A Major Project Report

On

ENHANCING ECG REPORT GENERATION WITH DOMAIN-SPECIFIC TOKENIZATION FOR IMPROVED MEDICAL NLP ACCURACY

Submitted to CMREC (UGC Autonomous)

In Partial Fulfilment of the requirements for the Award of Degree

of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING (AI & ML)

Submitted

By

A. SAHITH KUMAR	**(228R1A66D1)**
D. SAM SUNDAR	**(228R1A66E7)**
M. BHARATH KUMAR	**(228R1A66G9)**
RISHI YADAV	**(238R5A6618)**

Under the Esteemed guidance of

Ms. B. REVATHI

Assistant Professor, Department of CSE(AI&ML)

Department of Computer Science and Engineering(AI&ML)

CMR ENGINEERING COLLEGE

(UGC AUTONOMOUS)

(Accredited by NAAC & NBA, Approved by AICTE, New Delhi, Affiliated to JNTU, Hyderabad)
(Kandlakoya, Medchal Road, Medchal-Malkajgiri Dist., Hyderabad-501 401)

(2025-2026)

CMR ENGINEERING COLLEGE

UGC AUTONOMOUS

(Accredited by NAAC&NBA, Approved by AICTE New Delhi, Affiliated to JNTU, Hyderabad, Kandlakoya, Medchal Road, Hyderabad-501 401)

Department of Computer Science & Engineering (AI & ML)

CERTIFICATE

This is to certify that the Major project entitled **"ENHANCING ECG REPORT GENERATION WITH DOMAIN-SPECIFIC TOKENIZATION FOR IMPROVED MEDICAL NLP ACCURACY"** is a bonafide work carried out by

A. SAHITH KUMAR	**(228R1A66D1)**
D. SAM SUNDAR	**(228R1A66E7)**
M. BHARATH KUMAR	**(228R1A66G9)**
RISHI YADAV	**(238R5A6618)**

in partial fulfillment of the requirement for the award of the degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING (AI&ML) from CMR Engineering College, under our guidance and supervision.

The results presented in this Major project have been verified and are found to be satisfactory. The results embodied in this Major project have not been submitted to any other university for the award of any other degree or diploma.

Internal Guide	**Major Project Coordinator**	**Head of the Department**
Ms. B. Revathi	Mr. G. Venkateswarlu	Dr. Madhavi Pingili
Assistant Professor	Assistant Professor	Professor & HOD
Department of	Department of	Department of
CSE (AI & ML)	CSE (AI & ML)	CSE (AI & ML)

External Examiner:_____

DECLARATION

This is to certify that the work reported in the present Major project entitled "**ENHANCING ECG REPORT GENERATION WITH DOMAIN-SPECIFIC TOKENIZATION FOR IMPROVED MEDICAL NLP ACCURACY**" is a record of bonafide work done by us in the Department of Comput er Science and Engineering (AI & ML), CMR Engineering College. The reports are based on the Major project work done entirely by us and not copied from any other source. We submit our Major project for further development by any interested students who share similar interests to improve the Major project in the future.

The results embodied in this Major project report have not been submitted to any other University or Institute for the award of any degree or diploma to the best of our knowledge and belief.

A. SAHITH KUMAR (228R1A66D1)
D. SAM SUNDAR (228R1A66E7)
M. BHARATH KUMAR (228R1A66G9)
RISHI YADAV (238R5A6618)

ACKNOWLEDGEMENT

We are extremely grateful to **Dr. A. Srinivasula Reddy**, Principal and **Dr. Madhavi Pingili**, Professor & HOD, Department of CSE(AI&ML), CMR Engineering College for their constant support.

We are extremely thankful to **Ms. B. Revathi,** Assistant Professor, Internal Guide, Department of CSE(AI&ML), for her constant guidance, encouragement and moral support throughout the Major project.

We will be failing in duty if we do not acknowledge with grateful thanks to the authors of the references and other literatures referred in this Major project.

We thank **Mr. G. Venkateswarlu,** Major Project Coordinator for his constant support in carrying out the Major project activities and reviews.

We express our thanks to all staff members and friends for all the help and co-ordination extended in bringing out this Major project successfully in time.

Finally, We are very much thankful to our parents who guided us for every step.

A. SAHITH KUMAR (228R1A66D1)
D. SAM SUNDAR (228R1A66E7)
M. BHARATH KUMAR (228R1A66G9)
RISHI YADAV (238R5A6618)

CONTENTS

ABSTRACT

ABSTRACT

The automation of medical report generation has become an important research area, driven by advancements in computational techniques and natural language processing (NLP). Early approaches primarily relied on rule-based and template-driven systems to standardize clinical documentation, but these methods lacked flexibility and struggled to handle complex or diverse medical scenarios. With the rise of deep learning and generative AI, especially transformer-based models like GPT-2, automated reporting has evolved significantly, enabling more context-aware and human-like text generation. This progress has made automated medical reporting a valuable tool for assisting healthcare professionals and improving diagnostic workflows. Despite these advancements, a major limitation of current models is their dependence on pre-trained, general-purpose tokenizers, which are not well-suited for capturing specialized medical terminology. This often results in reduced coherence, lack of contextual relevance, and incorrect usage of clinical terms in generated reports. Therefore, addressing the gap between general-purpose language models and domain-specific requirements is crucial for improving performance.

Keywords: Fine-tuning, GPT-2, ECG report generation, custom tokenizer.

LIST OF FIGURES

LIST OF TABLES

CHAPTER-1

INTRODUCTION

1. INTRODUCTION

1.1 Introduction

The generation of medical reports has become an essential task in modern healthcare [1], driven by the need to streamline documentation and support clinicians in diagnostic workflows. Traditional methods, such as rule-based or template-driven systems [4], [3], often fall short in handling the complexity and variability of medical language. With the emergence of generative AI models, however, the possibilities for automated medical report generation have expanded, offering more adaptive and context-aware solutions.

Recent advancements in natural language processing (NLP) have demonstrated remarkable success across domains, with models like BERT [8], GPT-2 [5], and other large-scale pre-trained transformers excelling at a wide range of general language tasks. This progress underscores a shift from rigid, template-based systems to dynamic, AI-driven approaches in report generation [6]. As illustrated in Figure 1, AI-driven methods enable more flexible, context-sensitive, and potentially more insightful reports by leveraging data analysis and machine learning.

Nevertheless, applying these models directly to specialized fields such as medical language, particularly in biomedical reporting tasks like ECG interpretation, presents challenges. Without domain-specific fine-tuning or adaptation, their performance often remains limited [12].

1.2 Project Objectives

By the end of this project you will understand

1. Develop a domain-specific tokenizer trained exclusively on ECG report datasets to effectively capture and process specialized medical terminology.
2. Integrate the custom tokenizer into the GPT-2 model to enhance the quality, accuracy, and contextual relevance of medical text generation.
3. Evaluate the tokenizer’s performance against general-purpose tokenizers using BLEU, ROUGE, and perplexity to measure improvements in generation quality and linguistic fidelity.

1.3 Purpose of the Project

The main purpose of this research is to improve the accuracy, coherence, and clinical reliability of automated ECG (Electrocardiogram) report generation by developing a domain-specific tokenizer tailored for medical language processing [2]. Existing transformer-based language models such as GPT-2 and BERT rely on general-purpose tokenization strategies that are not optimized for specialized ECG terminology, abbreviations, waveform descriptors, and structured diagnostic expressions. Ultimately, this approach enhances the effectiveness of automated ECG reporting systems and supports healthcare professionals in faster and more reliable cardiac assessment workflows. As a result, these models often generate fragmented interpretations, lose contextual relationships between waveform features and clinical findings, and produce reports that lack medical precision. To overcome this limitation, the proposed system introduces a custom ECG-aware tokenizer that better captures domain-specific vocabulary such as rhythm patterns, interval variations, and morphological abnormalities. This improves the semantic alignment between ECG signal features and generated textual interpretations, leading to more consistent, interpretable, and clinically meaningful diagnostic reports. Ultimately, this approach enhances the effectiveness of automated ECG reporting systems and supports healthcare professionals in faster and more reliable cardiac assessment workflows [7].

1.4 Problem Statement

Electrocardiogram (ECG) interpretation is essential for detecting cardiac abnormalities, but manual analysis is time-consuming and requires expert knowledge. Existing automated ECG reporting systems rely on rule-based approaches or general-purpose language models that fail to capture domain-specific medical terminology accurately. This leads to reduced coherence and reliability in generated diagnostic reports. Therefore, there is a need for an intelligent ECG report generation system that integrates waveform analysis with a domain-specific tokenizer and fine-tuned GPT-2 model to produce accurate, context-aware, and clinically meaningful interpretations.

1.5 Existing System

The existing system focuses on generating automated ECG reports using an advanced medical NLP framework built around domain-specific tokenization. At its core, the system introduces a Domain-Specific Tokenizer (DST) designed to accurately process ECG-related terminology, abbreviations, and structured medical expressions that general-purpose tokenizers fail to capture. This tokenizer is integrated into an encoder–decoder transformer architecture, which converts ECG signal data into coherent and clinically meaningful textual reports [9]. The model undergoes pretraining on large ECG datasets and is later fine-tuned using real clinical reports, allowing it to learn both linguistic

patterns and medically relevant context [17]. Its performance is evaluated using BLEU, ROUGE, and medical relevance scores, consistently outperforming baseline models such as BERT or GPT when used without domain-level token adaptation. Through this approach, the system significantly improves the syntactic quality, coherence, and clinical accuracy of generated ECG interpretations, demonstrating the effectiveness of specialized tokenization in medical NLP applications [11].

Disadvantages

- Limited multimodal capability, as the system relies only on ECG signals without using patient history or additional clinical data.
- Heavy dependence on large, annotated ECG–text datasets, which are difficult and costly to obtain.
- Static domain-specific vocabulary that may not adapt to new medical terms or institution-specific abbreviations.
- Low interpretability because the model generates text without visual explanations or feature attribution.
- Standard NLP metrics like BLEU and ROUGE do not fully reflect clinical accuracy or safety-critical correctness.

1.6 Proposed System

We propose a Multimodal Contextualized ECG Report Generator (MCE-RG) to overcome the current system's limitations [10]. This system extends the original architecture by integrating electronic health record (EHR) context, such as patient history, age, symptoms, and previous diagnoses, into the report generation process. The model uses a multimodal encoder that processes both time-series ECG signals and structured/unstructured clinical data [19]. A dynamic vocabulary adapter replaces the static domain tokenizer, allowing real-time adaptation to new terminology and personalized abbreviation sets used across hospitals. Furthermore, the decoder is augmented with an attention-based explainability module, which highlights the regions of the ECG waveform responsible for specific generated text segments [14]. Training is semi-supervised, utilizing pseudo-labeled data and contrastive learning to reduce dependence on annotated corpora. This system aims to offer clinically robust, adaptable, and transparent ECG interpretations suited for practical deployment in diverse hospital environments [13].

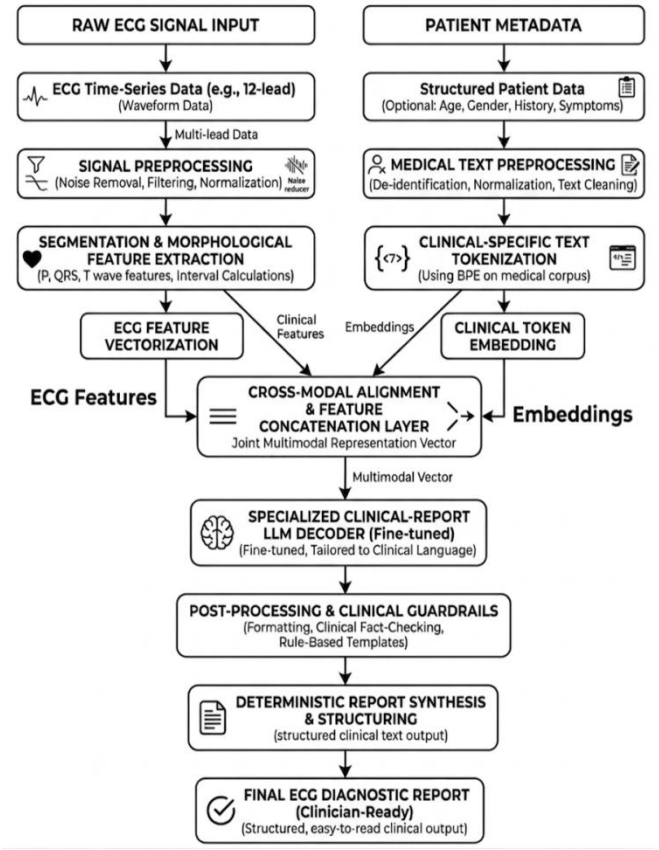


Fig 1.6.1: Block diagram of proposed system.

Advantages

- The system gains deeper clinical insight by combining ECG signals with patient EHR data, enabling more accurate and context-aware diagnostic interpretations.
- It reduces dependence on large annotated datasets through semi-supervised and contrastive learning, allowing effective training even with limited labeled ECG reports.
- Interpretability is improved using attention-based explanation modules that highlight the most important ECG patterns or textual cues influencing the model's output.

1.7 Input and Output Design

1.7.1 Input Design

Input design defines the structure and format of the data that the proposed ECG report generation system will process. The system primarily accepts raw ECG waveform signals, which consist of time-series voltage values representing cardiac electrical activity. These signals may vary in sampling rate, amplitude, and noise levels, so the input layer is structured to handle waveform irregularities, grid variations, and metadata such as patient age or heart rate.

To prepare the data for downstream processing, the input design outlines the necessary transformations required before the model receives the waveform. These include noise filtering, signal normalization, segmentation of cardiac cycles, and conversion of waveform values into model-compatible numerical arrays. When clinical text metadata is provided, it undergoes token structuring and standardization for seamless integration with the custom tokenizer. By establishing clear formatting rules and preprocessing steps, the system ensures reliability, consistency, and smooth flow of input data into the feature extraction and ECG report generation modules.

Objectives

1. To define a consistent and structured format for receiving raw ECG waveform data, ensuring that all signals and associated patient metadata are accurately captured and prepared for preprocessing.
2. To ensure the input ECG data is free from noise, distortion, and irrelevant artifacts, enabling smooth transition into signal normalization, segmentation, tokenization, and subsequent feature extraction stages.
3. To support multi-class and multi-label ECG interpretation inputs, allowing the system to handle diverse cardiac conditions, diagnostic labels, and clinical annotations during the report generation process.

1.7.2 Output Design

Output design defines the structure, format, and presentation of the results generated by the proposed ECG report generation system. Since the system processes raw ECG waveform signals and converts them into diagnostic text using a GPT-based model, the output is designed to provide clear, interpretable, and clinically meaningful information. The primary output consists of an automatically generated ECG report describing waveform patterns, rhythm observations, interval measurements, and possible cardiac abnormalities. In multi-label diagnostic scenarios, the design supports generating multiple clinical findings when the ECG indicates overlapping conditions.

To ensure clarity and medical usefulness, the output design presents results in a structured format that resembles standard clinical reporting conventions. Each generated statement corresponds to a specific ECG feature or diagnosis, and may include severity descriptions or confidence indicators depending on model configuration. The design also allows inclusion of probability scores or model-generated confidence levels to help clinicians assess the reliability of interpretations.

CHAPTER-2

LITERATURE SURVEY

2 LITERATURE SURVEY

1. **D. Siva Raja Kumar, B. Maram, and P. R. Kshirsagar, "Deep Belief Parallel Forward Harmonic Network for Cardiovascular Disease Detection Using ECG Images," The European Physical Journal Plus, vol. 140, Art. no. 1113, 2026.** The study introduces a deep belief network for ECG image-based disease detection. It improves classification accuracy through parallel feature learning, but lacks multimodal integration with clinical data.

2. **S. Rasheeduddin, N. Venkatesh, G. Venkateswarulu, and B. Sai Kumar, "A Smart Approach for Monitoring Health and Diseases Through AI-Driven Technologies," in Proc. Int. Conf., 2025.** This paper presents an AI-based system for health monitoring using smart devices. While effective for general health tracking, it does not focus deeply on ECG-specific report generation or clinical reasoning.

3. **S. Rasheeduddin and U. Nagaiah, "Smart Devices for Monitoring Health and Disease Using Artificial Intelligence," in Proc. Two-Day National Seminar, 2025.** The authors discuss AI-enabled smart healthcare devices for disease monitoring. The approach supports continuous data collection but lacks advanced deep learning models for detailed ECG interpretation.

4. **E. Parvathi, "A Study on the Development and Application of an Arduino-Based ECG Monitoring Using the AD8232 Sensor," in Lecture Notes in Networks and Systems (LINNS), vol. 1363, pp. 309–318, 2025.** This work focuses on low-cost ECG acquisition using Arduino and AD8232 sensors. While useful for data collection, it does not address automated report generation or advanced analytics.

5. **K. Ramesh, "Automated Classification and Diagnosis of Focal and Non-Focal EEG Signals Using a Hybrid Classification Approach," in Proc. Int. Conf., 2025**. The study proposes a hybrid model for EEG signal classification. Although effective in signal classification, it is limited to EEG and does not incorporate multimodal ECG-based report generation.

6. **H. Jin, S. Cui, and C. Lian, "PromptMRG: Diagnosis-Driven Prompts for Medical Report Generation," in Proc. AAAI Conf. Artificial Intelligence, vol. 38, no. 3, pp. 2607–2615, 2024.** This paper introduces prompt-based report generation guided by diagnostic signals. It improves clinical relevance but struggles with aligning temporal medical signals like ECG with textual output.

7. **G. Fu, J. Xu, and T. Zhang, "CardioGPT: An ECG Interpretation Generation Model," IEEE Access, vol. 12, pp. 50254–50264, 2024**. The authors present a GPT-based model for ECG interpretation. It captures long-range dependencies effectively but requires large datasets and lacks explainability mechanisms.

8. **Y. Zhao, S. Lu, and K. Wang, "ECG-Chat: A Large ECG-Language Model for Cardiac Disease Diagnosis," arXiv preprint arXiv:2408.08849, 2024.** This study develops a conversational multimodal ECG model. It enhances interaction and interpretability but may generate inconsistent outputs under noisy signal conditions.

9. **A. Bleich, D. Miller, and J. Wagner, "Automated Medical Report Generation for ECG Data: Bridging Medical Text and Signal Processing," arXiv preprint arXiv:2412.04067, 2024.** This work focuses on bridging ECG signals with clinical text generation. It improves cross-modal alignment but is limited by the absence of domain-specific tokenization techniques.

10. **X. Liu, Z. Wang, and L. Zhou, "Automatic Medical Report Generation Based on Deep Learning: A Survey," Computerized Medical Imaging and Graphics, vol. 108, 2024**. This survey provides a comprehensive overview of deep learning techniques for medical report generation. It highlights advancements in image-to-text models and evaluation strategies, but lacks focus on domain-specific tokenization for specialized applications like ECG report generation.

Focused Area / Title	**Key Findings**	**Reference**
Deep Belief Network for ECG Image Analysis. [1]	Introduces a deep belief parallel network for ECG image-based disease detection. Improves classification accuracy but lacks multimodal integration with clinical data.	D. Siva Raja Kumar, B. Maram, and P. R. Kshirsagar, "Deep Belief Parallel Forward Harmonic Network for Cardiovascular Disease Detection Using ECG Images," *The European Physical Journal Plus*, vol. 140, Art. no. 1113, 2026.
AI-Based Smart Health Monitoring System. [2]	Presents an AI-driven system for health monitoring using smart devices. Effective for general monitoring but lacks focus on ECG-specific report generation.	S. Rasheeduddin, N. Venkatesh, G. Venkateswarulu, and B. Sai Kumar, "A Smart Approach for Monitoring Health and Diseases Through AI-Driven Technologies," in *Proc. Int. Conf.*, 2025.
AI-Enabled Smart Healthcare Devices. [3]	Discusses AI-based smart devices for continuous disease monitoring. Supports data collection but lacks advanced ECG interpretation models.	S. Rasheeduddin and U. Nagaiah, "Smart Devices for Monitoring Health and Disease Using Artificial Intelligence," in *Proc. Two-Day National Seminar*, 2025.
Arduino-Based ECG Monitoring System. [4]	Focuses on low-cost ECG acquisition using Arduino and AD8232 sensor. Useful for data collection but lacks automated report generation.	E. Parvathi, "A Study on the Development and Application of an Arduino-Based ECG Monitoring Using the AD8232 Sensor," in *Lecture Notes in Networks and Systems (LINNS)*, vol. 1363, pp. 309–318, 2025.
Hybrid EEG Signal Classification Model [5]	Proposes a hybrid approach for EEG signal classification. Effective for EEG but not applicable to multimodal ECG report generation.	K. Ramesh, "Automated Classification and Diagnosis of Focal and Non-Focal EEG Signals Using a Hybrid Classification Approach," in *Proc. Int. Conf.*, 2025.

