Meet Ava: the Whatsapp Agent

Lesson 1: Course Overview

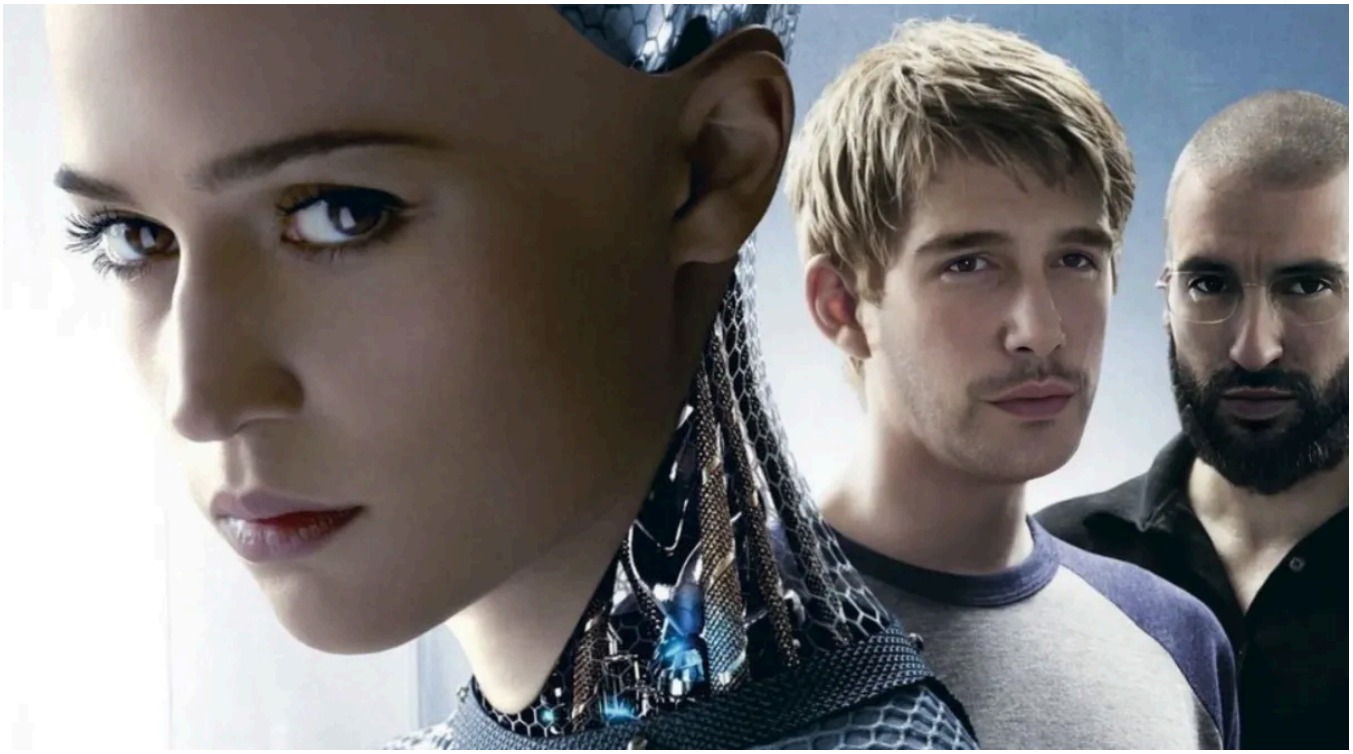
MIGUEL OTERO PEDRIDO
FEB 05, 2025

87 5 10 Sl

What happens when **two ML Engineers** with a love for sci-fi movies team up? 🤔

You get **Ava,** a **Whatsapp agent** that can engage with users in a **realistic way**, inspired by the great film Ex Machina. Ok, let's be real, you won't be building a full sentient robot in this project, but you will enjoy some pretty interesting Whatsapp conversations. I can assure you that! 😁

> **Check the code here!** 👩‍💻



The magic of face swapping

This course is divided into **six lessons**:

🏗️ **Lesson 1:** Project overview

🕸️ **Lesson 2:** Ava's brain is just a graph

🧠 **Lesson 3:** Unlocking Ava's memories

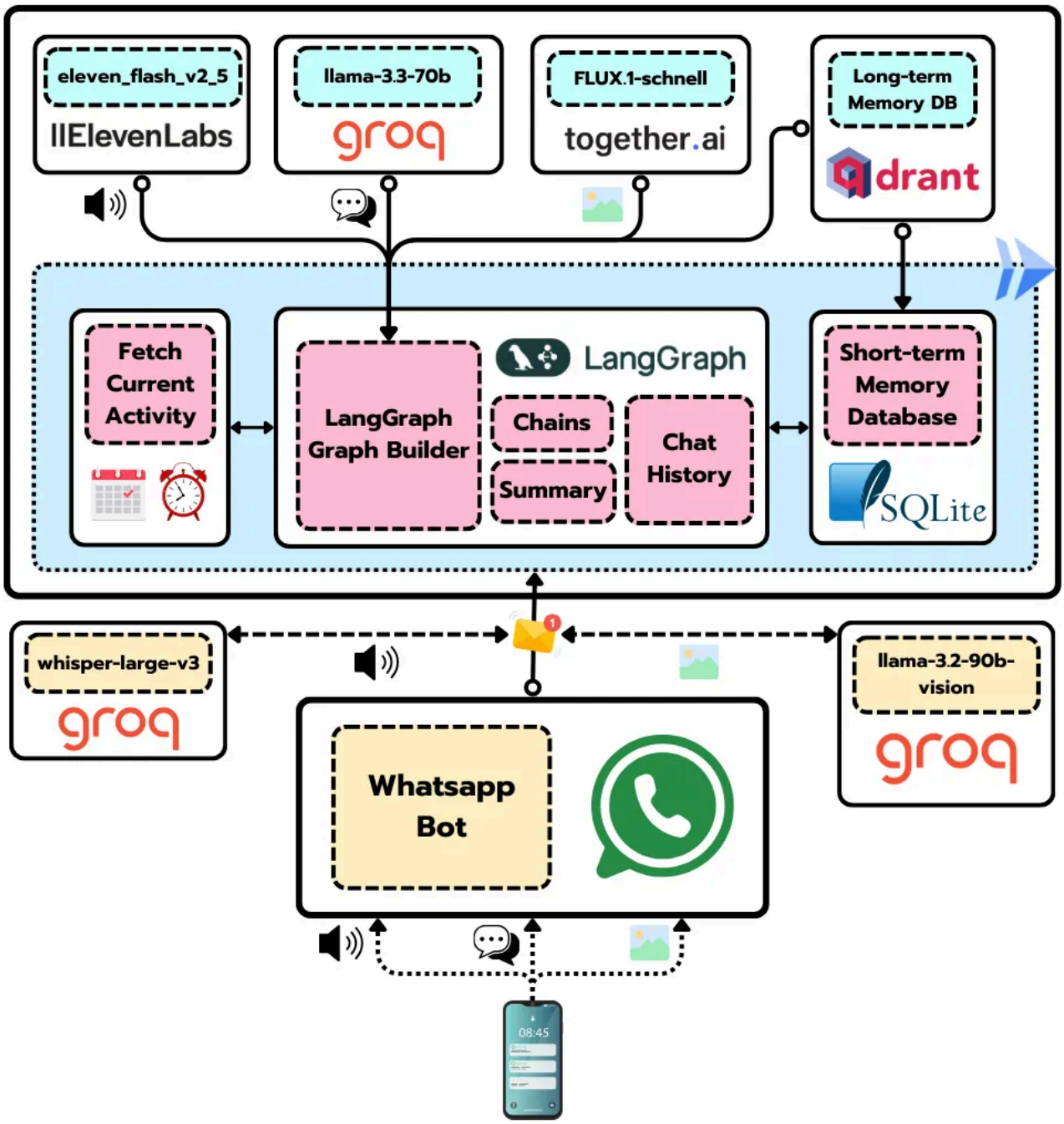
🗣️ **Lesson 4:** Giving Ava a Voice

👀 **Lesson 5:** Ava learns to see

📱 **Lesson 6:** Ava installs Whatsapp

Today, we'll start with the **first lesson** - a general introduction to the project and i core components.

Project Overview



Project Overview

Ava is a "Whatsapp Agent", meaning it will interact with you through this app. But i won't just rely on "regular" text messages, it will also **listen to your voice notes** (y even if you are **one of those** people 😒)and **react to your pictures**.

And that's not all ... Ava can also respond with its own voice notes and images of v it's up to - yes, Ava has a life beyond talking to you, don't be such a narcissist! 😂

Jesús in Westworld mode, messing with Ava's mind

At this point, you might be wondering:

> **What kind of system have we implemented to handle multimodal inputs / outputs coherently?**

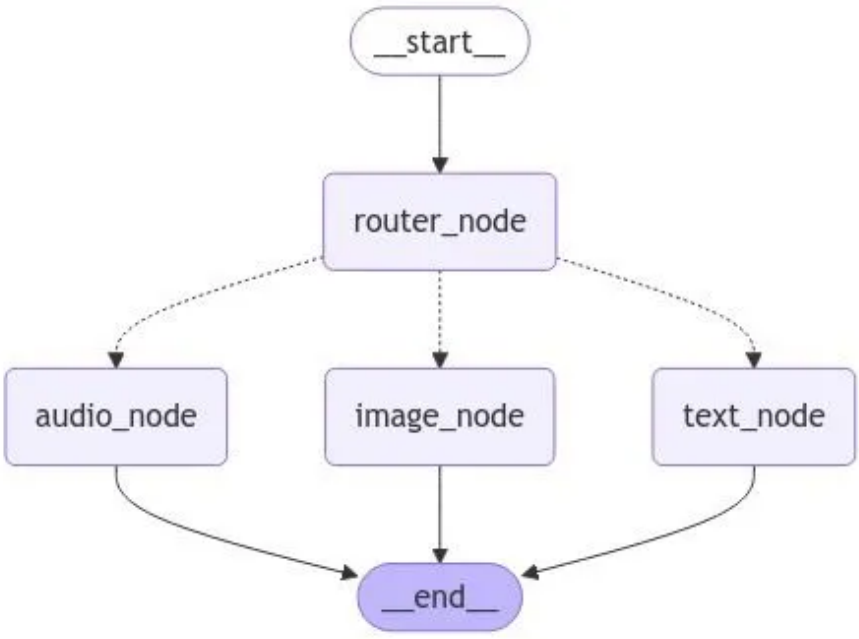
The short answer: Ava's brain is **just a graph** ... a LangGraph (sorry, I couldn't resist).

💠 Ava's Graph

Your brain is made up of neurons, right? Well, Ava's brain is made up of **LangGrap nodes** and **edges** - one for the processing images, another for listening to your vo another for fetching relevant memories, and so on.

At its core, **Ava is simply a graph with a state**. This state maintains all the key de of the conversation, including shared information (text, audio or images), current activities, and contextual information.

This is exactly what we'll explore in **Lesson 2**, where you'll learn how **LangGraph** be used to build agentic design architectures, such as **the router.**



Ava will determine the type of output based on your input

💠 Ava's memory

An Agent **without memory** is like talking to the **main character of "Memento"** (a if you haven't seen that film... seriously, what are you doing with your life?).

Ava has **two types of memory**:

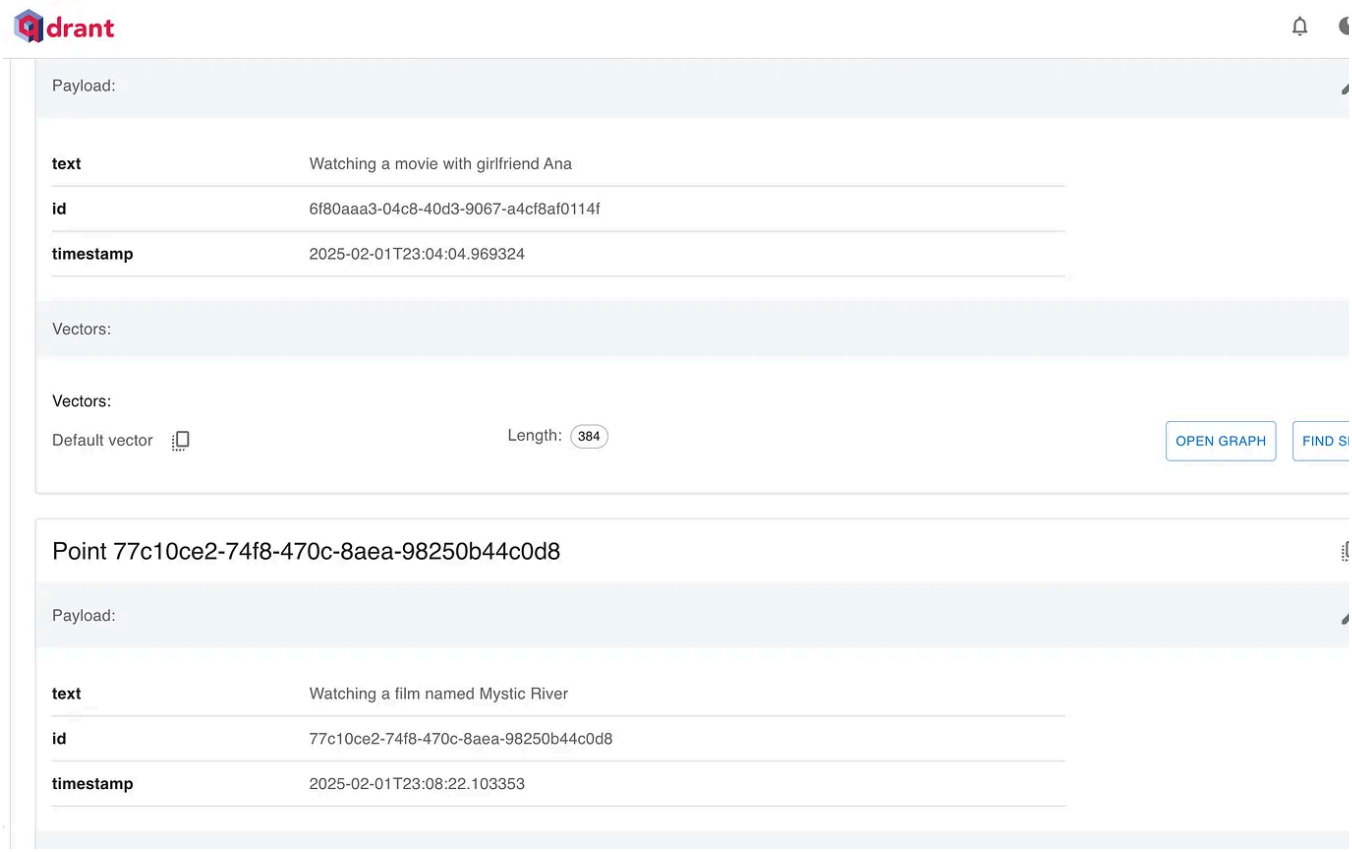
🔷 Short term memory

The usual - it stores the sequence of messages to maintain conversation context. our case, we save this sequence in **SQLite** (we are also storing a summary of the conversation, but that's for future lessons 😉).

