MailFokus

AI Powered Email Insights

Team Members:

Full Name : Swapnil Khandoker

Matr_no: k12215556

Full Name: Leonie Rothenbuchner

Matr_no: k42302685

Goal of our System 3
Requirements 5
AI Functional Requirements 7
Use case descriptions 9
Use case diagram 14
Which use cases were implemented in the source code 15
Traceability matrix 16
Domain Model 17
Architecture diagram 18
Components description 19
Design Questions and answers 21
Design Questions 21
Design Question Answers 21

Goal of our System

Q1. What is the domain of your system?
Ans: Management; Can be used in other domains and basically it would be an internal tool for a company.

Q2. Who could be the users of your systems?
Ans: Primarily managers and other people in a company can also use it.

Q3. What is the main goal of your system?
Ans: To help managers or other people in a company to extract important information from emails and manage things accordingly

Q4. How would your system achieve its goal?
Ans: It would be a tool , probably a chrome extension , to extract useful emails for example from gmail and then extract the important information from emails and store them in a database and if possible also post it to some other platforms such as Monday.com or notion and provide additional features such as summarization.

Q5. Which type of AI/ML strategy would be required/useful and why?
Ans: Natural Language processing to analyze the information from emails and LLM to summarize content and extract useful info.

Non-AI Goals:

- Enable managers and company staff to access, organize, and act upon critical email information.

- Store extracted email data in a database.

- Provide integration with third-party platforms (e.g., Monday.com, Notion).

- Offer a user-friendly interface (Chrome extension or web dashboard).

AI-Related Goals:

- Extract key information from unstructured email content.

- Summarize emails or threads for quick understanding.

- Classify and prioritize emails based on importance or category (e.g., task, update, spam).

- Auto-suggest actions or tags based on email content.

Note : *Our primary focus is to deal with customer emails and business email. With business email, we mean to extract information related to business meetings and appointments*
And with customer email , we mean to extract customer queries about products.

Why AI is needed:

- Emails are unstructured and vary widely in language and tone; rule-based systems would be rigid and error-prone.

- NLP and LLMs can understand contextual meaning, extract relevant entities, and summarize text.

- AI allows dynamic adaptability to different email formats and styles without constant manual updates.

Requirements

Stakeholders:

Stakeholder	**Description**
Users	Managers and employees using email for work communication.
People affected	Team members whose tasks depend on correctly extracted info from emails.
Managers	project supervisors overseeing this system’s development.
Regulators	Data privacy and compliance officers (e.g., GDPR authorities).

Function Requirements:

ID	Requirement
Req01	The EmailManager shall allow users to authenticate using their Gmail account.
Req02	When a new email is received, the EmailManager shall fetch the email content and metadata.
Req03	If the email is relevant, the EmailManager shall extract key information and store it in the internal database.
Req04	When the user opens the dashboard, the EmailManager shall display the latest extracted summaries.

Req05	If Monday.com integration is enabled and the email contains a task, the EmailManager shall create a task in Monday.com.
Req06	If Notion integration is enabled, the EmailManager shall post a summary of the email to the specified Notion page.
Req07	When the user clicks on a summarized item, the EmailManager shall display the full email context.
Req08	If the user selects multiple emails, the EmailManager shall generate a unified summary.

Non-Functional Requirements:

Id	**Category**	**Requirement**
NfReq01	External Interfaces	The EmailManager shall interface with the Gmail API to retrieve emails.
NfReq02	Performance	The EmailManager shall generate a summary within 3 seconds for emails under 1000 words.
NfReq03	Attributes	The EmailManager shall store extracted data with AES-256 encryption.

NfReq04	Constraints	The EmailManager shall comply with GDPR by not storing user-identifiable data longer than 30 days.
NfReq05	Attributes	If the summarization model fails, the EmailManager shall alert the user with a fallback message.
NfReq06	External Interfaces	The EmailManager shall support OAuth2.0 for secure user authentication.

AI Functional Requirements

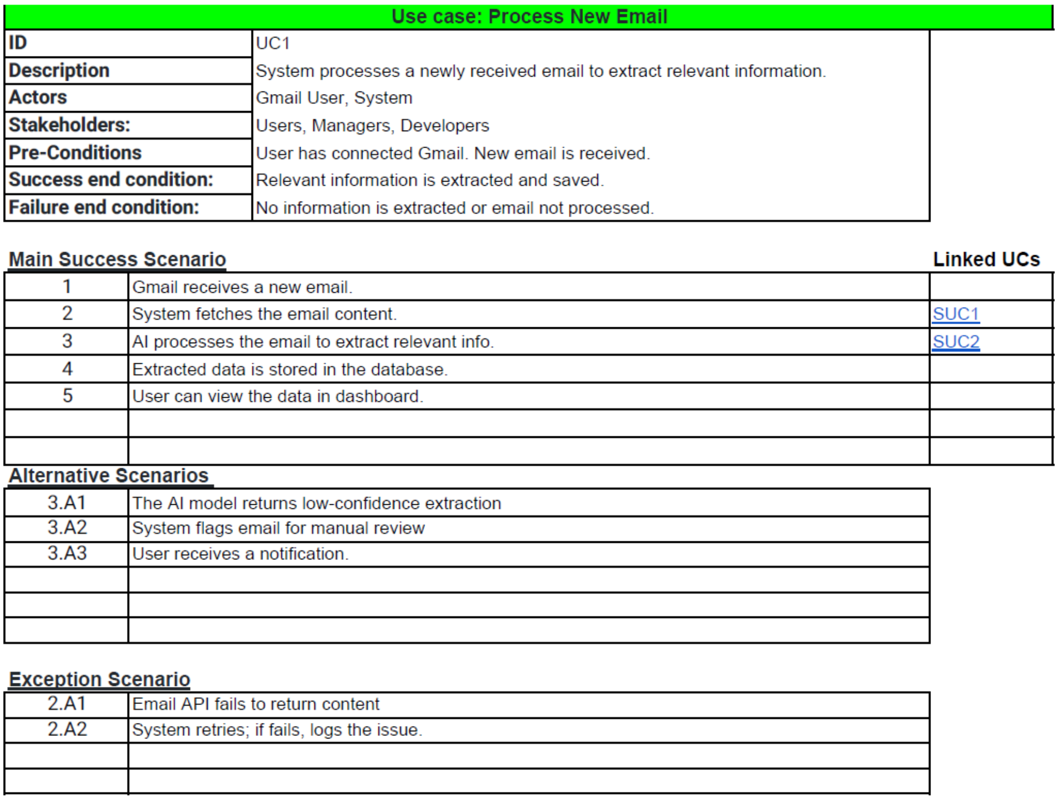
- **Req03:** Extracting information from emails.
  - Why AI? Rule-based systems are insufficient for handling varied sentence structures and contexts.
  - Thresholds: At least 85% accuracy in named entity recognition (NER) and extraction.
  - Implementation: Use pre-trained LLMs (like GPT or BERT) fine-tuned on a custom email dataset.
- **Req04, Req08**: Summarizing content.
  - Thresholds: 80% ROUGE-L score or user satisfaction metric.
  - Implementation: Use LLMs via APIs or fine-tune lightweight transformers (like DistilBERT or T5).
- **Req05:** Creating tasks in Monday.com from email content.

