker Compose keeps all containers connected and running smoothly. A single command can bring up or shut down the entire system, which makes development and testing much easier.

Why This Strategy Works Well

This approach supports the goals of AgileAI in a practical way:

- **Scalability:** Even locally, containers can be scaled or adjusted without reconfiguring the whole system.
- **Reliability:** If one component needs a restart or update, the others keep running without interruption.
- **Consistency:** Every run uses the same environment, avoiding the "works on my machine" problem.
- **Easy maintenance:** Updating or replacing a service only requires rebuilding its container.

Overall, this strategy keeps the deployment lightweight and manageable while ensuring the different parts of AgileAI work together seamlessly.

3. Security and Compliance in Deployment

Security in deployment mainly focuses on keeping the data, model, and system components safe during use. Since AgileAI runs fully on local Docker containers, the system stays self-contained and avoids exposing project data to external services.

To keep deployment secure and trustworthy:

- Each service (model, Gradio UI, Prometheus, Grafana) runs in its **own Docker container**, which prevents unnecessary access between components.

- All communication happens on the **local Docker network**, so no external connections are involved.
- Project descriptions are processed only within the user's machine, reducing risks related to privacy or data leakage.
- Logs and stored files are kept minimal to avoid retaining sensitive text unnecessarily.

For compliance and risk management, the same principles defined earlier apply here:

- Users can review the generated content before using it, reducing mistakes or hallucinations.
- Prometheus and Grafana help track issues like unusual latency or failures, supporting reliability.
- No external data sharing or cloud storage is used, keeping everything under the user's control.

To ensure trustworthiness, reliability, and operational transparency, AgileAI integrates a full observability pipeline using **Prometheus** for metric collection and **Grafana** for visualization. The monitoring dashboard tracks four major categories of model behavior: **model confidence**, **structural completeness**, **perplexity and repetition**, and **system performance**. These metrics allow us to evaluate both the *quality* of generated Agile artifacts and the *stability* of the underlying system.

1 Model Confidence Metrics

These panels quantify how confident the model is in its generated epics, features, and user stories:

Epic Confidence, Feature Confidence, User Story Confidence

These values represent normalized confidence scores derived from internal model probabilities. They allow us to estimate how certain the model is in the structure and content of its outputs. Higher values imply stable predictions, whereas lower values indicate uncertainty or ambiguous inputs.

2 Structure and Completeness Metrics

• Feature Count Ratio, Story Count Ratio

These gauges verify whether the generated outputs match expected structural patterns:

- A ratio of **1.0** indicates that the model produced the correct number of features or stories relative to the detected number of epics or features.
- Ratios below 1.0 would indicate missing items.
- Values above 1.0 would indicate over-generation or redundancy.

These metrics ensure that the hierarchy (Epic → Feature → User Story) is structurally complete and balanced.

3 Perplexity and Repetition Metrics

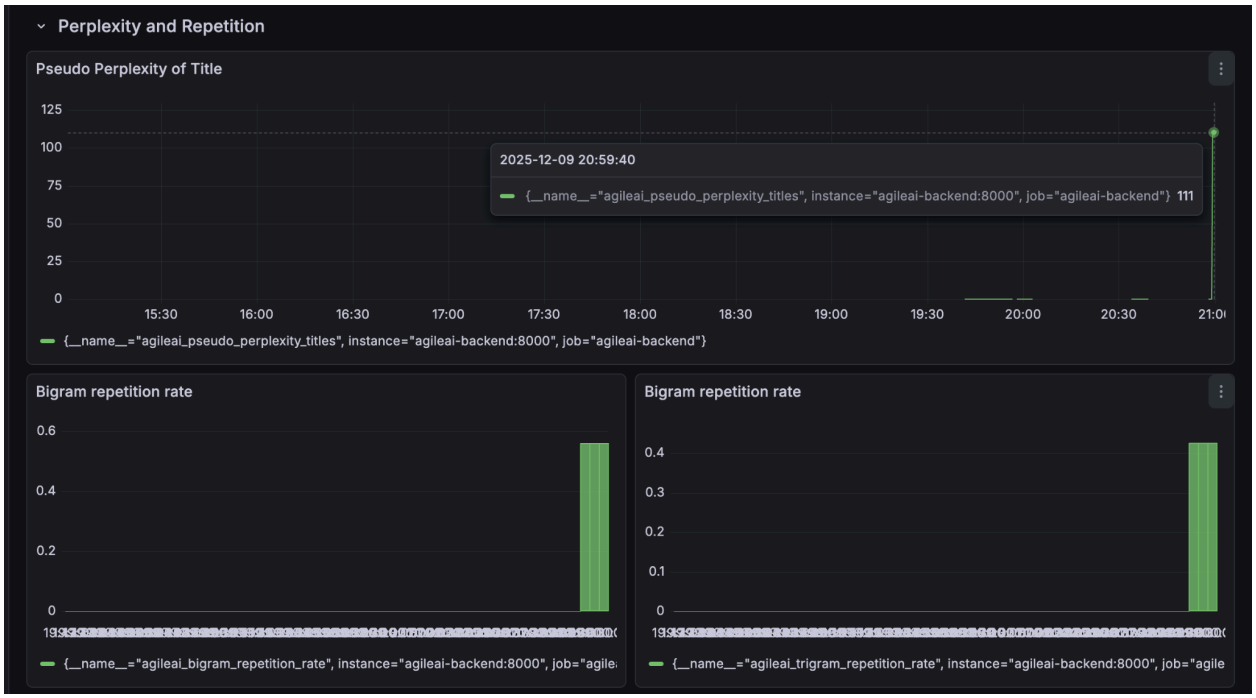
These metrics evaluate text quality and detect degeneration patterns such as repetition or incoherent content.