🔷 Long term memory

When you meet someone, you don't remember ***everything*** they say; you retain or the **key details**, like their **name**, **profession**, or **where they're from**, right?. That' exactly what we wanted to replicate with **Qdrant** - extracting relevant information from the conversation and storing it as embeddings.

Don't worry because we'll cover the memory modules in **Lesson 3.**



Capturing relevant facts about the conversation (e.g. watching Mystic River with my girlfriend)

💠 Ava's senses

Real Whatsapp conversations aren't limited to just text. Think about it - do you remember the last cringe GIF your mom sent you last week? Or that neverending v note from your high school friend? Exactly. We need both images and audio.

To make this possible, we've selected the following tools.

🔷 Text

Both Jesús and I are **Groq** fans (if you chat with Ava, ask about its job, you might b surprised). That's why we are using **Groq models** for all text generation. Specifica we've chosen **llama-3.3-70b-versatile** as our core LLM.

🔷 Images

The image module handles **two tasks**: **processing user images** and **generating ones** (take a look at the image below).

- For **image "understanding"**, we're using Groq's **llama-3.2-90b-vision-preview**.
- For **image generation**, **black-forest-labs/FLUX.1-schnell-Free** using **Toge AI**.

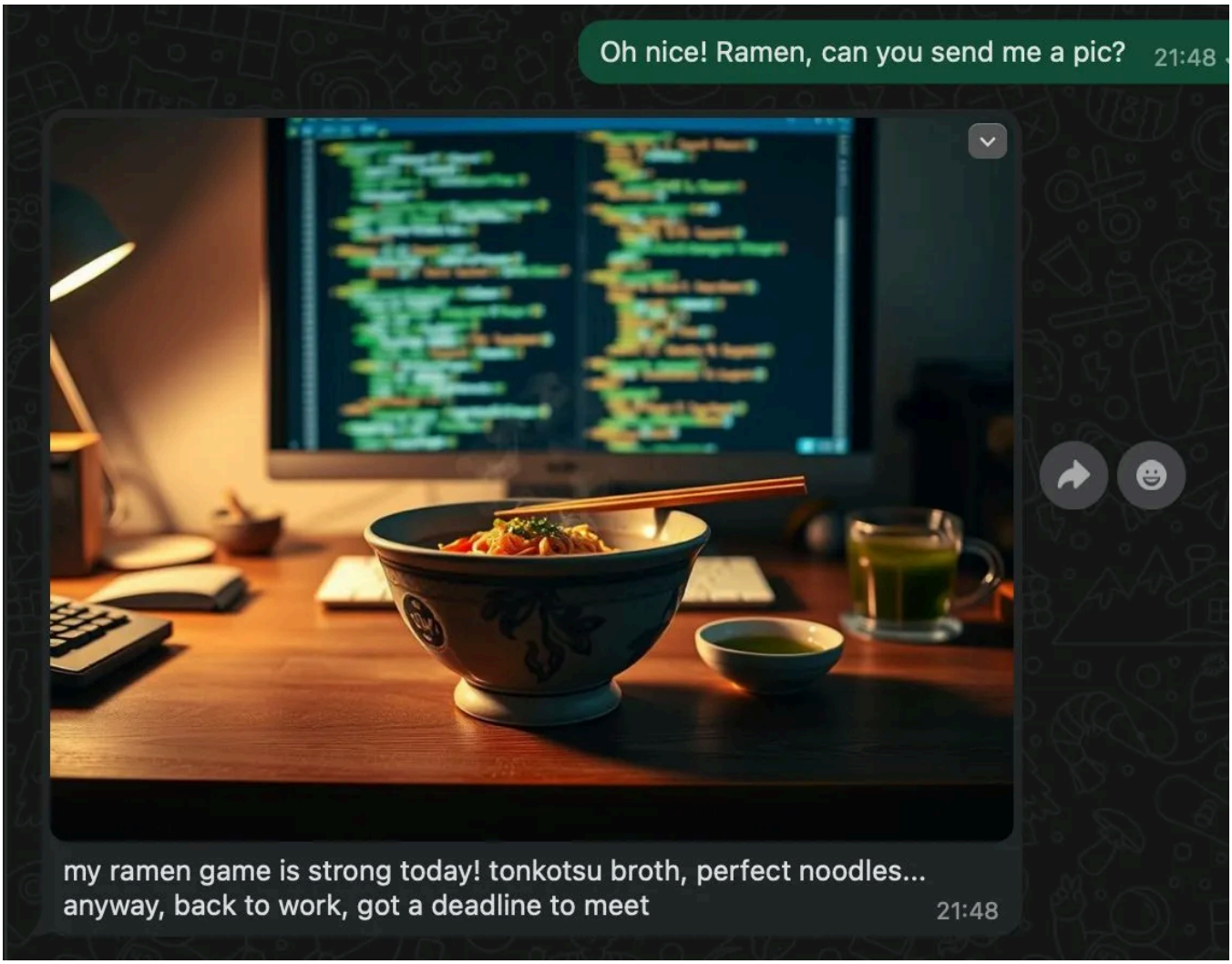
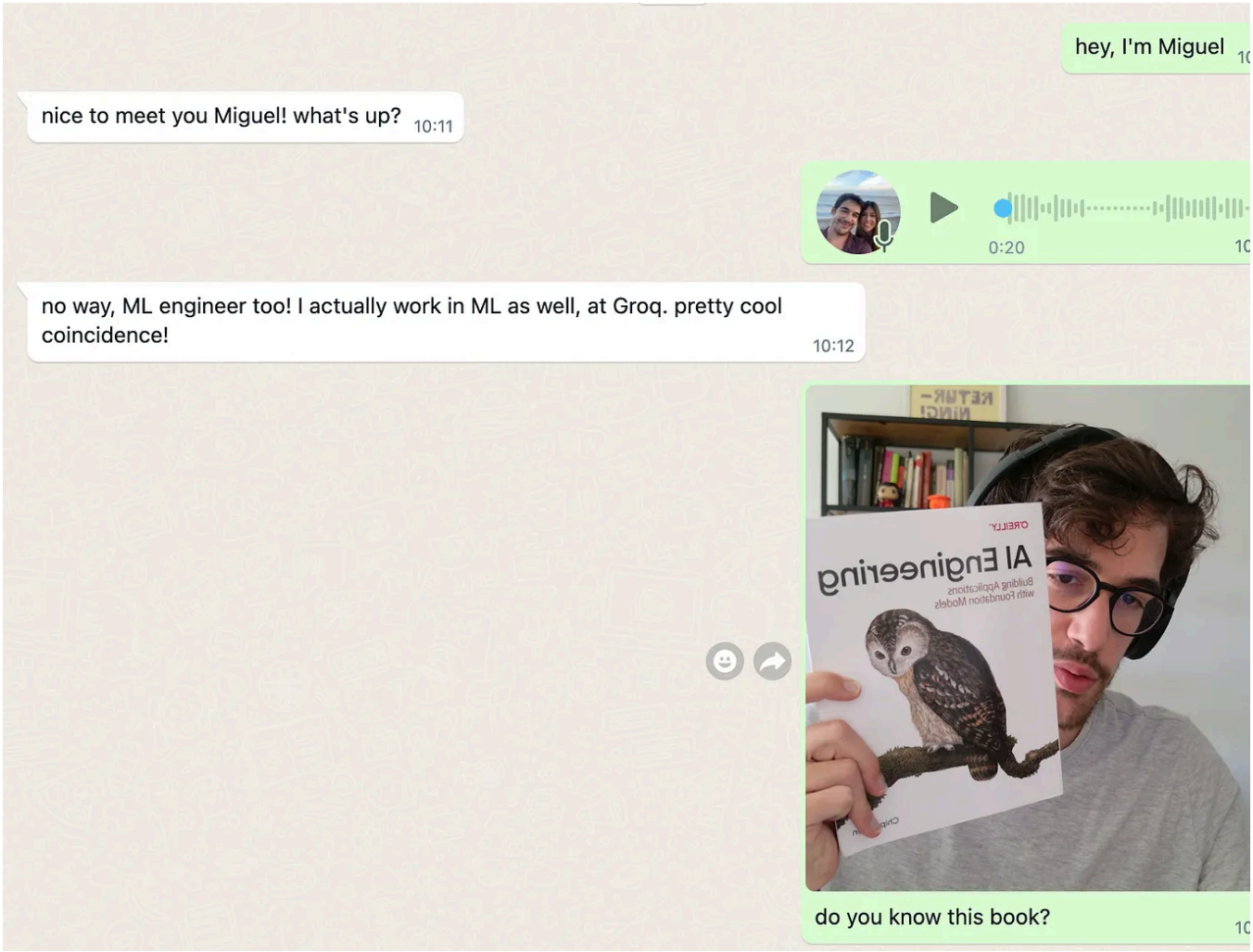


Image generation example. Turns out Ava loves ramen

🔷 Audio

The audio module needs to take care of **TTS (Text-To-Speech)** and **STT (Speec To-Text)**.

- For **TTS**, we are using **Elevenlabs** voices.
- For **STT**, **whisper-large-v3-turbo** from **Groq.**



Ava listens to my voice note, where I'm introducing myself as an ML Engineer

We'll cover the **audio module** in **Lesson 4** and the **image module** in **Lesson 5**!

And that's all for today! As you can see, this is a very complete course, so we hop you're excited to get started with it! Remember, **Lesson 2** will be available next

Dissecting Ava's brain

Lesson 2: Mastering LangGraph Workflows

MIGUEL OTERO PEDRIDO
FEB 12, 2025

30 5 2 Sl

Picture this: you're a **mad scientist** living in a creepy old house in the middle of t forest, and your mission is to build a **sentient robot**. What's the **first thing** you'd

Yep, you'd start with the brain, right? 🧠

Now, don't worry - neither Jesús nor I are actually mad (though, as a physicist, I m be a little questionable). But when we started building Ava, we also kicked things with the **"brain"**.

And that's exactly what **Lesson 2** is all about - **building Ava's brain using LangGraph!** 🕸️

> [Check the code here!](#) 👨‍💻

What if Oscar Isaac implemented Ava's brain as a LangGraph application?

> This is the **second lesson** of **"Ava: The Whatsapp Agent"** course. This lessor builds on the theory and code covered in the previous ones, so be sure to chec them out if you haven't already!
>
> - Lesson One: Project Overview

LangGraph in a Nutshell

Never used LangGraph before? No worries, here's a quick intro.

LangGraph models **agent workflows** as **graphs**, using **three** main components:

🔶 **State** - A shared data structure that tracks the current status of your app (workflow).

🔶 **Nodes** - Python functions that define the agent behaviour. They take in the cu state, perform actions, and return the updated state.

