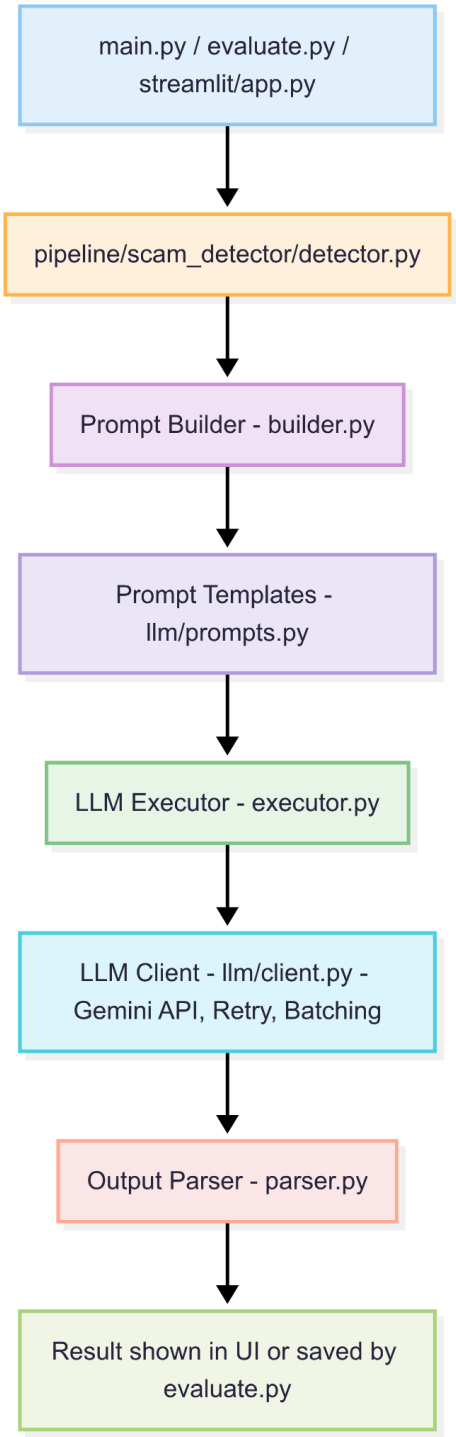
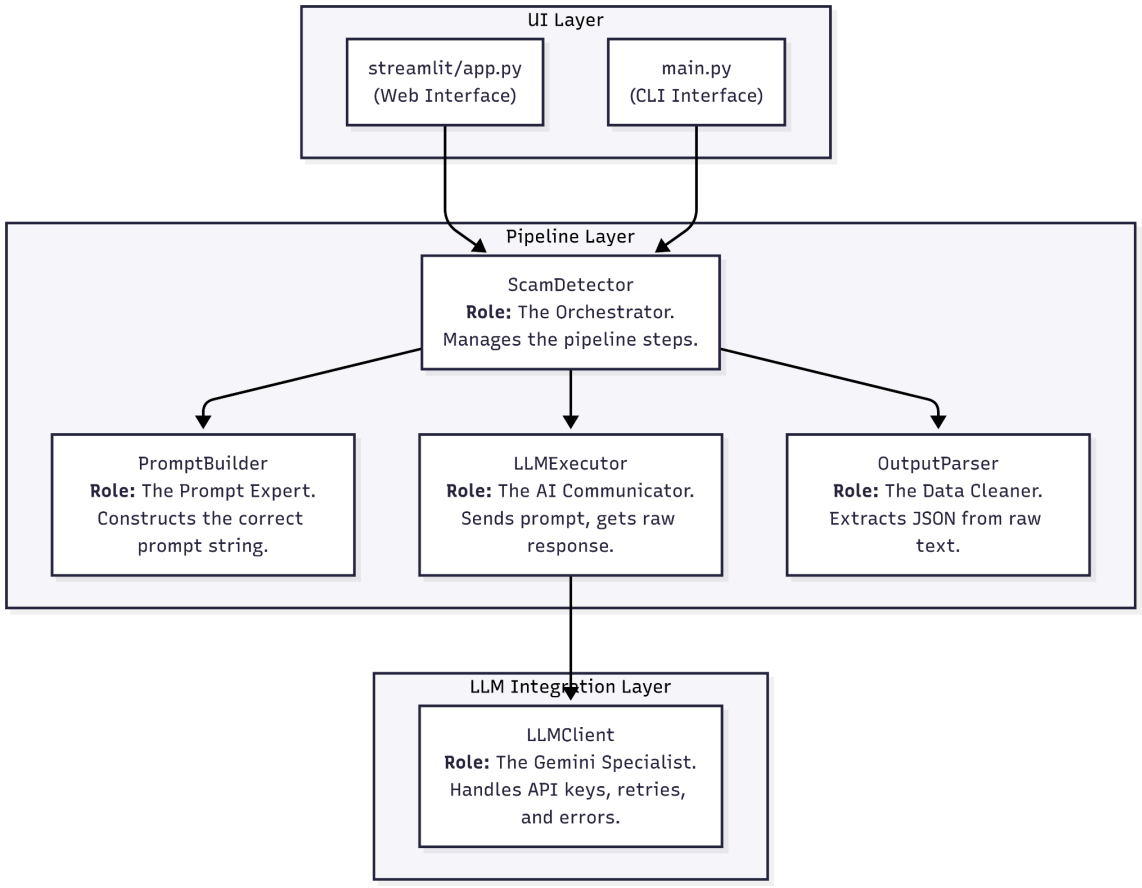
Reminder open up VS Code and in that open a new bash Terminal