- ○ Thresholds: Correct identification of task-related emails 90% of the time.
- ○ Implementation: Text classification model trained on labeled task vs. non-task emails.

AI Non-Functional Requirements:

Id	**Category**	**Requirement**
NfReq02	Safety	Ensure summarization is accurate and does not mislead the user with incorrect content.
NfReq03	Security and Privacy	Store data securely and comply with data retention rules.
NfReq04	Regulation	Ensure GDPR compliance in handling email data.
NfReq05	Explainability	Provide a "why this was extracted" button showing which part of the email led to a summary.
NfReq06	Transparency and Trust	Notify users that AI is being used and offer manual override if AI fails.

Use case descriptions

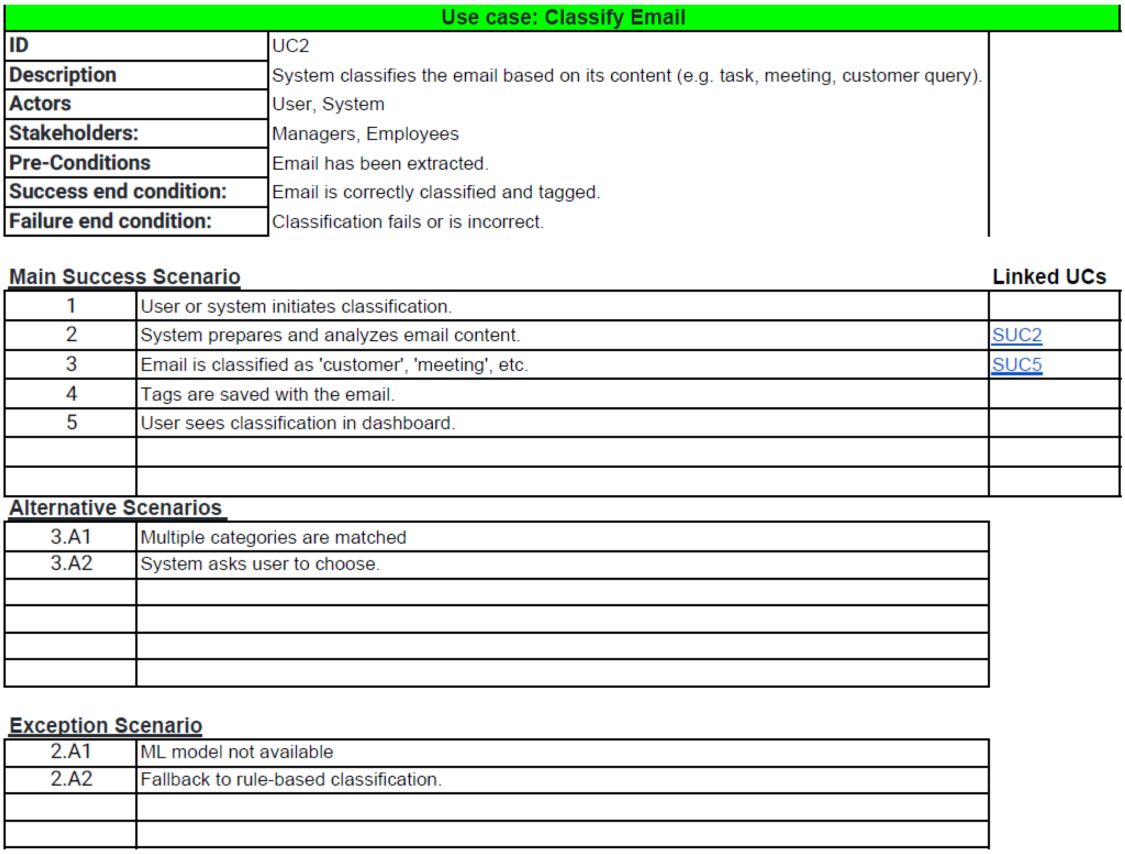


Use case: Process New Email	
ID	UC1
Description	System processes a newly received email to extract relevant information.
Actors	Gmail User, System
Stakeholders:	Users, Managers, Developers
Pre-Conditions	User has connected Gmail. New email is received.
Success end condition:	Relevant information is extracted and saved.
Failure end condition:	No information is extracted or email not processed.

Main Success Scenario		**Linked UCs**
1	Gmail receives a new email.	
2	System fetches the email content.	SUC1
3	AI processes the email to extract relevant info.	SUC2
4	Extracted data is stored in the database.	
5	User can view the data in dashboard.	

Alternative Scenarios	
3.A1	The AI model returns low-confidence extraction
3.A2	System flags email for manual review
3.A3	User receives a notification.

Exception Scenario	
2.A1	Email API fails to return content
2.A2	System retries; if fails, logs the issue.



Use case: Classify Email	
ID	UC2
Description	System classifies the email based on its content (e.g. task, meeting, customer query).
Actors	User, System
Stakeholders:	Managers, Employees
Pre-Conditions	Email has been extracted.
Success end condition:	Email is correctly classified and tagged.
Failure end condition:	Classification fails or is incorrect.

Main Success Scenario		**Linked UCs**
1	User or system initiates classification.	
2	System prepares and analyzes email content.	SUC2
3	Email is classified as 'customer', 'meeting', etc.	SUC5
4	Tags are saved with the email.	
5	User sees classification in dashboard.	

Alternative Scenarios	
3.A1	Multiple categories are matched
3.A2	System asks user to choose.

Exception Scenario	
2.A1	ML model not available
2.A2	Fallback to rule-based classification.

Use case: Summarize Emails	
ID	UC3
Description	User selects one or more emails to summarize.
Actors	User, AI Model
Stakeholders:	Managers, End Users
Pre-Conditions	Emails are available for summarization.
Success end condition:	Summary is generated and stored.
Failure end condition:	Summarization fails.