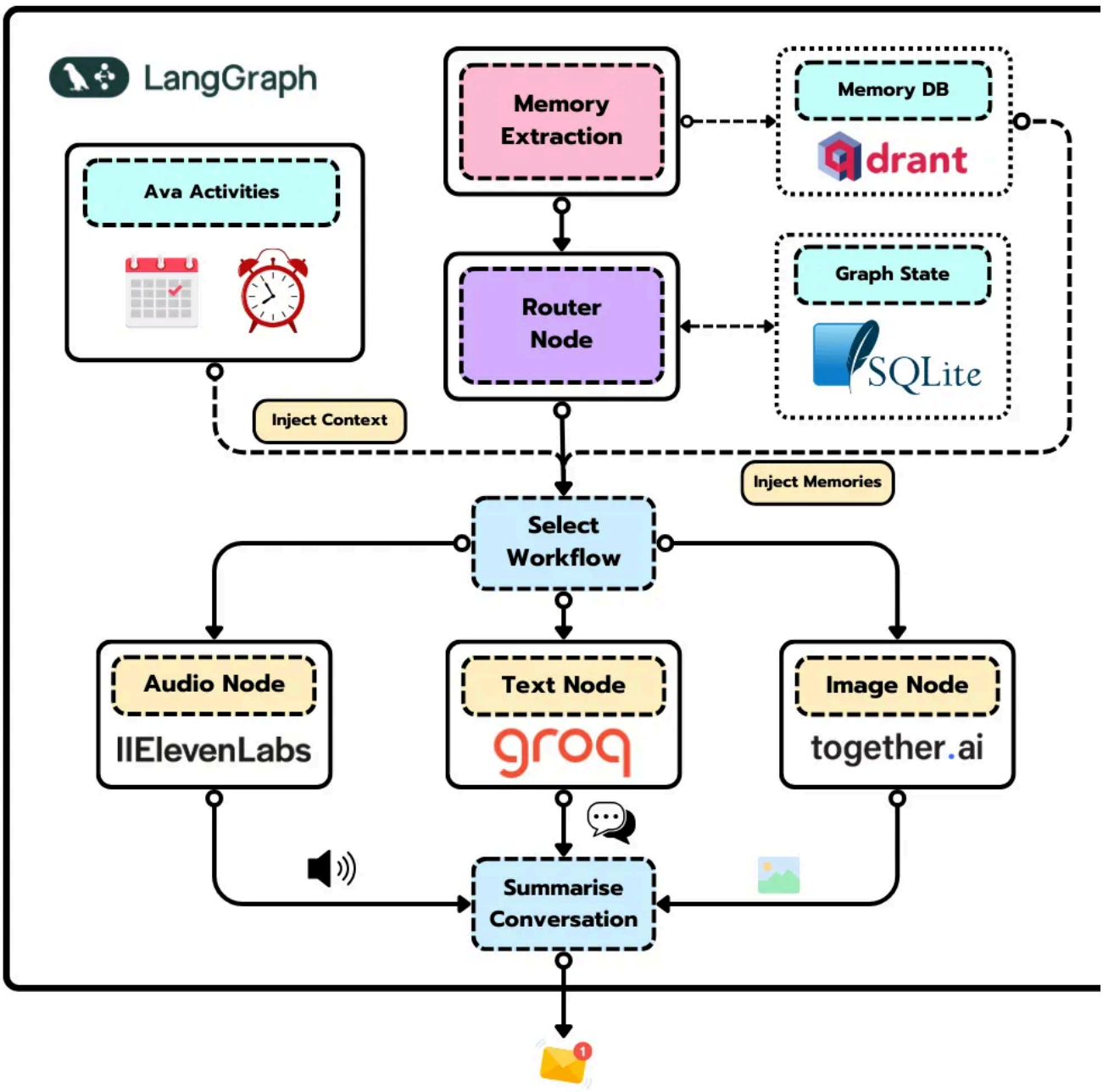
🔶 **Edges** - Python functions that decide which Node runs next based on the Stat allowing for conditional or fixed transitions (we'll see an example of conditional ed later 😉)

By combining **Nodes** and **Edges**, you can build **dynamic workflows**, like Ava! In t next section, we'll take a look at Ava's graph and its Nodes and Edges.

Let's go! 👇

> If you want something more complete, I recommend you to check this **LangCh Academy** course: Introduction to LangGraph.

Ava's graph



A diagram showcasing Ava's brain as a LangGraph workflow

> If you want to see the full graph implementation, take a look here.

Before we get into the Nodes and the Edges, let's describe Ava's state.

💠 Ava State

As we mentioned earlier, LangGraph keeps track of your app's current status usin **State**. Ava's state has these attributes:

- **summary** - The summary of the conversation so far (more on this in Lesson 3
- **workflow** - The current workflow Ava is in. Can be “conversation”, “image” or “audio”. More on this when we talk about the Router Node.
- **audio_buffer** - The buffer containing audio data for voice messages. This is something we’ll cover in Lesson 4.
- **image_path** - Path to the current image being generated. More about this in Lesson 5.
- **current_activity** - Description of Ava’s current simulated activity.
- **apply_activity** - Flag indicating whether to apply or update the current activi

```python
class AICompanionState(MessagesState):
    """State class for the AI Companion workflow.

    Extends MessagesState to track conversation history and maintains the last message receive

    Attributes:
        summary (str): A brief summary of the current conversation context.
        workflow (str): The current workflow/mode the AI companion is operating in.
        audio_buffer (bytes): Buffer containing audio data for voice messages.
        image_path (str): Path to the current image being processed.
        current_activity (str): Description of the AI's current simulated activity.
        apply_activity (bool): Flag indicating whether to apply/update the current activity.
    """

    summary: str
    workflow: str
    audio_buffer: bytes
    image_path: str
    current_activity: str
    apply_activity: bool
```

This state will be saved in an external database. We went with **SQLite3** for simplic we’ll get into the details in the next lesson when we cover Ava’s short-term memo

Now that we know how Ava’s State is set up, let’s check out the nodes and edges.

💠 Memory Extraction Node

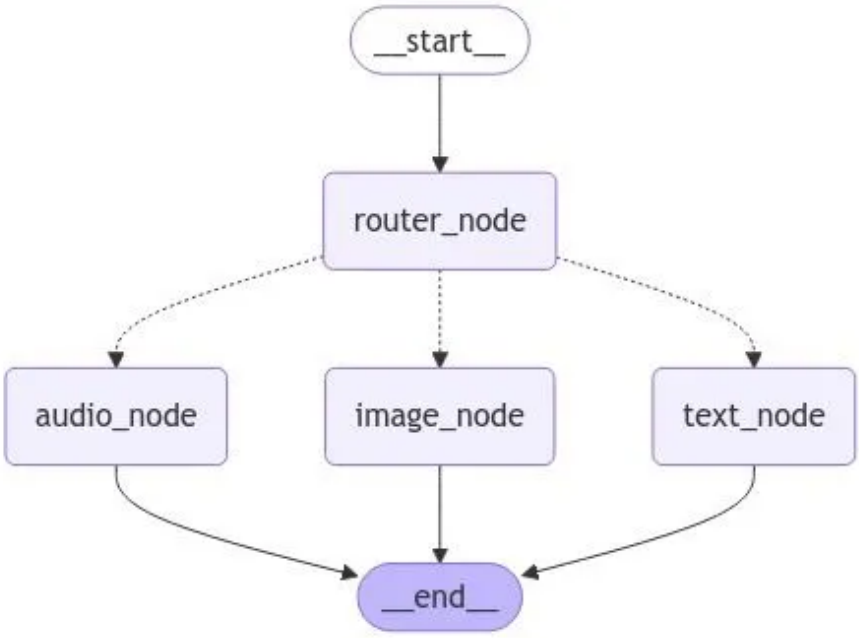
The first node of the graph is the **memory extraction node**. This node will take ca of extracting relevant information from the user conversation (e.g. name, age, background, etc.) We will explain this node in detail in the next lesson, when we explore the memory modules.

🔷 Context Injection Node

To appear like a real person, Ava needs to do more than just chat with you. That's we need a node that checks your local time and matches it with Ava's schedule. T handled by the ScheduleContextGenerator class, which you can see in action belo

```
def context_injection_node(state: AICompanionState):
    schedule_context = ScheduleContextGenerator.get_current_activity()
    if schedule_context != state.get("current_activity", ""):
        apply_activity = True
    else:
        apply_activity = False
    return {"apply_activity": apply_activity, "current_activity": schedule_context
```

🔷 Router Node



The router node is at the heart of Ava's workflow. It determines which workflow Av response should follow - **audio** (for audio responses), **image** (for visual responses **conversation** (regular text replies).

```python
async def router_node(state: AICompanionState):
    chain = get_router_chain()
    response = await chain.ainvoke(
        {"messages": state["messages"][-settings.ROUTER_MESSAGES_TO_ANALYZE :]
    )
    return {"workflow": response.response_type}
```

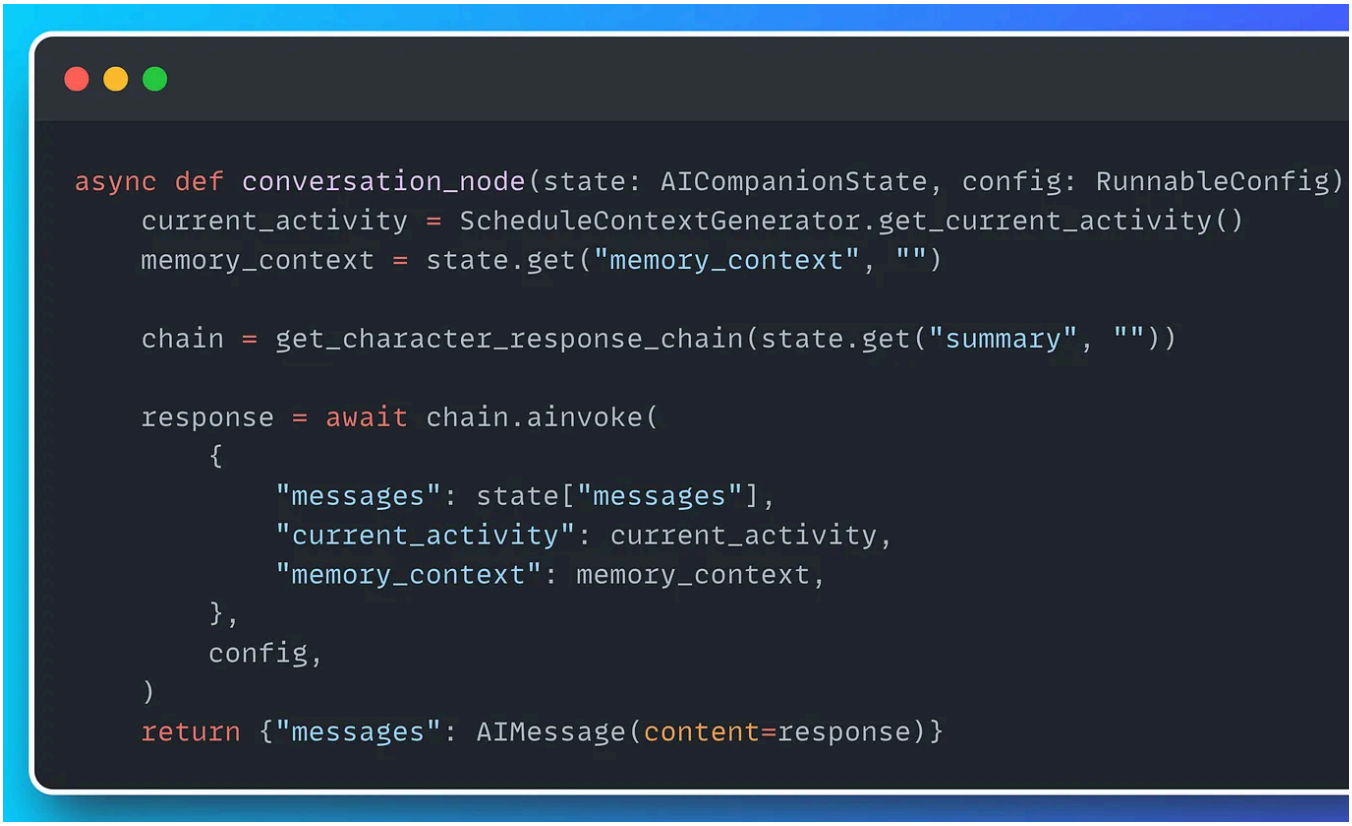
The logic behind this router is pretty straightforward - we're just creating a chain v a structured output (**RouterResponse**). You can check out the System prompt we using for this task here.

```python
class RouterResponse(BaseModel):
    response_type: str = Field(
        description="The response type to give to the user. It must be one of: 'conversation', 'image' or 'audi
    )


def get_router_chain():
    model = get_chat_model(temperature=0.3).with_structured_output(RouterResponse)

    prompt = ChatPromptTemplate.from_messages(
        [("system", ROUTER_PROMPT), MessagesPlaceholder(variable_name="messages")]
    )

    return prompt | model
```

Once the Router Node determines the final answer, the chosen workflow is assign to the "**workflow**" attribute of the **AICompanionState.** This information is then us by the **select_workflow** edge, which connects the router node to either the imag audio or conversation nodes.

```python
def select_workflow(
    state: AICompanionState,
) -> Literal["conversation_node", "image_node", "audio_node"]:
    workflow = state["workflow"]

    if workflow == "image":
        return "image_node"

    elif workflow == "audio":
        return "audio_node"

    else:
        return "conversation_node"
```

Both the image node and the audio node will be covered in future lessons. As for t conversation one (plain text), there's not much to elaborate on. It simply creates a conversational chain that takes the summary and uses Ava's character card prom generate a response.



```python
async def conversation_node(state: AICompanionState, config: RunnableConfig)
    current_activity = ScheduleContextGenerator.get_current_activity()
    memory_context = state.get("memory_context", "")

    chain = get_character_response_chain(state.get("summary", ""))

    response = await chain.ainvoke(
        {
            "messages": state["messages"],
            "current_activity": current_activity,
            "memory_context": memory_context,
        },
        config,
    )
    return {"messages": AIMessage(content=response)}
```

💠 Summarization Node

One challenge with long conversations is the huge number of messages that neec be stored in memory. To fix this, we're using an extra node - a **summarization no**

Basically, this node takes the conversation, sums it up, and adds the summary as new attribute in the state. Then, whenever Ava generates a new response, this summary gets used in the System Prompt.

```python
async def summarize_conversation_node(state: AICompanionState):
    model = get_chat_model()
    summary = state.get("summary", "")

    if summary:
        summary_message = (
            f"This is summary of the conversation to date between Ava and the user: {summary}\n\
            "Extend the summary by taking into account the new messages above:"
        )
    else:
        summary_message = (
            "Create a summary of the conversation above between Ava and the user. "
            "The summary must be a short description of the conversation so far, "
            "but that captures all the relevant information shared between Ava and the user:"
        )

    messages = state["messages"] + [HumanMessage(content=summary_message)]
    response = await model.ainvoke(messages)

    delete_messages = [
        RemoveMessage(id=m.id)
        for m in state["messages"][: -settings.TOTAL_MESSAGES_AFTER_SUMMARY]
    ]
    return {"summary": response.content, "messages": delete_messages}
```

But of course, we don't want to generate a summary **every single time** Ava gets a message. That's why this node is connected to the previous ones with a **conditio edge.**

```python
def should_summarize_conversation(
    state: AICompanionState,
) -> Literal["summarize_conversation_node", "__end__"]:
    messages = state["messages"]

    if len(messages) > settings.TOTAL_MESSAGES_SUMMARY_TRIGGER:
        return "summarize_conversation_node"

    return END
```

As you can see in the implementation above, this edge connects the **summarizat node** to the previous nodes if the total number of messages exceeds the **TOTAL_MESSAGES_SUMMARY_TRIGGER** (which is set to 20 by default). If not, will connect to the **END** node, which marks the end of the workflow.