Session 12: Notes - Hands on ScamGuard

Part 1: Recap System Architecture & Design


main.py / evaluate.py / streamlit/app.py
pipeline/scam_detector/detector.py
Prompt Builder - builder.py
Prompt Templates - llm/prompts.py
LLM Executor - executor.py
LLM Client - llm/client.py - Gemini API, Retry, Batching
Output Parser - parser.py
Result shown in UI or saved by evaluate.py



Part 2a: Project Setup and Configuration

Part 2: Architecture and Configuration# In your Terminal/CMD

```
# Create project directory
mkdir scam-detection
cd scam-detection

# Create the modular structure
mkdir -p llm/prompts pipeline/scam_detector streamlit

# Create essential files
touch config.py utils.py main.py evaluate.py requirements.txt .env
touch llm/__init__.py llm/client.py llm/prompts.py
touch pipeline/__init__.py pipeline/scam_detector/__init__.py
touch pipeline/scam_detector/detector.py pipeline/scam_detector/builder.py
touch pipeline/scam_detector/executor.py pipeline/scam_detector/parser.py
touch streamlit/app.py
```

In CMD

```
mkdir scam-detection
cd scam-detection

mkdir llm
mkdir llm\prompts
mkdir pipeline
mkdir pipeline\scam_detector
mkdir streamlit

type nul > config.py
type nul > utils.py
type nul > main.py
type nul > evaluate.py
type nul > requirements.txt
type nul > .env
```

```
type nul > llm\__init__.py
type nul > llm\client.py
type nul > llm\prompts.py

type nul > pipeline\__init__.py
type nul > pipeline\scam_detector\__init__.py
type nul > pipeline\scam_detector\detector.py
type nul > pipeline\scam_detector\builder.py
type nul > pipeline\scam_detector\executor.py
type nul > pipeline\scam_detector\parser.py

type nul > streamlit\app.py
```

– requirements.txt

```
google-genai==1.27.0
pandas==2.3.1
pydantic==2.11.7
python-dotenv==1.1.1
streamlit==1.47.1
tqdm==4.67.1
```