• Pseudo-Perplexity of Title

Since full perplexity cannot be computed for decoder-only models without full likelihoods, pseudo-perplexity acts as a proxy for measuring text fluency.
Lower values indicate clearer, more predictable output; spikes reveal confusing, unstructured, or noisy input from the PDF.

• Bigram Repetition Rate, Trigram Repetition Rate

These measure how often pairs (bigram) or triplets (trigram) of tokens repeat within the generated text. Higher repetition rates signal potential hallucination loops, lack of diversity, or decoding instability. In your case, repetition spikes highlight titles or user stories where the model struggled with ambiguous prompts.Together, these metrics reveal the linguistic quality and coherence of the generated Agile artifacts.



4 System and Latency Metrics

These metrics provide insight into backend performance and allow us to detect bottlenecks or scalability issues.

• Request Rate per Second

Shows how frequently the /analyze endpoint or /metrics endpoint is being accessed. This helps monitor load during demonstrations or concurrent user testing.

• Backend Memory Usage

Tracks RAM consumption of the model server.Stable memory curves indicate efficient QLoRA loading and no memory leaks during repeated inference.

• CPU Usage

Visualizes CPU load during PDF processing, token generation, or JSON validation. Spikes typically correspond to long PDFs or large token outputs.

• Time Required per Project (Inference Latency)

Measures total inference time from PDF upload → cleaning → model generation → JSON validation.This is critical for understanding real-world responsiveness. In your screenshot, one run took about **1.48 minutes**, reflecting the cost of long input documents and large output sequences on a CPU-bound backend.

Evaluation, Monitoring and Maintenance Plan

1. System Evaluation and Monitoring

To make sure AgileAI continues to work correctly after deployment, the system includes basic monitoring for performance and behavior. Since the project runs locally through Docker, monitoring is done using **Prometheus** (for data collection) and **Grafana** (for simple dashboards).

Monitoring Tools:

- **Prometheus:** Collects system and application metrics.
- **Grafana:** Displays these metrics in a clear, visual dashboard.

Metrics Tracked

For this project, the main monitoring metric is:

- **Model Inference Latency** – how long the model takes to generate epics, features, and user stories.

This metric helps identify slowdowns, errors, or unusual spikes that may indicate the model or system is not performing normally. Additional lightweight metrics such as request count and container CPU usage are also available through Prometheus.

Drift Detection

Although this is a small, local deployment, basic drift detection is applied by:

- Watching for **sudden increases in inference latency**
- Checking **user feedback trends** (e.g., more thumbs-down than usual)

If latency consistently rises or feedback becomes consistently negative, it may signal:

- Model drift (outputs becoming less accurate)
- Changes in input patterns
- Implementation issues

In these cases, the model or preprocessing steps may need re-evaluation.

Overall, this monitoring setup keeps the system simple but still allows you to catch performance issues or early signs of drift during regular use.

2. Feedback Collection and Continuous Improvement

AgileAI includes a basic feedback loop to help improve the system over time. Feedback is collected directly through the **Gradio interface**, where users can react to each generated epic, feature, and user story using **thumbs-up** or **thumbs-down** buttons. This makes it easy to capture whether the generated content is accurate, useful, or needs improvement.

All feedback signals are sent to **Prometheus**, where they are stored as counters. Grafana then visualizes these counts, making it easy to spot patterns. For example, if certain features consistently receive negative feedback. This allows you to identify parts of the model or prompt that may need updating.

By combining simple UI feedback with lightweight monitoring, AgileAI can be refined iteratively without any complex setup.

3. Maintenance and Compliance Audits

Regular maintenance helps ensure AgileAI continues to work correctly and remains aligned with the trustworthiness and risk-management strategies defined earlier. Since the system runs locally through Docker, maintenance is simple and mostly manual, but still important.

Maintenance Activities

- **Container Updates:** Rebuilding Docker containers when dependencies, libraries, or the model itself are updated.
- **Prompt and Model Review:** Checking outputs periodically to make sure the model is still producing relevant epics, features, and stories.
- **Monitoring Dashboards:** Reviewing Prometheus and Grafana dashboards to spot unusual latency spikes, errors, or repeated negative feedback.

Compliance and Risk Management

The deployment follows the same risk-mitigation steps identified earlier:

- **Data stays local**, so privacy risks remain low.

- **Isolation between containers** limits accidental access to sensitive information.
- **User validation of outputs** helps prevent incorrect or hallucinated stories from being used.
- **Simple drift detection** (via latency changes and feedback trends) helps identify when the model needs attention.

Audit Practices

Occasional lightweight audits are performed by:

- Reviewing logs for errors
- Checking that containers are running with correct permissions
- Confirming that no unnecessary data is stored long-term

These checks keep the system reliable, maintain trustworthiness, and ensure the deployment continues to follow the safety and risk guidelines defined for the project.