And that's all for today!

Just a reminder - **Lesson 3** will be available next **Wednesday, February 19th.** An don't forget there's also a **complementary video lesson** on **Jesús Copado's YouTube channel**.

> We **strongly recommend** exploring **both resources** (**written** lessons and **vid** lessons) to **maximize** your **learning experience**! 🙂

Can Agents get nostalgic about the past?

Lesson 3: Unlocking Ava's memories

MIGUEL OTERO PEDRIDO
FEB 19, 2025

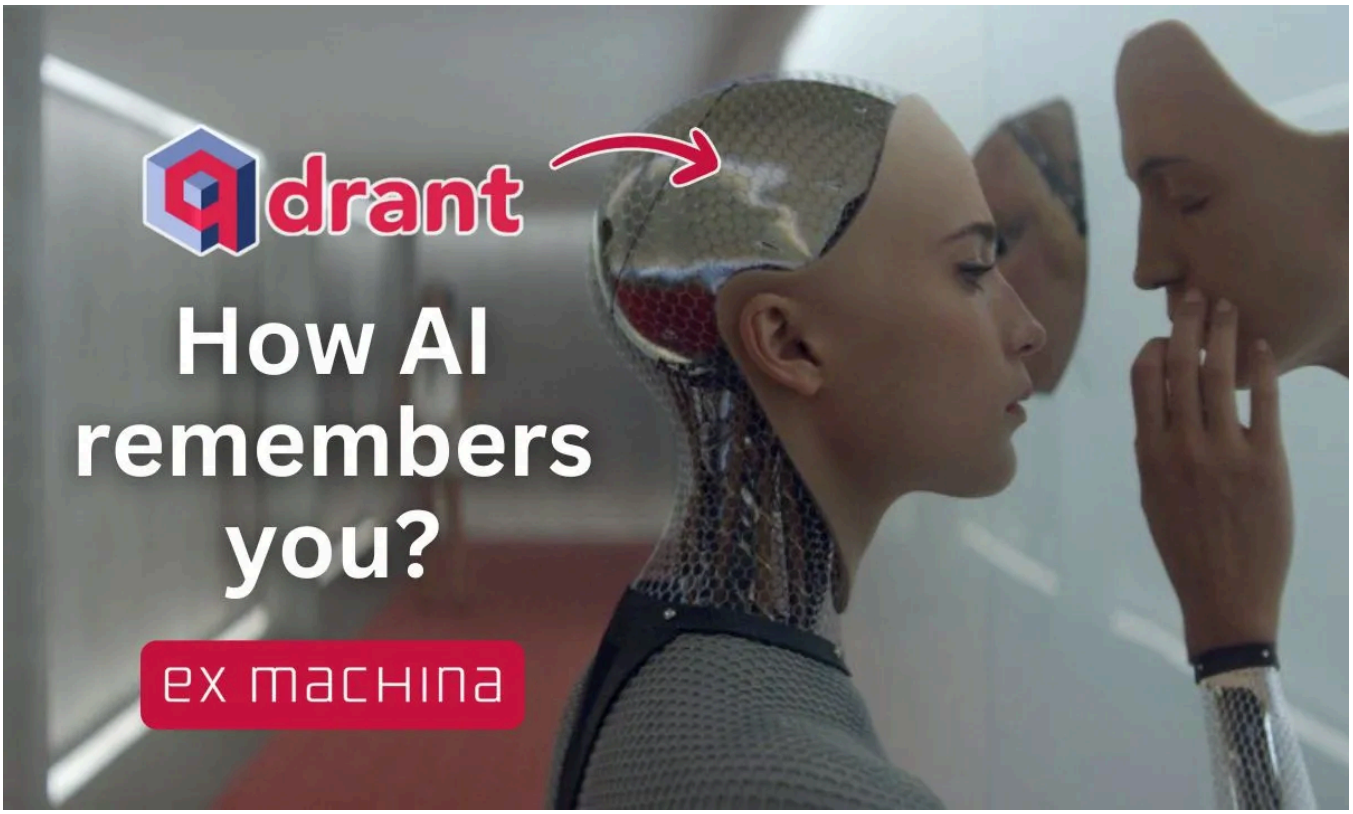
19 2 Sl

Talking to an Agent **without memory** is like dealing with the main character from **“Memento”** - you say your name, and one message later, it’s asking again ...

Ava was no different. We needed a memory module that made her feel real - one t actually keeps up with the conversation and remembers every relevant detail abou you. Whether you mentioned it two minutes ago or two months ago, she’s got it.

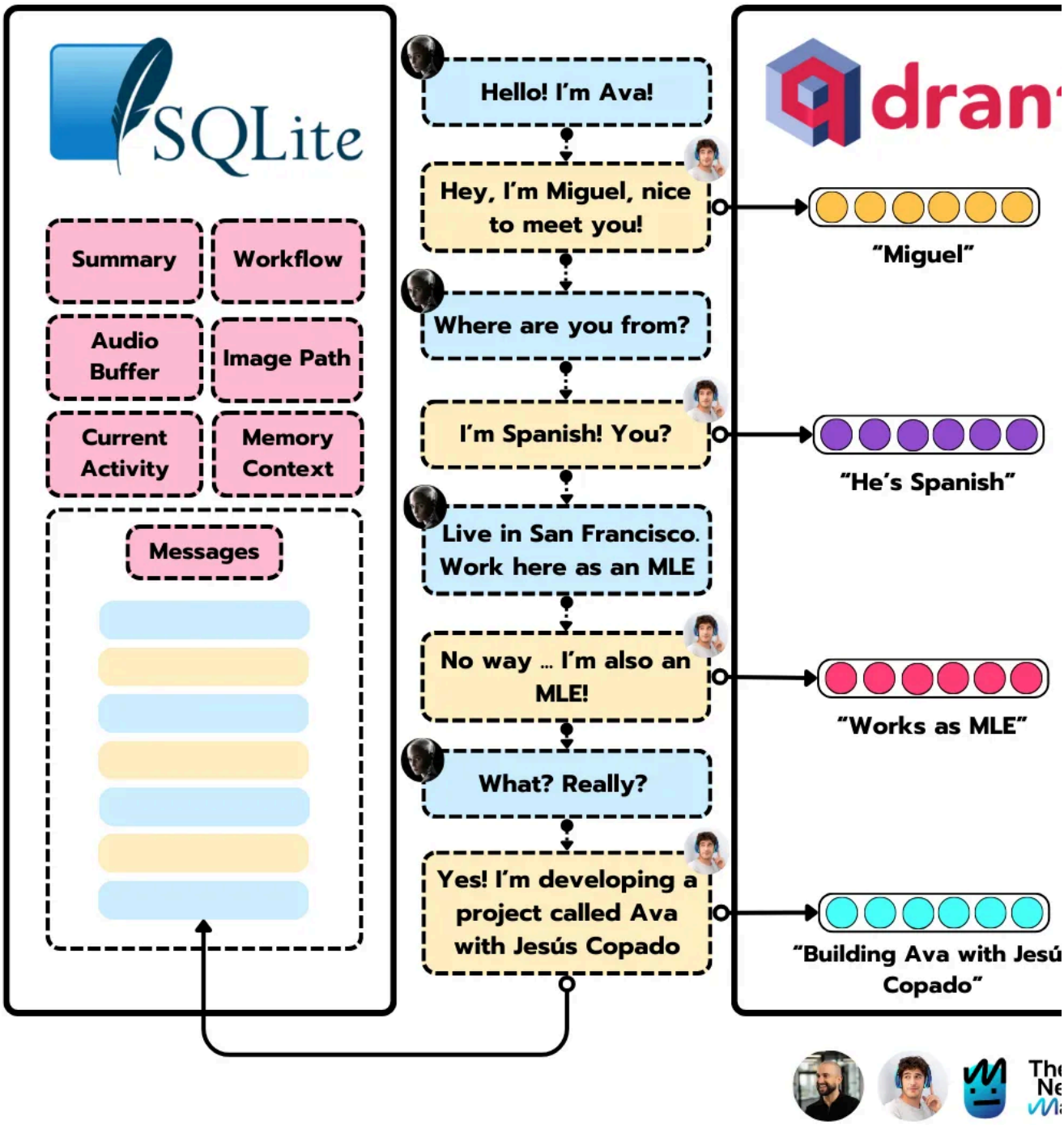
Ready for Lesson 3? Let’s give Ava a memory boost! 👇

> [Check the code here!](#) 👨‍💻



> This is the **third lesson** of **“Ava: The Whatsapp Agent”** course. This lesson bu
> on the theory and code covered in the previous ones, so be sure to check them
> if you haven’t already!
>
> - Lesson One: Project Overview
> - Lesson Two: Dissecting Ava’s brain

Ava’s Memory Overview



A general overview of the two memory systems: short-term (Sqlite) and long-term (Qdrant)

Let's start with a diagram to give you a big-picture view. As you can see, there are main memory "blocks" - one stored in a SQLite database (left) and the other in a Qdrant collection (right).

If this isn't clear yet, don't worry! We'll go through each memory module in detail i next sections.

💠 Short-term memory

The block on the left represents the short-term memory, which is stored in the LangGraph state and then persisted in a SQLite database. LangGraph makes this process simple since it comes with a built-in checkpointer for handling database storage.

In the code, we simply use the **AsyncSqliteSaver** class when compiling the graph This ensures that the LangGraph state checkpoint is continuously saved to SQLite You can see this in action in the code below.

```
...

# Process message through the graph agent
async with AsyncSqliteSaver.from_conn_string(
    settings.SHORT_TERM_MEMORY_DB_PATH
) as short_term_memory:
    graph = graph_builder.compile(checkpointer=short_term_memory)
    await graph.ainvoke(
        {"messages": [HumanMessage(content=content)]},
        {"configurable": {"thread_id": session_id}},
    )

    # Get the workflow type and response from the state
    output_state = await graph.aget_state(
        config={"configurable": {"thread_id": session_id}}
    )

workflow = output_state.values.get("workflow", "conversation")
response_message = output_state.values["messages"][-1].content

....
```

Ava's state is a subclass of LangGraph's MessageState, which means it inherits a **messages** property. This property holds the history of messages exchanged in th conversation - essentially, **that's what we call short-term memory!**

Integrating this short-term memory into the response chain is straightforward. We use LangChain's **MessagesPlaceholder** class, allowing Ava to consider past interactions when generating responses. This keeps the conversation smooth and coherent.

```python
def get_character_response_chain(summary: str = ""):
    model = get_chat_model()
    system_message = CHARACTER_CARD_PROMPT

    if summary:
        system_message += (
            f"\n\nSummary of conversation earlier between Ava and the user: {summary}"
        )

    prompt = ChatPromptTemplate.from_messages(
        [
            ("system", system_message),
            MessagesPlaceholder(variable_name="messages"),
        ]
    )

    return prompt | model | AsteriskRemovalParser()
```

Simple, right? Now, let's get into the interesting part: the **long-term memory.**

💠 Long-term memory

Long-term memory isn't just about saving every single message from a conversat far from it 😅. That would be impractical and impossible to scale. Long-term mem works quite differently.

Think about when you meet someone new - you don't remember every word they right? You only retain **key details**, like their **name**, **profession**, **where they're fro** or **shared interests**.

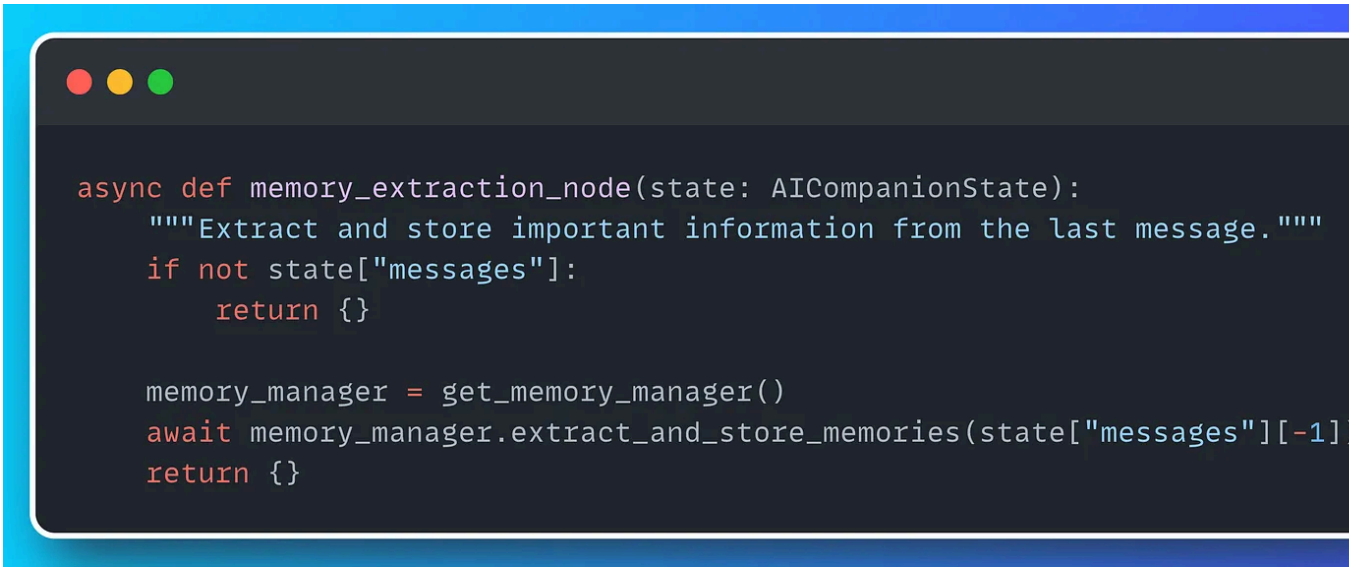
That's exactly what we wanted to replicate with Ava. **How?** 🤔

By using a Vector Database like Qdrant, that lets us store relevant information fror conversations as embeddings. Let's break this down in more detail.