Main Success Scenario

		Linked UCs
1	User selects one or more emails.	
2	System runs summarization on selected emails.	SUC3
3	Summary is generated.	
4	Summary is stored and linked to original emails.	
5	User sees the result.	

Alternative Scenarios

3.A1	Summary is too generic
3.A2	User is prompted to improve selection.

Exception Scenario

2.A1	Summarization service is unavailable
2.A2	System notifies user with fallback.

Use case: Forward Summary to Other Platforms	
ID	UC4
Description	System forwards a summary to platforms like Notion or Monday.com
Actors	User, External API
Stakeholders:	Managers, Project Teams
Pre-Conditions	Platform is connected. Summary exists.
Success end condition:	Summary is successfully posted.
Failure end condition:	Summary cannot be sent.

Main Success Scenario

		Linked UCs
1	User enables integration.	
2	System selects relevant summaries.	
3	System posts data to the platform.	SUC4
4	Post is confirmed by API.	
5	User is notified.	

Alternative Scenarios

3.A1	Only partial content fits platform format
3.A2	System trims or structures it.

Exception Scenario

2.A1	API call fails
2.A2	Retry or notify user.

Use case: View Email Summary	
ID	UC5
Description	User views a list of summarized emails and can expand for full details.
Actors	User
Stakeholders:	End users
Pre-Conditions	Summarized emails exist.
Success end condition:	User sees accurate summaries and can access full content.
Failure end condition:	Summary not found or can't be displayed.

Main Success Scenario

		Linked UCs
1	User opens dashboard.	
2	System loads summaries.	SUC3
3	User clicks on a summary.	
4	System expands the full email content.	
5	User navigates back.	

Alternative Scenarios

3.A1	Summary is outdated
3.A2	System re-generates it on request.

Exception Scenario

2.A1	Database returns error
2.A2	System shows fallback view.

Supporting Use case: Extract email content	
ID	SUC1
Description	Parses the full content of an email and prepares it for AI processing
Actors	EmailParser module
Stakeholders:	Developer, System Maintainer
Pre-Conditions	A raw email object is available via Gmail API
Success end condition:	Cleaned and structured email content is returned
Failure end condition:	Email content cannot be extracted

Main Success Scenario

1	The system receives a new raw email.	
2	The EmailParser extracts body text, metadata (sender, date), and attachments.	
3	The cleaned content is forwarded to the AI module.	
4	The system returns the structured content to the main use case.	

Alternative Scenarios

2.A1	Attachments are too large or unreadable.
2.A2	Attachments are skipped, only body text is used.

Exception Scenario

2.A1	Email body is encoded or corrupted.
2.A2	The system logs the error and discards the email.

Supporting Use case: Run entity extraction model

ID	SUC2
Description	Uses NLP/LLM model to extract named entities and key facts
Actors	AI Extractor
Stakeholders:	Users, Developers
Pre-Conditions	Clean email content is available
Success end condition:	Entities (person, task, time) are extracted successfully
Failure end condition:	No relevant entities found

Main Success Scenario

1	The email text is passed to the NER/LLM model.	
2	The model identifies key elements: people, dates, tasks.	
3	Extracted entities are returned to calling UC.	
4	Data is stored in the database.	

Alternative Scenarios

2.A1	Model is uncertain about entity classification.
2.A2	System flags entity with “Low confidence” for review.

Exception Scenario

2.A1	Model API fails to respond.
2.A2	Retry once, then return empty result.

Supporting Use case: Generate summary with LLM

ID	SUC3
Description	Passes email thread to summarization model to produce a human-readable summary
Actors	Summarization Engine
Stakeholders:	Users, Managers
Pre-Conditions	Email(s) are selected for summarization
Success end condition:	Meaningful summary is returned
Failure end condition:	No summary can be generated

Main Success Scenario

1	Emails are merged into a single text.	
2	Text is passed to the summarization model (e.g. GPT).	
3	Summary is returned and shown to the user.	
4	System saves the summary.	

Alternative Scenarios

3.A1	Summary exceeds character limit.
3.A2	System trims and displays note: “Shortened summary.”

Exception Scenario

2.A1	LLM model is unreachable.
2.A2	System displays: “Summary temporarily unavailable.”

Supporting Use case: Post summary to Notion/Monday.com

ID	SUC4
Description	Uses API to send a summary or task to third-party platform
Actors	Notion API / Monday API
Stakeholders:	Users
Pre-Conditions	Summary exists and API keys are configured
Success end condition:	Summary/task is posted successfully
Failure end condition:	Post fails or is rejected

Main Success Scenario

1	Summary/task data is packaged into API format.	
2	API call is made to the target platform.	
3	Target platform confirms receipt.	
4	System logs success.	

Alternative Scenarios

2.A1	Platform returns warning (e.g. page missing).
2.A2	System asks user to reconnect or change config.

Exception Scenario

2.A1	Auth token is expired.
2.A2	User is prompted to re-authenticate.

Supporting Use case: Classify email category using ML model

ID	SUC5
Description	Predicts the email type (task, customer query, irrelevant, ...) using classification model
Actors	ML Classifier
Stakeholders:	Users, AI Developers
Pre-Conditions	Email content is cleaned
Success end condition:	A category label is returned
Failure end condition:	Classification fails or returns "unknown"

Main Success Scenario

1	Email content is passed to the classification model.	
2	Model returns a category label.	
3	System tags the email accordingly.	
4	Label is stored for filtering/sorting.	