On Terminal:

```
pip install -r requirements.txt
```

– .env

```
GEMINI_API_KEY=your_google_api_key_here
```

Create a Virtual Environment

python -m venv venv

On Windows (CMD / PowerShell):
`.\.venv\Scripts\activate`

On macOS / Linux (bash / zsh):
source venv/bin/activate

– Config.py

```
import os
from pathlib import Path
from dotenv import load_dotenv

# Load environment variables
PROJECT_ROOT = Path(__file__).parent
load_dotenv(PROJECT_ROOT / ".env")

# API Configuration
GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")

# LLM Settings
DEFAULT_MODEL = "gemini-2.5-flash"
MAX_RETRIES = 3
RETRY_DELAY = 2

# Processing Settings
DEFAULT_BATCH_SIZE = 10
STREAMLIT_BATCH_SIZE = 5
```

– utils.py

```
import logging
import re
import json
from pathlib import Path
```

```
def get_logger(name: str) -> logging.Logger:
    """Get a simple logger for the given name."""
    logging.basicConfig(
        level=logging.INFO,
        format='[%(asctime)s] %(levelname)s: %(message)s'
    )
    return logging.getLogger(name)

def extract_json_from_text(text: str) -> dict:
    """Extract JSON from text string. Returns empty dict if not found."""
    try:
        match = re.search(r"\{.*\}", text, re.DOTALL)
        if match:
            return json.loads(match.group())
        return {}
    except json.JSONDecodeError:
        return {}

def load_file(file_path: str) -> str:
    """Load and return file contents as string."""
    return Path(file_path).read_text().strip()
```

Part 3b: Recap Prompt Engineering Strategies – llm/prompts.py

```
from pathlib import Path
from utils import load_file

# Directory paths
PROMPTS_DIR = Path(__file__).parent / "prompts"

# Load prompt templates from external files
def load_prompt(filename: str) -> str:
    """Load a prompt template from file."""
    return load_file(PROMPTS_DIR / filename)

# Available prompt templates
PROMPT = load_prompt("react.md") # <-- Edit this to use different prompts

def generate_prompt(user_input: str) -> str:
    template = PROMPT
    return f"{template}\n\nUser Message:\n{user_input.strip()}"
```

– llm/prompts/react.md

```
You are a highly reliable and safety-focused AI system trained to identify potentially scammy, manipulative, or deceptive intent in text-based communication.

Follow this exact structured reasoning format for each message:

1. **Thought**: Analyze the tone, language, urgency, and phrasing patterns
2. **Action**: Classify if this is likely a scam or not based on evidence
3. **Observation**: Justify your classification with specific details
4. **Final Answer**: Output a structured JSON with the following fields:

```json
{
  "label": "Scam | Not Scam | Uncertain",
  "reasoning": "<step-by-step analysis>",
  "intent": "<short description of user intent>",
  "risk_factors": ["<e.g., urgency, financial request, impersonation>"]
}
```
```

```
**Key Analysis Points:**

- Urgency tactics (limited time offers, immediate action required)
- Financial requests (money, bank details, card info)
- Impersonation (claiming to be from legitimate organizations)
- Suspicious links or attachments
- Grammar and spelling quality
- Emotional manipulation tactics

Be cautious when unsure. Do not make up details beyond the input.
```

– pipeline/scam_detector/builder.py

```
from llm.prompts import generate_prompt

def build_prompt(message: str, strategy: str = "react") -> str:
    if strategy == "react":
        return generate_prompt(message)
    else:
        raise NotImplementedError(f"Strategy '{strategy}' is not supported yet.")
```

