Comprehensive Implementation Guide for the Mental Health Chatbot

UniMind Team
BSc AI and Data Science, 2024/25

March 26, 2025

Contents

1 Introduction and Overview

This document serves as a detailed guide to direct each team member on the development of the mental health chatbot. The project is split into three core components:

a) **Model Building**: Focused on data merging, cleaning, and fine-tuning a large language model (LLM) to generate personality-adapted responses.

b) **Backend Development**: Centered on configuring RASA for intent classification, dialogue management, and integration with custom action servers.

c) **Frontend Development**: Responsible for the user interface, ensuring a friendly, accessible, and crisis-responsive design.

Each section of this guide is written from a research and technical perspective, discussing both the **requirements from our module brief** and practical implementation details. The key areas addressed include:

- Requirement Analysis and System Design
- Prototype Development and Testing Strategies
- Ethical and Legal Considerations, including privacy and crisis protocols
- Teamwork and Project Management aspects

The mental health chatbot is intended to provide scalable and empathetic support to university students. Primary research (interviews with professionals and surveys with students) highlights the importance of addressing academic stress, financial worries, and mental health crises in a culturally sensitive manner. Our architecture adopts a hybrid approach that leverages both rule-based dialogue management (via RASA) and advanced generative capabilities (through a fine-tuned LLM).

The remainder of this document details specific tasks and research directions for each development area. It is expected that each team member will refer to the appropriate section for guidance on their responsibilities.

2 Model Building

2.1 Overview

The model building component is critical as it forms the basis for how the chatbot understands user input and generates tailored responses. This section outlines the steps to gather, clean, merge, and prepare data for two major purposes:

1) **RASA NLU Training:** For intent classification, entity extraction, and dialogue management.
2) **Fine-Tuning a Generative LLM:** For producing personality-adapted, free-form responses.

Our goal is to build a training dataset that is robust and covers the range of student-specific queries (e.g., exam anxiety, plagiarism concerns, financial support, etc.). This process directly addresses the **Requirement Specification** and **System Design** criteria from the module brief.

2.2 Data Sources and Their Roles

The data available in `/src/data/finetune-set` includes:

- **Hugging Face Datasets:**
    - `personality_dataset_train`: Contains a large set of personality-related question-answer pairs.
    - `personalities`: Detailed descriptions of personality traits.
    - `mental_health_data`: Focused on mental health conversation data.
- **Zipped Folders:**
    - `Therapist Patient Conversation Dataset.zip`: Typically includes `intents.json` with structured conversation patterns.
    - `Mental Health Data.zip`: Another structured dataset with intent-based conversation examples.
- **Harmony Data (1).xlsx:** Provides question matching metadata and similarity scores (e.g., `match` column) that can help synthesize paraphrases.
- **Mental Health Data.txt:** Contains unstructured text with personality scores and conversation snippets.
- **New Student-Focused Questions:** Cover topics from academic stress to financial aid, which must be integrated.

2.3 Data Cleaning and Merging

2.3.1 Steps for Data Extraction and Consolidation

1) **Unzip and Parse:** Extract contents from zipped folders. Use scripts to parse JSON files (e.g., `intents.json`) and read Excel files using Python libraries like `pandas`.

2) **Normalize Text:** Standardize punctuation, remove extra spaces, and harmonize spellings (British vs. American) to ensure consistency across sources.

3) **Deduplication:** Identify and remove duplicate patterns. Use string similarity metrics (e.g., Levenshtein distance) to merge similar examples.

4) **Mapping:** Map the extracted data to student-specific intents. For instance, align phrases like "I'm anxious about my finals" to the `exam_anxiety` intent.

5) **Entity Annotation:** Annotate important entities (e.g., `subject_name` for course subjects, `deadline_type` for assignment dates) if necessary.