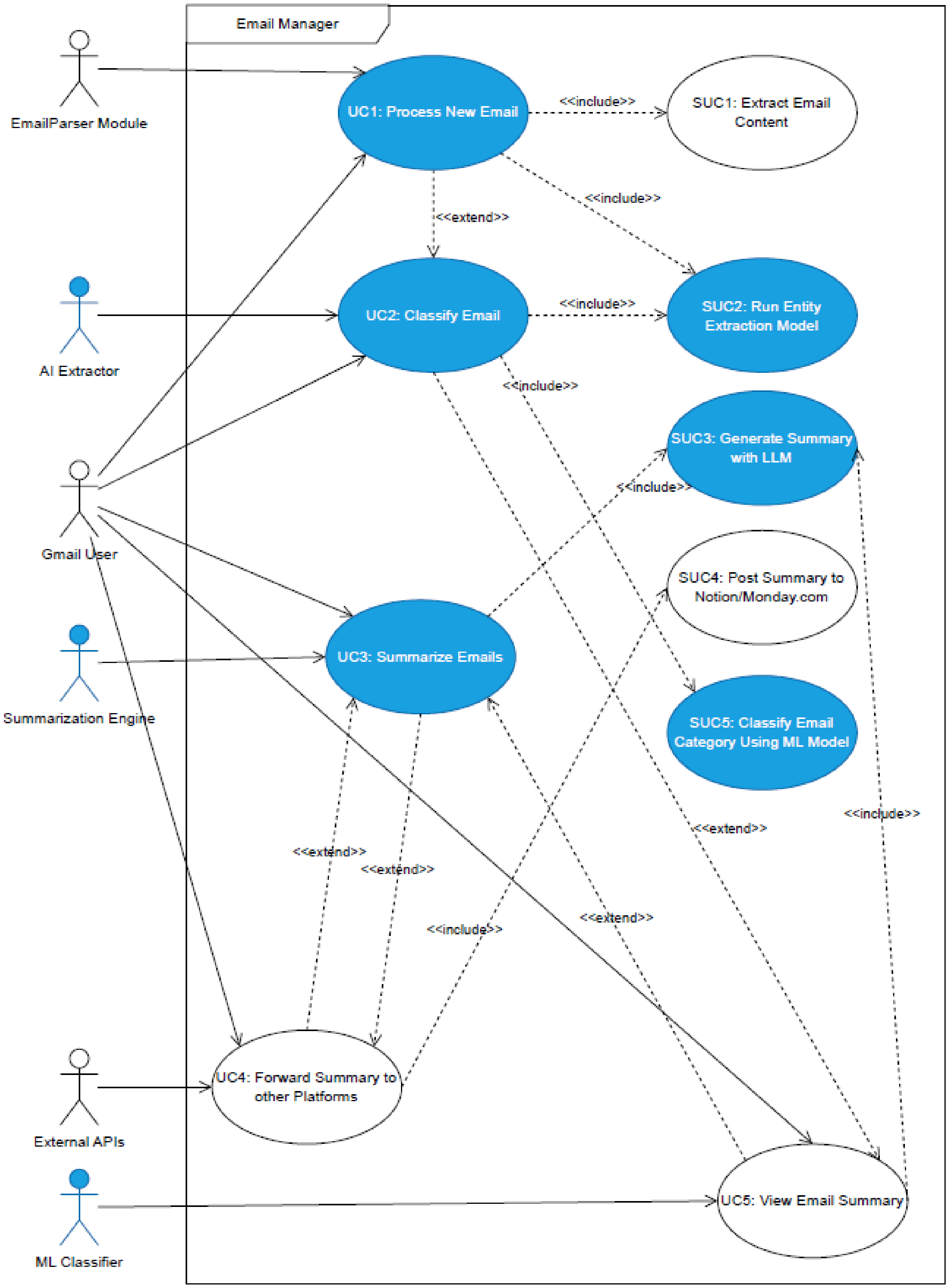
Alternative Scenarios

2.A1	Model returns multiple possible labels.
2.A2	System shows “Ambiguous” and user picks final label.

Exception Scenario

2.A1	Model is unavailable.
2.A2	System skips classification and tags as “Uncategorized”.

Use case diagram



Which use cases were implemented in the source code

AI Engine: Below given the source code for AI engine which is responsible for generating Summaries, categories and Named Entities extraction:

```
from transformers import pipeline,AutoTokenizer, AutoModelForSeq2SeqLM,set_seed
import torch
from langchain_huggingface import HuggingFacePipeline
from langchain.prompts import PromptTemplate
from summarizer import Summarizer



class AI_engine:

    def __init__(self):

        # summarizer:
        self.summarize_facebook_bart = pipeline("summarization", model="facebook/bart-large-cnn")
        self.tokenizer = AutoTokenizer.from_pretrained("facebook/bart-large-cnn")
        self.extractive_model_summarize=Summarizer()


        # classifier:
        # self.classifier = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")
        # self.classifier = pipeline("zero-shot-classification",
model="valhalla/distilbart-mnli-12-1")

        self.classifier = pipeline("zero-shot-classification",
model="MoritzLaurer/mDeBERTa-v3-base-mnli-xnli")

        # name-entity-extraction:
        # self.ner_bert = pipeline("ner", model="dslim/bert-base-NER",
aggregation_strategy="simple")

        self.ner_bert = pipeline(
    "ner",
    model="Davlan/bert-base-multilingual-cased-ner-hrl",
    tokenizer="Davlan/bert-base-multilingual-cased-ner-hrl",
    aggregation_strategy="simple"
)



    def summarizer_facebook_bart(self,text:str)->str:
        # word_count=len(text.split())
        input_tokens=len(self.tokenizer.encode(text))
        if input_tokens<142:
           return text

        hint = "Summarize the email and make sure to include any dates, deadlines, or scheduled
meetings.Give me important information in bullet points\n\n"
        augmented_text=hint+text


        summary=self.summarize_facebook_bart(
            augmented_text,
            max_length=142,
            min_length=30,
            do_sample=False,

        )[0]['summary_text']

        return summary

    def summarizer_extractive_model(self,text:str)->str:
        summary=self.extractive_model_summarize(text,ratio=0.40)
        return summary


    def categorization(self,text)->str:
        into a one category")
        # categorization_pipeline=HuggingFacePipeline(pipeline=self.category_generator_openai_gpt)
        # categorization_chain=template| categorization_pipeline
        # category=categorization_chain.invoke({'text':text})

        candidate_labels = ["meeting", "order", "invoice", "reminder", "event", "subscription",
"security", "policy", "legal", "finance"]
        category=self.classifier(text,candidate_labels,multi_label=False)

        return category

    def entity_extraction_xlm_roberta(self,text:str)->str:

        entities=self.ner_bert(text)

        return [(ent["word"], ent["entity_group"]) for ent in entities]
```