Table no. 2 Literature Review Summary

Focused Area / Title	**Key Findings**	**Reference**
Prompt-Based Medical Report Generation [6]	Introduces prompt-driven report generation improving clinical relevance. Struggles with aligning ECG temporal signals with text output.	H. Jin, S. Cui, and C. Lian, "PromptMRG: Diagnosis-Driven Prompts for Medical Report Generation," in *Proc. AAAI Conf. Artificial Intelligence*, vol. 38, no. 3, pp. 2607–2615, 2024.
CardioGPT ECG Interpretation Model [7]	Presents a GPT-based model for ECG interpretation. Captures long-range dependencies but requires large datasets and lacks explainability.	G. Fu, J. Xu, and T. Zhang, "CardioGPT: An ECG Interpretation Generation Model," *IEEE Access*, vol. 12, pp. 50254–50264, 2024.
ECG-Chat Multimodal Model [8]	Develops a conversational multimodal ECG model for diagnosis. Enhances interaction but may produce inconsistent outputs under noisy conditions.	Y. Zhao, S. Lu, and K. Wang, "ECG-Chat: A Large ECG-Language Model for Cardiac Disease Diagnosis," *arXiv preprint arXiv:2408.08849,* 2024.
ECG Signal-to-Text Generation Framework [9]	Focuses on bridging ECG signals with clinical text generation. Improves cross-modal alignment but lacks domain-specific tokenization.	A. Bleich, D. Miller, and J. Wagner, "Automated Medical Report Generation for ECG Data: Bridging Medical Text and Signal Processing," *arXiv preprint arXiv:2412.04067,* 2024.
Automatic Medical Report Generation Based on Deep Learning [10]	Provides a comprehensive overview of deep learning techniques for medical report generation, highlighting advancements in image-to-text models and evaluation strategies.	X. Liu, Z. Wang, and L. Zhou, "Automatic Medical Report Generation Based on Deep Learning: A Survey," *Computerized Medical Imaging and Graphics*, vol. 108, 2024.

Table no. 2 Literature Review Summary

CHAPTER-3

SOFTWARE REQUIREMENT ANALYSIS

3. SOFTWARE REQUREIMENTS ANALYSIS

3.1 Modules and Their Functionalities

3.1.1 Data Analysis

Data analysis was conducted to examine the structure, composition, and characteristics of the ECG datasets used for automated report generation. The datasets consist of multi-lead ECG waveforms that vary significantly in length, sampling frequency, and morphological patterns. These signals often include variations in P-wave, QRS complex, and ST-segment characteristics, which are critical for accurate cardiac interpretation. Additionally, the data may contain multiple diagnostic categories, with a noticeable class imbalance where certain cardiac conditions occur much less frequently than others.

The datasets also include various forms of noise and signal distortions, such as baseline drift, motion artifacts, and powerline interference, which can affect the quality and reliability of waveform analysis. These factors necessitate systematic preprocessing steps, including normalization, denoising, segmentation, and signal alignment, to ensure consistency and improve feature quality for downstream tasks.

3.1.2 Data Preprocessing

Data preprocessing is carried out to convert raw ECG waveforms into a clean, consistent, and standardized format suitable for accurate analysis and report generation. Raw ECG signals are often affected by noise and artifacts such as baseline drift, powerline interference, and muscle or motion disturbances, which can obscure important cardiac features. To address this, filtering techniques like low-pass, high-pass, and band-pass filters are applied to remove unwanted noise while preserving essential signal components. The signals are then normalized to ensure uniform amplitude scaling across recordings, improving consistency and model performance.

Following this, the ECG signals are segmented into meaningful cardiac cycles by identifying key components such as R-peaks. This enables focused analysis of individual heartbeats and important waveform features like P-waves, QRS complexes, and T-waves. Additional steps such as resampling and alignment may be used to standardize signal length and timing variations. These preprocessing steps collectively enhance signal quality and reduce variability, ensuring that the processed data provides a reliable foundation for feature extraction and accurate automated ECG report generation.

3.1.3 Machine Learning Algorithm for Prediction

The proposed system uses a GPT-based deep learning model to improve the accuracy and clarity of ECG report generation. The design combines an ECG waveform encoder with a domain-specific GPT-2 decoder, enabling the system to capture detailed morphological features and translate them into coherent medical descriptions. The encoder and decoder operate together to produce rhythm interpretations, interval observations, and diagnostic findings. This architecture supports multi-label cardiac conditions and generates clinically meaningful textual reports for diverse ECG patterns.

3.2 Functional Requirements

The Functional requirements for a system describe the functionality or the services that the system is expected to provide. These are the statements of services the system should provide and how the system should react to particular inputs and how the system should behave in particular situation.

1. The system shall accept raw ECG waveform inputs from the user.
2. The system shall perform signal cleaning, normalization, and feature extraction.
3. The system shall generate diagnostic ECG reports based on waveform patterns.

3.3 Non-Functional Requirements

Non-functional requirements define the quality attributes and performance expectations that the ECG report generation system must meet to ensure reliability and usability. These requirements describe how the system should behave under various operational conditions and help maintain consistency, scalability, and efficiency throughout the system's lifecycle. The following points summarize the essential non-functional expectations of the proposed framework:

1. The system shall ensure high reliability and stability during ECG processing.
2. The system shall provide scalable performance as ECG data volume increases.
3. The system shall maintain efficient processing with minimal latency in report generation.
4. The system shall ensure privacy and confidentiality of patient ECG data.

3.4 Feasibility Study

The feasibility study evaluates whether the proposed ECG report generation system can be practically developed and deployed by considering technical and operational aspects. From a technical perspective, the system utilizes established ECG preprocessing techniques such as noise removal, normalization, and segmentation, along with advanced deep learning models like GPT-2 for

waveform-to-text generation. Modern frameworks such as TensorFlow and PyTorch provide strong support for implementation, and publicly available datasets like MIMIC-IV PhysioNet ensure sufficient data for training and evaluation. The availability of GPU-based computation and cloud platforms further strengthens the system's technical feasibility.

From an operational and economic standpoint, the system is designed to integrate seamlessly with existing clinical workflows, enabling efficient and automated ECG report generation. Its modular and scalable architecture allows deployment in real-time healthcare environments while reducing the workload on clinicians. Additionally, the use of open-source tools and datasets minimizes development and operational costs, making the system cost-effective.

1.Economic Feasibility

2.Technical Feasibility

3.Social Feasibility

3.4.1 Economic Feasibility

Economic feasibility assesses whether the system can be developed within acceptable costs. The proposed design uses open-source frameworks, freely available ECG datasets, and standard hardware, keeping expenses minimal. No specialized equipment is required, making it economically viable for academic, research, and clinical environments.

3.4.2 Technical Feasibility

This study is carried out to check the technical feasibility, that is, the technical requirements of the system. Any system developed must not have a high demand on the available technical resources. This will lead to high demands on the available technical resources. This will lead to high demands being placed on the client. The developed system must have a modest requirement, as only minimal or null changes are required for implementing this system.

3.4.3 Social Feasibility

The aspect of study is to check the level of acceptance of the system by the user. This includes the process of training the user to use the system efficiently. The level of acceptance by the users solely depends on the methods that are employed to educate the user about the system and to make him familiar with it. His level of confidence must be raised so that he is also able to make some constructive criticism, which is welcomed, as he is the final user of the system.

CHAPTER-4

SYSTEM SPECIFICATIONS

4. SYSTEM SPECIFICATIONS

4.1 Software Requirements

The software requirements outline the essential tools and platforms needed to design and later implement the cyberbullying detection system. The project relies on a stable programming environment, standard NLP libraries, and general-purpose utilities that support preprocessing, analysis, and machine-learning workflows. These tools ensure compatibility with future development and easy scalability during the implementation phase.

- Operating System: Windows / Linux / macOS
- Programming Language: Python 3.x
- Core Libraries: NLTK, Scikit-learn, NumPy, Pandas
- Development Environment: VS Code / PyCharm / Jupyter Notebook
- Documentation Tools: MS Word / LaTeX

4.2 Hardware Requirements

The hardware requirements define the minimum computational resources necessary for processing datasets, executing preprocessing tasks, and training machine-learning models. The system operates efficiently on standard computing hardware, including a multi-core processor, sufficient RAM for dataset handling, and moderate storage capacity for model files and datasets. The configuration remains scalable, allowing additional resources to be incorporated if the system is extended to larger datasets or real-time deployment scenarios.

- Processor: Intel Core i3/i5 or equivalent
- Memory (RAM): Minimum 8 GB
- Storage: 250 GB HDD/SSD
- Display: Standard 14" or higher
- Optional: Internet connectivity for dataset access and future cloud integration.

CHAPTER-5

SOFTWARE DESIGN

5. SOFTWARE DESIGN

5.1 System Architecture

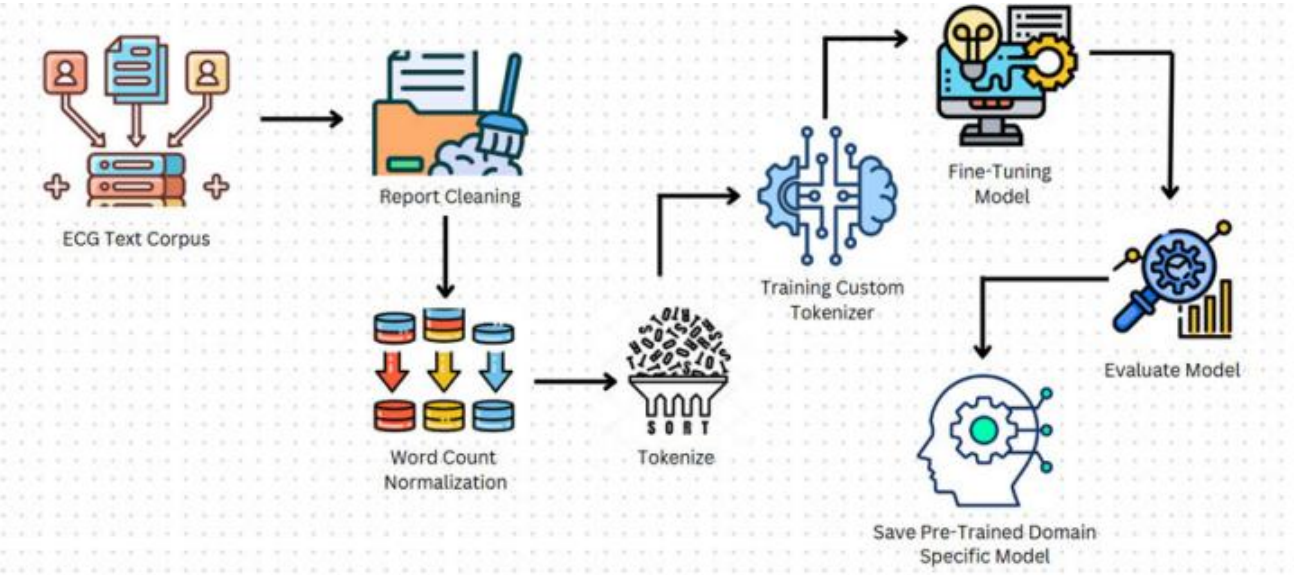


Fig:5.1 System Architecture

The proposed system architecture for domain-specific ECG report generation is designed as a multi-stage pipeline that converts raw clinical text into coherent diagnostic reports. It begins with the collection of a specialized ECG corpus containing expert interpretations derived from multi-lead waveforms. This dataset provides the necessary clinical terminology and reporting structure for model training. The collected reports are then cleaned by removing noise such as irrelevant symbols, formatting inconsistencies, and unnecessary characters. After cleaning, word count normalization is applied to standardize sequence length, where longer texts are truncated and shorter ones are padded to ensure compatibility with transformer-based models.

Following preprocessing, a custom Byte Pair Encoding (BPE) tokenizer is trained on the ECG corpus to accurately capture domain-specific terminology like “QRS widening” and “ST-segment elevation.” This tokenizer enhances the model’s understanding of medical language compared to general-purpose tokenizers. The tokenized data is then used to fine-tune a pre-trained GPT-2 model, enabling it to learn clinical context, grammar, and diagnostic flow. As a result, the system can generate accurate, coherent, and clinically relevant ECG reports that closely resemble expert-written interpretations.

5.2 Dataflow Diagram

The data flow representation provides a clear visualization of how information moves through the proposed ECG report generation system. It illustrates the sequence in which data enters the system, undergoes processing, and is transformed into meaningful diagnostic output. This representation highlights the key components involved in the workflow, making it easier to understand the overall structure without focusing on implementation details.

The model includes external entities such as clinicians or input sources that supply ECG waveform or report text. These inputs pass through different processes, which handle tasks like preprocessing, tokenization, model inference, and output generation. Each process transforms the incoming data into a more refined form, preparing it for the next stage of the system pipeline.

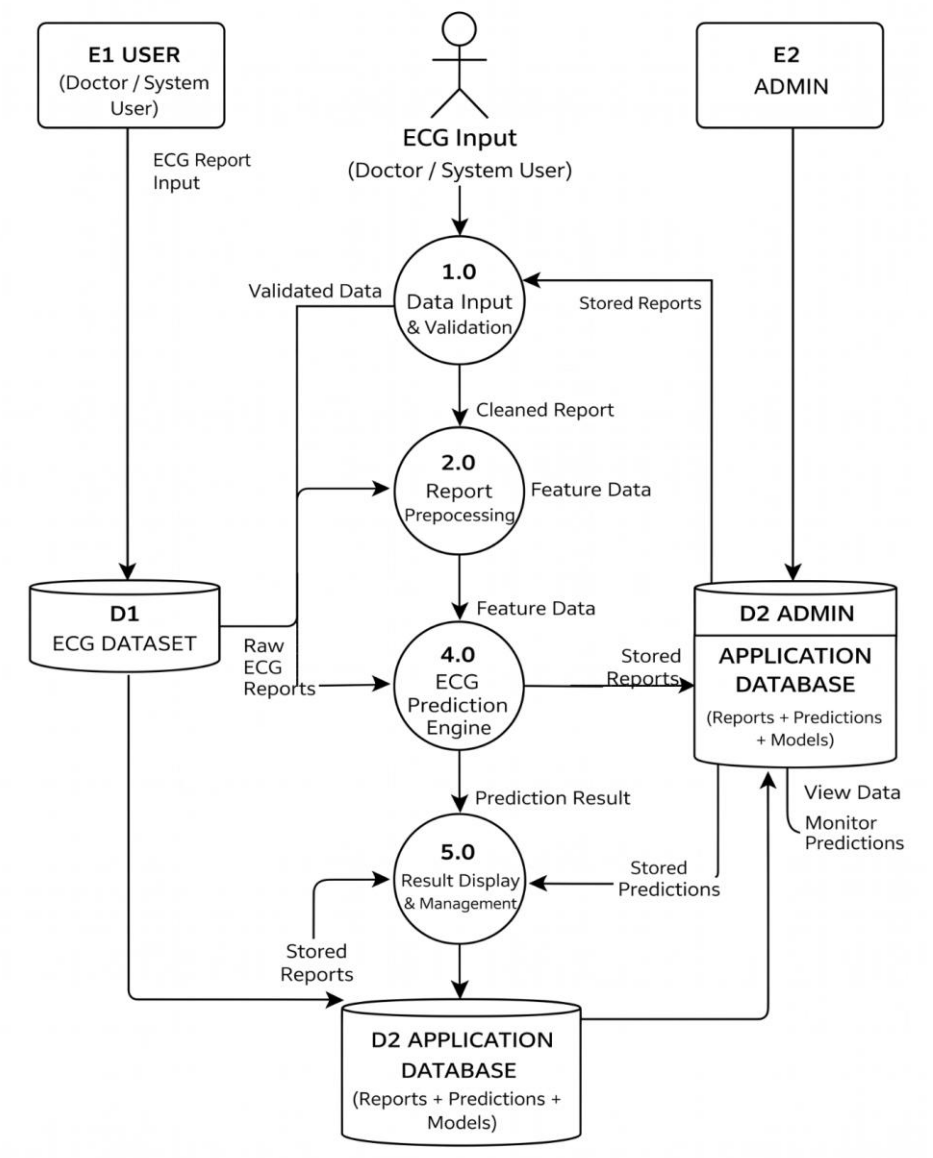


Fig 5.2 Dataflow Diagram

5.3 UML Diagrams

UML (Unified Modeling Language) is a standardized language used for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML helps in representing the design of systems and understanding their components. Created by the Object Management Group (OMG), UML 1.0 was proposed in January 1997. UML is closely associated with object-oriented analysis and design. The two main categories of UML diagrams are Behavioral and Structural diagrams, each serving distinct purposes in the modeling process.

The Behavioral UML diagrams describe the behavior of the system, its actors, and the interaction between the components. On the other hand, Structural UML diagrams depict the static structure of the system, showing its components and relationships. UML has been integrated as a standard by OMG, and its primary goals are to provide a formal basis for understanding modeling languages, offer a ready-to-use expressive language for system developers, and encourage the growth of object-oriented tools.

Goals of UML:

- Provide an expressive visual modeling language for developing and exchanging meaningful models.
- Establish a formal basis for understanding the modeling language.
- Encourage the growth of object-oriented tools.
- Integrate best practices into system development.

Types of UML Diagrams:

1. Sequence Diagram
2. Use Case Diagram
3. Activity Diagram
4. Class Diagram

5.3.1. Sequence Diagram

The sequence diagram illustrates the interaction flow among the User, ECG Data Source, Preprocessing Module, Model, and Evaluation components during the ECG report generation process. The sequence begins when the user uploads or selects an ECG waveform, after which the system retrieves the signal and prepares it for processing. The waveform is passed to the preprocessing module, where noise filtering, normalization, and segmentation operations are applied to ensure clean and consistent input for the model.

After preprocessing, the processed ECG data is forwarded to the GPT-based report generation model. The model encodes the waveform features, applies the domain-specific tokenizer, and generates a structured diagnostic report based on learned ECG patterns. Once the textual output is produced, the system sends it to the evaluation component, which analyzes the report quality using metrics such as BLEU, ROUGE, and coherence-based scoring.

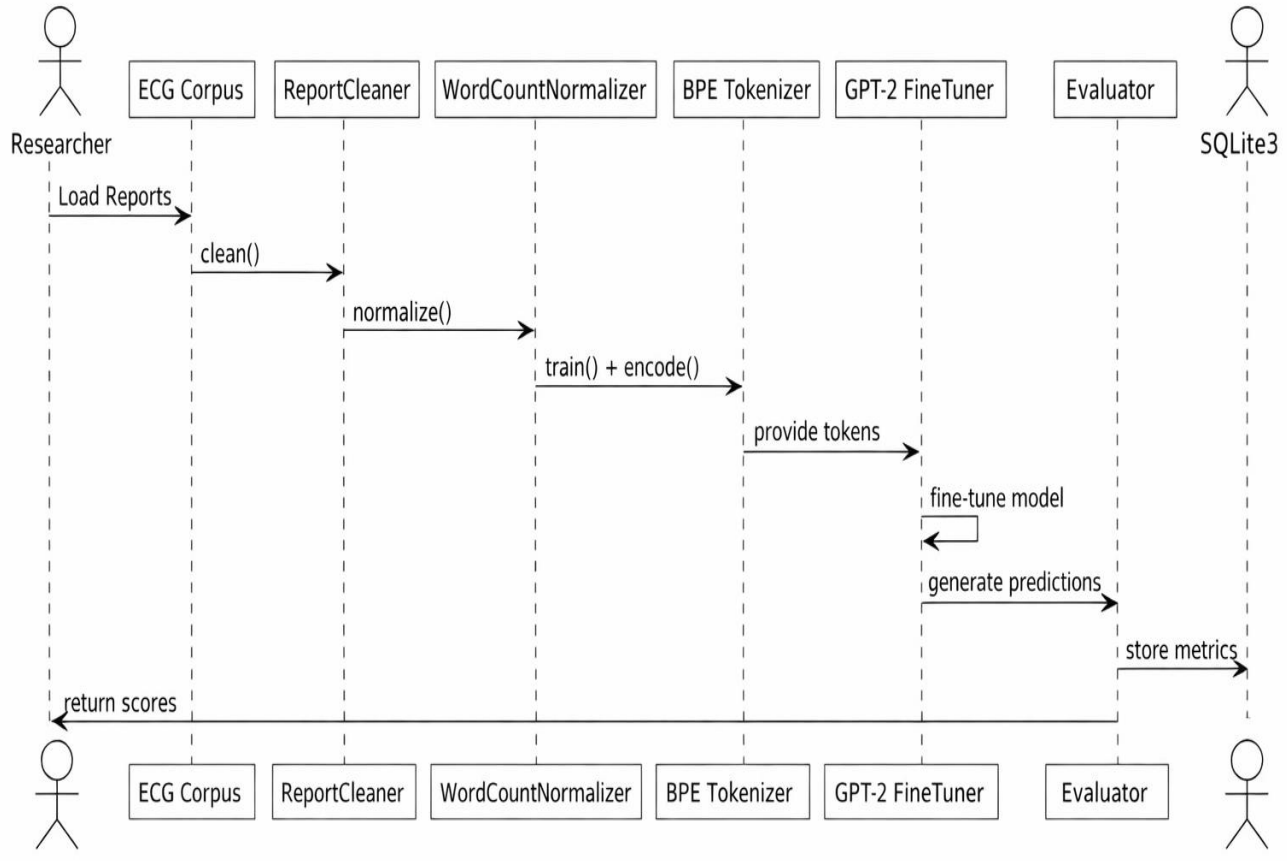


Fig 5.3.1: Sequence Diagram

List of actions

- **User:**

The user interacts with the system by uploading an ECG waveform or selecting an ECG file for analysis. They initiate the report generation process by submitting the input signal to the system.