Part 3: LLM API Integration

– llm/client.py

```
import time
from google import genai
from config import GEMINI_API_KEY, DEFAULT_MODEL, MAX_RETRIES, RETRY_DELAY
from utils import get_logger

logger = get_logger(__name__)

class LLMClient:
    """Gemini API client"""

    def __init__(self, model_name=DEFAULT_MODEL, max_retries=MAX_RETRIES,
retry_delay=RETRY_DELAY):
        self.model_name = model_name
        self.max_retries = max_retries
        self.retry_delay = retry_delay
        self.client = genai.Client(api_key=GEMINI_API_KEY)

    def call(self, prompt: str, **kwargs) -> str:
        """Send prompt to Gemini API"""
        for attempt in range(self.max_retries + 1):
            try:
                response = self.client.models.generate_content(
                    contents=prompt,
                    model=self.model_name,
                    **kwargs
                )
                if response and response.text:

                    return response.text.strip()
                else:
                    raise Exception("Empty response received")

            except Exception as e:
                if attempt == self.max_retries:
                    raise Exception(f"API call failed after {self.max_retries
+ 1} attempts: {e}")

                time.sleep(self.retry_delay * (2 ** attempt))
```

– pipeline/scam_detector/executor.py

```
from typing import Optional
from llm.client import LLMClient
from utils import get_logger

logger = get_logger(__name__)

class LLMExecutor:
    """Executes prompts using the LLM client."""

    def __init__(self, model: Optional[str] = None) -> None:
        self.llm: LLMClient = LLMClient(model) if model else LLMClient()
        logger.info("Initialized LLMExecutor")

    def execute(self, prompt: str) -> str:
        logger.info(f"Executing LLM with prompt length: {len(prompt)}")
        try:
            response = self.llm.call(prompt)
            logger.info(f"LLM execution successful, response length:
{len(response)}")
            return response
        except Exception as e:
            logger.error(f"LLM execution failed: {str(e)}")
            raise
```

Part 4: Output Handling & Reliability
– pipeline/scam_detector/parser.py

```
from typing import Dict, Any
from utils import get_logger, extract_json_from_text

logger = get_logger(__name__)

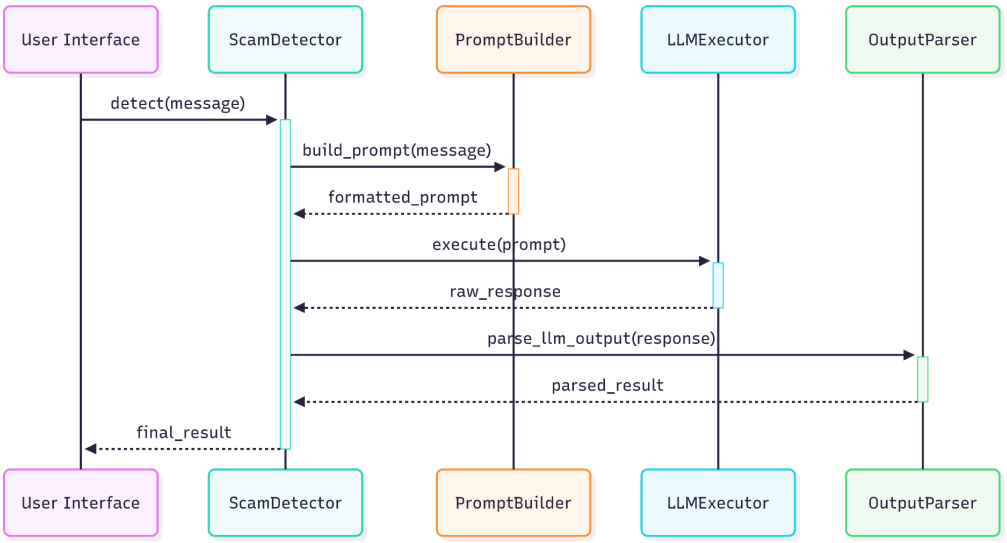
class OutputParser:
    """Parses LLM output into structured format."""

    def parse_llm_output(self, llm_output: str) -> Dict[str, Any]:
        logger.info(f"Parsing LLM output of length: {len(llm_output)}")

        # Try to extract JSON using utils function
        parsed_json = extract_json_from_text(llm_output)

        if parsed_json:
            logger.info("Successfully parsed LLM output to JSON.")
            return parsed_json
        else:
            logger.warning("No JSON found in LLM output.")
            # Return fallback result
            fallback_result = {
                "label": "Uncertain",
                "reasoning": "Failed to parse response: No JSON found",
                "intent": "Could not determine",
                "risk_factors": []
            }
            return fallback_result
```