Database Model:

```
from sqlalchemy.orm import Session
from datetime import datetime,timedelta
from . import models

def create_email_into_db(
        db:Session,
        gmail_id:str,
        sender:str,
        email_subject:str,
        original_email_label:str,
        original_email:str,
        summary:str,
        category:str,
        named_entities:str,
        user_id:str
):
    existing=db.query(models.EmailRecord).filter(
        models.EmailRecord.gmail_id==gmail_id,
        models.EmailRecord.user_id==user_id
    ).first()

    if existing:
        return existing


    db_emails=models.EmailRecord(
        gmail_id=gmail_id,
        sender=sender,
        email_subject=email_subject,
        original_email_label=original_email_label,
        original_email=original_email,
        summary=summary,
        category=category,
        named_entities=named_entities,
        user_id=user_id
    )

    db.add(db_emails)
    db.commit()
    db.refresh(db_emails)
    return db_emails






def get_emails_from_db(db:Session,user_id:str):
    return db.query(models.EmailRecord).filter(models.EmailRecord.user_id==user_id).all()


def get_email_by_id_from_db(db:Session,user_id:str,email_id:int):
    return
db.query(models.EmailRecord).filter(models.EmailRecord.id==email_id,models.EmailRecord.user_id==user
_id).first()


def post_email_summary_by_id(db:Session,user_id:str,email_id:int,summary:str):
    email_record=db.query(models.EmailRecord).filter(
        models.EmailRecord.id==email_id,
        models.EmailRecord.user_id==user_id
    ).first()

    if email_record:
        email_record.summary=summary
        db.commit()
        db.refresh(email_record)
        return email_record
    else:
        return None


def post_email_categories_by_id(db:Session,user_id:str,email_id:int,category:str):
    email_record=db.query(models.EmailRecord).filter(
        models.EmailRecord.id==email_id,
        models.EmailRecord.user_id==user_id
    ).first()

    if email_record:
        email_record.category=category
        db.commit()
        db.refresh(email_record)
        return email_record

    else:
        return None

def post_email_named_entities_by_id(db:Session,user_id:str,email_id:int,namedEntities:str):
    email_record=db.query(models.EmailRecord).filter(
        models.EmailRecord.id==email_id,
        models.EmailRecord.user_id==user_id
    ).first()
    if email_record:
        email_record.named_entities=namedEntities
        db.commit()
        db.refresh(email_record)
        return email_record

    else:
        return None
```

Routes: Below given the source code different routes responsible for calling respective function from AI engine and performing the task and updating the changes to the database.

```
from fastapi import APIRouter,Depends,HTTPException,Request
from pydantic import BaseModel
from sqlalchemy.orm import Session
from bs4 import BeautifulSoup
from database.models import EmailRecord, SessionLocal
from collections import defaultdict

from utility.extractor import get_message_details, api_call, authenticiate

from database.db import *

from utility.utils import authenticate_and_get_user_details
from database.models import get_db
import json
from datetime import datetime
from AI_engine.ai_engine import AI_engine
from fastapi import BackgroundTasks

router=APIRouter()
engine=AI_engine()
class EmailRequest(BaseModel):
    email_subject:str
    original_email: str

    class Config:
        json_schema_extra = {
            "example": {
                "email_subject":"Developer team meeting on recent progress",
                "original_email": "Hi team, please find the notes from our last meeting 
attached...",
            }
        }

# @router.get("/extract-original-emails")
def extract_original_emails(user_id,service):
    db=SessionLocal()
    try:

        print("Starting email extraction...")
        print("User ID:", user_id)
        print("Service:", service)

        existing_subjects=set(
            email.gmail_id for email in get_emails_from_db(db,user_id) if email.gmail_id
        )




        if not service:
            return {"error": "Gmail service could not be initialized"}


        emails=get_message_details(service,user_id='me',max_results=50)
        print("Fetched emails from Gmail:", len(emails))
        stored_ids=[]

        for email in emails:
            print("Email")
            soup=BeautifulSoup(email['body'],'html.parser')
            clean_text=soup.get_text()
            # summary=engine.summarizer_extractive_model(clean_text)
            # print(summary)

            # categories=engine.categorization(clean_text)
            # category=categories['label'][0]
            # print(category)

            # entities=engine.entity_extraction_xlm_roberta(clean_text)
            # print("Named Entities:", entities)

            # formatted_entities = [f"{word} ({ent_type})" for word, ent_type in
entities]
            # entity_str = ", ".join(formatted_entities)


            # print(entities)
            db_record=create_email_into_db(
                db=db,
                gmail_id=email['gmail_id'],
                sender=email['sender'],
                email_subject=email['subject'],
                original_email=clean_text,
                original_email_label=",".join(email.get('label',[])),
                summary=None,
                category=None,
                named_entities=None,
                user_id=user_id

            )
            stored_ids.append(db_record.id)

        print("📦 All stored IDs:", stored_ids)


        return{
            "user_id":user_id,
            "stored_email_ids":stored_ids
        }



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






@router.get("/get-emails")
async def get_emails(request:Request,background_tasks: BackgroundTasks,
db:Session=Depends(get_db)):
    user_details=authenticate_and_get_user_details(request)
    user_id=user_details.get('user_id')
    user_service=user_details.get('service')

    # creating_emails into db:

    # existing_emails=get_emails_from_db(db,user_id)
    # background_tasks.add_task(extract_original_emails,user_id, user_service)
    extract_result = extract_original_emails(user_id, user_service)
    emails = get_emails_from_db(db, user_id)




    return emails


@router.get("/get-email-by-email-id/{email_id}")
async def
get_email_by_email_id(email_id:int,request:Request,db:Session=Depends(get_db)):
    user_details=authenticate_and_get_user_details(request)
    user_id=user_details.get('user_id')
    user_service=user_details.get('service')
    email_record=get_email_by_id_from_db(db,user_id,email_id=email_id)

    if not email_record:
        raise HTTPException(status_code=404,detail='Email not found')

    return email_record
```