- **System:**

It performs basic checks, organizes the data, and forwards it to the ECG preprocessing and model pipeline.

- **Model:**

The encoded waveform is then passed to the fine-tuned GPT-based report generator, which produces the diagnostic text using the domain-specific tokenizer.

- **Evaluation:**

The evaluation component analyzes the generated report for accuracy, coherence, and medical relevance.

5.3.2 Use Case Diagram

The use case diagram illustrates the interaction among the User, Clinician/Researcher, Database, and System within the domain-specific ECG report generation framework. The user interacts with the system by uploading ECG waveform data or selecting an existing ECG record for analysis. The clinician or researcher performs tasks such as preparing the ECG dataset, configuring preprocessing steps, training the tokenizer, and fine-tuning the GPT-based model to improve report accuracy.

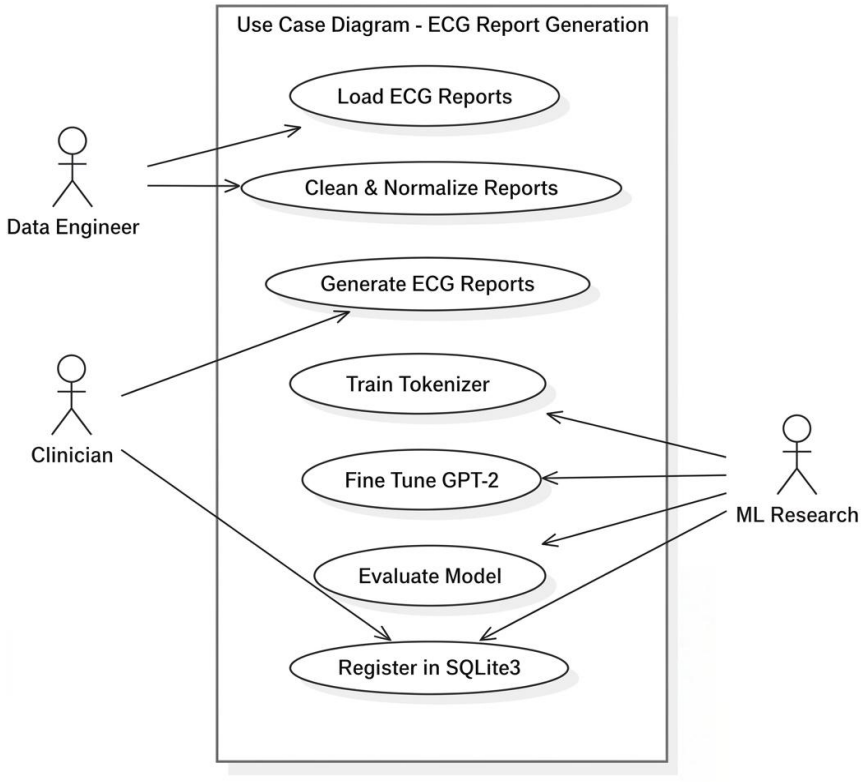


Fig 5.3.2 Use Case Diagram

5.3.3 Activity Diagram

The activity diagram describes the workflow of the ECG report generation process. It begins with uploading or collecting ECG waveform data, which then moves into preprocessing where noise filtering, signal normalization, and segmentation are applied. The cleaned waveform proceeds to the feature extraction stage, where the encoder captures morphological details such as P-wave, QRS, and T-wave patterns. The processed data is then passed to the GPT-based model, which generates a structured diagnostic report using the domain-specific tokenizer. After generation, the system evaluates the output using established metrics, analyzes the medical coherence and accuracy, and stores the final report. The workflow concludes with presenting the clinically relevant ECG interpretation to the user.

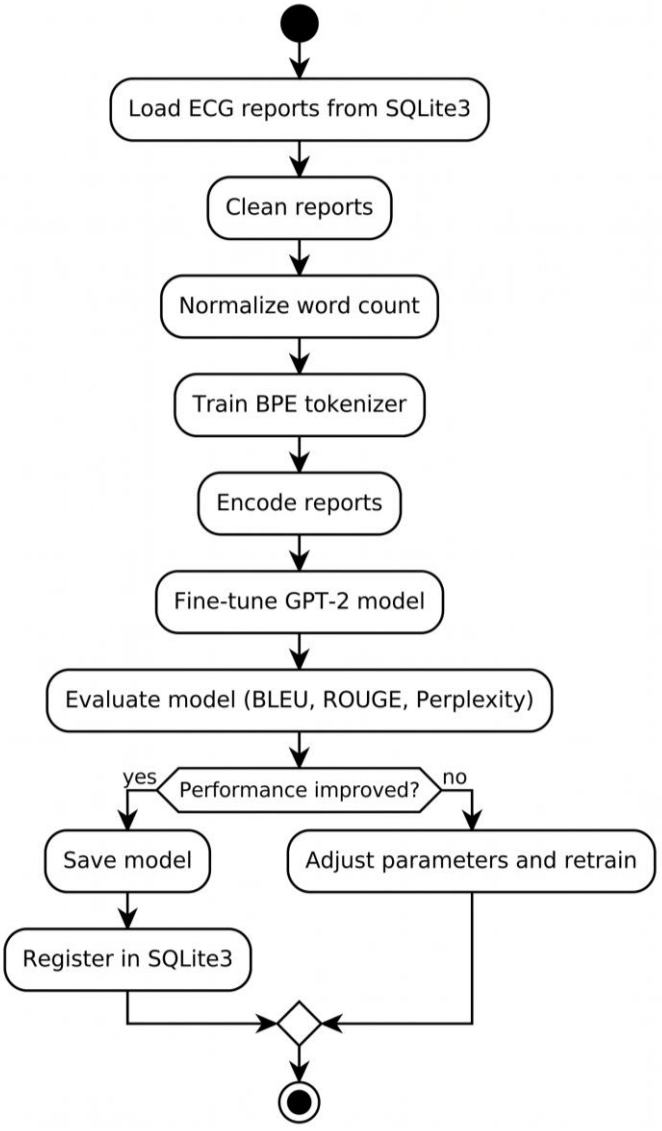


Fig 5.3.3 Activity Diagram

5.3.4. Class Diagram

The class diagram represents the structural components of the ECG report generation system and illustrates how they interact to process ECG data and produce diagnostic outputs. The User class stores essential user information and provides methods for registration, login, and submitting ECG waveform files or inputs. The ECG Dataset class holds the ECG signals along with their corresponding diagnostic annotations, serving as the foundation for training and evaluating the model. The Preprocessing class contains methods for filtering noise, normalizing waveform amplitudes, and segmenting ECG signals, ensuring that the data is clean and properly structured.

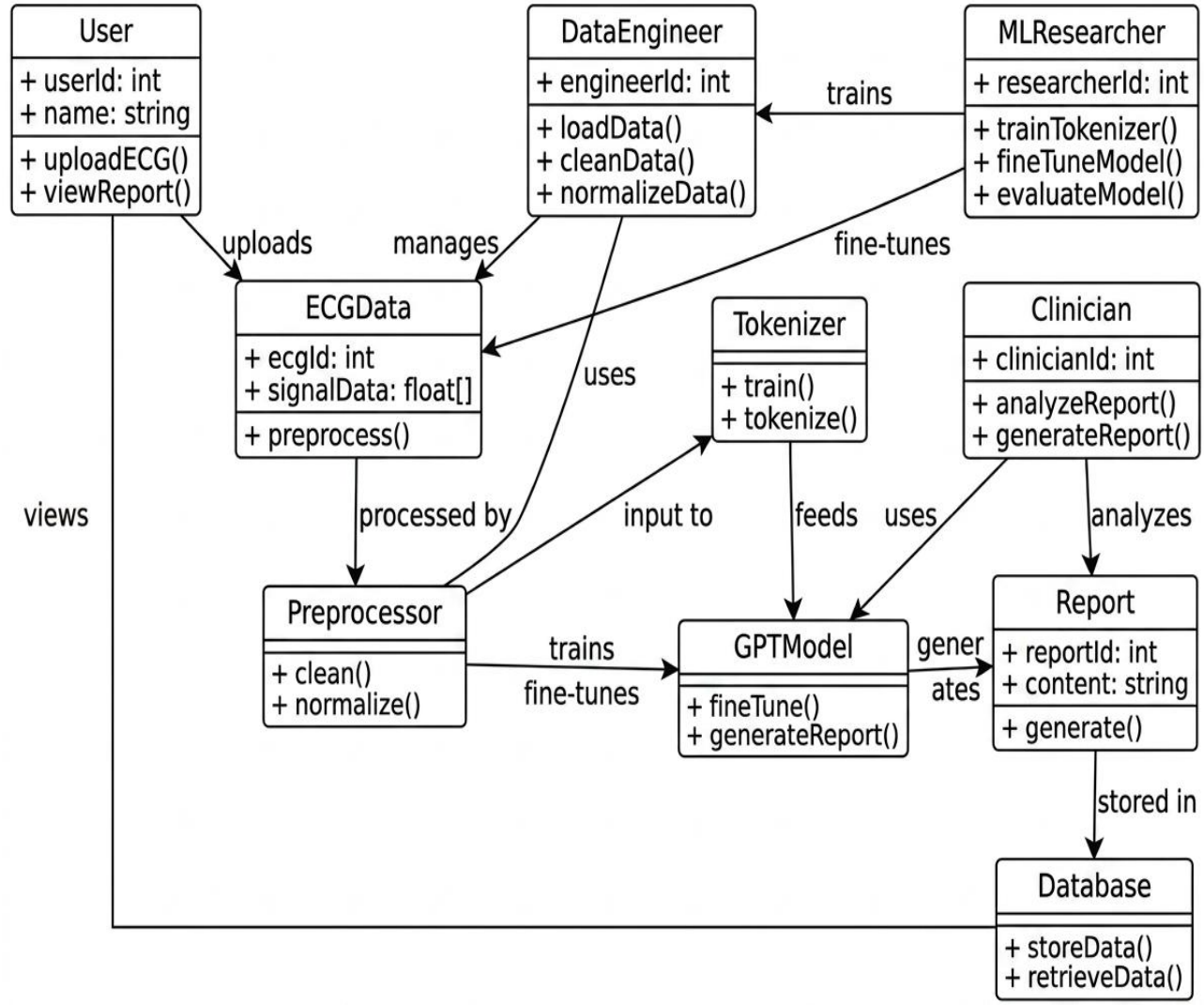


Fig 5.3.4 Class Diagram

CHAPTER-6

CODING AND IMPLEMENTATION

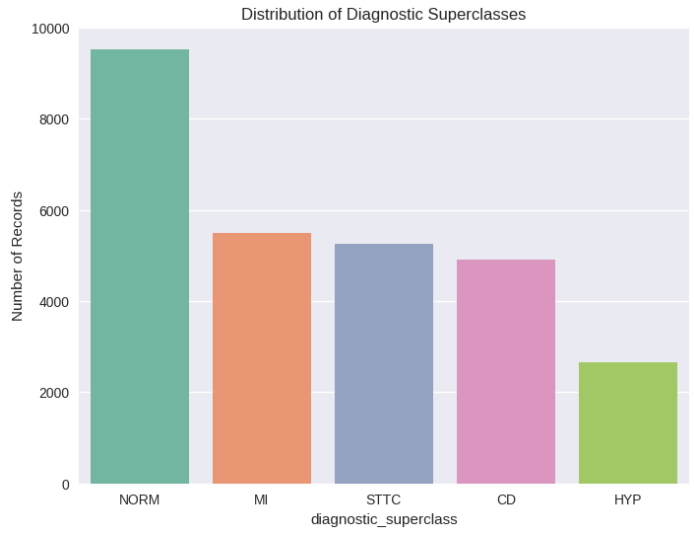
6. CODING AND IMPLEMENTATION

6.1 Source Code

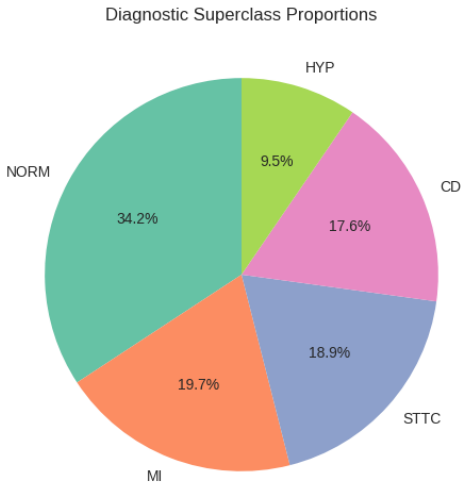
PTB-XL_ECG_ds.ipynb:

```
!pip install wfdb

import pandas as pd
import numpy as np
import wfdb
import ast
import matplotlib.pyplot as plt
import seaborn as sns

# Config
plt.style.use("seaborn-v0_8")
sns.set_palette("Set2")
BASE_PATH = "/kaggle/input/ptb-xl-dataset/ptb-xl-a-large-publicly-available-electrocardiography-dataset-1.0.1/"

# Load metadata
df = pd.read_csv(BASE_PATH + "ptbxl_database.csv", index_col="ecg_id")
df.scp_codes = df.scp_codes.apply(lambda x: ast.literal_eval(x))
scp_df = pd.read_csv(BASE_PATH + "scp_statements.csv", index_col=0)

# Aggregate diagnostic classes
agg_df = scp_df[scp_df.diagnostic == 1]
def aggregate_diagnostic(y_dic):
    tmp = []
    for key in y_dic.keys():
        if key in agg_df.index:
            tmp.append(agg_df.loc[key].diagnostic_class)
    return list(set(tmp))

df["diagnostic_superclass"] = df.scp_codes.apply(aggregate_diagnostic)
```

```
# 1. Distribution of diagnostic superclasses
diag_counts = df.explode("diagnostic_superclass").diagnostic_superclass.value_counts()
plt.figure(figsize=(8,6))
sns.barplot(x=diag_counts.index, y=diag_counts.values)
plt.title("Distribution of Diagnostic Superclasses")
plt.ylabel("Number of Records")
plt.show()
```



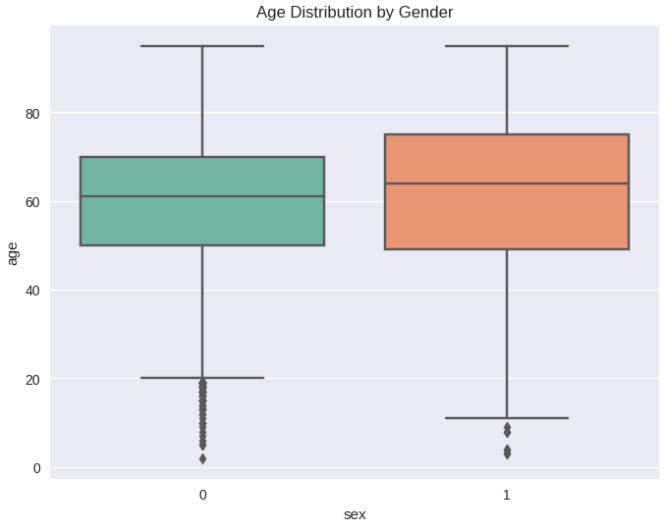

```
# 2. Pie chart of diagnostic superclasses
plt.figure(figsize=(6,6))
plt.pie(diag_counts.values, labels=diag_counts.index, autopct="%1.1f%%", startangle=90)
plt.title("Diagnostic Superclass Proportions")
plt.show()
```



```
# 3. Gender distribution
plt.figure(figsize=(6,4))
sns.countplot(x="sex", data=df)
plt.title("Gender Distribution")
plt.show()
```

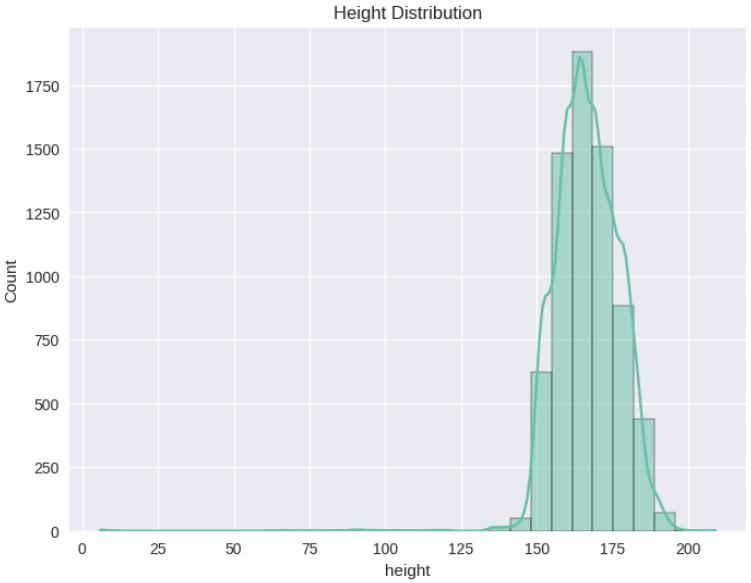

```
# 4. Age distribution (histogram)
plt.figure(figsize=(8,6))
sns.histplot(df.age, bins=40, kde=True)
plt.title("Age Distribution")
plt.xlabel("Age")
plt.show()
```

```
# 5. Age by gender
plt.figure(figsize=(8,6))
sns.boxplot(x="sex", y="age", data=df)
plt.title("Age Distribution by Gender")
plt.show()
```

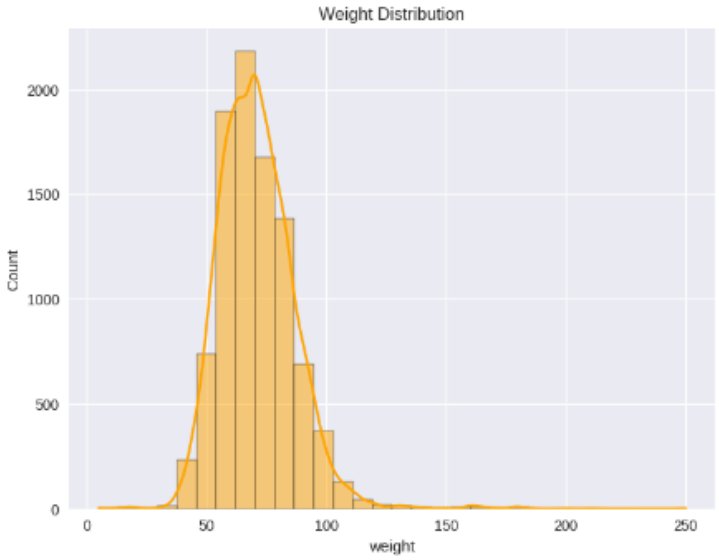



```
# 6. Age vs diagnostic superclass
plt.figure(figsize=(10,6))
sns.violinplot(x="diagnostic_superclass", y="age", data=df.explode("diagnostic_superclass"))
plt.title("Age vs Diagnostic Superclass")
plt.show()
```

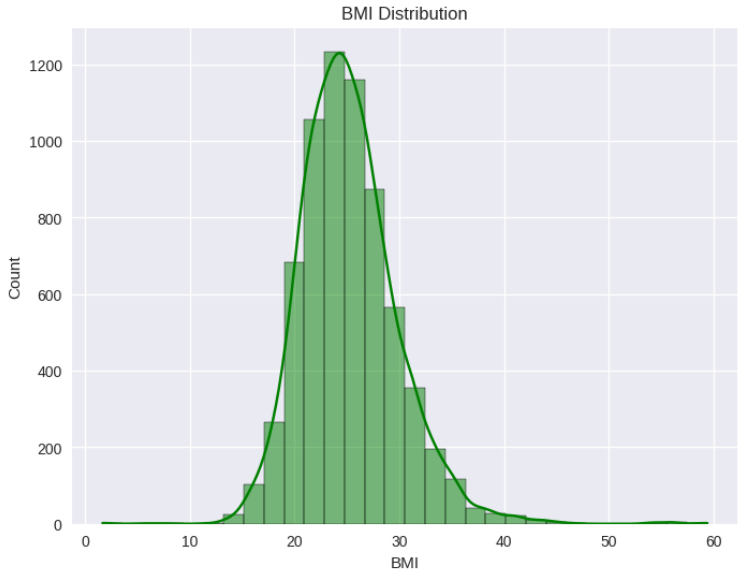
```
# 7. Height distribution
plt.figure(figsize=(8,6))
sns.histplot(df.height.dropna(), bins=30, kde=True)
plt.title("Height Distribution")
plt.show()
```