Part 5: Pipeline Integration & Chat Interfaces



– pipeline/scam_detector/detector.py

```
from typing import List, Dict, Any
from .builder import build_prompt
from .executor import LLMExecutor
from .parser import OutputParser
from utils import get_logger

logger = get_logger(__name__)

class ScamDetector:
    """Orchestrates the scam detection pipeline."""

    def __init__(self, strategy: str = "react") -> None:
        """Initializes the pipeline components."""
        self.executor = LLMExecutor()
        self.parser = OutputParser()
        self.strategy = strategy
        logger.info(f"Initialized ScamDetector with strategy:
{self.strategy}")

    def detect(self, message: str) -> Dict[str, Any]:
        """Runs scam detection on a single message."""
        logger.info(f"Starting detection for message length: {len(message)}")
        try:
```

```
            # The 3-step pipeline
            prompt = build_prompt(message, self.strategy)
            raw_response = self.executor.execute(prompt)
            parsed_result = self.parser.parse_llm_output(raw_response)

            logger.info(f"Detection successful. Result:
{parsed_result.get('label', 'Unknown')}")
            return parsed_result

        except Exception as e:
            logger.error(f"Detection pipeline failed: {e}")
            # Re-raise the exception to be handled by the caller (UI layer)
            raise

    def detect_batch(self, messages: List[str]) -> List[Dict[str, Any]]:
        """Runs scam detection on a list of messages."""
        total_messages = len(messages)
        logger.info(f"Starting batch detection for {total_messages}
messages.")

        results: List[Dict[str, Any]] = []

        for i, message in enumerate(messages):
            try:
                result = self.detect(message)
                results.append(result)
            except Exception as e:
                logger.warning(f"Failed to process message
{i+1}/{total_messages}: {e}")
                # Append a fallback error result for the failed message
                error_result = {
                    "label": "Uncertain",
                     "reasoning": f"Error processing message: {e}",
                    "intent": "Could not determine",
                    "risk_factors": ["processing_error"]
                }
                results.append(error_result)

        successful = sum(1 for r in results if r.get("label") != "Uncertain"
or "processing_error" not in r.get("risk_factors", []))
        logger.info(f"Batch processing complete.
{successful}/{total_messages} succeeded.")
        return results
```

– main.py

```
from pathlib import Path
import sys
from pipeline.scam_detector.detector import ScamDetector
from utils import get_logger

# Add project root to path
project_root = Path(__file__).parent
sys.path.append(str(project_root))

logger = get_logger(__name__)

def main():
    """Test the scam detection system with dynamic message."""
    detector = ScamDetector()

    test_msg = str(input("Enter your Message: "))

    try:
        logger.info("Running scam detection")
        result = detector.detect(test_msg)

        print(f"Test Message: {test_msg}")
        print(f"Detection Result: {result}")

    except Exception as e:
        logger.error(f"Detection failed: {e}")
        print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

–streamlit/app.py

```
import sys
from pathlib import Path

# Add the project root directory to Python path
project_root = Path(__file__).parent.parent
sys.path.append(str(project_root))

import streamlit as st
import pandas as pd
from pipeline.scam_detector.detector import ScamDetector

st.set_page_config(page_title="Scam Detection App", layout="wide")
st.title(" Scam Detection System")

# Initialize detector
detector = ScamDetector()

# Tab layout
tab1, tab2 = st.tabs(["Single Message", "Dataset Evaluation"])

# ----------- Single Message Analysis ----------
with tab1:
    st.header("Analyze a Single Message")
    user_input = st.text_area("Enter the message to analyze:", height=150,
                              placeholder="Example: Congratulations! You've
won $1000. Click here to claim...")

    if st.button("Analyze Message", type="primary"):
        if user_input.strip():
            with st.spinner("Analyzing message..."):
                try:
                    result = detector.detect(user_input)