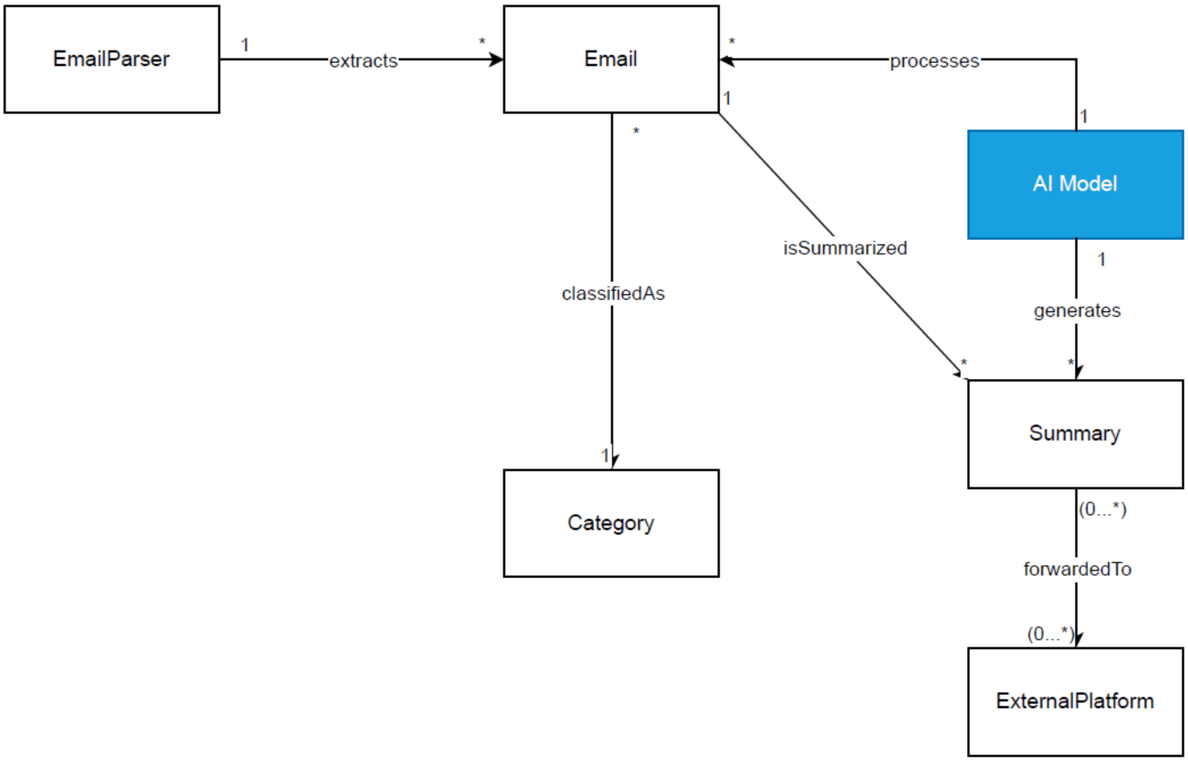
```
@router.post("/generate-summary/{email_id}")
async def generate_summary(email_id:int,request_obj:Request,db:Session=Depends(get_db)):
    try:
        user_details=authenticate_and_get_user_details(request_obj)
        user_id=user_details.get('user_id')
        user_service=user_details.get('service')

        email_record=get_email_by_id_from_db(db,user_id,email_id=email_id)


        if not email_record:
            raise HTTPException(status_code=404,detail='Email not found')


        soup=BeautifulSoup(email_record.original_email,'html.parser')
        clean_text=soup.get_text()

        if email_record.summary is None:
            summary=engine.summarizer_extractive_model(clean_text)

email_record_with_summary=post_email_summary_by_id(db,user_id,email_id=email_id,summary=
summary)
            return email_record_with_summary


        # summary=engine.summarizer_extractive_model(clean_text)

        #
email_record=post_email_summary_by_id(db,user_id,email_id=email_id,summary=summary)

        return {"message": "Email summary already exists for this email"}

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))


# @router.get("/get-summarized-emails")
# def get_summarized_emails(request: Request, db: Session = Depends(get_db)):
#     user_details = authenticate_and_get_user_details(request)
#     user_id = user_details.get("user_id")

#     emails = db.query(EmailRecord).filter(
#         EmailRecord.user_id == user_id,
#         EmailRecord.summary.isnot(None)
#     ).all()

#     return emails


@router.post("/generate-category/{email_id}")
async def generate_category(email_id,request_obj:Request,db:Session=Depends(get_db)):
    try:
        user_details=authenticate_and_get_user_details(request_obj)
        user_id=user_details.get('user_id')
        user_service=user_details.get('service')

        email_record=get_email_by_id_from_db(db,user_id,email_id=email_id)

        if email_record.category is None:
            clean_text=email_record.email_subject.strip()
            category=engine.categorization(clean_text)
            category=category['labels'][0]

email_record_with_category=post_email_categories_by_id(db,user_id,email_id,category)
            return email_record_with_category



        # soup=BeautifulSoup(email_record.email_subject,'html.parser')



        return {"message": "Email category already exists for this email"}



    except Exception as e:
        print("Error in generating categories")
        raise HTTPException(status_code=500,detail=str(e))



@router.post("/generate-named-entities/{email_id}")
async def
generate_named_entities(email_id:int,request_obj:Request,db:Session=Depends(get_db)):
    try:
        user_details=authenticate_and_get_user_details(request_obj)
        user_id=user_details.get('user_id')
        user_service=user_details.get('service')

        email_record=get_email_by_id_from_db(db,user_id,email_id)

        if email_record.named_entities is None:
            soup=BeautifulSoup(email_record.original_email,'html.parser')
            clean_text=soup.get_text()
            entities=engine.entity_extraction_xlm_roberta(clean_text)
            print("Named Entities:", entities)
            entity_dict = defaultdict(list)
            for word, ent_type in entities:
               entity_dict[ent_type].append(word)
            # formatted_entities = [f"{word} ({ent_type})" for word, ent_type in
entities]
            # entity_str = ", ".join(formatted_entities)
            entities_json_str = json.dumps([list(e) for e in entities])

email_record=post_email_named_entities_by_id(db,user_id,email_id,entities_json_str)
            print(entities_json_str)
            return email_record



        # soup=BeautifulSoup(email_record.original_email,'html.parser')
        # clean_text=soup.get_text()

        # entities=engine.entity_extraction_xlm_roberta(clean_text)
        # entities_json_str=json.dumps(entities)
        #
email_record=post_email_named_entities_by_id(db,user_id,email_id,entities_json_str)
        return {"message": "Email NER already exists for this email"}


    except Exception as e:
        print("Error in NER generation")
        raise HTTPException(status_code=500,detail=str(e))


@router.get("/auth-gmail")
def auth_gmail(user_id: str):
    creds = authenticiate(user_id)  # This opens browser
    if creds:
        return {"success": True}
    else:
        raise HTTPException(status_code=400, detail="Failed to authenticate Gmail")
```

You can find the full source code in the following github repo:
https://github.com/SwapnilKhandoker101/MailFokus