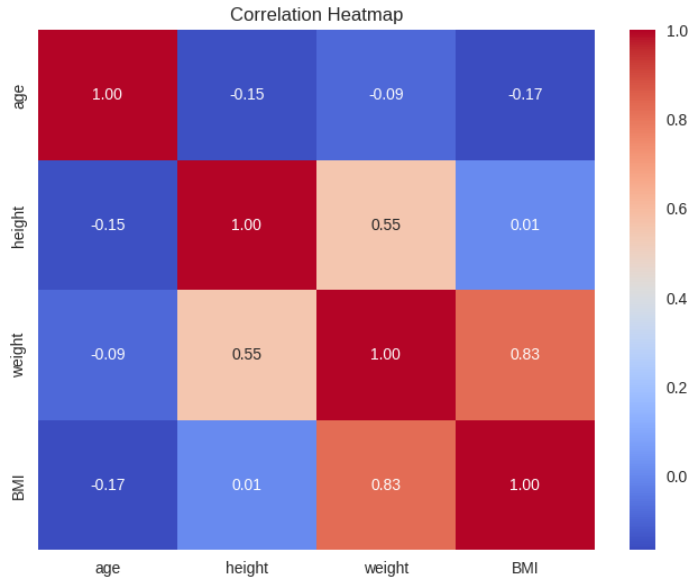

```
# 8. Weight distribution
plt.figure(figsize=(8,6))
sns.histplot(df.weight.dropna(), bins=30, kde=True, color="orange")
plt.title("Weight Distribution")
plt.show()
```



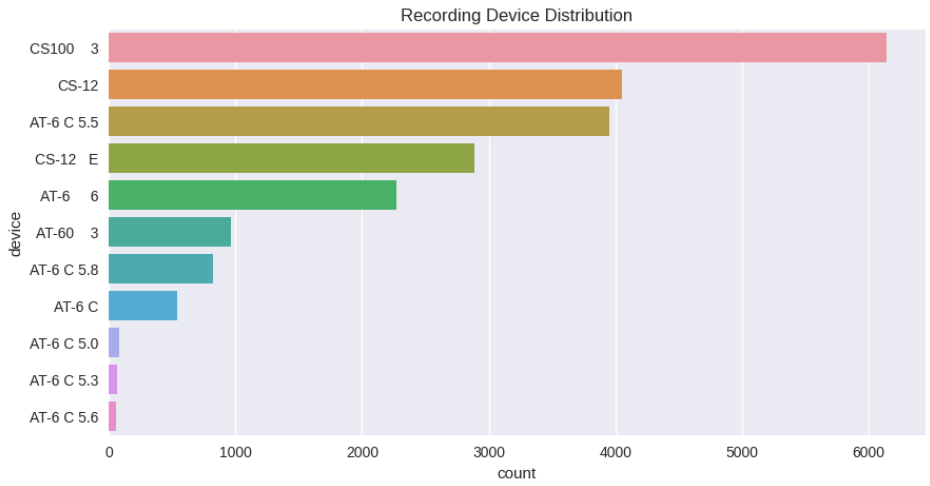
```
# 9. BMI approximation (weight/height^2)
df["BMI"] = df.apply(lambda r: r.weight/(r.height/100)**2 if pd.notnull(r.weight) and
pd.notnull(r.height) else np.nan, axis=1)
plt.figure(figsize=(8,6))
sns.histplot(df.BMI.dropna(), bins=30, kde=True, color="green")
plt.title("BMI Distribution")
plt.show()
```



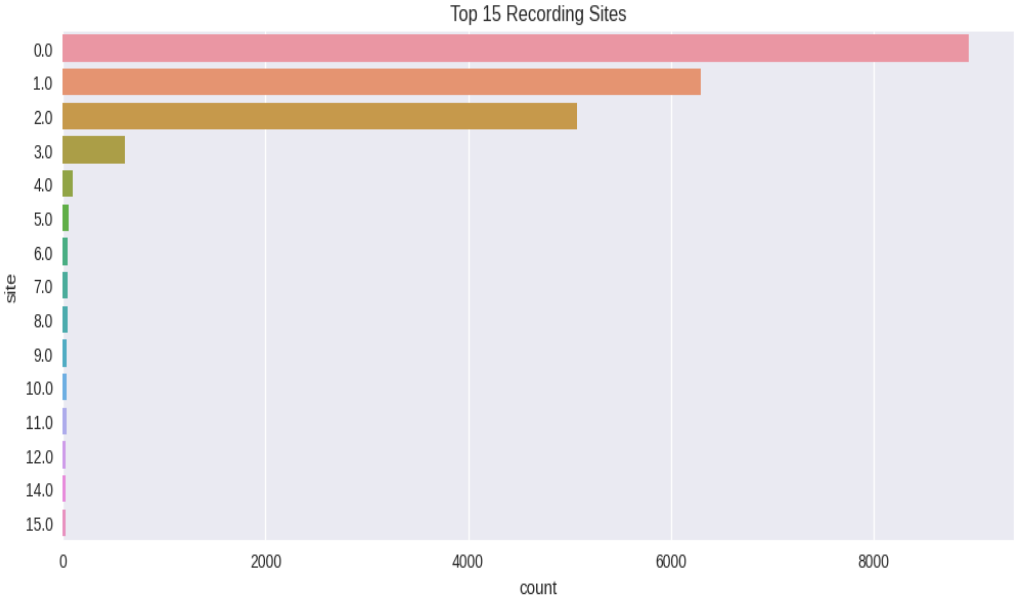

```
# 10. Correlation heatmap of demographics
plt.figure(figsize=(8,6))
sns.heatmap(df[["age","height","weight","BMI"]].corr(), annot=True, cmap="coolwarm",
fmt=".2f")
plt.title("Correlation Heatmap")
plt.show()
```



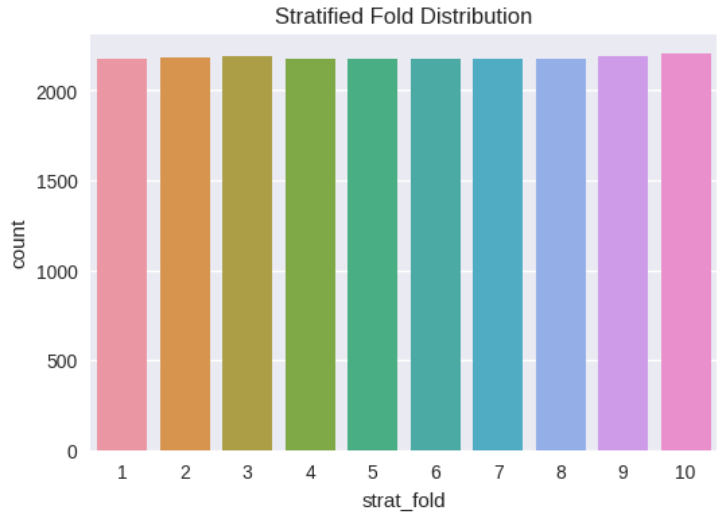
```
# 11. Recording device distribution
plt.figure(figsize=(10,5))
sns.countplot(y="device", data=df, order=df.device.value_counts().index)
plt.title("Recording Device Distribution")
plt.show()
```



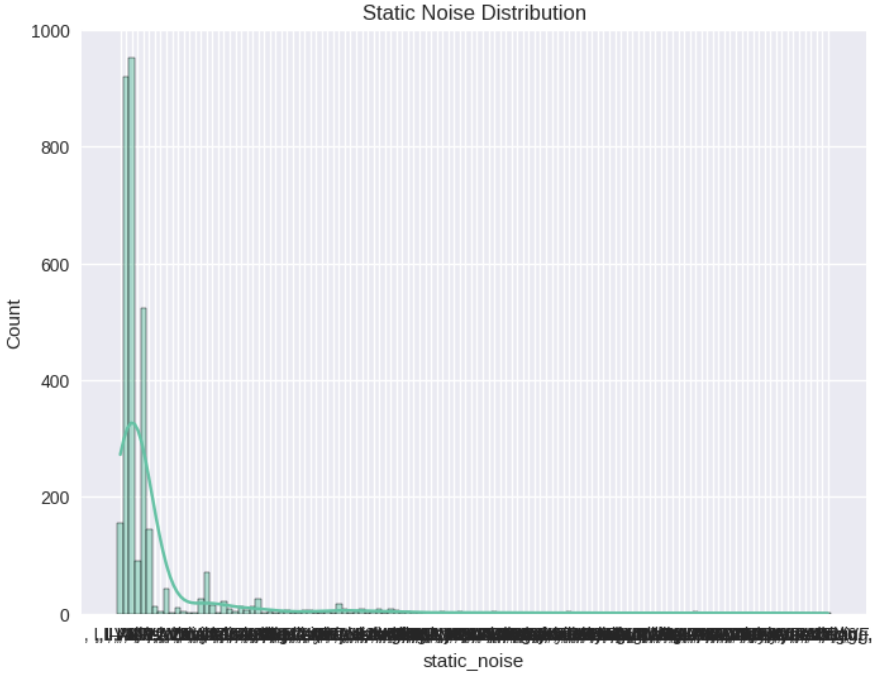

```
# 12. Recording site distribution
plt.figure(figsize=(12,5))
sns.countplot(y="site", data=df, order=df.site.value_counts().index[:15])
plt.title("Top 15 Recording Sites")
plt.show()
```



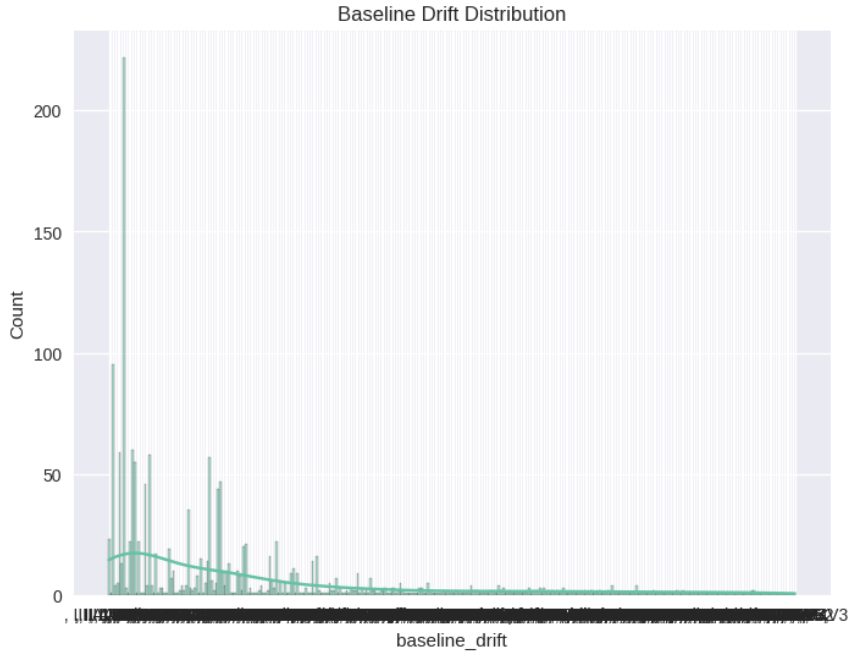
```
# 13. Fold distribution
plt.figure(figsize=(6,4))
sns.countplot(x="strat_fold", data=df)
plt.title("Stratified Fold Distribution")
plt.show()
```



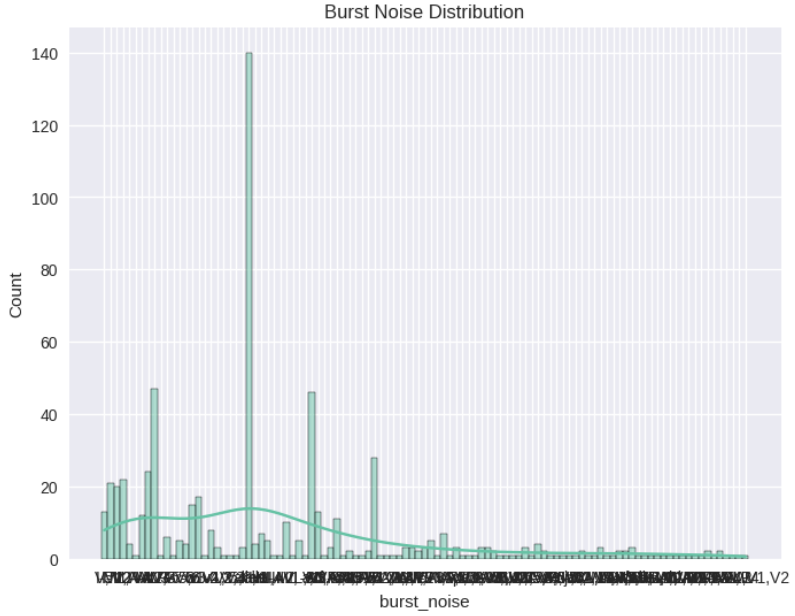

```
# 14. Noise distribution (static)
plt.figure(figsize=(8,6))
sns.histplot(df.static_noise.dropna(), bins=30, kde=True)
plt.title("Static Noise Distribution")
plt.show()
```



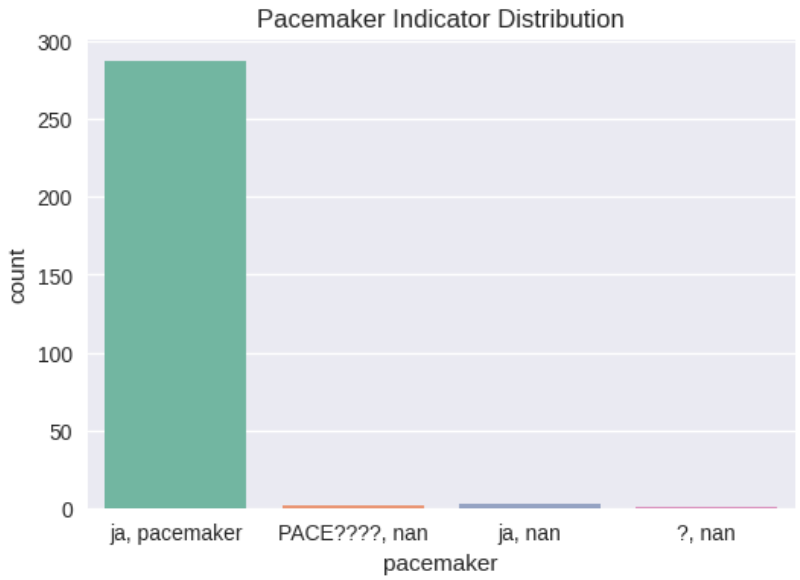
```
# 15. Baseline drift distribution
plt.figure(figsize=(8,6))
sns.histplot(df.baseline_drift.dropna(), bins=30, kde=True)
plt.title("Baseline Drift Distribution")
plt.show()
```




```
# 16. Burst noise distribution
plt.figure(figsize=(8,6))
sns.histplot(df.burst_noise.dropna(), bins=30, kde=True)
plt.title("Burst Noise Distribution")
plt.show()
```

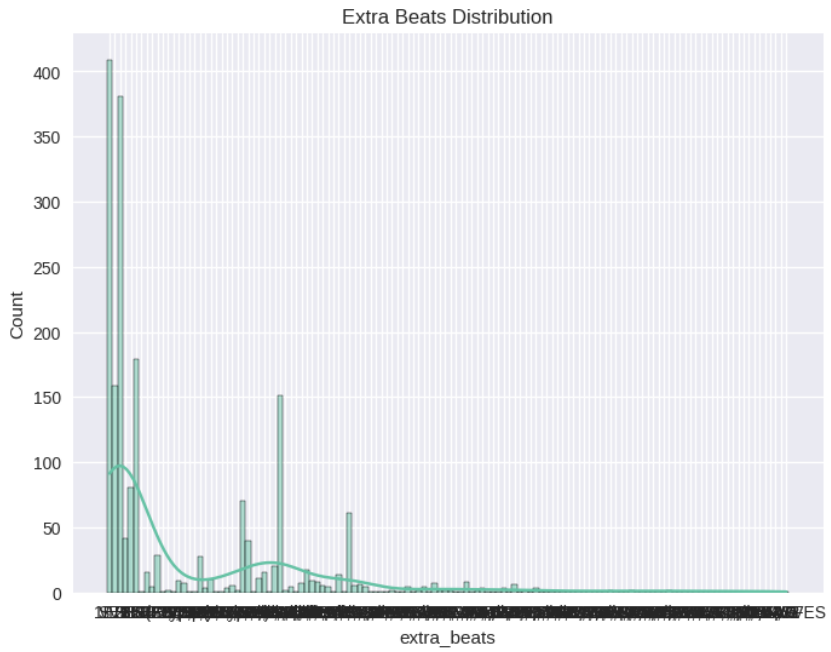


```
# 17. Pacemaker presence
plt.figure(figsize=(6,4))
sns.countplot(x="pacemaker", data=df)
plt.title("Pacemaker Indicator Distribution")
```

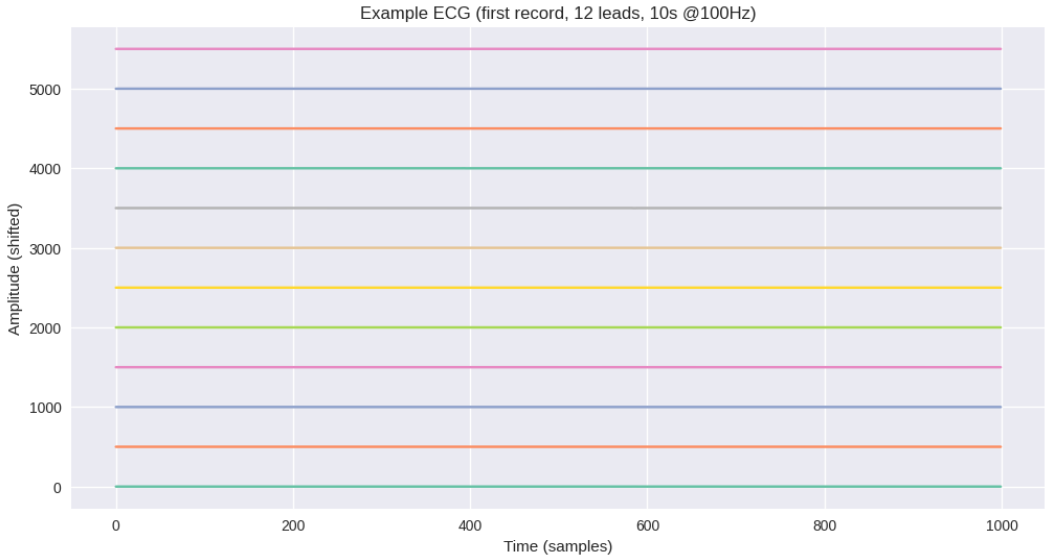



```
plt.show()
```

```
# 18. Extra beats distribution
plt.figure(figsize=(8,6))
sns.histplot(df.extra_beats.dropna(), bins=30, kde=True)
plt.title("Extra Beats Distribution")
plt.show()
```

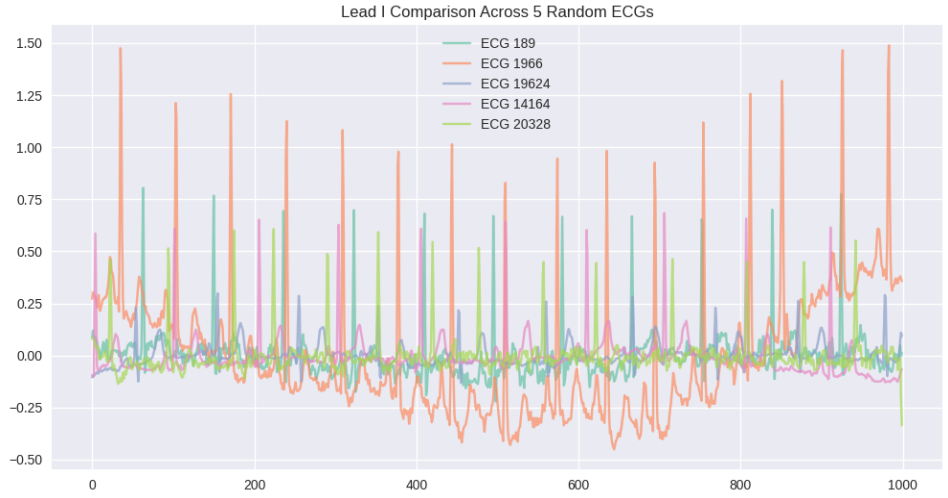


```
# 19. Example ECG signal (100Hz)
example_file = BASE_PATH + df.filename_lr.iloc[0]
sig, fields = wfdb.rdsamp(example_file)
plt.figure(figsize=(12,6))
for i in range(12):
plt.plot(sig[:1000,i] + 500*i)  # shift each lead for clarity
plt.title("Example ECG (first record, 12 leads, 10s @100Hz)")
plt.xlabel("Time (samples)")
plt.ylabel("Amplitude (shifted)")
```