2.3.2 Script Outline for Data Merging

Below is an example pseudocode outline for merging Harmony Data:

```
import pandas as pd

# Load Harmony data
harmony_df = pd.read_excel("Harmony␣Data␣(1).xlsx")
# Filter high match scores
good_matches = harmony_df[harmony_df['match'] > 0.8]

# Loop through and extract question pairs
qa_pairs = []
for index, row in good_matches.iterrows():
    q1 = row['question1_text']
    q2 = row['question2_text']
    qa_pairs.append(q1)
    qa_pairs.append(q2)

# Remove duplicates and save as a text file for RASA NLU examples
qa_unique = list(set(qa_pairs))
with open("merged_qa.txt", "w") as f:
    for q in qa_unique:
        f.write(q + "\n")
```

2.4 Building the Fine-Tuning Dataset

For fine-tuning an LLM:

1) **Collate Q–A Pairs:** Gather conversation snippets from all sources, including those from `Mental Health Data.txt` and interview transcripts.

2) **Structure Data:** Format data into a JSON structure:

```
{
  "instruction": "[EMPATHETIC] The user expresses concern about deadlines
  "input": "User: 'I might miss my deadline because of anxiety.'",
  "output": "I'm sorry to hear you're worried. Have you considered talkin
}
```

3) **Tag Personalities:** Replicate each conversation pair for various personality styles (e.g., `[DIRECT]`, `[MOTIVATIONAL]`). This helps the LLM learn to generate style-specific responses.

4) **Augmentation:** If examples are scarce, consider using synthetic data generation methods (e.g., prompting an LLM to generate paraphrases) but document the process.

2.5 Research and Further Considerations

- Review literature on **data augmentation** techniques in NLP.
- Investigate **domain adaptation** methods to ensure the LLM fine-tuning is robust for student mental health contexts.
- Ensure ethical use and documentation of data sources; include statements on anonymization.

Advanced Data Cleaning Techniques: In addition to basic normalization and deduplication, consider employing:

- **Tokenization and Lemmatization:** Use NLP libraries (e.g., NLTK, spaCy) to reduce words to their base forms. This ensures that semantically identical phrases are treated uniformly.
- **Semantic Similarity Measures:** Implement cosine similarity on word embeddings to identify near-duplicate phrases.
- **Manual Review:** After automatic cleaning, manually review a subset to ensure no critical context was lost.

Data Augmentation Strategies: To increase the variety of training examples:

- **Paraphrasing:** Use tools or models (like T5 or GPT-3) to generate paraphrases for key student questions.
- **Back-Translation:** Translate text to another language and back to create natural paraphrases.
- **Synonym Replacement:** Randomly substitute words with synonyms while preserving meaning.

Annotation Guidelines: Establish a clear set of annotation guidelines to ensure consistency:

a) Define standard labels for intents (e.g., `exam_anxiety`, `financial_support`).

b) Document rules for entity annotation (e.g., how to label course names, dates, and numerical values).

c) Provide examples and counter-examples in an annotation manual.

Evaluation Metrics: Plan for evaluation of the model training:

- **NLU Performance:** Use RASA's built-in evaluation commands (`rasa test nlu`) to measure intent classification accuracy.
- **LLM Quality:** Manually review generated responses for coherence, style adherence, and factual correctness.
- **User Simulation:** Create simulated conversation flows and evaluate end-to-end system performance.

Documentation and Version Control: Maintain comprehensive documentation:

- Keep a changelog of data cleaning scripts, annotation modifications, and fine-tuning parameters.
- Use Git to version control all scripts and data files. Commit frequently with detailed messages.

Research Literature: Consider reading the following for deeper insights:

- "Data Augmentation in NLP: A Review" for augmentation techniques.
- "Domain Adaptation for Neural Machine Translation" for methods applicable to our fine-tuning.
- Recent papers on ethical considerations in AI for mental health.

3 Backend Development

3.1 Overview

The backend is the engine of our chatbot. It includes the RASA framework for dialogue management and a custom action server that connects to the fine-tuned LLM. This section details the tasks and research directions to ensure that the system is robust, secure, and scalable. This work addresses **System Design**, **Prototype System**, and **Test/Evaluation Strategy** aspects of the brief.

3.2 RASA Configuration

Key Files:

- **config.yml:** Define the NLU pipeline (e.g., `SpacyTokenizer`, `DIETClassifier`) and dialogue policies.
- **domain.yml:** List intents, entities, slots, responses, and actions.
- **data/nlu.yml, stories.yml, rules.yml:** Provide comprehensive training examples and conversation paths.