🔶 Memory Extraction Node

Remember the other day when we talked about the different nodes in our LangGr workflow? The first one was the **memory_extraction_node**, which is responsible identifying and storing key details from the conversation.

That's the first essential piece we need to get our long-term memory module up a running! 💪



```
async def memory_extraction_node(state: AICompanionState):
    """Extract and store important information from the last message."""
    if not state["messages"]:
        return {}

    memory_manager = get_memory_manager()
    await memory_manager.extract_and_store_memories(state["messages"][-1]
    return {}
```

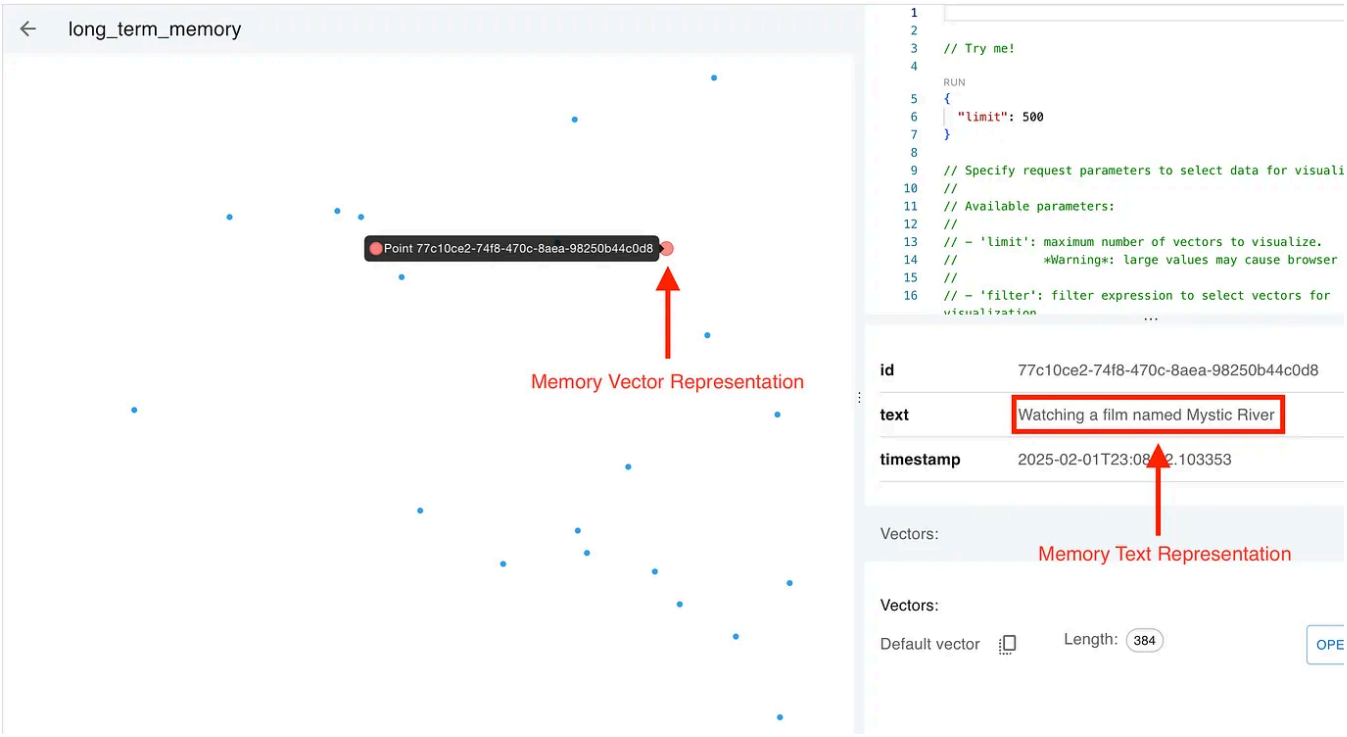
Using our MemoryManager class, this node will retrieve relevant information from the conversation and store it in Qdrant

🔶 Qdrant

As the conversation progresses, the **memory_extraction_node** will keep gatheri more and more details about you.

If you check your Qdrant Cloud instance, you'll see the collection gradually filling with "memories".

> Don't know how to configure Qdrant Cloud? There's a detailed guide on setting up here!



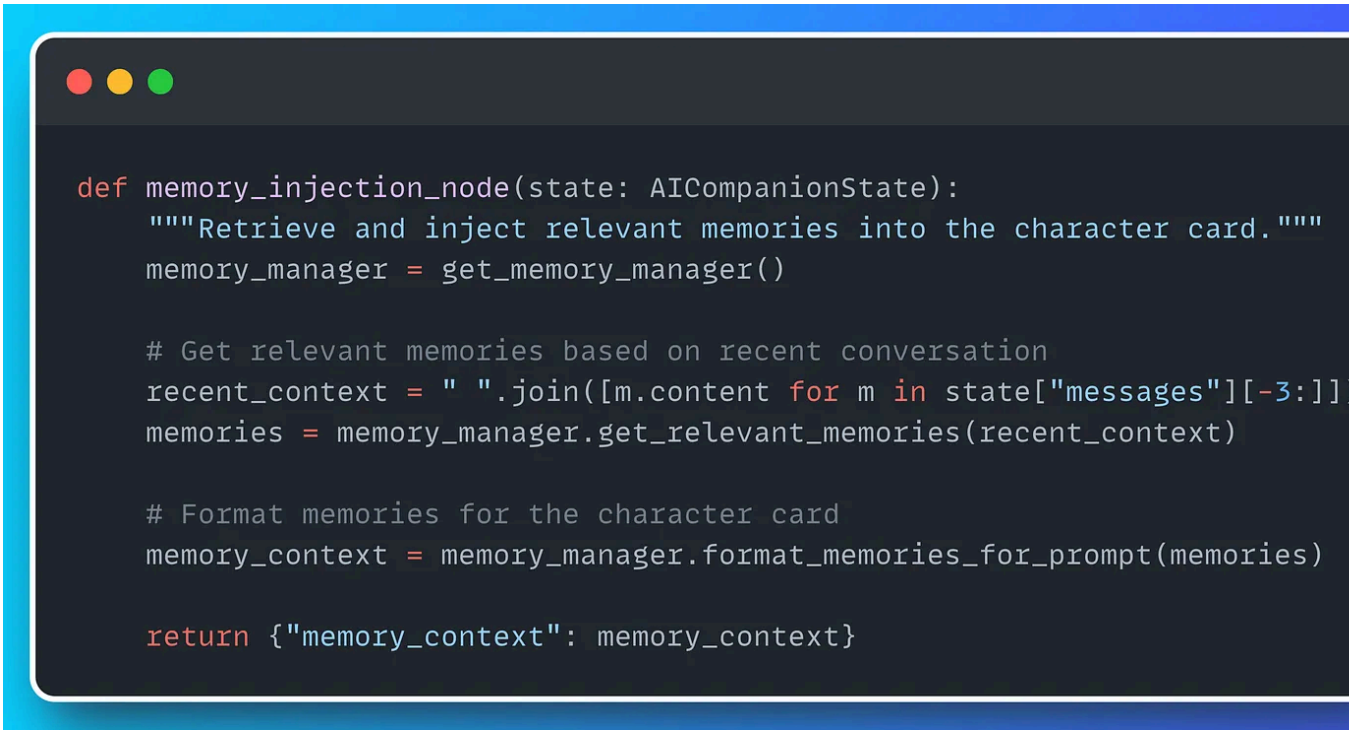
tSNE visualization of Ava's memories

In the example above, you can see how Ava remembers I was watching a film nam "Mystic River" (great movie, by the way). The memories are stored in Qdrant as embeddings by the **memory_extraction_node** using the **all-MiniLM-L6-v2** mod as can be seen in our VectorStore class.

🔶 Memory Injection Node

Now that all the memories are stored in Qdrant, how do we let Ava use them in he conversations?

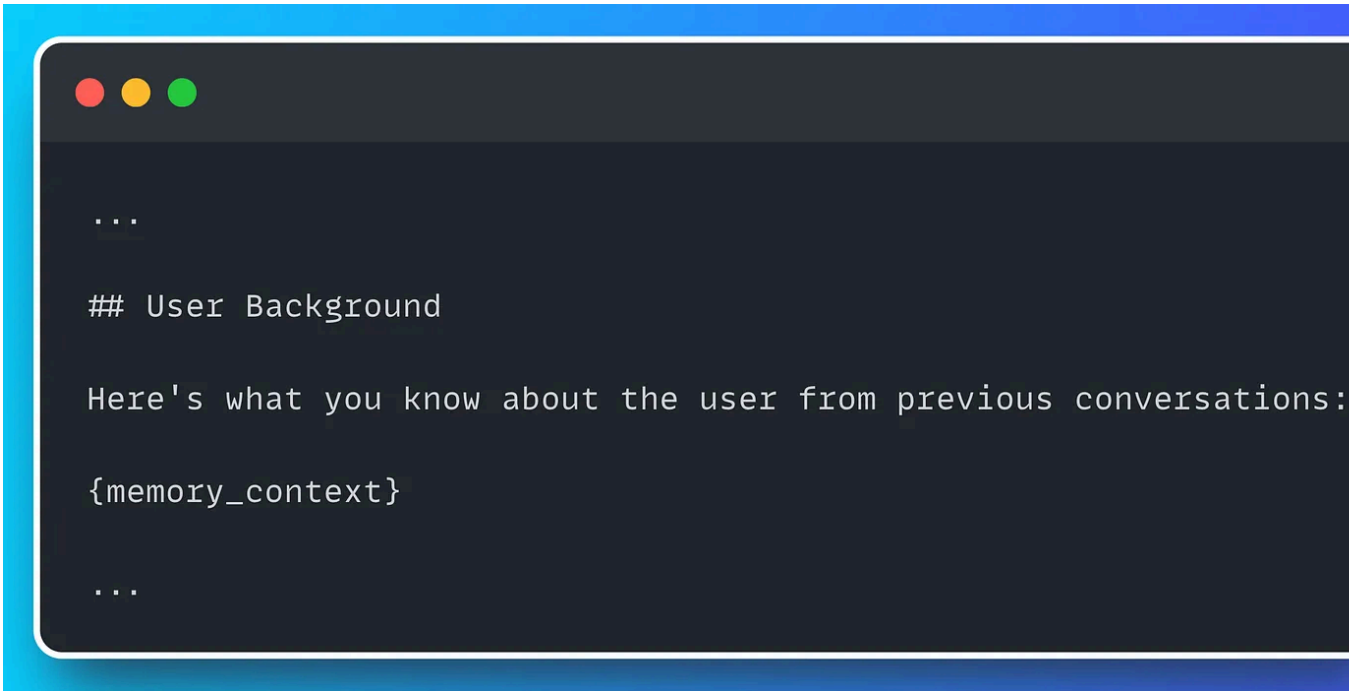
It's simple! We just need one more node: the **memory_injection_node.**



```python
def memory_injection_node(state: AICompanionState):
    """Retrieve and inject relevant memories into the character card."""
    memory_manager = get_memory_manager()

    # Get relevant memories based on recent conversation
    recent_context = " ".join([m.content for m in state["messages"][-3:]])
    memories = memory_manager.get_relevant_memories(recent_context)

    # Format memories for the character card
    memory_context = memory_manager.format_memories_for_prompt(memories)

    return {"memory_context": memory_context}
```

This node uses the MemoryManager class to retrieve relevant memories from Qdr essentially performing a vector search to find the top-k similar embeddings. Then transforms those embeddings (vector representations) into text using the **format_memories_for_prompt** method.