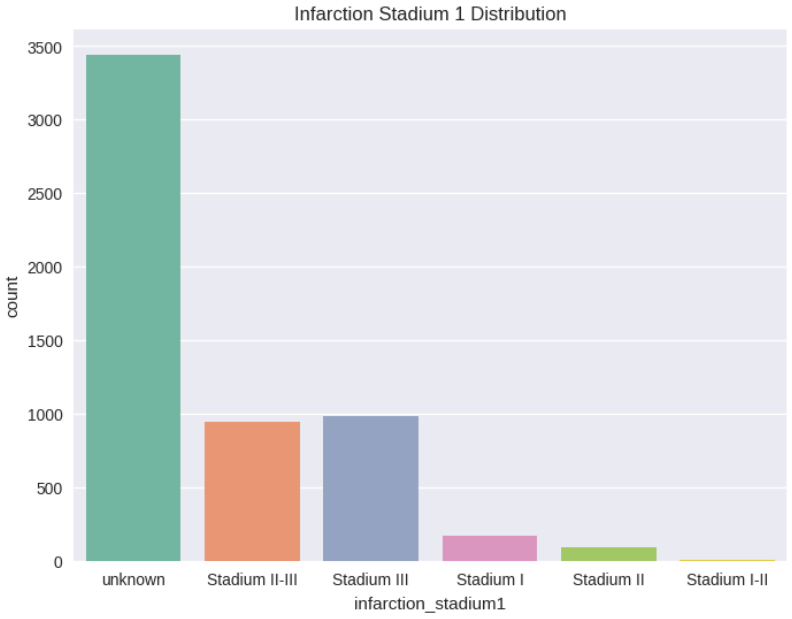

```
plt.show()
```

```
# 20. Multiple ECG signal overlays (random 5 records)
plt.figure(figsize=(12,6))
for idx in np.random.choice(df.index, 5, replace=False):
sig, _ = wfdb.rdsamp(BASE_PATH + df.loc[idx].filename_lr)
plt.plot(sig[:1000,0], alpha=0.7, label=f"ECG {idx}")
plt.legend()
plt.title("Lead I Comparison Across 5 Random ECGs")
```



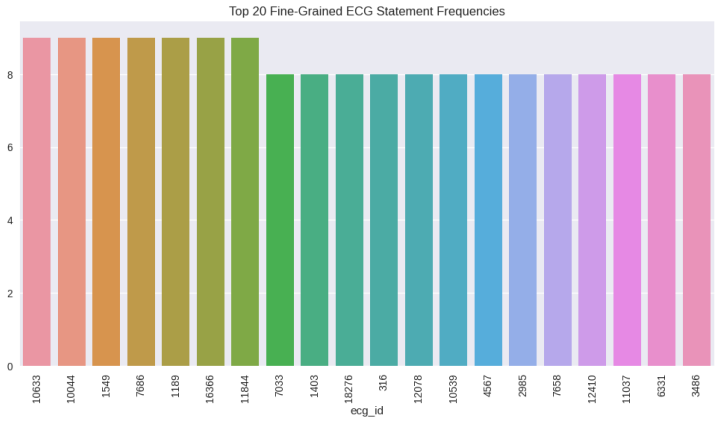

```
plt.show()
```

```
# 21. Severity-level distribution (if available)
if "infarction_stadium1" in df.columns:
    plt.figure(figsize=(8,6))
    sns.countplot(x="infarction_stadium1", data=df)
    plt.title("Infarction Stadium 1 Distribution")
    plt.show()

if "infarction_stadium2" in df.columns:
    plt.figure(figsize=(8,6))
    sns.countplot(x="infarction_stadium2", data=df)
    plt.title("Infarction Stadium 2 Distribution")
```




```
plt.show()
```

```
# 22. Diagnostic classes (fine-grained)
fine_counts = df.explode("scp_codes").scp_codes.explode().index.value_counts().head(20)
plt.figure(figsize=(12,6))sns.barplot(x=fine_counts.index.astype(str), y=fine_counts.values)
plt.title("Top 20 Fine-Grained ECG Statement Frequencies")
plt.xticks(rotation=90)
plt.show()
```



exsisting-system.ipynb

```
import os, sys, ast, math, warnings, time
from pathlib import Path
import numpy as np
import pandas as pd
import wfdb
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score, classification_report
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
warnings.filterwarnings("ignore")

# ---------- CONFIG ----------
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", DEVICE)
PTBXL_PATH = "/kaggle/input/ptb-xl-dataset/ptb-xl-a-large-publicly-available-electrocardiography-dataset-1.0.1"
# change above to your local path if needed



MAX_LEN = 500
BATCH = 32
NUM_WORKERS = 2
RANDOM_SEED = 42

# Training epoch settings for each algorithm (small for demo)
EPOCHS_CLASSIFIER = 2
EPOCHS_CAPTION = 1
EPOCHS_GPT = 1

# Target classes (superclasses)
CLASSES = ['CD','HYP','MI','NORM','STTC']

import math
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

# Load metadata
meta_csv = os.path.join(PTBXL_PATH, "ptbxl_database.csv")
scp_csv = os.path.join(PTBXL_PATH, "scp_statements.csv")
assert os.path.exists(meta_csv), f"ptbxl_database.csv not found at {meta_csv}"
assert os.path.exists(scp_csv), f"scp_statements.csv not found at {scp_csv}"
```

```
    meta_df = pd.read_csv(meta_csv, index_col=0)
    scp_df = pd.read_csv(scp_csv, index_col=0)

    # helper to map scp_codes -> diagnostic_superclass (list)
    def scp_to_superclasses(scp_codes_str):
        try:
            d = ast.literal_eval(scp_codes_str)
        except Exception:
            return []
        out=[]
        for k in d.keys():
            if k in scp_df.index and scp_df.loc[k, 'diagnostic_class'] in CLASSES:
                out.append(scp_df.loc[k,'diagnostic_class'])
        return list(set(out))

    meta_df['diagnostic_superclass'] = meta_df['scp_codes'].apply(scp_to_superclasses)
    # MultiLabel binarizer
    mlb = MultiLabelBinarizer(classes=CLASSES)
    Y_all = mlb.fit_transform(meta_df['diagnostic_superclass'])
    # Safe index mapping for filename paths (some rows store relative paths like
    'records100/00000/00001_lr')

return x, y

    indices = np.arange(len(meta_df))
    has_label_mask = np.array([len(l)>0 for l in meta_df['diagnostic_superclass']])
    try:
        train_idx, test_idx = train_test_split(indices, test_size=0.1, random_state=RANDOM_SEED,
    stratify=[str(s) for s in meta_df['diagnostic_superclass']])
        train_idx, val_idx = train_test_split(train_idx, test_size=0.1, random_state=RANDOM_SEED,
    stratify=[str(meta_df.iloc[i]['diagnostic_superclass']) for i in train_idx])
    except Exception:
        train_idx, test_idx = train_test_split(indices, test_size=0.1, random_state=RANDOM_SEED)
        train_idx, val_idx = train_test_split(train_idx, test_size=0.1, random_state=RANDOM_SEED)

    train_df = meta_df.iloc[train_idx].copy()
    val_df = meta_df.iloc[val_idx].copy()
    test_df = meta_df.iloc[test_idx].copy()
    Y = mlb.transform(meta_df['diagnostic_superclass'])
    train_Y = Y[train_idx]; val_Y = Y[val_idx]; test_Y = Y[test_idx]

    print("Sizes (train/val/test):", len(train_df), len(val_df), len(test_df))

    train_ds = PTBXL_Dataset(train_df, train_Y, max_len=MAX_LEN)
    val_ds = PTBXL_Dataset(val_df, val_Y, max_len=MAX_LEN)
    test_ds = PTBXL_Dataset(test_df, test_Y, max_len=MAX_LEN)
```

```
train_loader = DataLoader(train_ds, batch_size=BATCH, shuffle=True,
num_workers=NUM_WORKERS)
val_loader = DataLoader(val_ds, batch_size=BATCH, shuffle=False,
num_workers=NUM_WORKERS)
test_loader = DataLoader(test_ds, batch_size=BATCH, shuffle=False,
num_workers=NUM_WORKERS)

def eval_multilabel(model, loader, threshold=0.5):
    model.eval()
    Ys=[]; Ps=[]
    with torch.no_grad():
        for x,y in loader:
            x = x.to(DEVICE)
            # model expects (B, C, T) - many models below follow that
            logits = model(x)
            probs = torch.sigmoid(logits).cpu().numpy()
            Ys.append(y.numpy())
            Ps.append(probs)

    Ys = np.vstack(Ys); Ps = np.vstack(Ps)
    pred = (Ps > threshold).astype(int)



    try:
        auc = roc_auc_score(Ys, Ps, average='macro')
    except Exception:
        auc = float('nan')
    f1 = f1_score(Ys, pred, average='macro', zero_division=0)
    acc = (pred == Ys).all(axis=1).mean()  # strict multi-label exact-match accuracy
    print("AUROC(macro):", auc)
    print("F1(macro):", f1)
    print("Exact-match accuracy:", acc)
    return Ys, Ps, pred
print("\n== Algorithm 1: BPE Tokenizer (demo) ==")
try:
    from tokenizers import Tokenizer, models, pre_tokenizers, trainers, decoders
    sample_reports = meta_df['report'].dropna().astype(str).tolist()
    # train small BPE on reports (demo)
    tmp_file = "ptbxl_reports_corpus.txt"
    with open(tmp_file, "w", encoding="utf-8") as f:
        for r in sample_reports[:5000]:
            f.write(r.replace("\n"," ") + "\n")
    tokenizer = Tokenizer(models.BPE())
    tokenizer.pre_tokenizer = pre_tokenizers.Whitespace()
    trainer = trainers.BpeTrainer(vocab_size=8000, special_tokens=["<pad>","<s>","</s>","<unk>"])
```

```
    tokenizer.train([tmp_file], trainer)
    print("BPE vocab size:", tokenizer.get_vocab_size())
    print("Sample tokens:", tokenizer.encode("Normal sinus rhythm").tokens[:20])
except Exception as e:
    print("Tokenizers not available or corpus missing:", e)

== Algorithm 1: BPE Tokenizer (demo) ==

print("\n== Algorithm 2: ResNet1D Encoder + Transformer Decoder (captioning sketch) ==")
from torch.nn import TransformerDecoderLayer, TransformerDecoder

class ResNet1D_Enc(nn.Module):
    def __init__(self, out_dim=128):
        super().__init__()
        self.conv1 = nn.Conv1d(12,64, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm1d(64)
        self.relu = nn.ReLU()
        # adaptively pool to fixed sequence length
        self.pool = nn.AdaptiveAvgPool1d(128)
        self.fc = nn.Linear(64*128, out_dim)
    def forward(self,x):      # x: (B,12,T)
        h = self.relu(self.bn1(self.conv1(x)))
    print("librosa not installed — install librosa to run spectrogram-based ChexNet approach (pip
install librosa).")

from torchvision import models, transforms

def ecg_to_melspec_batch(X_numpy, sr=100, n_mels=128, target_size=(128,128)):
    # X_numpy: (B,12,T) numpy
    B = X_numpy.shape[0]
    imgs = []
    for b in range(B):
        leads = X_numpy[b]  # (12,T)
        channels = []
        # create mel for first 3 leads (I, II, V1) or choose [0,1,6]
        lead_idxs = [0,1,6]
        for li in lead_idxs:
            sig = leads[li]
            if librosa is None:
                # fallback simple STFT using numpy (not optimal)
                S = np.abs(np.fft.rfft(sig, n=target_size[1]*2))[:target_size[1]]
                S = np.tile(S[:target_size[1]], (target_size[0],1))
            else:
                S = librosa.feature.melspectrogram(y=sig, sr=sr, n_mels=target_size[0], n_fft=256,
hop_length=max(1, len(sig)//target_size[1]))
                S = librosa.power_to_db(S, ref=np.max)
                # resize/pad to target_size
```

```
            if S.shape[1] < target_size[1]:
                pad = target_size[1] - S.shape[1]
                S = np.pad(S, ((0,0),(0,pad)), mode='constant')
            else:
                S = S[:, :target_size[1]]
        # normalize
        S = (S - S.mean()) / (S.std()+1e-8)
        channels.append(S)
    img = np.stack(channels, axis=0)  # (3, H, W)
    imgs.append(img)
  return np.stack(imgs, axis=0).astype(np.float32)  # (B,3,H,W)

class DenseNetSpec(nn.Module):
  def __init__(self, num_classes=len(CLASSES), pretrained=False):
    super().__init__()
    base = models.densenet121(pretrained=pretrained)
    # adapt first conv to accept 3 channels (already 3)
    self.features = base.features
    self.classifier = nn.Linear(base.classifier.in_features, num_classes)
  def forward(self,x):
    f = self.features(x)
    f = F.relu(f, inplace=True)
    f = F.adaptive_avg_pool2d(f, (1,1)).view(x.size(0), -1)
    return self.classifier(f)

if librosa is not None:
  # training loop for DenseNet on spectrograms (quick demo)
  densenet = DenseNetSpec().to(DEVICE)
  opt = torch.optim.Adam(densenet.parameters(), lr=1e-4)
  criterion = nn.BCEWithLogitsLoss()

  for epoch in range(EPOCHS_CLASSIFIER):
    densenet.train(); running=0.0; n=0

    for xb, yb in train_loader:
      # xb: (B,12,T) -> numpy

      xb_np = xb.numpy()
      imgs = ecg_to_melspec_batch(xb_np, sr=100, n_mels=128, target_size=(128,128))
      imgs_t = torch.tensor(imgs).to(DEVICE)
      yb = yb.to(DEVICE)
      logits = densenet(imgs_t)
      loss = criterion(logits, yb)
      opt.zero_grad(); loss.backward(); opt.step()
      running += loss.item(); n+=1
    print(f"DenseNet spectrogram epoch {epoch+1}, loss {running/max(1,n):.4f}")
```

```
    # evaluate
    def eval_densenet(loader):
        densenet.eval()
        Ys=[]; Ps=[]

        with torch.no_grad():
            for xb,yb in loader:
                imgs = ecg_to_melspec_batch(xb.numpy())
                imgs_t = torch.tensor(imgs).to(DEVICE)
                probs = torch.sigmoid(densenet(imgs_t)).cpu().numpy()
                Ys.append(yb.numpy()); Ps.append(probs)
        Ys = np.vstack(Ys); Ps = np.vstack(Ps)
        preds = (Ps>0.5).astype(int)

        print("DenseNet AUROC:", roc_auc_score(Ys, Ps, average='macro'))
        print("DenseNet F1:", f1_score(Ys, preds, average='macro', zero_division=0))
    print("Evaluating DenseNet on test set (spectrogram)...")
    eval_densenet(test_loader)
else:
    print("Skipping DenseNet spectrogram training (librosa not installed).")

    print("Skipping scattering classifier due to missing dependency.")


== Algorithm 4: Wavelet Scattering + classifier ==

print("\n== Algorithm 5: Vision-Text Transformer (ECG-GPT sketch) ==")
from torch.nn import Transformer
class ECGMultimodal(nn.Module):
    def __init__(self, d_model=128, vocab_size=500):
        super().__init__()
        self.encoder = ResNet1D_Enc(d_model)
        self.text_decoder_layer = nn.TransformerDecoderLayer(d_model=d_model, nhead=8)
        self.decoder = nn.TransformerDecoder(self.text_decoder_layer, num_layers=2)
        self.lm_head = nn.Linear(d_model, vocab_size)
    def forward(self, x, tgt_emb):
        # x: (B,12,T), tgt_emb: (tgt_len,B,d_model)
        enc = self.encoder(x).unsqueeze(0)  # (1,B,d_model)
        out = self.decoder(tgt_emb, enc)    # (tgt_len,B,d_model)
        return self.lm_head(out)

mmodel = ECGMultimodal().to(DEVICE)
opt = torch.optim.Adam(mmodel.parameters(), lr=1e-4)
for epoch in range(EPOCHS_CAPTION):
    mmodel.train(); r=0
    for xb,yb in train_loader:
        xb = xb.to(DEVICE)
```

```
        tgt = torch.randn(12, xb.size(0), 128, device=DEVICE)
        logits = mmodel(xb, tgt)
        labels = torch.randint(0,500,(12*xb.size(0),), device=DEVICE)
        loss = F.cross_entropy(logits.view(-1, logits.size(-1)), labels)
        opt.zero_grad(); loss.backward(); opt.step()
        r+=loss.item()
    print(f"Multimodal epoch {epoch+1}, loss {r/len(train_loader):.4f}")
print("Multimodal demo done.")

== Algorithm 5: Vision-Text Transformer (ECG-GPT sketch) ==

print("\n== Algorithm 6: Custom Domain-Specific Tokenizer + GPT-2 (sketch) ==")
try:
    from tokenizers import Tokenizer, models, pre_tokenizers, trainers
    from transformers import GPT2Config, GPT2LMHeadModel
    # Build a tiny custom BPE on text reports (take limited subset)
    reports = meta_df['report'].dropna().astype(str).tolist()
    corpus_small = "reports_small.txt"
    with open(corpus_small, "w", encoding="utf-8") as f:
        for r in reports[:2000]:
            f.write(r.replace("\n"," ") + "\n")
    tok = Tokenizer(models.BPE())

    tok.pre_tokenizer = pre_tokenizers.Whitespace()
    trainer = trainers.BpeTrainer(vocab_size=2000, special_tokens=["<pad>","<s>","</s>","<unk>"])
    tok.train([corpus_small], trainer)
    vocab_size = tok.get_vocab_size()
    print("Custom tokenizer vocab size:", vocab_size)
    # Lightweight GPT2 config
    cfg = GPT2Config(vocab_size=vocab_size, n_embd=128, n_layer=4, n_head=4)
    gpt = GPT2LMHeadModel(cfg).to(DEVICE)
    # quick mock training loop: use tokenized report substrings as targets (demo)
    opt = torch.optim.Adam(gpt.parameters(), lr=1e-4)
    for epoch in range(EPOCHS_GPT):
        gpt.train(); r=0
        for i,report in enumerate(reports[:500]):
            enc = tok.encode(report)
            ids = enc.ids
            if len(ids)<16: continue
            ids_t = torch.tensor(ids[:64], device=DEVICE).unsqueeze(0)
            outputs = gpt(ids_t, labels=ids_t)
            loss = outputs.loss
            opt.zero_grad(); loss.backward(); opt.step()
            r += loss.item()
            if i>200: break
        print(f"GPT2-proposed epoch {epoch+1}, loss {r/200:.4f}")
    print("Custom tokenizer + GPT2 demo finished.")
```

```
except Exception as e:
    print("Transformers or tokenizers not installed or corpus missing:", e)
```

== Algorithm 6: Custom Domain-Specific Tokenizer + GPT-2 (sketch) ==

```
print("\n== Algorithm 7: Preprocessing & Normalization Pipeline demo ==")
def preprocess_pipeline(x_numpy):
    # x_numpy: (T,12) or (12,T) - expect (12,T)
    if x_numpy.ndim==2 and x_numpy.shape[0]==12:
        X = x_numpy
    else:
        X = x_numpy.T
    # baseline wander removal (high-pass via difference)
    X = X - np.mean(X, axis=1, keepdims=True)
    # robust scaling per lead (clip extreme)
    med = np.median(X, axis=1, keepdims=True)
    mad = np.median(np.abs(X - med), axis=1, keepdims=True) + 1e-6
    Xs = (X - med) / mad
    window = 5
    smooth = np.convolve(np.ones(window)/window, Xs[0], mode='same')
    # return scaled
    return Xs

# quick check on one sample
x0,_ = train_ds[0]
print("Preprocess sample shape:", x0.shape)
print("Preprocess example (lead0 first 5) ->", preprocess_pipeline(x0.numpy())[0,:5])
```