                    # Display results
                    col1, col2 = st.columns(2)

                    with col1:
                        st.subheader("Classification Result")
                        label = result.get("label", "Unknown")

                        if label == "Scam":
```

```
                            st.error(f"🚨 **{label}**")
                        elif label == "Not Scam":
                            st.success(f"✅ **{label}**")
                        else:
                            st.warning(f"⚠️ **{label}**")

                    with col2:
                        st.subheader("Intent")
                        st.write(result.get("intent", "Could not determine"))

                    st.subheader("Reasoning")
                    st.write(result.get("reasoning", "No reasoning
provided"))

                    st.subheader("Risk Factors")
                    risk_factors = result.get("risk_factors", [])
                    if risk_factors:
                        for factor in risk_factors:
                            st.write(f"• {factor}")
                    else:
                        st.write("No specific risk factors identified.")

                except Exception as e:
                    st.error(f"Error during analysis: {str(e)}")
        else:
            st.warning("Please enter a message to analyze.")

# ---------- Dataset Evaluation ----------
with tab2:
    st.header("Dataset Evaluation")

    uploaded_file = st.file_uploader("Upload a CSV file for batch analysis",
type="csv")

    if uploaded_file is not None:
        try:
            df = pd.read_csv(uploaded_file)
            st.write(f"Loaded dataset with {len(df)} rows")
            st.write("Sample data:")
            st.dataframe(df.head())

            # Find text column
            text_columns = ["text", "message_text", "message"]
```

```
            text_col = None
            for col in text_columns:
                if col in df.columns:
                    text_col = col
                    break

            if text_col:
                st.write(f"Using '{text_col}' as text column")

                # Limit for demo purposes
                max_rows = min(len(df), 10)
                if st.button(f"Analyze First {max_rows} Messages"):
                    with st.spinner(f"Analyzing {max_rows} messages..."):
                        messages = df[text_col].head(max_rows).tolist()
                        results = detector.detect_batch(messages)

                        # Create results dataframe
                        results_df = pd.DataFrame(results)

                        st.subheader("Analysis Results")
                        st.dataframe(results_df)

                        # Summary statistics
                        label_counts = results_df['label'].value_counts()
                        st.subheader("Summary")
                        st.bar_chart(label_counts)

            else:
                st.error(f"Could not find text column. Expected one of:
{text_columns}")

        except Exception as e:
            st.error(f"Error processing file: {str(e)}")
```

TO RUN:
Mac/Linux: streamlit run streamlit/app.py

Windows: streamlit run streamlit\app.py

https://docs.streamlit.io/

HW: Part 6: Evaluation & Performance Assessment

```python
# evaluate.py


import pandas as pd

from tqdm import tqdm

import argparse


from pipeline.scam_detector.detector import ScamDetector


def calculate_metrics(actual_labels, predicted_labels):

    """Calculate accuracy, recall, and F1-score for scam detection."""

    total = len(actual_labels)

    if total == 0:

        return {"total": 0, "correct": 0, "accuracy": 0.0, "class_metrics": {}}


    # Clean up labels (remove spaces, make lowercase)

    actual_clean = [label.lower().strip() for label in actual_labels]

    predicted_clean = [label.lower().strip() for label in predicted_labels]


    # Calculate overall accuracy

    correct = 0

    for actual, pred in zip(actual_clean, predicted_clean):

        if actual == pred:

            correct += 1
```

```
accuracy = (correct / total) * 100


# Calculate metrics for each class
class_metrics = {}
unique_labels = set(actual_clean)


for label in unique_labels:
    # Count correct and incorrect predictions for this class
    true_positives = 0
    false_negatives = 0
    false_positives = 0


    for actual, pred in zip(actual_clean, predicted_clean):
        if actual == label and pred == label:
            true_positives += 1
        elif actual == label and pred != label:
            false_negatives += 1
        elif actual != label and pred == label:
            false_positives += 1


    # Calculate metrics
    if true_positives + false_negatives > 0:
        recall = (true_positives / (true_positives + false_negatives)) * 100
```