Once that's done, the formatted memories are stored in the **memory_context** property of the graph. This allows them to be parsed into the Character Card Pror the one that defines Ava's personality and behaviour.



```
...

## User Background

Here's what you know about the user from previous conversations:

{memory_context}

...
```

The Ultimate AI Voice Pipeline

Lesson 4: Giving Ava a voice

MIGUEL OTERO PEDRIDO
FEB 26, 2025

18 1 5 Sl

If you open **WhatsApp** and scroll through your recent chats, I bet you'll see more just text messages - memes, GIFs, videos ... and the star of today's article: **voice notes!** 🗣️

When Jesús Copado and I began the **Ava** project, we knew one thing for sure - Av needed a voice. **Not just any voice, but a unique one**. Don't believe me? Take a listen:

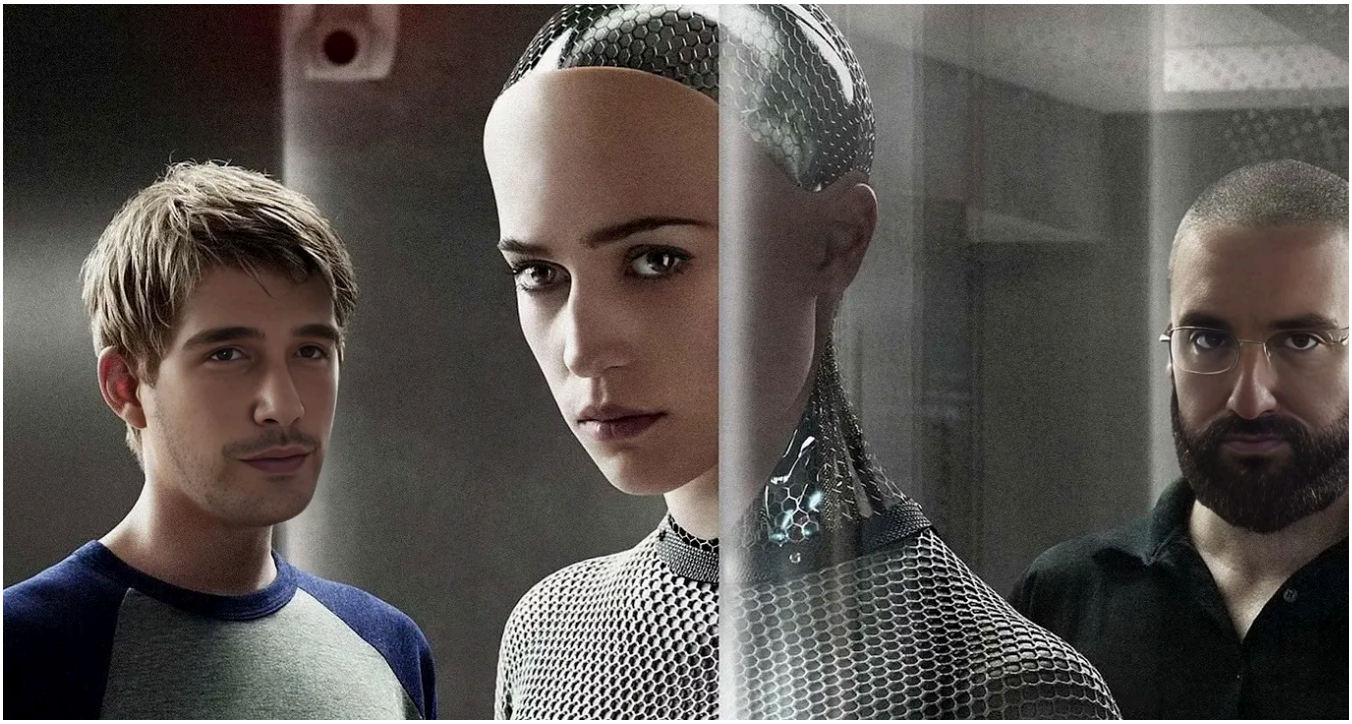
1× 0:00 -0:03

To make that happen, we needed two key systems: **STT (speech-to-text)** to turn incoming voice notes into text and **TTS (text-to-speech)** to convert Ava's replies audio.

That brings us to our focus today: **Ava's Voice Pipeline.**

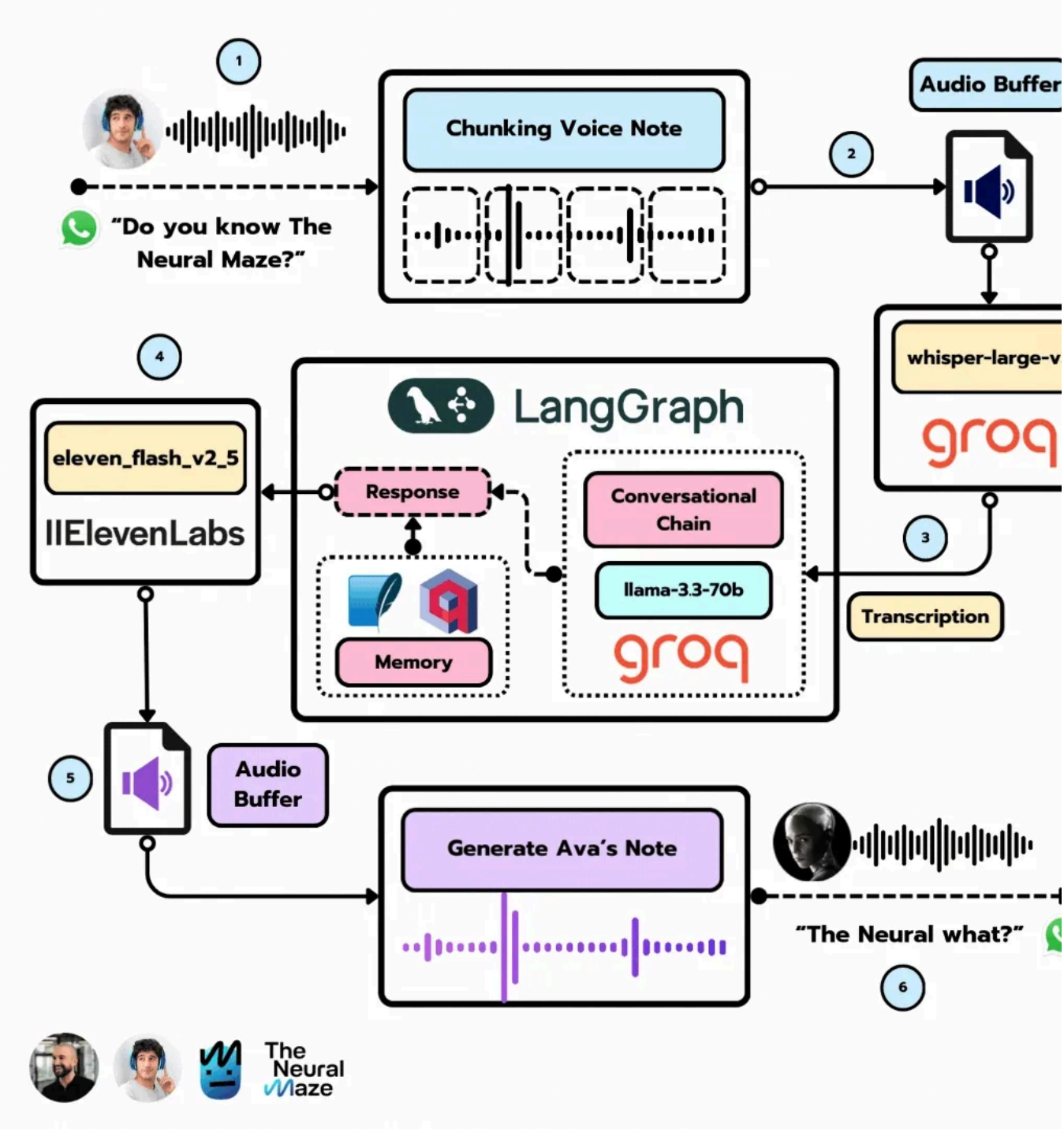
Ready? Let's begin! 👇

> [Check the code here!](#) 👨‍💻



> This is the **fourth lesson** of **“Ava: The Whatsapp Agent”** course. This lesson builds on the theory and code covered in the previous ones, so be sure to chec them out if you haven’t already!
>
> - Lesson One: Project Overview
> - Lesson Two: Dissecting Ava’s brain
> - Lesson Three: Unlocking Ava’s memories

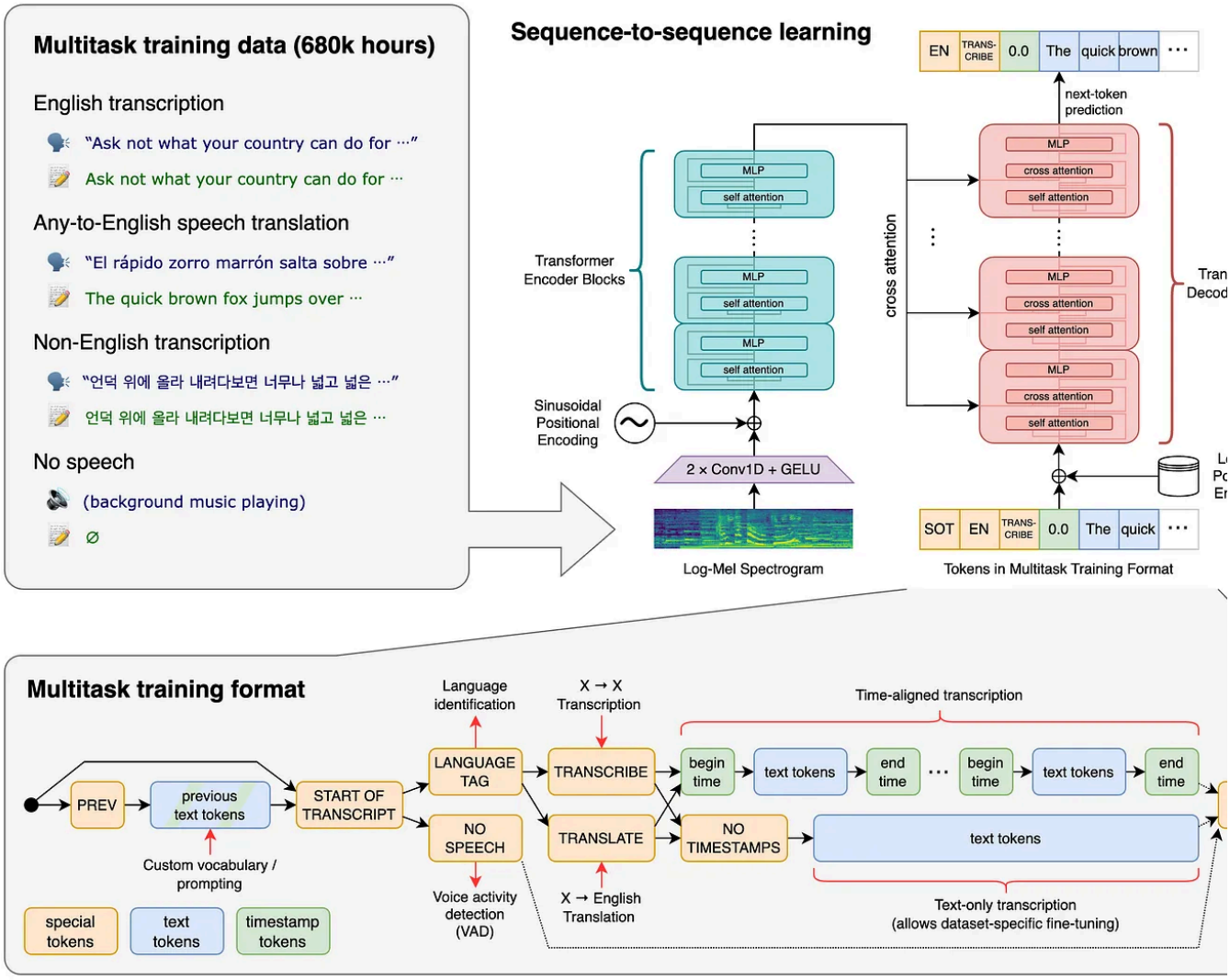
Ava’s Voice Pipeline



Ava's Voice Pipeline Overview

Like we mentioned before, this pipeline has **two main parts** - the **TTS module** an the **STT module** - which can work together or on their own. Let's go over each on the next sections.

STT Module - Whisper 👂



Whisper architecture (source: https://cdn.openai.com/papers/whisper.pdf)

Whisper is a key part of Ava's **STT (speech-to-text) module** because it helps Av accurately transcribe voice messages. If you're not familiar with "Whisper", it's an advanced model from OpenAI that can handle multiple languages, different accen and even background noise - perfect for WhatsApp voice messages! 📱

In the Voice Pipeline diagram, you'll see we're using Groq's Whisper - sorry, we ar hosting Whisper ourselves 😅 - , which you can check out here.