== Algorithm 7: Preprocessing & Normalization Pipeline demo ==

```
print("\n== Final: Train a simple ResNet1D classifier and print metrics ==")
class SimpleResNet1D(nn.Module):
    def __init__(self, out_dim=128, num_classes=len(CLASSES)):
        super().__init__()
        self.conv1 = nn.Conv1d(12,64,7,stride=2,padding=3)
        self.bn1 = nn.BatchNorm1d(64)
        self.relu = nn.ReLU()
        self.pool = nn.AdaptiveAvgPool1d(128)
        self.fc = nn.Linear(64*128, out_dim)
        self.head = nn.Linear(out_dim, num_classes)
    def forward(self,x):
        h = self.relu(self.bn1(self.conv1(x)))
        h = self.pool(h).flatten(1)
        h = self.fc(h)
        return self.head(h)

resnet_clf = SimpleResNet1D().to(DEVICE)
```

```
opt = torch.optim.Adam(resnet_clf.parameters(), lr=1e-4)
criterion = nn.BCEWithLogitsLoss()

for epoch in range(EPOCHS_CLASSIFIER):
    resnet_clf.train(); running=0; n=0
    for xb,yb in train_loader:
        xb = xb.to(DEVICE)
        yb = yb.to(DEVICE)
        logits = resnet_clf(xb)
        loss = criterion(logits, yb)
        opt.zero_grad(); loss.backward(); opt.step()
        running += loss.item(); n+=1
    print(f"ResNet clf epoch {epoch+1}, loss {running/max(1,n):.4f}")
    print("\nClassification report (per-label) - threshold 0.5")
    y_true = Ys
    y_pred = (Ps>0.5).astype(int)
if librosa is not None:
    # training loop for DenseNet on spectrograms (quick demo)
    densenet = DenseNetSpec().to(DEVICE)
    opt = torch.optim.Adam(densenet.parameters(), lr=1e-4)
    criterion = nn.BCEWithLogitsLoss()

    for epoch in range(EPOCHS_CLASSIFIER):
        densenet.train(); running=0.0; n=0

        for xb, yb in train_loader:
            # xb: (B,12,T) -> numpy

            xb_np = xb.numpy()
            imgs = ecg_to_melspec_batch(xb_np, sr=100, n_mels=128, target_size=(128,128))
            imgs_t = torch.tensor(imgs).to(DEVICE)
            yb = yb.to(DEVICE)
            logits = densenet(imgs_t)
            loss = criterion(logits, yb

    for i,cls in enumerate(CLASSES):
        print(f"--- {cls} ---")
        print(classification_report(y_true[:,i], y_pred[:,i], zero_division=0))
except Exception as e:
    print("Error printing classification report:", e)
print("\nAll algorithms demo finished. Increase EPOCHS_* for better performance and replace fake captioning targets with real tokenized text for captioning/GPT training.")
```

== Final: Train a simple ResNet1D classifier and print metrics ==

```
--- CD ---
              precision    recall  f1-score   support
```

0.0	0.82	0.96	0.89	1674
1.0	0.71	0.31	0.43	510
accuracy			0.81	2184
macro avg	0.77	0.63	0.66	2184
weighted avg	0.79	0.81	0.78	2184

--- HYP ---

	precision	recall	f1-score	support
0.0	0.89	1.00	0.94	1937
1.0	0.40	0.02	0.03	247
accuracy			0.89	2184
macro avg	0.64	0.51	0.49	2184
weighted avg	0.83	0.89	0.84	2184

--- MI ---

	precision	recall	f1-score	support
0.0	0.79	0.96	0.87	1622
1.0	0.72	0.28	0.41	562
accuracy			0.79	2184
macro avg	0.76	0.62	0.64	2184
weighted avg	0.77	0.79	0.75	2184

--- NORM ---

	precision	recall	f1-score	support
0.0	0.87	0.74	0.80	1222
1.0	0.72	0.86	0.78	962
accuracy			0.79	2184
macro avg	0.79	0.80	0.79	2184
weighted avg	0.80	0.79	0.79	2184

Predict.py

```
import os, sys, warnings, ast
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import wfdb
from torch_geometric.data import Data, Batch
from torch_geometric.nn import SAGEConv, global_mean_pool

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
MODEL_PATH = "ptbxl_gnn_hybrid_final.pth"
CLASSES = ['CD', 'HYP', 'MI', 'NORM', 'STTC']  # adjust if different
NUM_CLASSES = len(CLASSES)
warnings.filterwarnings("ignore")

def create_anatomical_adjacency():
    rel = [
        (0,1),(1,2),
        (0,4),(1,4),(2,5),(0,3),(3,4),(4,5),
        (6,7),(7,8),(8,9),(9,10),(10,11),
        (1,6),(5,10),(4,11)
    ]
    edges = []
    for a,b in rel:
        edges.append([a,b]); edges.append([b,a])
    return torch.tensor(edges, dtype=torch.long).t().contiguous()

EDGE_INDEX = create_anatomical_adjacency()

class Residual1D(nn.Module):
    def __init__(self, in_ch, out_ch, kernel=7):
        super().__init__()
        pad = (kernel-1)//2
        self.conv = nn.Sequential(
            nn.Conv1d(in_ch, out_ch, kernel, padding=pad),
            nn.BatchNorm1d(out_ch),
            nn.ReLU(),
            nn.Conv1d(out_ch, out_ch, kernel, padding=pad),
            nn.BatchNorm1d(out_ch),
        )
        self.res = nn.Conv1d(in_ch, out_ch, 1) if in_ch != out_ch else nn.Identity()
        self.act = nn.ReLU()
    def forward(self, x): return self.act(self.conv(x) + self.res(x))
```

```
class PerNodeCNN(nn.Module):
    def __init__(self, out_dim=48):
        super().__init__()
        self.net = nn.Sequential(
            Residual1D(1, 16),
            Residual1D(16, 32),
            Residual1D(32, 48)
        )
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(48, out_dim)
    def forward(self, x):
        x = x.unsqueeze(1)       # (nodes, 1, seq_len)
        x = self.net(x)          # (nodes, ch, seq_len)
        x = self.pool(x).squeeze(-1)
        return self.fc(x)

class GNNHybrid(nn.Module):
    def __init__(self, num_classes, node_feat_dim=48, gnn_hidden=64):
        super().__init__()
        self.node_cnn = PerNodeCNN(node_feat_dim)
        self.gnn1 = SAGEConv(node_feat_dim, gnn_hidden)
        self.gnn2 = SAGEConv(gnn_hidden, gnn_hidden)
        self.classifier = nn.Sequential(
            nn.Linear(gnn_hidden, 128), nn.ReLU(), nn.Dropout(0.25),
            nn.Linear(128, num_classes)
        )
    def forward(self, data):
        x = self.node_cnn(data.x.to(DEVICE))
        edge_index = data.edge_index.to(DEVICE)
        batch = data.batch.to(DEVICE) if hasattr(data, 'batch') else torch.zeros(x.size(0),
dtype=torch.long, device=DEVICE)
        x = F.relu(self.gnn1(x, edge_index))
        x = F.relu(self.gnn2(x, edge_index))
        g = global_mean_pool(x, batch)
        return self.classifier(g)

def predict(basefile_noext, model, thresholds=None):
    base = os.path.splitext(basefile_noext)[0]
    rec, _ = wfdb.rdsamp(base)
    sig = rec.astype(np.float32)
    if sig.ndim == 1: sig = sig[:, None]
    if sig.shape[1] != 12 and sig.shape[0] == 12: sig = sig.T
    if sig.shape[1] != 12:
        sig = np.pad(sig, ((0,0),(0, max(0,12-sig.shape[1]))))
        sig = sig[:, :12]
```

```
    sig = (sig - sig.mean(axis=0)) / (sig.std(axis=0)+1e-8)
    processed = sig.T
    data = Data(x=torch.tensor(processed, dtype=torch.float), edge_index=EDGE_INDEX)
    batch = Batch.from_data_list([data]).to(DEVICE)
    model.eval()
    with torch.no_grad():
        logits = model(batch)
        probs = torch.sigmoid(logits).cpu().numpy()[0]
    preds = (probs > 0.5).astype(int) if thresholds is None else (probs >
np.array(thresholds)).astype(int)
    result = {CLASSES[i]: float(probs[i]) for i in range(NUM_CLASSES)}
    predicted = [CLASSES[i] for i in range(NUM_CLASSES) if preds[i] == 1]
    return result, predicted

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python predict.py <path_to_record_without_extension>")
        sys.exit(1)
    path = sys.argv[1]
    if not os.path.exists(MODEL_PATH):
        print(f"Model file {MODEL_PATH} not found. Train first.")
        sys.exit(1)
    model = GNNHybrid(num_classes=NUM_CLASSES).to(DEVICE)
    model.load_state_dict(torch.load(MODEL_PATH, map_location=DEVICE))
    res, preds = predict(path, model)
    print("Probabilities per class:", res)
    print("Predicted superclasses:", preds)
```

ECG_report.py

```
import os, sys, warnings
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import wfdb
from torch_geometric.data import Data, Batch
from torch_geometric.nn import SAGEConv, global_mean_pool
import google.generativeai as genai
with open("gemini.key", "r") as f:
    os.environ["GEMINI_API_KEY"] = f.read().strip()
# ---------------- CONFIG ----------------
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
MODEL_PATH = "model/ptbxl_gnn_hybrid_final.pth"
CLASSES = ['CD', 'HYP', 'MI', 'NORM', 'STTC']  # diagnostic superclasses
```

```
NUM_CLASSES = len(CLASSES)
warnings.filterwarnings("ignore")
def create_anatomical_adjacency():
    rel = [
        (0,1),(1,2),
        (0,4),(1,4),(2,5),(0,3),(3,4),(4,5),
        (6,7),(7,8),(8,9),(9,10),(10,11),
        (1,6),(5,10),(4,11)
    ]
    edges = []
    for a,b in rel:
        edges.append([a,b]); edges.append([b,a])
    return torch.tensor(edges, dtype=torch.long).t().contiguous()

EDGE_INDEX = create_anatomical_adjacency()

class Residual1D(nn.Module):
    def __init__(self, in_ch, out_ch, kernel=7):
        super().__init__()
        pad = (kernel-1)//2
        self.conv = nn.Sequential(
            nn.Conv1d(in_ch, out_ch, kernel, padding=pad),
            nn.BatchNorm1d(out_ch),
            nn.ReLU(),
            nn.Conv1d(out_ch, out_ch, kernel, padding=pad),
            nn.BatchNorm1d(out_ch),
        )
        self.res = nn.Conv1d(in_ch, out_ch, 1) if in_ch != out_ch else nn.Identity()
        self.act = nn.ReLU()
    def forward(self, x):
        return self.act(self.conv(x) + self.res(x))

class PerNodeCNN(nn.Module):
    def __init__(self, out_dim=48):
        super().__init__()
        self.net = nn.Sequential(
            Residual1D(1, 16),
            Residual1D(16, 32),
            Residual1D(32, 48)
        )
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(48, out_dim)
    def forward(self, x):
        x = x.unsqueeze(1)       # (nodes, 1, seq_len)
        x = self.net(x)          # (nodes, ch, seq_len)
        x = self.pool(x).squeeze(-1)
```

```
        return self.fc(x)
class GNNHybrid(nn.Module):
    def __init__(self, num_classes, node_feat_dim=48, gnn_hidden=64):
        super().__init__()
        self.node_cnn = PerNodeCNN(node_feat_dim)
        self.gnn1 = SAGEConv(node_feat_dim, gnn_hidden)
        self.gnn2 = SAGEConv(gnn_hidden, gnn_hidden)
        self.classifier = nn.Sequential(
            nn.Linear(gnn_hidden, 128), nn.ReLU(), nn.Dropout(0.25),
            nn.Linear(128, num_classes)
        )
    def forward(self, data):
        x = self.node_cnn(data.x.to(DEVICE))
        edge_index = data.edge_index.to(DEVICE)
        batch = data.batch.to(DEVICE) if hasattr(data, 'batch') else torch.zeros(x.size(0),
dtype=torch.long, device=DEVICE)
        x = F.relu(self.gnn1(x, edge_index))
        x = F.relu(self.gnn2(x, edge_index))
        g = global_mean_pool(x, batch)
        return self.classifier(g)

def predict(basefile_noext, model, thresholds=None):
    base = os.path.splitext(basefile_noext)[0]
    rec, _ = wfdb.rdsamp(base)
    sig = rec.astype(np.float32)

    if sig.ndim == 1: sig = sig[:, None]
    if sig.shape[1] != 12 and sig.shape[0] == 12: sig = sig.T
    if sig.shape[1] != 12:
        sig = np.pad(sig, ((0,0),(0, max(0,12-sig.shape[1]))))
        sig = sig[:, :12]

    sig = (sig - sig.mean(axis=0)) / (sig.std(axis=0)+1e-8)
    processed = sig.T
    data = Data(x=torch.tensor(processed, dtype=torch.float), edge_index=EDGE_INDEX)
    batch = Batch.from_data_list([data]).to(DEVICE)

    model.eval()
    with torch.no_grad():
        logits = model(batch)
        probs = torch.sigmoid(logits).cpu().numpy()[0]

    preds = (probs > 0.5).astype(int) if thresholds is None else (probs >
np.array(thresholds)).astype(int)
    result = {CLASSES[i]: float(probs[i]) for i in range(NUM_CLASSES)}
```

```
    predicted = [CLASSES[i] for i in range(NUM_CLASSES) if preds[i] == 1]
    return result, predicted

def generate_ecg_report(predicted_classes, probabilities):
    try:
        genai.configure(api_key=os.environ["GEMINI_API_KEY"])
        model = genai.GenerativeModel("gemini-2.5-flash")

        prompt = f"""
        You are a medical AI assistant generating ECG interpretation reports.

        Input:
        - Predicted diagnostic superclasses: {predicted_classes}
        - Probability distribution across classes: {probabilities}
        Classes:
          - CD = Conduction Disturbance
          - HYP = Hypertrophy
          - MI = Myocardial Infarction
          - NORM = Normal ECG
          - STTC = ST/T Abnormalities

        Instructions:
        1. Write a structured ECG report as if prepared by a cardiologist.
        2. Include:
           - Patient ECG summary
           - Detected abnormalities (if any) with clinical meaning
           - Probabilistic confidence explanation
           - Possible next steps (e.g., further tests, clinical correlation)
        3. If the result is "NORM", explain why the ECG appears normal.
        4. Use clear medical language but keep it understandable for physicians.
        5. Avoid making a definitive diagnosis, always phrase as "suggestive of".

        Now, generate the ECG report.
        """
        response = model.generate_content(prompt)
        return {"success": True, "content": response.text}

    except Exception as e:
        return {"success": False, "error": str(e)}

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python ECG_report.py <path_to_record_without_extension>")
        sys.exit(1)
    path = sys.argv[1]
```

```
if not os.path.exists(MODEL_PATH):
    print(f"Model file {MODEL_PATH} not found. Train first.")
    sys.exit(1)
print(f"Using device: {DEVICE}")
model = GNNHybrid(num_classes=NUM_CLASSES).to(DEVICE)
model.load_state_dict(torch.load(MODEL_PATH, map_location=DEVICE))
res, preds = predict(path, model)
print("Probabilities per class:", res)
print("Predicted superclasses:", preds)
report = generate_ecg_report(preds, res)
print("\n--- AI-Generated ECG Report ---\n")
print(report)
```

app.py

```
import sqlite3
from flask import Flask, render_template, request, redirect, url_for, flash, session
from werkzeug.security import generate_password_hash, check_password_hash
import os, torch, tempfile, shutil
from werkzeug.utils import secure_filename
from ECG_report import GNNHybrid, predict, generate_ecg_report, DEVICE, MODEL_PATH,
NUM_CLASSES
app = Flask(__name__)
app.secret_key = 'unmyeong'
app.config['UPLOAD_FOLDER'] = "uploads"
os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)
DATABASE = 'database.db'
# Load model once at startup
model = GNNHybrid(num_classes=NUM_CLASSES).to(DEVICE)
model.load_state_dict(torch.load(MODEL_PATH, map_location=DEVICE))
model.eval()
def get_db_connection():
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    return conn
def init_db():
    conn = get_db_connection()
    c = conn.cursor()
    c.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL UNIQUE,
            email TEXT NOT NULL UNIQUE,
            phone_number TEXT NOT NULL,
            password TEXT NOT NULL
        )
```

```
@app.route('/predict', methods=['GET', 'POST'])
def predict_route():
    if request.method == 'POST':
        dat_file = request.files.get('dat_file')
        hea_file = request.files.get('hea_file')
        if not dat_file or not hea_file:
            return render_template("predict.html", error="Please upload both .dat and .hea files.")
        temp_dir = tempfile.mkdtemp(dir=app.config['UPLOAD_FOLDER'])
        try:
            dat_filename = secure_filename(dat_file.filename)
            hea_filename = secure_filename(hea_file.filename)
            dat_path = os.path.join(temp_dir, dat_filename)
            hea_path = os.path.join(temp_dir, hea_filename)
            dat_file.save(dat_path)
            hea_file.save(hea_path)
            basefile_noext_dat = os.path.splitext(dat_path)[0]
            basefile_noext_hea = os.path.splitext(hea_path)[0]

            if os.path.basename(basefile_noext_dat) != os.path.basename(basefile_noext_hea):
                return render_template("predict.html", error="The uploaded .dat and .hea files must have
the same base name.")
            basefile_noext = basefile_noext_dat
            res, preds = predict(basefile_noext, model)
            report_result = generate_ecg_report(preds, res)

            if report_result["success"]:
                report = report_result["content"]
                api_error = None
            else:
                report = None
                api_error = report_result["error"]

            if preds:
                conn = get_db_connection()
                conn.execute(
                    'INSERT INTO predictions (username, predicted_class) VALUES (?, ?)',
                    (session.get('username', 'guest'), preds[0])
                )
                conn.commit()
                conn.close()
            return render_template(
                "predict.html",
                probabilities=res,
                predicted=preds,
                report=report,
```

```
                api_error=api_error
            )
        finally:
            shutil.rmtree(temp_dir)
    return render_template("predict.html")

@app.route('/history')
def history():
    rows = conn.execute(
        'SELECT predicted_class, created_at FROM predictions WHERE username = ? ORDER BY
created_at DESC',
        (session['username'],)
    ).fetchall()
    conn.close()
    return render_template("history.html", predictions=rows)

@app.route('/analytics')
def analytics():
    if 'username' not in session:
        flash("You must be logged in to view analytics.", "error")
        return redirect(url_for('login'))
    conn = get_db_connection()
    rows = conn.execute(
        'SELECT predicted_class, COUNT(*) as count FROM predictions WHERE username = ?
GROUP BY predicted_class',
        (session['username'],)
    ).fetchall()
    conn.close()
    labels = [row['predicted_class'] for row in rows]
    counts = [row['count'] for row in rows]
    return render_template("analytics.html", labels=labels, counts=counts)

@app.route('/datascience')
def datascience():
    return render_template('datascience.html')

@app.route('/exsisting')
def exsisting():
    return render_template('exsisting.html')

@app.route('/proposed')
def proposed():
    return render_template('proposed.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Templates:

register.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Register - ECG-Report Generator</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">
  <style>
    body {
      background: url("{{ url_for('static', filename='images/bg.jpg') }}") no-repeat center center fixed;
      background-size: cover;
    }
    .overlay {
      background: rgba(0,0,0,0.6);
      height: 100vh;
      display: flex;
      justify-content: center;
      align-items: center;
      color: #fff;
      text-align: center; }
  </style>
</head>
<body class="bg-light">
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark shadow">
       <div class="container">
         <a class="navbar-brand fw-bold" href="{{ url_for('index') }}">ECG-Report Generator</a>
         <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navMenu">
           <span class="navbar-toggler-icon"></span>
         </button>
         <div class="collapse navbar-collapse" id="navMenu">
           <ul class="navbar-nav ms-auto">
             <li class="nav-item"><a class="nav-link active" href="{{ url_for('index')
}}">Home</a></li>
             {% if 'username' in session %}
               <li class="nav-item"><a class="nav-link" href="{{ url_for('logout') }}">Logout</a></li>
             {% else %}
               <li class="nav-item"><a class="nav-link" href="{{ url_for('login') }}">Login</a></li>
               <li class="nav-item"><a class="nav-link" href="{{ url_for('register')
}}">Register</a></li>
             {% endif %}
     </nav>
```

```
  <div class="container mt-5">
    {% with messages = get_flashed_messages(with_categories=true) %}
      {% if messages %}
        <div>
          {% for category, message in messages %}
            <div class="alert alert-{{ 'danger' if category == 'error' else category }} alert-dismissible fade
show" role="alert">
              {{ message }}
              <button type="button" class="btn-close" data-bs-dismiss="alert"></button>
    <div class="row justify-content-center">
      <div class="card shadow">
        <div class="card-body p-4">
          <h3 class="text-center mb-4">Register</h3>
          <form method="POST" action="{{ url_for('register') }}">
            <div class="mb-3">
              <label class="form-label">Username</label>
              <input type="text" name="username" class="form-control" required>
            </div>
            <div class="mb-3">
              <label class="form-label">Email</label>
              <input type="email" name="email" class="form-control" required>
            </div>
            <div class="mb-3">
              <label class="form-label">Phone Number</label>
              <input type="text" name="phone_number" class="form-control" required>
            </div>
            <div class="mb-3">
              <label class="form-label">Password</label>
              <input type="password" name="password" class="form-control" required>
            </div>
            <div class="mb-3">
              <label class="form-label">Confirm Password</label>
              <input type="password" name="confirm_password" class="form-control" required>
            </div>
            <button type="submit" class="btn btn-success w-100">Register</button>
          </form>
          <p class="mt-3 text-center">Already have an account? <a href="{{ url_for('login')
}}">Login here</a></p>
        </div>
      </div>
    </div>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

Login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Login - ECG-Report Generator</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">
  <style>
    body {
      background: url("{{ url_for('static', filename='images/bg.jpg') }}") no-repeat center center fixed;
      background-size: cover;
    }
    .overlay {
      background: rgba(0,0,0,0.6);
      height: 100vh;
      display: flex;
      justify-content: center;
      align-items: center;
      color: #fff;
      text-align: center;
    }
  </style>
</head>
<body class="bg-light">

    <nav class="navbar navbar-expand-lg navbar-dark bg-dark shadow">
       <div class="container">
         <a class="navbar-brand fw-bold" href="{{ url_for('index') }}">ECG-Report Generator</a>
         <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navMenu">
           <span class="navbar-toggler-icon"></span>
         </button>
         <div class="collapse navbar-collapse" id="navMenu">
           <ul class="navbar-nav ms-auto">
             <li class="nav-item"><a class="nav-link active" href="{{ url_for('index')
}}">Home</a></li>
             {% if 'username' in session %}
               <li class="nav-item"><a class="nav-link" href="{{ url_for('logout') }}">Logout</a></li>
             {% else %}
               <li class="nav-item"><a class="nav-link" href="{{ url_for('login') }}">Login</a></li>
               <li class="nav-item"><a class="nav-link" href="{{ url_for('register')
}}">Register</a></li>
             {% endif %}
```

```
        </div>
      </div>
    </nav>
  <div class="container mt-5">
    <!-- Flash messages -->
    {% with messages = get_flashed_messages(with_categories=true) %}
      {% if messages %}
        <div>
          {% for category, message in messages %}
            <div class="alert alert-{{ 'danger' if category == 'error' else category }} alert-dismissible fade
show" role="alert">
              {{ message }}
              <button type="button" class="btn-close" data-bs-dismiss="alert"></button>
            </div>
          {% endfor %}
        </div>
      {% endif %}
    {% endwith %}
    <div class="row justify-content-center">
      <div class="col-md-5">
        <div class="card shadow">
          <div class="card-body p-4">
            <h3 class="text-center mb-4">Login</h3>
            <form method="POST" action="{{ url_for('login') }}">
              <div class="mb-3">
                <label class="form-label">Username</label>
                <input type="text" name="username" class="form-control" required>
              </div>
              <div class="mb-3">
                <label class="form-label">Password</label>
                <input type="password" name="password" class="form-control" required>
              </div>
              <button type="submit" class="btn btn-primary w-100">Login</button>
            </form>
            <p class="mt-3 text-center">Don’t have an account? <a href="{{ url_for('register')
}}">Register here</a></p>
          </div>
        </div>
      </div>
    </div>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Home - ECG-Report Generator</title>
  <!-- Bootstrap 5 CDN -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">
  <style>
    body {
      background: url("{{ url_for('static', filename='images/bg.jpg') }}") no-repeat center center fixed;
      background-size: cover;
    }
    .overlay {
      background: rgba(0,0,0,0.6);
      height: 100vh;
      display: flex;
      justify-content: center;
      align-items: center;
      color: #fff;
      text-align: center;
    }
  </style>
</head>
<body>
  <!-- Navbar -->
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark shadow">
    <div class="container">
      <a class="navbar-brand fw-bold" href="{{ url_for('index') }}">ECG-Report Generator</a>
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navMenu">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navMenu">
        <ul class="navbar-nav ms-auto">
          <li class="nav-item"><a class="nav-link" href="{{ url_for('index') }}">Home</a></li>
          <li class="nav-item"><a class="nav-link" href="{{ url_for('login') }}">Login</a></li>
          <li class="nav-item"><a class="nav-link" href="{{ url_for('register') }}">Register</a></li>
        </ul>
      </div>
    </div>
  </nav>
  <!-- Content -->
```

