Paystand AI Project: iMessageAI

Caden Roberts
cawrober@ucsc.edu
https://github.com/cadenroberts/iMessageAI

Problem Statement

As a full-time researcher and student, I often work in long, uninterrupted stretches. Friends, family, and loved ones message me, but I frequently forget to respond or reply hours later. This is a real personal bottleneck. Since iMessage is deeply integrated into macOS, building an AI assistant that monitors messages, surfaces replies, and generates mood-aligned drafts directly improves my daily communication without breaking focus.

1 Approach

I propose an AI messaging assistant: **iMessageAI**. It is a UI for an iMessage AI agent where you can set/edit/delete your name, a personal description about yourself, up to 5 moods and descriptions of those moods, and a list of phone numbers to include or exclude. This data is written to config.json by the Swift UI and read from by the python model script. You can select from and edit the AI (Ollama 3.1 8B) suggested responses corresponding to each of your moods and send one, regenerate them, or ignore the entire iMessage. There is also a toggle for notifications, which are sent on every round of response generation. Data is written to and read from replies.json by both the Swift UI and the Python script. To make the pipeline simple:

1. A person sends you a text message. That sender's phone number and the text content are written to /Library/Messages/chat.db.

2. We use a Python script and sqlite3 to monitor for changes in the most recent text, phone number, and is_from_me of chat.db:

```
SELECT text
FROM message
ORDER BY date
DESC LIMIT 1;
```

```
SELECT id
FROM handle
WHERE ROWID=(
  SELECT handle_id
  FROM message
  ORDER BY date
  DESC LIMIT 1
);
```

```
SELECT is_from_me
FROM message
ORDER BY date
DESC LIMIT 1;
```

3. We have the Python script call Ollama in json format mode until it returns responses with names according to the set moods. We validate this in the Python script but use this as our system prompt:

```
system_prompt = f"""You are {config['name']}. {config['name']} was
    asked about their personality so take notes and form a base
    tone of {config['name']}: "{config['personalDescription']}" You
    have {len(config['moods'])} moods. Your moods are: {{{", ".
    join(f'"{mood}": "{config['moods'][mood]}"' for mood in config[
    'moods'])}}}. As {config['name']}, you will be given new texts
    from a sender. You MUST output **EXACTLY {len(config['moods'])}
```

```
RESPONSES**. Each text response should be a response as though
you were in the given mood. If moods were {{"Happy": "Very nice
and upbeat.", "Sad": "Very short and pessimistic", "Angry":
"Quick to snap and not very nice"}} and the text was "Hi", you
respond with {{"Happy": "Hi! How are you, is everything good?",
"Sad": "Hey, how are you? I'm hanging in there ...", "Angry":
"Yeah, what do you need?"}}. The goal is to always return a
dictionary and it have {len(config['moods'])} entries."""
```

4. After writing to replies.json, we can display the AI responses from that .json file and wait for the user to choose, regenerate, or ignore the iMessage as requested. This process is repeated *1-3*.

2 AI Usage

I used GPT-5 for this project. For the Swift UI, I used the built in GPT-5 assistant and vibe-coded through it, though I will say that there was a significant degree of prompt engineering. For the Python model/logic file, I used ChatGPT to debug errors but the logic flow and most of the code itself was written by me.

3 Discussion of Challenges

When you are creating an app in XCode, the default setting is to sandbox the app. Additionally, restrictions on the chat.db file itself caused file issues. Resolving these was a challenge and I settled with the user being required to give the app (or XCode and/or terminal) full disk access and to keep the project files in their root (the app can move anywhere though). I decided that the swift UI and Python would communicate by reading and writing to 2 json files. I also encountered Python environment issues here, but I just had to ensure that ollama was installed in the system Python and or any venv either the app or terminal calls. Initially, I wanted the app to be a pop-up-style widget from its notification, but settled with a full app UI given the scope of the project. Making this app reproducible would require sandboxing, allowing it access to chat.db, and giving it a much better model than Ollama to call. Selecting which texts to choose to respond to was also an issue. chat.db provides a *is_from_me* flag which I used to check if the most recent text is *from* the user and not *to* the user. I also added an include/exclude toggle for a list of phone numbers to contain who the auto generative AI can be used on. The Python logic needed to only ever respond to the most recent message (i.e. 5 messages come in while iMessageAI is waiting for you to select a response to a text, it will generate responses to the 5th message only). It also ensured that blank texts and duplicate texts were not responded to, so it gave a natural, human response vibe. The only limitation is that if different numbers are texting you at the same time you may not get prompted to reply to the most recent text in conversation 2 if you are already replying to conversation 1 and another text comes in after the conversation 2 text. Handling individual recent texts by *handle_id*'s is a great next step. Rolling context for more than one text based on a *handle_id* would greatly strengthen response quality. I added moderately extensive debug logs in the model python file, which can also be viewed in the demo video. In summary, I learned a lot about this project and really enjoyed creating it. To bring it to a larger scale, I need developer permissions, a better chat model, and *time*. Time is one of the most important and constraining factors in the engingeering design cycle.