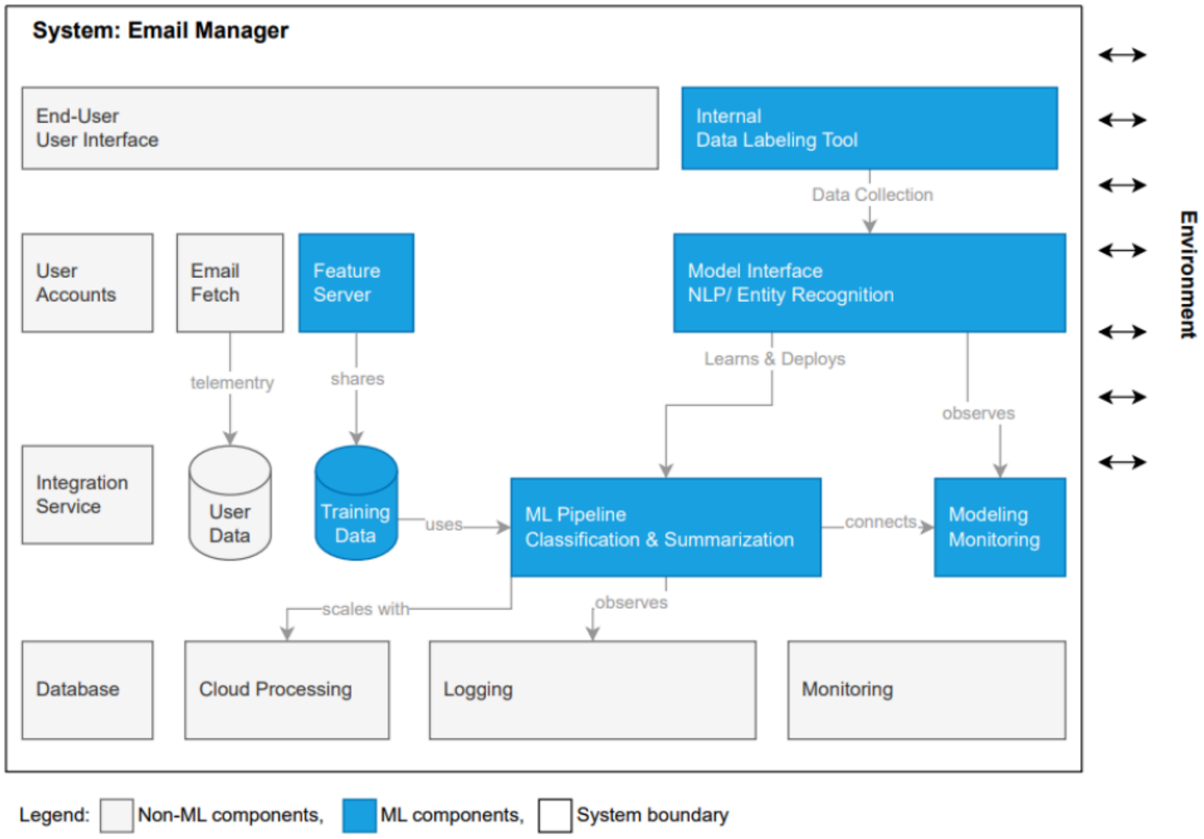
Traceability matrix

Use cases	UC1	UC2	UC3	UC4	UC5	SUC1	SUC2	SUC3	SUC4	SUC5
Req01	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH
Req02	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	FALSCH
Req03	WAHR	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	WAHR
Req04	FALSCH	FALSCH	WAHR	FALSCH	WAHR	FALSCH	FALSCH	WAHR	FALSCH	FALSCH
Req05	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	WAHR	FALSCH
Req06	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	WAHR	FALSCH
Req07	FALSCH	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH
Req08	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSCH
NfReq01	WAHR	FALSCH	WAHR	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	FALSCH
NfReq02	WAHR	FALSCH	WAHR	FALSCH	WAHR	FALSCH	FALSCH	WAHR	FALSCH	FALSCH
NfReq03	WAHR	FALSCH	FALSCH	FALSCH	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH
NfReq04	WAHR	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH	FALSCH
NfReq05	FALSCH	WAHR	WAHR	FALSCH	FALSCH	FALSCH	WAHR	WAHR	FALSCH	WAHR
NfReq06	FALSCH	WAHR	WAHR	FALSCH	FALSCH	FALSCH	WAHR	WAHR	FALSCH	WAHR

Domain Model



Architecture diagram



Components description

Non-ML Components

1. **End-User User Interface**
   The End-User Interface provides access to the system via Chrome extension or web dashboard, allowing users to view email summaries, classifications, and extracted information. It serves as the main interaction point for users to authenticate, configure settings, and view processed email data.
   Related Requirements: Req01, Req04, Req07, Req08, NfReq06
2. **User Accounts**
   The User Accounts component manages user profiles, authentication credentials, and system access permissions. It handles OAuth2.0 authentication with Gmail and stores user preferences for email processing and integration settings.
   Related Requirements: Req01, NfReq06
3. **Email Fetch**
   The Email Fetch component interfaces with the Gmail API to retrieve email content and metadata when new messages arrive. It prepares raw email content for processing by the AI components and triggers the information extraction pipeline.
   Related Requirements: Req02, NfReq01
4. **Integration Service**
   The Integration Service connects the Email Manager with third-party platforms like Monday.com and Notion, formatting extracted information appropriately for each platform. It handles API authentication, error handling, and ensures data is properly transferred to external services based on user configuration.
   Related Requirements: Req05, Req06, SUC4
5. **User Data**
   The User Data component stores user information, preferences, and system configuration in a secure database with proper encryption. It maintains records of user settings, connected services, and processing preferences while ensuring GDPR compliance for data retention.
   Related Requirements: NfReq03, NfReq04
6. **Database**
   Persistently stores user-relevant data including extracted info and email metadata (with encryption).
   Related Requirements: Req03, NfReq03, NfReq04
7. **Cloud Processing**
   Provides scalable compute resources to handle inference and training workloads.
   Related Requirements: NfReq02, AI Req03, AI Req04, AI Req05
8. **Logging**
   Captures system and model activity to ensure traceability and auditability.
   Related Requirements: NfReq03, NfReq05
9. **Monitoring**
   Tracks system uptime and performance metrics, alerting on system issues.
   Related Requirements: NfReq02, NfReq06