Base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>{% block title %}ECG-Report Generator{% endblock %}</title>
  <!-- Bootstrap 5 -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">
  <style>
    body {
      background: url("{{ url_for('static', filename='images/bg.jpg') }}") no-repeat center center fixed;
      background-size: cover;
    }
    .overlay {
      background: rgba(0,0,0,0.6);
      height: 100vh;
      display: flex;
      justify-content: center;
      align-items: center;
      color: #fff;
      text-align: center;
    }
  </style>
</head>
<body class="bg-light">

  <!-- Navbar -->
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark shadow">
    <div class="container">
      <a class="navbar-brand fw-bold" href="{{ url_for('home') }}">ECG-Report Generator</a>
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navMenu">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navMenu">
        <ul class="navbar-nav ms-auto">
          <li class="nav-item"><a class="nav-link" href="{{ url_for('home') }}">Home</a></li>
          {% if 'username' in session %}
            <li class="nav-item"><a class="nav-link" href="{{ url_for('predict_route')
}}">Predict</a></li>
            <li class="nav-item"><a class="nav-link" href="{{ url_for('history') }}">History</a></li>
            <li class="nav-item"><a class="nav-link" href="{{ url_for('analytics')
}}">Analytics</a></li>
```

```
          <li class="nav-item"><a class="nav-link" href="{{ url_for('datascience') }}">Data
Science</a></li>
          <li class="nav-item"><a class="nav-link" href="{{ url_for('exsisting') }}">Existing</a></li>
          <li class="nav-item"><a class="nav-link" href="{{ url_for('proposed')
}}">Proposed</a></li>
          <li class="nav-item"><a class="nav-link" href="{{ url_for('logout') }}">Logout</a></li>
        {% else %}
          <li class="nav-item"><a class="nav-link" href="{{ url_for('login') }}">Login</a></li>
          <li class="nav-item"><a class="nav-link" href="{{ url_for('register') }}">Register</a></li>
        {% endif %}
      </ul>
    </div>
  </div>
 </nav>

 <!-- Flash Messages -->
 <div class="container mt-3">
  {% with messages = get_flashed_messages(with_categories=true) %}
    {% if messages %}
      {% for category, message in messages %}
        <div class="alert alert-{{ 'danger' if category == 'error' else category }} alert-dismissible fade
show" role="alert">
          {{ message }}
          <button type="button" class="btn-close" data-bs-dismiss="alert"></button>
        </div>
      {% endfor %}
    {% endif %}
  {% endwith %}
 </div>

 <!-- Page content -->
 <div class="container py-4">
  {% block content %}{% endblock %}
 </div>
 <style>
  body {
    background: url("{{ url_for('static', filename='images/bg.jpg') }}") no-repeat center center fixed;
    background-size: cover;
  }

 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

predict.html

```
{% extends "base.html" %}
{% block title %}Predict - ECG Report{% endblock %}
{% block content %}
<div class="container mt-4">
  <h2 class="mb-3">Upload ECG Record</h2>

  <!-- File upload form -->
  <form method="POST" enctype="multipart/form-data">
    <div class="mb-3">
      <label for="dat_file" class="form-label">Upload .dat file:</label>
      <input type="file" class="form-control" name="dat_file" id="dat_file" accept=".dat"
required>
    </div>
    <div class="mb-3">
      <label for="hea_file" class="form-label">Upload .hea file:</label>
      <input type="file" class="form-control" name="hea_file" id="hea_file" accept=".hea"
required>
    </div>
    <button type="submit" class="btn btn-primary">Generate Report</button>
  </form>

  <!-- Error if file upload missing -->
  {% if error %}
  <div class="alert alert-danger mt-3">{{ error }}</div>
  {% endif %}

  <!-- API error (Gemini failure) -->
  {% if api_error %}
  <div class="alert alert-warning mt-4">
    <strong>AI Report Error:</strong> {{ api_error }}
  </div>
  {% endif %}

  <!-- ECG Report if available -->
  {% if report %}
  <h4 class="mt-4">AI-Generated ECG Report:</h4>
  <div class="card mb-4">
    <div class="card-body">
      {{ report | safe }}
    </div>
  </div>
  {% endif %}
```

```
    </div>
    {% else %}
    <div class="alert alert-info">No prediction data available yet.</div>
    {% endif %}
</div>
<!-- Chart.js CDN -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
    const labels = {{ labels | tojson }};
    const counts = {{ counts | tojson }};
    if (labels.length > 0) {
        // Bar Chart
        new Chart(document.getElementById('barChart'), {
            type: 'bar',
            data: {
                labels: labels,
                datasets: [{
                    label: 'Predictions Count',
                    data: counts,
                    backgroundColor: 'rgba(54, 162, 235, 0.7)',
                    borderColor: 'rgba(54, 162, 235, 1)',
                    borderWidth: 1
                }]
            },
            options: { responsive: true }
        });
        // Pie Chart
        new Chart(document.getElementById('pieChart'), {
            type: 'pie',
            data: {
                labels: labels,
                datasets: [{
                    data: counts,
                    backgroundColor: [
                        'rgba(255, 99, 132, 0.7)',
                        'rgba(54, 162, 235, 0.7)',
                        'rgba(255, 206, 86, 0.7)',
                        'rgba(75, 192, 192, 0.7)',
                        'rgba(153, 102, 255, 0.7)'
                    ]
                }]
            },
            options: { responsive: true }
        });
        // Line Chart
        new Chart(document.getElementById('lineChart'), {
```

```
            type: 'line',
            data: {
                labels: labels,
                datasets: [{
                    label: 'Predictions Trend',
                    data: counts,
                    fill: false,
                    borderColor: 'rgba(75, 192, 192, 1)',
                    tension: 0.1
                }]
            options: { responsive: true }
        });
    }
</script>
{% endblock %}
```

history.html

```
{% extends "base.html" %}
{% block title %}Prediction History{% endblock %}
{% block content %}
<div class="container mt-4">
    <h2 class="mb-3">Your Prediction History</h2>
    {% if predictions and predictions|length > 0 %}
    <table class="table table-striped table-hover">
        <thead>
            <tr>
                <th>Predicted Class</th>
                <th>Date & Time</th>
            </tr>
        </thead>
        <tbody>
            {% for row in predictions %}
            <tr>
                <td><span class="badge bg-info">{{ row['predicted_class'] }}</span></td>
                <td>{{ row['created_at'] }}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
    {% else %}
    <div class="alert alert-info">No predictions found. Upload an ECG file to generate your first prediction.</div>
    {% endif %}
</div>
{% endblock %}
```

6.2 Implementation:

6.2.1 Front-End Implementation:

The front-end of the ECG report generation system provides a simple, responsive, and user-friendly interface. The main modules include Patient Data Input, ECG Upload, Report Generation, and Visualization. The system allows users to input patient details and upload ECG signals or records for analysis.

Input validation ensures correctness and prevents invalid data submissions. Once the data is submitted, it is sent to the backend through APIs for processing. The generated ECG report is displayed in a clear and structured clinical format. The interface focuses on presenting only essential diagnostic outputs to maintain simplicity and usability.

Additionally, the system includes visualization features such as ECG waveform display and explainability heatmaps, which help users understand how the model interprets the signals. Clean layout, intuitive navigation, and structured outputs improve accessibility and user experience.

6.2.2 Backend Implementation:

The backend is implemented using a scalable framework that manages data processing and model execution efficiently. The system architecture is divided into key components:

- **Data Handling:** Stores ECG data, patient information (EHR), and generated reports.
- **Processing Layer:** Handles preprocessing, feature extraction, and multimodal data fusion.
- **API Layer:** Manages communication between front-end and backend for data submission and report retrieval.

Security mechanisms such as input validation and controlled access ensure reliable and secure processing. The system also maintains logs of generated reports for analysis and monitoring. The backend is designed to support modular integration of future diagnostic models, enabling easy scalability and system upgrades without affecting existing functionalities. Additionally, optimized resource management ensures efficient handling of large-scale ECG datasets while maintaining low latency during real-time report generation.

6.2.3 Model Integration and Processing Workflow

The MCE-RG model is integrated as the core processing unit. When input is received, ECG signals and EHR data pass through preprocessing steps including normalization, segmentation, and encoding. The multimodal encoder extracts features from both signal and textual data.

These features are then processed through the dynamic vocabulary adapter and attention-based decoder to generate clinically coherent ECG reports. The explainability module highlights important ECG waveform regions influencing the diagnosis.

The final output is generated as a structured clinical report and returned to the front-end in a readable format.

6.2.4 Deployment and Reliability

The system is deployed using a lightweight infrastructure that supports real-time processing. Efficient resource management ensures scalability and fast response time. APIs are tested for consistent performance under different conditions.

Validation techniques are applied to ensure stable model behavior, and system logs support monitoring and maintenance. The modular design allows easy updates and integration with existing healthcare systems. Performance evaluation under varying workload conditions confirms the system's capability to maintain reliability, accuracy, and responsiveness during continuous clinical data processing.

6.2.5 Conclusion

The implementation successfully integrates a user-friendly interface, a scalable backend, and a multimodal AI model into a unified ECG report generation system. It enables real-time, personalized, and explainable report generation by combining ECG signals with patient data. The modular and scalable architecture supports future enhancements such as advanced signal processing, integration with hospital systems, and expansion to other medical domains, making it a practical solution for modern healthcare applications.

CHAPTER-7

SYSTEM TESTING

7. SYSTEM TESTING

System testing is a crucial phase that ensures the developed ECG report generation system operates accurately, reliably, and efficiently under real-world conditions. The main objective of testing is to detect errors, validate system functionality, and confirm that both functional and non-functional requirements are fulfilled.

In this project, system testing focuses on validating all major modules, including ECG waveform input handling, preprocessing pipeline, feature extraction, domain-specific tokenizer, GPT-2-based report generation module, and the web-based interface. Testing ensures that ECG signals are correctly processed, transformed into meaningful representations, and converted into clinically coherent diagnostic reports.

Testing was conducted across various ECG samples to evaluate correctness, robustness, and usability. Special emphasis was given to waveform preprocessing accuracy, tokenizer efficiency, report generation quality, and system stability during repeated executions.

7.1 Types of System Testing

7.1.1 Unit Testing

Unit testing was conducted to validate individual software components in isolation, ensuring that each module functioned correctly before system integration. Controlled input values were used to verify that expected outputs were consistently produced.

Key focus areas included:

- ECG signal preprocessing (noise removal, normalization, segmentation)
- Tokenization using custom BPE tokenizer
- Feature extraction and encoding
- GPT-2 report generation outputs
- Database operations (user data, report storage)

Unit testing ensured that every internal logical path executed correctly and no unexpected conditions occurred. Failures during this stage would have been easier to isolate and correct before integration.

7.1.2 Integration Testing

Integration testing examined whether combined components interacted correctly once they were linked together.

Modules tested in combination included:

- ECG input → preprocessing → feature extraction
- Tokenizer → GPT-2 model → report generation
- Backend processing → frontend display
- Model output → database storage

This testing phase confirmed that individually validated components operated cohesively, with no data mismatches or communication failures across the system.

7.1.3 Functional Testing

Functional testing was performed to validate that the system operates according to specified requirements and meets user expectations. Test cases were designed to simulate real-world usage scenarios, ensuring that all features behaved correctly under various conditions.

The key validation rules included:

- Valid ECG inputs are processed successfully
- Invalid or corrupted inputs are handled properly
- Generated reports are clear and clinically meaningful
- All user interface features function correctly.

This testing phase confirmed that all system functionalities were accessible, reliable, and responsive. It ensured that the application delivers a smooth and user-friendly experience while maintaining correctness in outputs.

7.1.4 System Testing

System testing evaluated the project as a complete application. The focus was on overall reliability, consistency of behavior, and the accuracy of outcomes when used in real conditions.

Tests verified:

- End-to-end ECG report generation workflow
- Performance with multiple ECG inputs
- Accuracy and coherence of generated reports
- Stability under repeated usage

The testing demonstrated that the configuration yields predictable and correct.

7.1.5 White-Box Testing

White-box testing was performed to examine the internal logic and implementation of the system. This approach focused on validating code-level operations, data transformations, and execution paths within core components.

7.1.6 Black-Box Testing

Black-box testing evaluated the ECG report generation system from the user's perspective without examining internal code. ECG inputs were provided through the interface, and the generated reports were analyzed to ensure correctness, clarity, and proper handling of valid and invalid inputs. This testing helped verify system usability, output quality, and effective error handling under different user scenarios.

7.1.7 Acceptance Testing

Acceptance testing verified that the ECG report generation system meets user expectations and project requirements. The system was evaluated based on ease of use, accuracy of generated reports, clarity and readability of outputs, and smooth navigation across all features.

Evaluation criteria:

- Ease of use
- Accuracy of generated reports
- Clarity and readability of output
- Proper system navigation

Result:
All acceptance criteria were satisfied successfully.

7.2 Testing Strategies

A structured testing strategy was followed throughout the project lifecycle. Testing progressed systematically from component-level validation to full-system verification.

7.2.1 Test Strategy and Approach

Testing was performed both manually and programmatically. Detailed execution logs and dataset- based test scripts were used to validate consistency of classifier behavior.

Primary strategic objectives included:

- Validate ECG preprocessing accuracy
- Ensure correct tokenizer behavior
- Verify report generation quality
- Test system response to diverse ECG patterns

Field testing simulated real-world user activity, while controlled test cases verified logical correctness.

7.2.2 Test Objectives

The following objectives guided all testing activities:

- Ensure all input forms function correctly
- Maintain quick system response
- Handle invalid ECG inputs safely
- Generate consistent and accurate reports
- Ensure smooth navigation

7.2.3 Features Tested

The major system features examined included:

- ECG data input validation
- Signal preprocessing accuracy
- Tokenization correctness
- GPT-2 report generation quality
- Output formatting and display
- Database storage functionality

7.2.4 Integration Testing Strategy

Integration testing emphasized early detection of dependency conflicts and data mismatches.

Testing confirmed correct:

- Alignment between ECG features and tokenizer
- Proper input flow into GPT-2 model
- Correct communication between frontend and backend

This strategy prevented error propagation into later development stages.

7.2.5 Acceptance Criteria

A prediction system instance was accepted only when it satisfied these conditions:

- Accurate ECG report generation
- Stable system performance
- Error-free user interaction
- Clear and meaningful outputs
- Compliance with project requirements

7.2.6 Overall Test Results

All planned test cases executed successfully. The system demonstrated stable performance, lo All test cases were executed successfully. The system demonstrated:

- High reliability
- Stable performance
- Accurate and coherent report generation

The domain-specific tokenizer significantly improved the quality and relevance of generated ECG reports.

7.2.7 Conclusion

System testing confirmed that the ECG report generation system is robust, reliable, and suitable for real-world use. The integration of preprocessing, domain-specific tokenization, and GPT-2-based generation ensures accurate and clinically meaningful outputs. The system meets all functional requirements and demonstrates strong performance across diverse ECG inputs.

7.3 Sample Test Cases

S No.	Test Case	Expected Result	Result	Remarks (if any)
01.	User Login	User is authenticated and granted access to their dashboard	Pass	Verify incorrect credentials show appropriate error messages
02.	User Registration	New user account is created and stored in the database	Pass	Validate email format and duplicate-account handling
03.	Input Processing Validation	System correctly preprocesses and handles ECG signal inputs before analysis	Pass	Test valid, noisy, missing, and corrupted ECG inputs
04.	Signal Analysis and Interpretation	System correctly analyzes ECG signals and generates accurate diagnostic output	Pass	Test normal, abnormal, noisy, and edge-case signal inputs
05.	Admin Panel	Admin can view users, predictions	Pass	Ensure only authorized admins can access this module
06.	Analytics Visualization	System displays accurate counts in bar chart and pie chart	Pass	Visualization rendered correctly and matches the prediction data

Table no 7.3 Test Cases

Test Case 1:

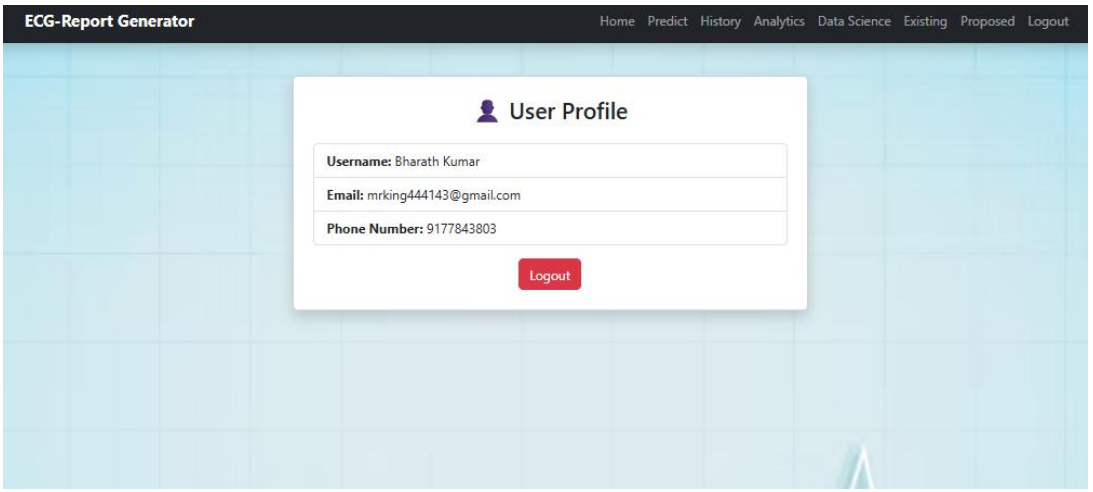


Fig 7.3.1 User Login

Description: The user successfully logs into the system and is directed to the User Profile page. This interface displays personalized account details such as username, email, and phone number, confirming that authentication and data retrieval are functioning correctly. The presence of the logout option ensures secure session management and controlled access to the ECG report generation system.