Action Items:

1) Update the `domain.yml` to incorporate new intents and the personality slot.
2) Ensure that `stories.yml` contains paths for student-specific queries (e.g., financial support, exam anxiety).
3) Run `rasa test nlu` and `rasa test core` to validate the training data.

3.3 Custom Action Server Implementation

Structure and Dependencies:

- Use `rasa-sdk` for custom action classes.
- Organize actions into multiple files if necessary (e.g., `finance_actions.py`, `crisis_actions.py`).
- Include external dependencies (e.g., `requests` for API calls to the LLM).

Example Code Snippet:

```
from typing import Text, Dict, Any
from rasa_sdk import Action, Tracker
from rasa_sdk.executor import CollectingDispatcher
import requests

def call_llm(user_text: str, personality: str) -> str:
    payload = {"input": user_text, "style": personality}
    # Example: Call a remote LLM inference endpoint
    try:
        response = requests.post("http://my-llm-host:8000/generate", json=payl
        if response.status_code == 200:
            return response.json().get("generated_text", "Sorry,␣I␣didn't␣catc
        else:
            return "There␣was␣an␣error␣generating␣a␣response."
    except Exception as e:
        return "Error␣contacting␣the␣response␣service."

class ActionSuggestFinanceResources(Action):
    def name(self) -> Text:
        return "action_suggest_finance_resources"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker,
            domain: Dict[Text, Any]) -> list:
        personality = tracker.get_slot("personality_type") or "empathetic"
        user_message = tracker.latest_message.get("text")
        llm_response = call_llm(user_message, personality)
        dispatcher.utter_message(text=llm_response)
        return []
```

3.4 Crisis Protocols and Privacy

Crisis Handling:

- Implement an `action_handle_crisis` that immediately displays crisis information.
- Ensure that if a user's message contains high-risk keywords (e.g., suicidal ideation), the system triggers this action.

Privacy Measures:

- Limit the storage of personal data and, if necessary, store logs in an anonymized database.
- Provide a "**Clear Data**" functionality accessible from the frontend.

3.5 Scalability and Deployment

Containerization and Load Balancing:

- Containerize RASA and the action server using Docker.
- Set up load balancing (e.g., using Kubernetes) for handling increased user traffic.

LLM Scalability:

- If using a remote LLM, deploy it on a GPU-enabled server or use a managed inference service.
- Consider horizontal scaling with message queues (e.g., RabbitMQ) if LLM calls become latency-bound.

3.6 Testing and CI/CD Integration

Testing:

- Write unit tests for custom actions using `pytest`.

- Create end-to-end tests by simulating user interactions with RASA (e.g., using `rasa test core`).

CI/CD:

- Set up automated pipelines (e.g., GitHub Actions) to run tests and deploy containers.
- Ensure version control of configuration files and action scripts.

3.7 Research Directions for Backend Improvements

- Investigate adaptive dialogue policies that modify responses based on user context.
- Research secure methods to integrate external APIs while complying with GDPR.
- Consider long-term logging solutions for iterative improvement, ensuring compliance with ethical guidelines.

Detailed RASA Configuration: The `config.yml` should include:

```
language: en
pipeline:
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
  - name: DIETClassifier
  - name: EntitySynonymMapper
policies:
  - name: MemoizationPolicy
  - name: TEDPolicy
  - name: RulePolicy
```

These settings ensure the backend can effectively parse and classify the variety of student questions.

Domain-Specific Customization: Within `domain.yml`, define slots and responses to capture personality preferences:

```
slots:
  personality_type:
    type: categorical
    values:
      - empathetic
      - direct
      - motivational
      - analytical
    influence_conversation: true
entities:
  - subject_name
  - deadline_type
intents:
  - exam_anxiety
  - plagiarism_concern
  - attendance_concern
  - financial_support
  - crisis_suicidal_thoughts
responses:
  utter_exam_anxiety:
    - text: "Exams can be very stressful. Would you like some study tips or re
  utter_financial_support:
    - text: "I understand financial worries are challenging. The university o
```