ML- Components

1. **Internal Data Labeling Tool**
   The Internal Data Labeling Tool allows system administrators to create labeled training data from email samples to improve AI model performance. It provides an interface for annotating entities, classifying emails, and rating summarization quality to enhance model training.
   Related Requirements: AI Functional Requirements (Req03, Req04, Req05, Req08)
2. **Feature Server**
   The Feature Server extracts and prepares relevant features from email content for use by ML models. It transforms raw email text into structured features that the classification and entity extraction models can process effectively.
   Related Requirements: Req03, SUC2
3. **Model Interface (NLP/Entity Recognition)**
   This component uses NLP models to identify and extract key information from emails, such as people, dates, tasks, and other relevant entities. It achieves at least 85% accuracy in named entity recognition and forms the core of the information extraction capability.
   Related Requirements: Req03, SUC2, NfReq02
4. **Training Data**
   The Training Data component stores labeled examples used to train and improve the ML models for entity extraction, classification, and summarization. It contains curated examples that help the system learn patterns in email content and improve accuracy over time.
   Related Requirements: Req03, Req04, Req05, Req08
5. **ML Pipeline (Classification & Summarization)**
   The ML Pipeline processes emails through classification and summarization models, determining email categories and generating concise summaries. It handles the workflow of model inference, achieves 90% accuracy in task identification, and creates summaries with 80% ROUGE-L scores.
   Related Requirements: Req03, Req04, Req05, Req08, SUC2, SUC3, SUC5, NfReq02
6. **Model Monitoring**
   The Model Monitoring component tracks AI model performance, detects accuracy drift, and alerts when models need retraining. It ensures AI components maintain their performance thresholds and provides transparency about model confidence levels for user trust.
   Related Requirements: NfReq05, NfReq06

System Boundary

The System Boundary clearly delineates all components that make up the Email Manager application, separating them from external systems and the environment. It defines the scope of the system and identifies integration points with external services and APIs.
Related Requirements: All requirements are contained within the system boundary

Design Questions and answers

Design Questions

1. How do we define and measure the accuracy of information extraction from emails, and what accuracy threshold would satisfy users before they abandon the system?
2. How should we present model confidence levels to users when displaying extracted information or summaries, and should low-confidence extractions be shown at all?
3. How can we determine which emails are truly important to different types of users, and how might this vary across departments or roles?
4. What mechanisms should we implement to collect user feedback on incorrect classifications or summaries, and how quickly can this feedback improve our models?
5. How should the system handle failures when posting to external platforms like Monday.com or Notion, and what recovery mechanisms are needed?
6. As the email volume grows, how can we maintain response times within the 3-second requirement for summarization while controlling infrastructure costs?
7. What specific techniques should be implemented to prevent sensitive information leakage when processing emails containing personal or confidential data?
8. When and how should we deploy improved AI models without disrupting ongoing service, and what rollback mechanisms do we need?
9. How can our models adapt to different industry-specific terminology and email formats without requiring extensive retraining?
10. What logging and monitoring strategies should we implement to debug AI model failures without accessing the actual email content that caused them?

Design Question Answers

1. We will define accuracy using a multi-metric approach that combines Named Entity Recognition (NER) F1 scores for factual extraction and user satisfaction ratings for overall usefulness. Our target threshold is 85% NER accuracy and 80% user satisfaction, which our preliminary user research indicates will satisfy most business users. We'll implement progressive improvement by starting with high-precision extractions (fewer but more accurate results) and gradually increasing recall as our models improve.
2. We'll implement a three-tier visual system: high-confidence extractions (green) shown normally, medium-confidence (yellow) shown with subtle highlighting, and low-confidence (red) shown with explicit uncertainty indicators. Users can adjust confidence thresholds in settings, with defaults set to hide extractions below 60% confidence. A "Why this extraction?" feature will explain which parts of the email led to specific conclusions, enhancing user trust through transparency.
3. Our priority classification will combine explicit signals (sender importance, keyword matching) with implicit signals (user interaction patterns, response times) using a personalized ML approach. We'll build department-specific classification models starting with management, sales, and customer service, allowing organization

admins to define custom priority rules. The system will learn from user behavior - emails that users consistently open first or respond to quickly will automatically receive higher priority scores over time.

4. We'll implement multiple feedback channels: thumbs up/down buttons on each extraction and summary, an annotation tool for incorrect entities, and implicit feedback from user corrections. This feedback will enter a continuous improvement pipeline, with high-impact corrections (affecting many users) deployed weekly and personalization adjustments applied daily. Critical fixes for systematic errors will be prioritized based on frequency and severity, with clear tracking of improvement metrics over time.
5. We'll implement a three-stage failure handling system: immediate retry (3 attempts with exponential backoff), persistent queue storage for failed operations, and user notification with manual retry options. Integration failures will be stored in a dedicated database table with error details, and a background service will attempt to resolve them automatically every 30 minutes. Users will have a dashboard showing integration status and options to manually force retries or modify data before retrying.
6. We'll implement a tiered architecture with static caching for common operations, dynamic scaling of inference workers, and asynchronous processing for non-urgent summaries. To meet the 3-second requirement for typical emails, we'll use lightweight models for initial processing while queuing longer emails for background processing. We'll also implement predictive scaling based on historical email volume patterns (e.g., Monday morning influx) and use cloud spot instances to optimize costs during peak periods.
7. We'll implement entity-based PII detection and redaction before storing any content, with all processing occurring in isolated compute environments. Email data will be encrypted at-rest and in-transit using AES-256, with keys rotated regularly. We'll implement strict data retention policies (30 day default with configurable periods) and provide organization admins with audit logs showing all system access to email content. For GDPR compliance, we'll support subject access requests with automated data retrieval and deletion capabilities.
8. We'll use a canary deployment approach, gradually rolling out model updates to increasing percentages of traffic while monitoring key metrics. New models will initially process only 5% of emails with side-by-side comparison to current models, expanding to 20%, 50%, and 100% if performance metrics remain stable or improve. We'll maintain the previous model version in standby for immediate rollback capability and implement feature flags to disable specific model components if issues arise in production.
9. We'll implement domain adaptation using a combination of transfer learning and few-shot learning techniques. Our base models will be supplemented with domain-specific layers that can be fine-tuned with minimal examples (50-100 labeled emails per domain). We'll provide industry-specific terminology configuration options and build a collaborative learning system where similar organizations can contribute to shared domain models while preserving privacy. For organizations with unique terminology, we'll offer a custom entity definition interface.
10. We'll implement anonymized error fingerprinting that captures model input/output patterns and feature statistics without storing raw email content. Detailed logging will track model decisions, confidence scores, and feature importance values for each prediction. When errors occur, the system will generate synthetic examples

mimicking the problematic pattern for debugging. We'll create a secure debugging workflow where authorized developers can request temporary access to anonymized versions of problematic emails, requiring explicit admin approval and full audit trails.