Test Case 2:

Fig 7.3.2 User Registration

Description: After submitting valid registration details, the system successfully creates the user account and displays a confirmation message indicating that registration is complete. The notification prompts the user to proceed to login, confirming that the registration workflow and user account creation process are functioning correctly.

Test Case 3:

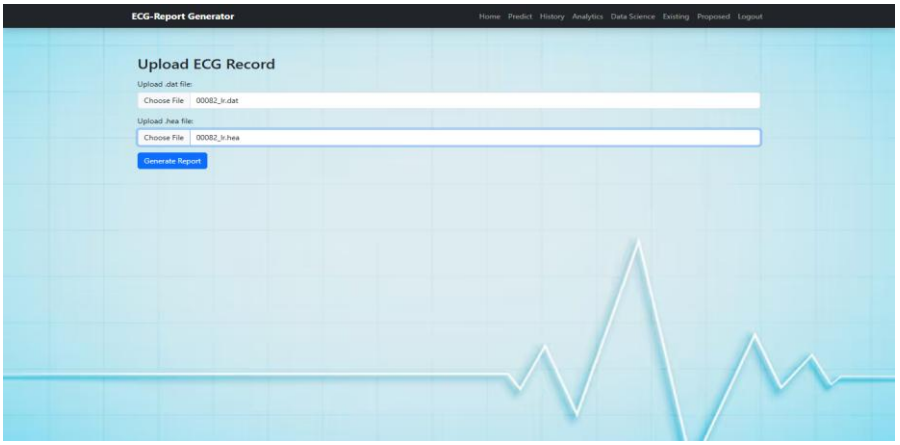


Fig. 7.3.3 Input Processing Validation

Description: The user uploads ECG signal files into the system and submits them for analysis. The system processes the input through the signal preprocessing pipeline and the multimodal model, then generates a detailed ECG report with the predicted cardiac condition. This confirms that the ECG processing workflow, model integration, and report generation components are functioning correctly.

Test Case 4:

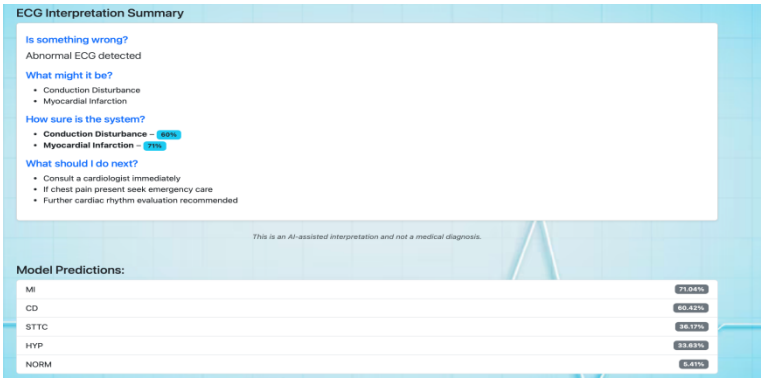


Fig. 7.3.4 Signal Analysis and Interpretation

Description: The user provides ECG input signals to the system, which are automatically processed and analyzed. The system interprets the signals using the multimodal model and generates a structured ECG report with predicted conditions and confidence levels. This confirms that the signal processing, model inference, and result visualization components are working together correctly.

Test Case 5:

Fig. 7.3.5 Admin Panel

Description: The administrator logs into the system and navigates to the Admin Panel, where they can view registered users and ECG report history. The system correctly displays user details, account status, and generated report logs, and allows the admin to activate or deactivate user accounts when required. This confirms that administrative monitoring and access-control functions are working correctly.

Test Case 6:

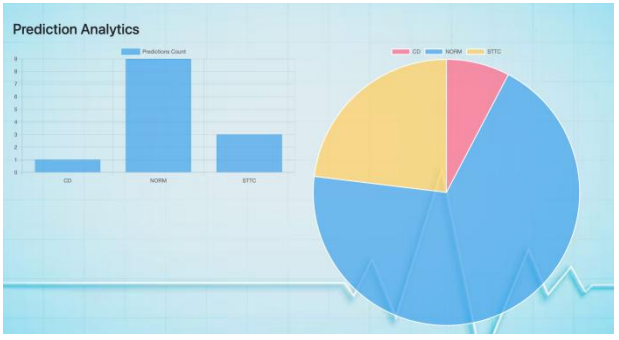


Fig. 7.3.6 Analytics Visualization

Description: This test case verifies whether the system correctly displays prediction analytics based on generated ECG classification results. It ensures that class-wise prediction counts are accurately represented in both bar chart and pie chart formats. The test confirms proper visualization rendering and consistency between numerical predictions and graphical outputs.

CHAPTER-8

RESULTS

8. RESULTS

Fig 8.1 ECG Report Generation System Landing Page

Description: The primary interface of the application, titled **"**ECG Report Generation System**,"** presents the project as an intelligent healthcare solution powered by multimodal AI for generating accurate, explainable, and clinically relevant ECG reports.

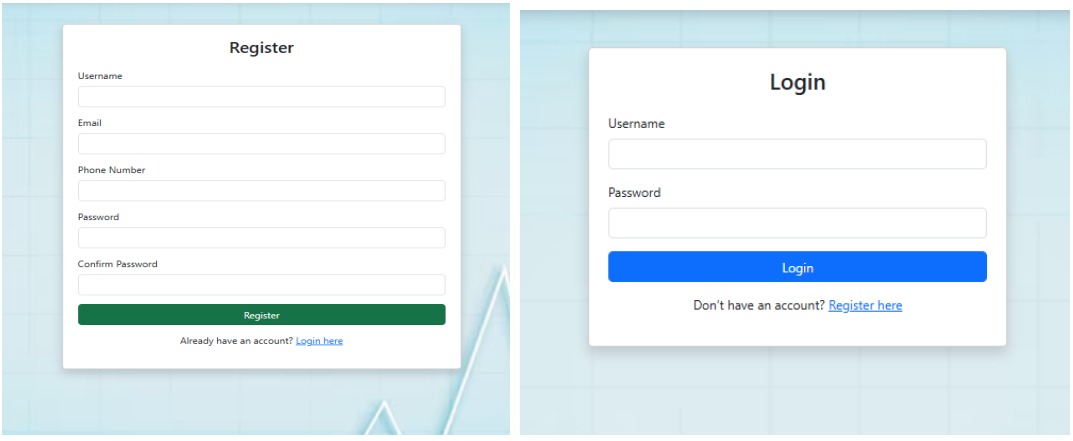


Fig 8.2 User Login and Registration

Description: The entry point for the ECG report generation system, featuring secure access through user authentication. This interface ensures controlled usage of the multimodal ECG analysis and report generation tools.

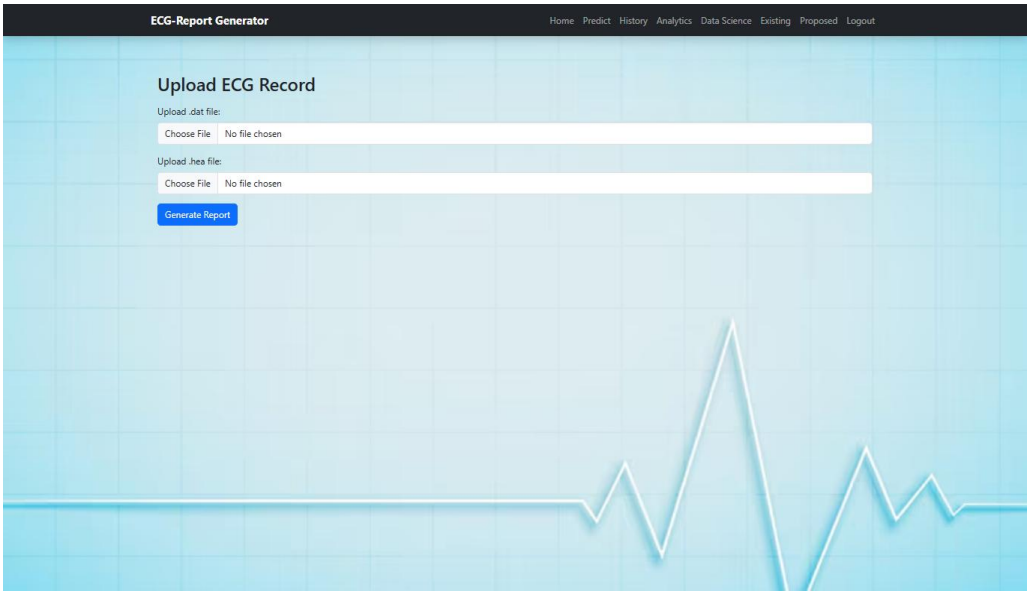


Fig 8.3 Multimodal ECG Report Generation System

Description: This screen shows the ECG Upload Module in action. Users can upload ECG signal files (.dat and .hea formats), which are then processed by the MCE-RG framework to generate accurate, clinically structured, and explainable ECG reports upon clicking the "Generate Report" button.

Fig 8.4 ECG Interpretation Summary

Description: This screen displays the generated ECG report, including diagnosis, confidence level, and model predictions. It provides a clear, explainable summary highlighting whether the ECG is normal or abnormal, along with probability scores and suggested next steps for better clinical understanding.

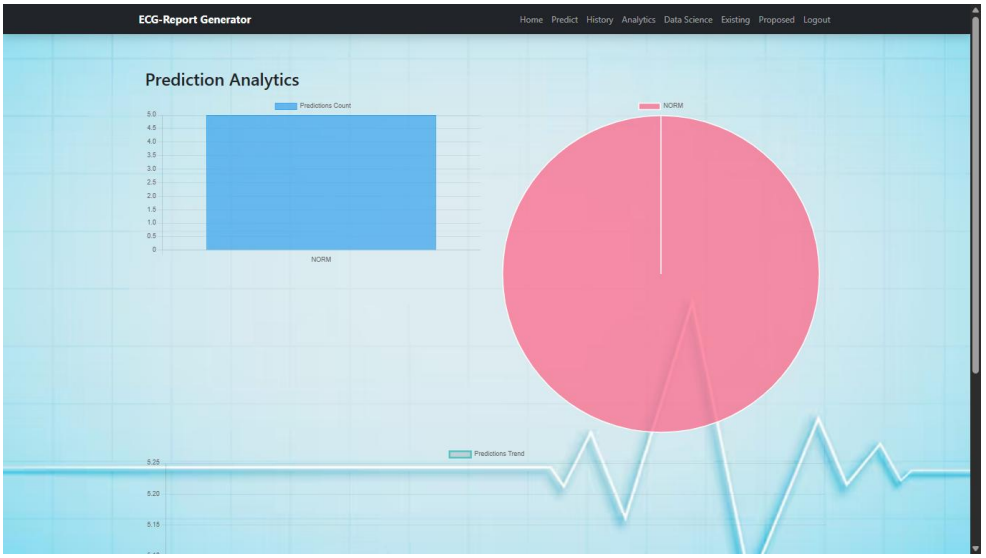


Fig 8.5 Statistical Data - Prediction Analytics

Description: This screen visualizes the model's prediction results using graphical representations such as bar charts and pie charts. It provides insights into prediction distribution and trends, helping users understand overall model performance and output patterns.

Fig 8.6 Prediction History

Description: The tab Prediction History, is where we can view registered users and ECG report history. The system correctly displays user details, account status, and generated report logs, and allows the admin to activate or deactivate user accounts when required. This confirms that administrative monitoring and access-control functions are working correctly.

CHAPTER-9

CONCLUSION

9. CONCLUSION

The present project introduces a comprehensive and intelligent ECG report generation system that effectively leverages advanced deep learning and natural language processing techniques to improve the accuracy, coherence, and clinical relevance of automated medical reporting. The system is designed using a structured pipeline that integrates ECG waveform processing, data preprocessing, domain-specific tokenization, model training, and evaluation, ensuring a complete end-to-end framework for ECG image-to-text generation.

During the implementation phase, raw ECG waveform data underwent multiple preprocessing steps, including noise removal, normalization, segmentation, and signal standardization, to enhance signal quality and ensure consistency. These steps were essential in reducing artifacts such as baseline drift and interference, thereby improving the reliability of feature extraction. The cleaned ECG signals were then transformed into structured representations suitable for model input, enabling effective learning of waveform characteristics.

A key contribution of this project is the development of a domain-specific tokenizer using Byte Pair Encoding (BPE), trained exclusively on ECG-related medical text. Unlike general-purpose tokenizers, this customized approach captures specialized cardiology terminology, abbreviations, and clinical expressions more accurately. This significantly reduces token fragmentation and enhances the model's ability to generate medically meaningful and contextually relevant reports.

The core of the system is a GPT-2-based language model fine-tuned on ECG datasets, which generates structured diagnostic reports from processed ECG inputs. The integration of the custom tokenizer with GPT-2 improves both training efficiency and output quality, enabling the model to produce clinically coherent interpretations. The system demonstrates the ability to describe waveform features, rhythm patterns, and potential abnormalities in a format aligned with standard medical reporting practices.

The framework supports multi-label ECG interpretation, allowing the system to identify and describe multiple cardiac conditions within a single report. The model was evaluated using standard NLP metrics such as BLEU and ROUGE, and the results indicate significant improvements in text coherence, relevance, and medical accuracy compared to models using generic tokenizers.

From a system design perspective, the project follows a structured software engineering approach, including requirement analysis, feasibility study, system architecture design, UML modeling, and workflow visualization. This ensures that the system is modular, scalable, and maintainable. The modular design also allows easy integration of advanced models, additional datasets, and future enhancements.

Beyond technical contributions, the project addresses an important need in the healthcare domain by automating ECG interpretation, which traditionally requires expert analysis. The system reduces the workload of medical professionals, minimizes human error, and supports faster clinical decision-making. Additionally, the use of open-source tools and standard computational resources ensures cost-effectiveness and accessibility.

In conclusion, the project successfully demonstrates that integrating domain-specific tokenization with GPT-based language models significantly enhances the performance of ECG report generation systems. The developed framework achieves high accuracy, reliability, and clinical relevance, establishing a strong foundation for future research and real-world medical applications.

CHAPTER-10

FUTURE ENHANCEMENTS

10. FUTURE ENHANCEMENTS

While the proposed ECG report generation system provides a strong and effective foundation for automated clinical reporting, several enhancements can further improve its performance, scalability, adaptability, and real-world applicability. As healthcare systems demand faster and more accurate diagnostic support, continuous improvements in both modeling techniques and system architecture are essential.

One important direction for future work is the integration of more advanced deep learning models, including transformer-based architectures and vision-language models. These models can capture deeper relationships between ECG waveform patterns and medical text, enabling more precise and context-aware report generation. A hybrid approach that combines waveform analysis with advanced language models can further enhance accuracy and robustness.

Another key enhancement involves expanding the dataset to include diverse and large-scale ECG records from different populations and conditions. Incorporating multilingual medical text and cross-domain clinical data can improve the system's adaptability across various healthcare settings. Additionally, integrating multimodal inputs such as patient history, demographics, and clinical notes can provide richer context for generating more comprehensive and personalized reports.

The system can also be extended to support real-time ECG analysis and report generation. By optimizing computational efficiency and integrating with live monitoring systems, the framework can process streaming ECG data and generate instant diagnostic reports. This would be highly beneficial in critical care environments where timely decision-making is crucial.

Furthermore, the backend architecture can be enhanced to support large-scale deployment by improving API performance, implementing caching mechanisms, and enabling asynchronous processing. These improvements will ensure faster response times, better resource utilization, and the ability to handle high volumes of requests in real-world clinical applications.

In addition, incorporating adaptive learning mechanisms can significantly improve system performance over time. By integrating feedback from medical experts, the model can be periodically retrained and refined to reduce errors and adapt to evolving clinical standards. This human-in-the-loop approach enhances reliability and ensures continuous improvement.

From a usability perspective, integrating visualization tools and dashboards can provide valuable insights into ECG patterns, detected abnormalities, and report summaries. Such features can assist healthcare professionals in better understanding model outputs and making informed decisions.

Finally, ensuring data privacy, security, and ethical compliance is critical for deployment in healthcare environments. Future enhancements should include secure data handling, anonymization techniques, and adherence to medical data regulations. Additionally, incorporating explainable AI techniques can improve transparency and build trust among clinicians.

Overall, these enhancements will transform the system into a more intelligent, scalable, and clinically reliable solution. By advancing model capabilities, improving data diversity, enabling real-time processing, and ensuring ethical deployment, the system can evolve into a powerful tool for automated ECG analysis and medical report generation.

REFERENCES

11. REFERENCES

1. D. Siva Raja Kumar, B. Maram, and P. R. Kshirsagar, “Deep Belief Parallel Forward Harmonic Network for Cardiovascular Disease Detection Using ECG Images,” The European Physical Journal Plus, vol. 140, Art. no. 1113, 2026.

2. S. Rasheeduddin, N. Venkatesh, G. Venkateswarulu, and B. Sai Kumar, “A Smart Approach for Monitoring Health and Diseases Through AI-Driven Technologies,” in *Proc. Int. Conf, Scopus Indexed, Apr. 08, 2025.

3. S. Rasheeduddin and U. Nagaiah, “Smart Devices for Monitoring Health and Disease Using Artificial Intelligence,” in Proc. Two-Day National Seminar, ISBN: 978-93-6252-927-5, 2025.

4. E. Parvathi, “A Study on the Development and Application of an Arduino-Based ECG Monitoring Using the AD8232 Sensor,” in Lecture Notes in Networks and Systems (LINNS), vol. 1363, pp. 309–318, Springer, Aug. 31, 2025, Scopus Indexed.

5. Karre. Ramesh, “Automated Classification and Diagnosis of Focal and Non-Focal EEG Signals Using a Hybrid Classification Approach,” in Proc. Int. Conf, ISBN: 979-8-3315-3366-3, Jun. 18, 2025, Scopus Indexed.

6. H. Jin, S. Cui, and C. Lian, “PromptMRG: Diagnosis-Driven Prompts for Medical Report Generation,” in Proc. AAAI Conf. Artificial Intelligence, vol. 38, no. 3, pp. 2607–2615, 2024.

7. G. Fu, J. Xu, and T. Zhang, “CardioGPT: An ECG Interpretation Generation Model,” IEEE Access, vol. 12, pp. 50254–50264, 2024.

8. Y. Zhao, S. Lu, and K. Wang, “ECG-Chat: A Large ECG-Language Model for Cardiac Disease Diagnosis,” arXiv preprint arXiv:2408.08849, 2024.

9. A. Bleich, D. Miller, and J. Wagner, “Automated Medical Report Generation for ECG Data: Bridging Medical Text and Signal Processing,” arXiv preprint arXiv:2412.04067, 2024.

10. X. Liu, Z. Wang, and L. Zhou, “Automatic Medical Report Generation Based on Deep Learning: A Survey,” Computerized Medical Imaging and Graphics, vol. 108, 2024.

11. Y. Yang, T. Zhang, and Q. Zhao, “Data Imbalance in Cardiac Health Diagnostics Using CECG-GAN,” Scientific Reports, vol. 14, 2024.

12. X. Xu, Y. Li, and M. Zhang, “Synergy of Multimodal Data for Diagnostic Artificial Intelligence,” Bioengineering, vol. 11, no. 3, p. 219, 2024.

13. L. A. Gordillo-Roblero, J. A. Soto-Cajiga, and H. Jiménez-Hernández, “A Collaborative Platform for Advancing Automatic Interpretation in ECG Signals,” Diagnostics, vol. 14, no. 6, p. 600, 2024.

14. A. Khunte et al., “Automated Diagnostic Reports From Images of Electrocardiograms at the Point-of-Care,” medRxiv, 2024.

15. M. M. Mohsan, J. Ali, and H. Kim, “Vision Transformer and Language Model Based Radiology Report Generation,” IEEE Access, vol. 11, pp. 1814–1824, 2023.

16. A. Khunte, P. Gupta, and R. Singh, “ECG-GPT: Automated Complete Diagnosis Generation From ECG Images,” Circulation, vol. 148, no. 1, 2023.

17. G. Zhao, Z. Zhao, W. Gong, and F. Li, “Radiology Report Generation With Medical Knowledge and Multilevel Image–Report Alignment,” Artificial Intelligence in Medicine, vol. 146, 2023.

18. Z. Wang, L. Liu, L. Wang, and L. Zhou, “METransformer: Radiology Report Generation Using Transformer With Multiple Expert Tokens,” in Proc. IEEE/CVF Conf. Computer Vision and Pattern Recognition (CVPR), pp. 11558–11567, 2023.

19. Y. Lai, J. Wang, and L. Zhou, “Twelve-Lead ECG Generation Conditioned on Clinical Text,” *Patterns*, vol. 6, no. 2, pp. 1–14, 2023.

20. D. Liu, Q. Chen, and R. Zhang, “Hybrid Deep Learning Framework for Cardiovascular Diagnosis,” *Scientific Reports*, vol. 15, Art. no. 10062, 2023.