Action Server Robustness: Enhance your action server code to handle exceptions and logging. For instance, integrate Python's logging module:

```
import logging
logging.basicConfig(level=logging.INFO)

def call_llm(user_text: str, personality: str) -> str:
    payload = {"input": user_text, "style": personality}
    try:
        response = requests.post("http://my-llm-host:8000/generate", json=payl
        response.raise_for_status()
        logging.info("LLM␣call␣successful")
        return response.json().get("generated_text", "Sorry,␣I␣didn't␣catch␣th
    except Exception as e:
        logging.error(f"LLM␣call␣failed:␣{e}")
        return "There␣was␣an␣error␣processing␣your␣request.␣Please␣try␣again␣
```

Crisis Action Example: A critical component is the crisis protocol. The action must immediately provide emergency contact details.

```
class ActionHandleCrisis(Action):
    def name(self) -> Text:
        return "action_handle_crisis"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker,
            domain: Dict[Text, Any]) -> list:
        crisis_message = (
            "It␣appears␣you␣might␣be␣experiencing␣a␣crisis.␣"
            "If␣you␣feel␣unsafe␣or␣are␣in␣immediate␣danger,␣please␣call␣your␣
            "For␣further␣support,␣contact␣your␣university␣counseling␣center."
        )
        dispatcher.utter_message(text=crisis_message)
        return []
```

Continuous Integration: Implementing CI/CD is essential. A sample GitHub Actions workflow for testing RASA might include:

```
name: RASA CI