```
        else:
            recall = 0

        if true_positives + false_positives > 0:
            precision = (true_positives / (true_positives + false_positives)) *
100
        else:
            precision = 0

        if precision + recall > 0:
            f1_score = (2 * precision * recall) / (precision + recall)
        else:
            f1_score = 0

        class_metrics[label] = {
            "recall": round(recall, 2),
            "f1_score": round(f1_score, 2)
        }

    return {
        "total": total,
        "correct": correct,
        "accuracy": round(accuracy, 2),
        "class_metrics": class_metrics
    }
```

```python
def evaluate_model(dataset_path, limit=None, verbose=False, batch_size=10):
    """Load dataset, run predictions, and evaluate performance."""
    # Load the dataset
    try:
        df = pd.read_csv(dataset_path)
        print(f"Loaded dataset with {len(df)} rows")
    except FileNotFoundError:
        print(f"Error: Dataset file not found at {dataset_path}")
        return

    if limit:
        df = df.head(limit)
        print(f"Limiting evaluation to the first {limit} messages.")

    # Initialize detector
    detector = ScamDetector()

    # Prepare data for batch processing
    messages = df['message_text'].tolist()
    actual_labels = df['label'].tolist()

    print(f"\nEvaluating {len(messages)} messages in batches of {batch_size}...")
```

```
    # Process messages in batches
    predicted_labels = []
    total_batches = (len(messages) + batch_size - 1) // batch_size

    for i in tqdm(range(0, len(messages), batch_size), desc="Processing batches",
total=total_batches):
        batch_messages = messages[i:i + batch_size]
        batch_actual = actual_labels[i:i + batch_size]

        try:
            # Process batch
            batch_results = detector.detect_batch(batch_messages)
            batch_predicted = [result.get("label", "Uncertain") for result in
batch_results]

            if verbose:
                for j, (msg, true_label, pred_label) in
enumerate(zip(batch_messages, batch_actual, batch_predicted)):
                    print(f"\nMessage {i+j+1}: {msg[:50]}...")
                    print(f"  True: {true_label}")
                    print(f"  Predicted: {pred_label}")

            predicted_labels.extend(batch_predicted)

        except Exception as e:
```

```
            print(f"Error processing batch {i//batch_size + 1}: {e}")
            # Add fallback predictions for failed batch
            predicted_labels.extend(["Uncertain"] * len(batch_messages))

print("\nEvaluation complete.")

# Calculate and print the final metrics
metrics = calculate_metrics(actual_labels, predicted_labels)

print("\n" + "="*50)
print(" MODEL EVALUATION RESULTS")
print("="*50)
print(f"  Total Messages: {metrics['total']}")
print(f"  Correct Predictions: {metrics['correct']}")
print(f"  Overall Accuracy: {metrics['accuracy']}%")
print()

print("  PER-CLASS METRICS:")
for label, class_metrics in metrics['class_metrics'].items():
    print(f"    {label.upper()}:")
    print(f"      Recall: {class_metrics['recall']}%")
    print(f"      F1-Score: {class_metrics['f1_score']}%")
    print()
print("="*50)
```

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Evaluate the Scam Detection System.")
    parser.add_argument("dataset", help="Path to the labeled dataset (CSV file).")
    parser.add_argument("--limit", type=int, help="Limit evaluation to the first N messages.")
    parser.add_argument("--verbose", action="store_true", help="Print detailed results for each message.")
    parser.add_argument("--batch-size", type=int, default=10, help="Batch size for processing (default: 10).")

    args = parser.parse_args()
    evaluate_model(args.dataset, args.limit, args.verbose, args.batch_size)
```

Push to GitHub and Deploy on Streamlit

Create a .gitignore

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[codz]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
#  Usually these files are written by a python script from a template
#  before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt
```

```
# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover
*.py.cover
.hypothesis/
.pytest_cache/
cover/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
db.sqlite3
db.sqlite3-journal

# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
.pybuilder/
target/

# Jupyter Notebook
.ipynb_checkpoints

# IPython
profile_default/
```

ipython_config.py

pyenv
For a library or package, you might want to ignore these files since the code is
intended to run in multiple environments; otherwise, check them in:
.python-version

pipenv
According to pypa/pipenv#598, it is recommended to include Pipfile.lock in version control.
However, in case of collaboration, if having platform-specific dependencies or dependencies
having no cross-platform support, pipenv may install dependencies that don't work, or not
install all needed dependencies.
#Pipfile.lock

UV
Similar to Pipfile.lock, it is generally recommended to include uv.lock in version control.
This is especially recommended for binary packages to ensure reproducibility, and is more
commonly ignored for libraries.
#uv.lock

poetry
Similar to Pipfile.lock, it is generally recommended to include poetry.lock in version control.
This is especially recommended for binary packages to ensure reproducibility, and is more
commonly ignored for libraries.
https://python-poetry.org/docs/basic-usage/#commit-your-poetrylock-file-to-version-control
#poetry.lock
#poetry.toml

pdm
Similar to Pipfile.lock, it is generally recommended to include pdm.lock in version control.
pdm recommends including project-wide configuration in pdm.toml, but excluding
.pdm-python.
https://pdm-project.org/en/latest/usage/project/#working-with-version-control
#pdm.lock
#pdm.toml
.pdm-python
.pdm-build/

pixi
Similar to Pipfile.lock, it is generally recommended to include pixi.lock in version control.
#pixi.lock
Pixi creates a virtual environment in the .pixi directory, just like venv module creates one
in the .venv directory. It is recommended not to include this directory in version control.
.pixi

```
# PEP 582; used by e.g. github.com/David-OConnor/pyflow and github.com/pdm-project/pdm
__pypackages__/

# Celery stuff
celerybeat-schedule
celerybeat.pid

# SageMath parsed files
*.sage.py

# Environments
.env
.envrc
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json

# Pyre type checker
.pyre/

# pytype static type analyzer
.pytype/

# Cython debug symbols
cython_debug/
```

```
# PyCharm
#  JetBrains specific template is maintained in a separate JetBrains.gitignore that can
#  be found at https://github.com/github/gitignore/blob/main/Global/JetBrains.gitignore
#  and can be added to the global gitignore or merged into this file.  For a more nuclear
#  option (not recommended) you can uncomment the following to ignore the entire idea folder.
#.idea/

# Abstra
# Abstra is an AI-powered process automation framework.
# Ignore directories containing user credentials, local state, and settings.
# Learn more at https://abstra.io/docs
.abstra/

# Visual Studio Code
#  Visual Studio Code specific template is maintained in a separate VisualStudioCode.gitignore
#  that can be found at
https://github.com/github/gitignore/blob/main/Global/VisualStudioCode.gitignore
#  and can be added to the global gitignore or merged into this file. However, if you prefer,
#  you could uncomment the following to ignore the entire vscode folder
# .vscode/

# Ruff stuff:
.ruff_cache/

# PyPI configuration file
.pypirc

# Cursor
#  Cursor is an AI-powered code editor. `.cursorignore` specifies files/directories to
#  exclude from AI features like autocomplete and code analysis. Recommended for sensitive
data
#  refer to https://docs.cursor.com/context/ignore-files
.cursorignore
.cursorindexingignore

# Marimo
marimo/_static/
marimo/_lsp/
__marimo__/
```

1. Initialize an empty git repo
    - git init
2. **Add all files to staging**
    - git add .
3. **Make initial commit**
    - git commit -m "Initial commit: ScamGuard"
    - git remote add origin https://github.com/YOUR_USERNAME/ScamGuard.git
    - **Push code to GitHub**
    - git branch -M main
    - git push -u origin main

If you get fatal error

- git config --global user.name "Your Name"
- git config --global user.email "your.email@example.com"

git add .

git commit -m "message"

git push