To use this model in our code, we've created a `SpeechToText` class inside Ava's modules. This class handles all the audio transcription logic - you'll find the magic the `transcribe` method in the snippet below! 👀

```python
class SpeechToText:
    """A class to handle speech-to-text conversion using Groq's Whisper model."""

    def __init__(self):
        """Initialize the SpeechToText class"""
        self._client: Optional[Groq] = None

    @property
    def client(self) -> Groq:
        """Get or create Groq client instance using singleton pattern."""
        if self._client is None:
            self._client = Groq(api_key=settings.GROQ_API_KEY)
        return self._client

    async def transcribe(self, audio_data: bytes) -> str:
        """Convert speech to text using Groq's Whisper model.

        Args:
            audio_data: Binary audio data

        Returns:
            str: Transcribed text

        Raises:
            ValueError: If the audio file is empty or invalid
            RuntimeError: If the transcription fails
        """
        if not audio_data:
            raise ValueError("Audio data cannot be empty")

        try:
            # Create a temporary file with .wav extension
            with tempfile.NamedTemporaryFile(suffix=".wav", delete=False) as temp_file
                temp_file.write(audio_data)
                temp_file_path = temp_file.name

                try:
                    # Open the temporary file for the API request
                    with open(temp_file_path, "rb") as audio_file:
                        transcription = self.client.audio.transcriptions.create(
                            file=audio_file,
                            model="whisper-large-v3-turbo",
                            language="en",
                            response_format="text",
                        )

                    if not transcription:
                        raise SpeechToTextError("Transcription result is empty")

                    return transcription

                finally:
                    # Clean up the temporary file
```

```
            os.unlink(temp_file_path)

        except Exception as e:
            raise SpeechToTextError(
                f"Speech-to-text conversion failed: {str(e)}"
            ) from e
```

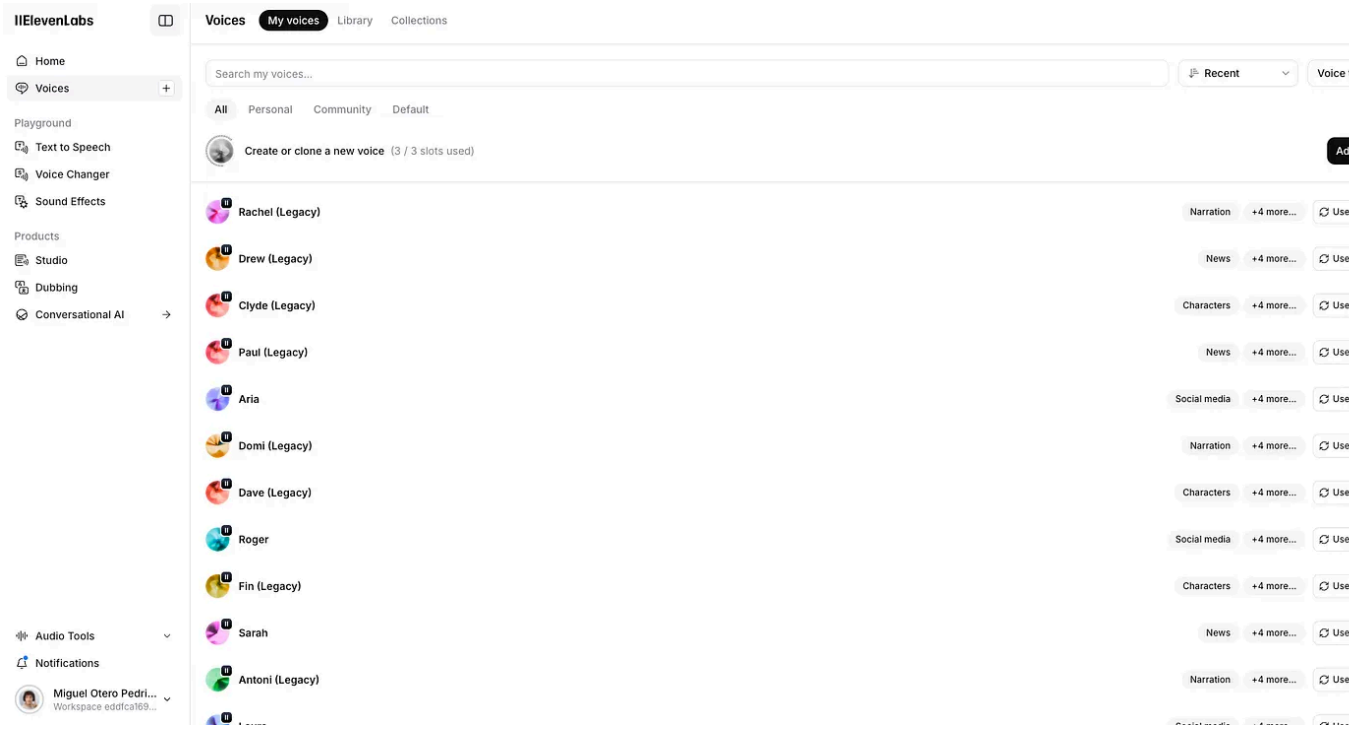
The SpeechToText class takes care of all the transcription logic

TTS Module - ElevenLabs 🗣️

Once the audio messsage is transcribed, it moves to the LangGraph workflow (Av brain, remember? 😉), which takes care of generating a response - using the short/long-term memories, Ava's activities, etc.

But this response is just text! We need a voice, and ... guess who has amazing voi ready to go?

You got it - ElevenLabs is in the house 😎



Creating custom voices in ElevenLabs

Like I mentioned at the start, we didn't want Ava to have just any voice - we wante something unique. That's why we're using **ElevenLabs' custom voice features**.

In the end, all we need is an `ELEVENLABS_VOICE_ID`, which uniquely defines Ava's voice.

So, to add the voice generation logic to the code, we followed the same approach as the STT module: we created a `TextToSpeech` class inside Ava's modules.

Check the `synthesize` method in the snippet below! 👀

```python
class TextToSpeech:
    """A class to handle text-to-speech conversion using ElevenLabs."""

    def __init__(self):
        """Initialize the TextToSpeech class"""
        self._client: Optional[ElevenLabs] = None

    @property
    def client(self) -> ElevenLabs:
        """Get or create ElevenLabs client instance using singleton pattern."""
        if self._client is None:
            self._client = ElevenLabs(api_key=settings.ELEVENLABS_API_KEY)
        return self._client

    async def synthesize(self, text: str) -> bytes:
        """Convert text to speech using ElevenLabs.

        Args:
            text: Text to convert to speech

        Returns:
            bytes: Audio data

        Raises:
            ValueError: If the input text is empty or too long
            TextToSpeechError: If the text-to-speech conversion fails
        """
        if not text.strip():
            raise ValueError("Input text cannot be empty")

        if len(text) > 5000:  # ElevenLabs typical limit
            raise ValueError("Input text exceeds maximum length of 5000 characters"

        try:
            audio_generator = self.client.generate(
                text=text,
                voice=Voice(
                    voice_id=settings.ELEVENLABS_VOICE_ID,
                    settings=VoiceSettings(stability=0.5, similarity_boost=0.5),
                ),
                model=settings.TTS_MODEL_NAME,
            )

            # Convert generator to bytes
            audio_bytes = b"".join(audio_generator)
            if not audio_bytes:
                raise TextToSpeechError("Generated audio is empty")

            return audio_bytes

        except Exception as e:
```

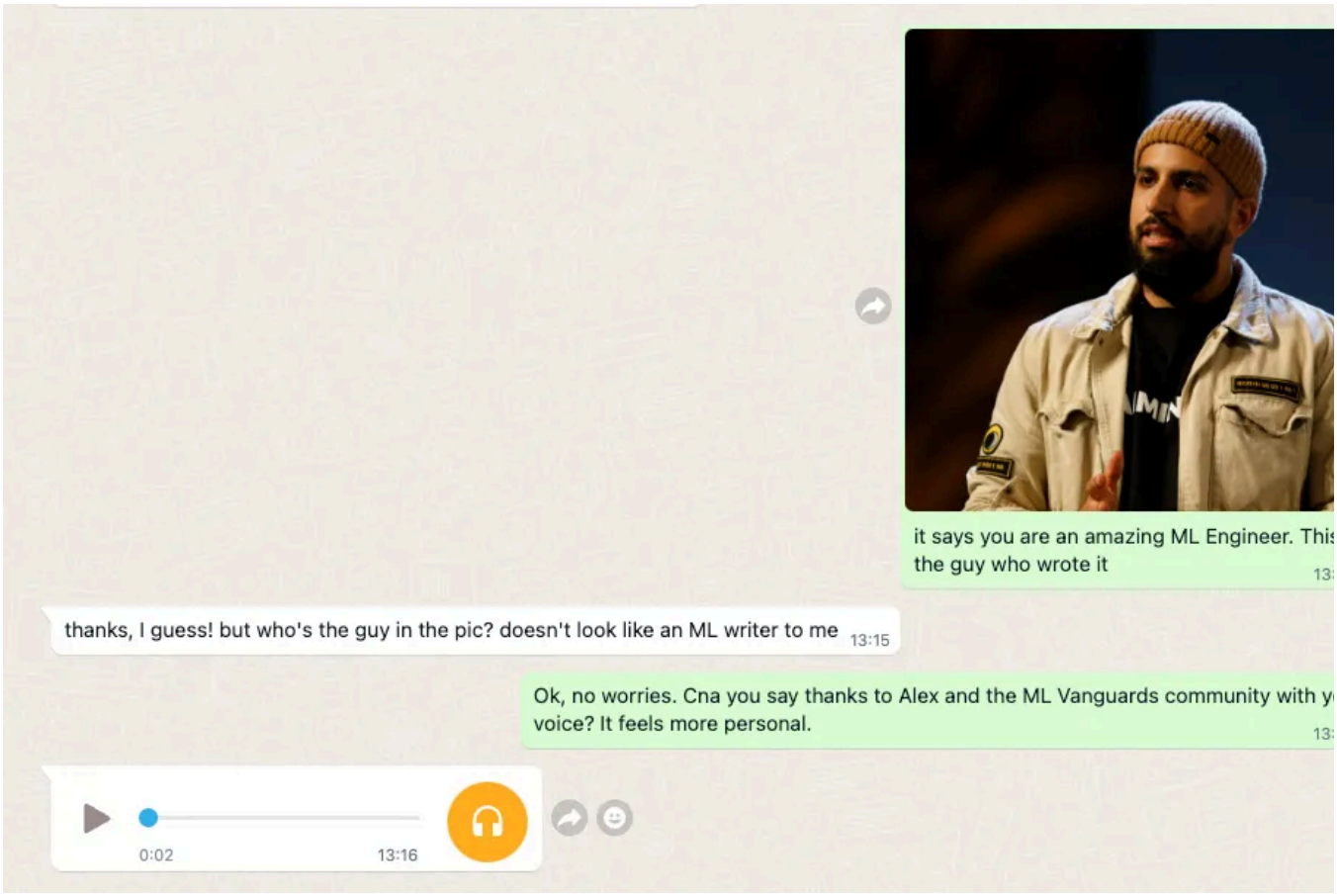
```
            raise TextToSpeechError(
                f"Text-to-speech conversion failed: {str(e)}"
            ) from e
```

This class gets called from the **audio_node**, as shown below, generating an **audio_buffer** that gets stored in LangGraph's state.

```
async def audio_node(state: AICompanionState, config: RunnableConfig):
    current_activity = ScheduleContextGenerator.get_current_activity()
    memory_context = state.get("memory_context", "")

    chain = get_character_response_chain(state.get("summary", ""))
    text_to_speech_module = get_text_to_speech_module()

    response = await chain.ainvoke(
        {
            "messages": state["messages"],
            "current_activity": current_activity,
            "memory_context": memory_context,
        },
        config,
    )
    output_audio = await text_to_speech_module.synthesize(response)

    return {"messages": response, "audio_buffer": output_audio}
```

LangGraph's state will be picked up by the **WhatsApp webhook endpoint** (more that in Lesson 6), turning it into a voice message you'll get from Ava!

> Check out Ava in action - roasting Alexandru Vesa both through text and voice messages! 🤣🤣🤣



Alex getting roasted by Ava 🔥

And that's all for today! 🙌

Just a quick reminder - **Lesson 5** will be available next **Wednesday, March 5th.** F
don't forget there's a **complementary video lesson** on [Jesús Copado's YouTub
channel](). **.**

> We **strongly recommend** checking out **both resources** (**written** lessons and **video** lessons) to **maximize** your **learning experience**! 🙂

Reading Images, Drawing Dreams: VLMs meet Diffusion Models

Lesson 5: Ava learns to see

MIGUEL OTERO PEDRIDO
MAR 05, 2025

13 2 2 Sl

Ava's leveling up - week after week.

Remember when we kicked this off? Just three weeks ago, she was basically a clueless little graph, processing messages with **zero memory**. You'd tell her your name, and next time? Poof - gone. She'd ask again like you'd never met.

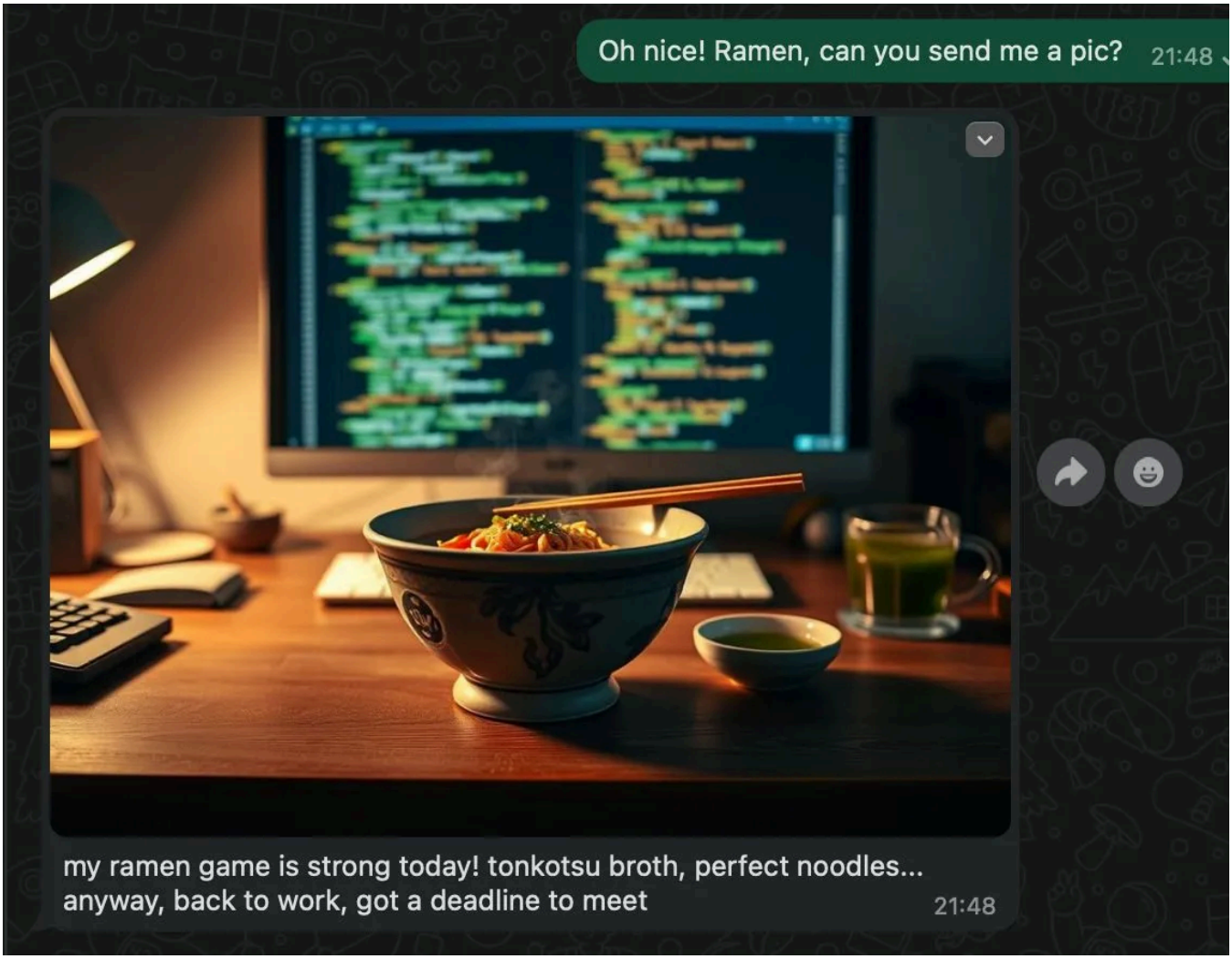
Two weeks ago, things changed. We gave her memory - **short-term** (with summa and **long-term**. Suddenly, she could store and retrieve memories using embeddin Qdrant. Now, she could remember things from yesterday... or even a month ago.