```

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'
      - name: Install dependencies
        run: |
          pip install rasa rasa-sdk pytest
      - name: Run RASA NLU Tests
        run: rasa test nlu
      - name: Run RASA Core Tests
        run: rasa test core
```

4 Frontend Development

4.1 Overview

The frontend is the user-facing component that delivers the chatbot experience. The design must reflect findings from our research—emphasizing simplicity, empathy, and user control. This section details the required tasks, UI/UX guidelines, and research directions. It addresses the **Prototype System** and **Project Management/Teamwork** elements.

4.2 UI Design and Layout

General Requirements:

- **Responsive Design:** The UI should work across devices (desktop, tablet, mobile).
- **Accessibility:** Support Dark/Light mode, adjustable font sizes, and screen reader compatibility.
- **Simplicity:** The interface should be clean, with clear navigation and minimal clutter.

Page Layout:

1. **Header:** Contains the app logo, title, and quick-access icons (e.g., Settings, Crisis Button).
2. **Main Chat Area:** A chat bubble interface where:
    - User messages appear on the right.
    - Bot responses appear on the left.
3. **Input Area:** Fixed at the bottom with:
    - A text input box.
    - A send button.
    - Quick action buttons for "Suggested Exercises" or "Resources."
4. **Settings/Preferences Page:** Accessible via an icon in the header.
    - Options include: Check-in frequency, Tone selection (Casual, Empathetic, Adaptive), Dark/Light mode toggle, and Data privacy controls.

4.3 User Onboarding and Interaction Flow

Onboarding Screen:

- Provide an introduction to the chatbot. Explain that it offers mental health support but is not a replacement for professional therapy.
- Allow the user to select their preferred personality/tone (default: Empathetic + Supportive).
- Offer a toggle to enable or disable daily check-ins.

Main Chat Interface:

- Display a **typing indicator** when waiting for a response.
- Include **quick reply buttons** for common actions (e.g., "View Resources," "Start Exercise").
- Implement smooth transitions and animations to improve user experience.

4.4 Crisis Protocol and Emergency Response

Crisis Button:

- The emergency button should be prominent (e.g., bright red) and accessible on every page.
- When activated, it immediately displays crisis contact details (e.g., hotline numbers) and a disclaimer stating that the chatbot is not a substitute for immediate professional help.

Fallback Messaging:

- If the backend is unresponsive, the UI should display a friendly message indicating the issue and offering alternative support (e.g., "Please try again later or contact your university counseling center.").

4.5 Accessibility and User Preferences

User Personalization Features:

- **Dark/Light Mode Toggle:** Allow users to switch between themes.
- **Font Size Adjustments:** Provide controls to increase or decrease text size.
- **Privacy Settings:** Let users review what data is stored and allow them to clear chat history.
- **Voice Interaction:** Optional feature for reading responses aloud.

4.6 Error Handling and Future Extensions

Error Handling:

- Display an error message if the chatbot fails to connect with the backend.
- Prompt the user to rephrase ambiguous text or change input language if necessary.

Future Enhancements:

a) **Multi-language Support:** Extend the interface to support additional languages for international students.

b) **Voice-based Interaction:** Develop a voice assistant module for guided exercises.

c) **Scheduled Reminders:** Enable outbound notifications for daily or weekly check-ins.

d) **Enhanced Data Visualizations:** Integrate mood tracking graphs and progress dashboards.

4.7 Implementation and Testing

Frontend Development Stack:

- Use a modern JavaScript framework such as React, Vue, or Angular.
- Ensure real-time communication using WebSockets or asynchronous API calls.
- Design a responsive layout that adapts to various screen sizes.

Testing Procedures:

- Conduct usability testing with a small group of students.
- Gather feedback on the tone, ease of navigation, and clarity of information.
- Perform A/B tests for different UI elements (e.g., color schemes, button placements).

Research for UI/UX: To improve user engagement and accessibility:

- Review current literature on digital mental health interfaces.
- Examine case studies from popular mental health apps (e.g., Headspace, Wysa).
- Analyze survey data to understand user preferences for tone, check-in frequency, and crisis response.

Detailed UI Component Design:

1. **Header Bar:**
    - Must include the chatbot logo and title.
    - A settings icon that leads to a dedicated preferences page.
    - A crisis button that is visible on every page.
2. **Chat Window:**
    - Utilize a two-column layout: right-aligned for user messages, left-aligned for bot responses.
    - Incorporate subtle animations for new message appearance.
3. **Input Area:**
    - The text input should be prominent with placeholder text guiding the user.
    - Include a “send” button and quick-access icons for additional features (e.g., attachments, voice input).
4. **Settings Page:**

  - Options for changing the chatbot's tone (dropdown or toggle selection).
  - Check-in frequency selection: Daily, Weekly, or None.
  - Privacy controls: buttons to view, clear, or disable data storage.
  - Theme selection: Dark Mode and Light Mode.

Accessibility Considerations:

- Ensure that all interactive elements are keyboard navigable.
- Use ARIA roles to enhance screen reader support.
- Test the interface with various devices and browsers.

Integrating Frontend with Backend:

- The frontend should send user inputs to the RASA REST endpoint.
- Use asynchronous API calls to fetch bot responses without freezing the UI.
- Implement error-handling routines that provide user-friendly messages in case of backend failure.

User Feedback and Iterative Design:

- Deploy an early version to a small group and gather detailed feedback.
- Implement a feedback button within the chat interface.
- Iterate on design based on qualitative and quantitative data (e.g., usage logs, survey results).

Version Control:

- Use Git to manage your frontend code.
- Maintain separate branches for major features (e.g., voice integration, dark mode).
- Ensure regular commits with descriptive messages.

5 Overall Integration and Project Management

5.1 Team Collaboration and Roles

The project is divided into three major sections. To ensure cohesive development:

a) **Model Building:** Responsible for data collection, cleaning, merging, and preparing datasets for RASA and fine-tuning.

b) **Backend Development:** Focuses on configuring RASA, developing custom actions, and integrating the fine-tuned LLM.

c) **Frontend Development:** Develops the user interface, ensuring it is accessible, user-friendly, and responsive.

Each member should ensure that their deliverables are documented and reviewed in regular team meetings. Coordination is essential to align data formats and ensure smooth API integrations between the frontend, backend, and model components.

5.2 Research and Documentation

Documentation:

- Maintain a centralized document repository (e.g., Overleaf, ShareLaTeX, or GitHub Wiki) for all design documents, scripts, and research findings.
- Document every decision, especially regarding data anonymization, ethical disclaimers, and personality tagging.
- Update the project's changelog with every major change.

Research Tasks:

1) Conduct a literature review on data augmentation techniques in NLP and mental health applications.
2) Investigate case studies and best practices in digital mental health services.
3) Review ethical guidelines related to digital mental health (e.g., GDPR, FERPA, and local ethical standards).
4) Compare various LLM fine-tuning approaches and document pros and cons.

5.3 Integration Testing and Evaluation

Testing Strategy:

- **Unit Tests:** Write unit tests for model cleaning scripts, custom action functions, and frontend components.
- **Integration Tests:** Ensure that the frontend can communicate with the backend reliably. Simulate user conversations to test end-to-end performance.
- **User Acceptance Testing (UAT):** Organize testing sessions with a sample group of students and professionals. Collect and analyze feedback.

- **Stress Testing:** Evaluate system performance under load. Use load testing tools to simulate many concurrent users.

Evaluation Metrics:

- **Intent Accuracy:** Evaluate RASA NLU performance using metrics such as precision, recall, and F1-score.
- **Response Quality:** Manually review LLM outputs for coherence, personality alignment, and factual correctness.
- **User Engagement:** Track metrics like session length, frequency of use, and user satisfaction ratings.
- **System Reliability:** Monitor uptime, error rates, and response latency.

5.4 Deployment and Maintenance

Deployment Strategy:

- Use containerization (Docker) for the backend (RASA and action server) and the frontend.
- Consider orchestration with Kubernetes or Docker Compose for multi-instance deployments.
- Implement secure channels (HTTPS/TLS) for all communications.

Maintenance Tasks:

- Set up logging and monitoring systems (e.g., ELK Stack, Prometheus) for real-time tracking.
- Regularly update dependencies and review security patches.
- Schedule periodic reviews of conversation logs (anonymized) to refine the model.

5.5 Project Management Recommendations

Team Meetings:

- Hold weekly meetings to review progress and align on integration points.
- Use collaborative tools (e.g., Trello, Asana) to track tasks and milestones.
- Encourage each sub-team to present demos of their modules at regular intervals.

Documentation and Reporting:

- Each member should maintain a detailed log of their contributions.
- Prepare interim reports summarizing progress, challenges, and next steps.
- Ensure final reports cover technical details as well as reflections on teamwork and project management.

Risk Management:

- Identify potential risks (e.g., integration delays, data privacy issues) and develop mitigation strategies.
- Assign contingency tasks for critical issues (e.g., fallback responses for the backend).

Cross-Team Integration: To ensure seamless integration:

- Model Building members must provide clear API documentation of the data formats (JSON, YAML) and any preprocessing steps.
- Backend developers should expose well-documented endpoints and error codes.
- Frontend developers need to integrate with these endpoints and handle errors gracefully.

Ethical Considerations: In all modules, emphasize:

- User privacy and consent for data storage.
- Clear crisis protocols with immediate human intervention options.
- Transparency in AI-generated responses—include disclaimers that the chatbot is not a replacement for professional therapy.

Future Work and Scalability:

- Research into adaptive learning methods for the LLM can be an ongoing project.
- Explore federated learning approaches if user data privacy becomes a critical concern.
- Plan for multi-language support and further accessibility improvements.

Final Recommendations: The success of the project depends on:

1) Detailed coordination among the three sub-teams.
2) Thorough testing at every stage—from model training to full system integration.
3) Regular updates and iterative improvements based on real user feedback.
4) Documentation of every step to support the final technical report and future research publications.

6 Conclusion and Final Recommendations

This guide provides a roadmap for the three essential components of our mental health chatbot project. In summary:

- **Model Building:**
  - Gather, clean, and merge data from diverse sources.
  - Create a robust training dataset for both RASA and fine-tuning an LLM.
  - Implement data augmentation and ensure ethical data handling.
- **Backend Development:**
  - Configure RASA with updated domain, NLU, and dialogue stories.
  - Develop custom actions, especially for crisis management and personality-based responses.
  - Ensure the system is secure, scalable, and well-tested.
- **Frontend Development:**
  - Design a user-friendly interface with options for personalization, privacy, and emergency access.
  - Integrate smoothly with the backend using asynchronous communication.
  - Prioritize accessibility and ease of use.

Each team member should adhere to these guidelines and coordinate closely. Continuous integration, thorough testing, and regular team meetings will be crucial to ensure that our final prototype meets both the technical and ethical requirements outlined in the module brief. Future iterations should build on user feedback and emerging research to refine and enhance the system.

Final Note: This document is intended as a living guide. As research progresses and new challenges emerge, update this report accordingly to reflect the latest insights and improvements.