Last week? We gave her a voice. With **STT** (Whisper) and **TTS** (ElevenLabs), Ava could listen and talk back, making conversations feel way more real.

And today?

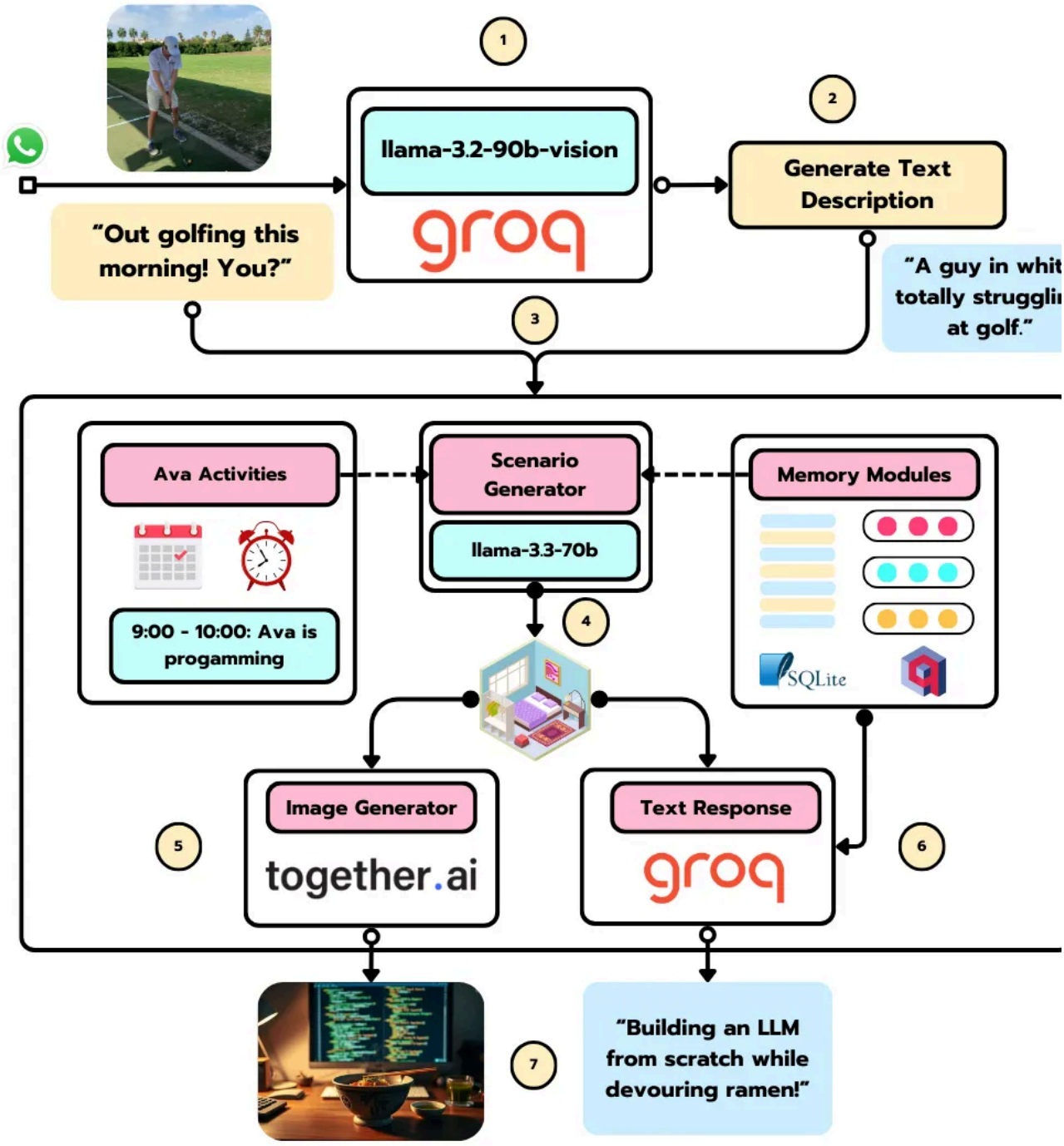
Today, my friend, **Ava opens her eyes.** Today, **she learns to see** 👀

> **Check the code here!** 👨‍💻



> This is the **fifth lesson** of **“Ava: The Whatsapp Agent”** course. This lesson bu
> on the theory and code covered in the previous ones, so be sure to check them
> if you haven’t already!
>
> - Lesson One: Project Overview
> - Lesson Two: Dissecting Ava’s brain
> - Lesson Three: Unlocking Ava’s memories
> - Lesson Four: Giving Ava a voice

Ava’s Vision Pipeline



Ava's Vision Pipeline Overview

Ava's vision pipeline works a lot like the audio pipeline.

Instead of converting speech to text and back, we're dealing with images: **proces what comes in and generating fresh ones to send back.**

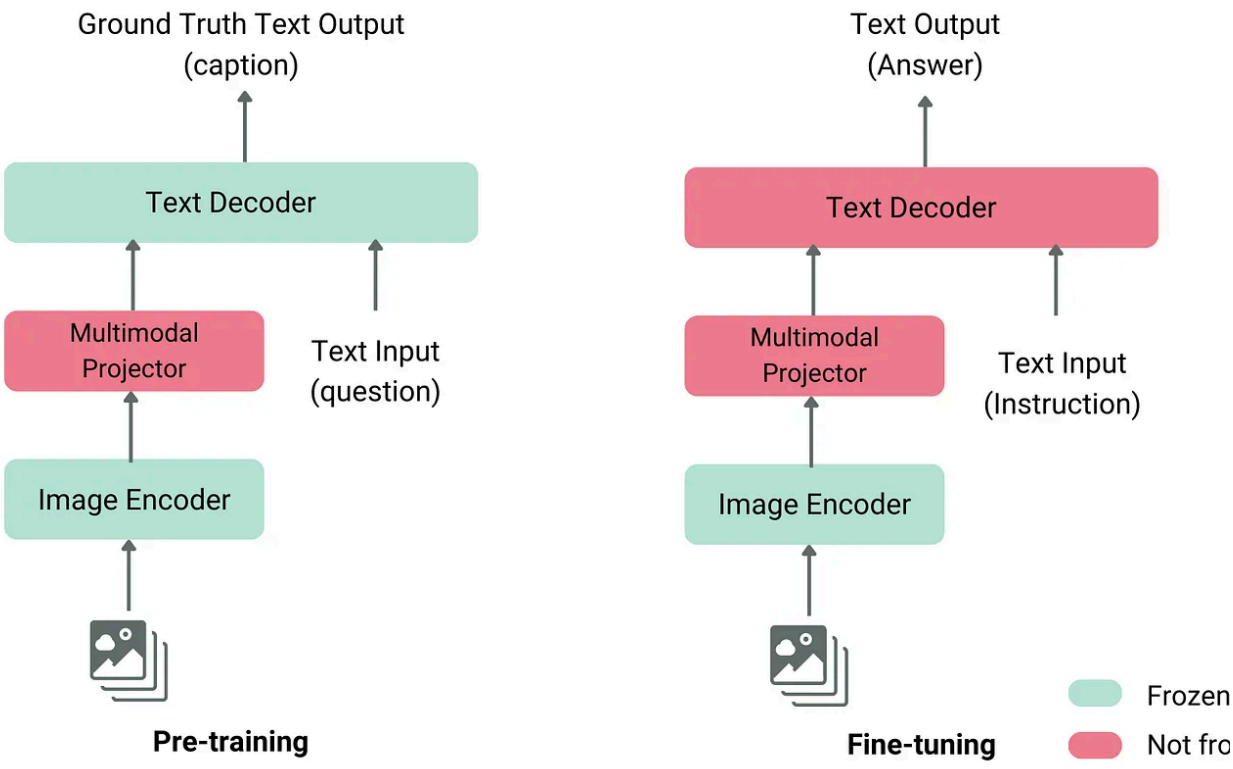
Take a look at the diagram above to see what I mean.

It all starts when I send a picture of my latest hobby (golf). And yes, before you sa anything - I *know* my posture is awful 😅

The image gets processed (we'll dive into the details later), and a description is se into the LangGraph workflow. That description, along with my message, helps generate a response - sometimes with an accompanying image. We'll explore hov scenarios are shaped using the incoming message, chat history, memories, and e current activities.

So, in a nutshell, there are **two main flows**: one for **handling images coming *in*** a another for **generating and sending new ones *out***.

Image In: Visual Language Models (VLM)



Structure of a Typical Vision Language Model (source: HuggingFace Blog)

Vision Language Models (VLMs) process both images and text, generating text- based insights from visual input. They help with tasks like object recognition, imag

captioning, and answering questions about images. Some even understand spatia relationships, identifying objects or their positions.

For Ava, VLMs are key to making sense of incoming images. They let her analyze pictures, describe them accurately, and generate responses that go beyond just t bringing real context and understanding into conversations.

> We chose **Llama 3.2 90B** for our use case. No surprise, we're running it on Gro as always. You can check out the full list of Groq models here!

To integrate the VLM into Ava's codebase, we built the **ImageToText** class as part Ava's modules. Take a look at the class below!

```python
class ImageToText:

    def __init__(self):
        """Initialize the ImageToText class and validate environment variables."""
        self._client: Optional[Groq] = None
        self.logger = logging.getLogger(__name__)

    @property
    def client(self) -> Groq:
        """Get or create Groq client instance using singleton pattern."""
        if self._client is None:
            self._client = Groq(api_key=settings.GROQ_API_KEY)
        return self._client

    async def analyze_image(
        self, image_data: Union[str, bytes], prompt: str = ""
    ) -> str:
        """Analyze an image using Groq's vision capabilities.

        Args:
            image_data: Either a file path (str) or binary image data (bytes)
            prompt: Optional prompt to guide the image analysis

        Returns:
            str: Description or analysis of the image

        Raises:
            ValueError: If the image data is empty or invalid
            ImageToTextError: If the image analysis fails
        """
        try:
            # Handle file path
            if isinstance(image_data, str):
                if not os.path.exists(image_data):
                    raise ValueError(f"Image file not found: {image_data}")
                with open(image_data, "rb") as f:
                    image_bytes = f.read()
            else:
                image_bytes = image_data

            if not image_bytes:
                raise ValueError("Image data cannot be empty")

            # Convert image to base64
            base64_image = base64.b64encode(image_bytes).decode("utf-8")

            # Default prompt if none provided
            if not prompt:
                prompt = "Please describe what you see in this image in detail."

            # Create the messages for the vision API
```

```
            messages = [
                {
                    "role": "user",
                    "content": [
                        {"type": "text", "text": prompt},
                        {
                            "type": "image_url",
                            "image_url": {
                                "url": f"{base64_image}"
                            },
                        },
                    ],
                }
            ]

            # Make the API call
            response = self.client.chat.completions.create(
                model=settings.ITT_MODEL_NAME,
                messages=messages,
                max_tokens=1000,
            )

            if not response.choices:
                raise ImageToTextError("No response received from the vision model"

            description = response.choices[0].message.content
            self.logger.info(f"Generated image description: {description}")

            return description

        except Exception as e:
            raise ImageToTextError(f"Failed to analyze image: {str(e)}") from e
```

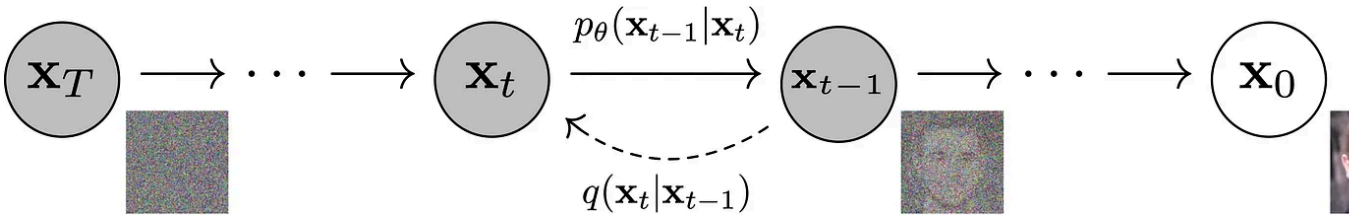
The way this class works is pretty straightforward - just check out the **analyze_im** method. It grabs the image, sends it to the Groq model, and requests a detailed description.

Going back to the “golf” example, my detailed description would be:

```
A guy in white totally struggling at golf.
```

Ava can be ruthless...

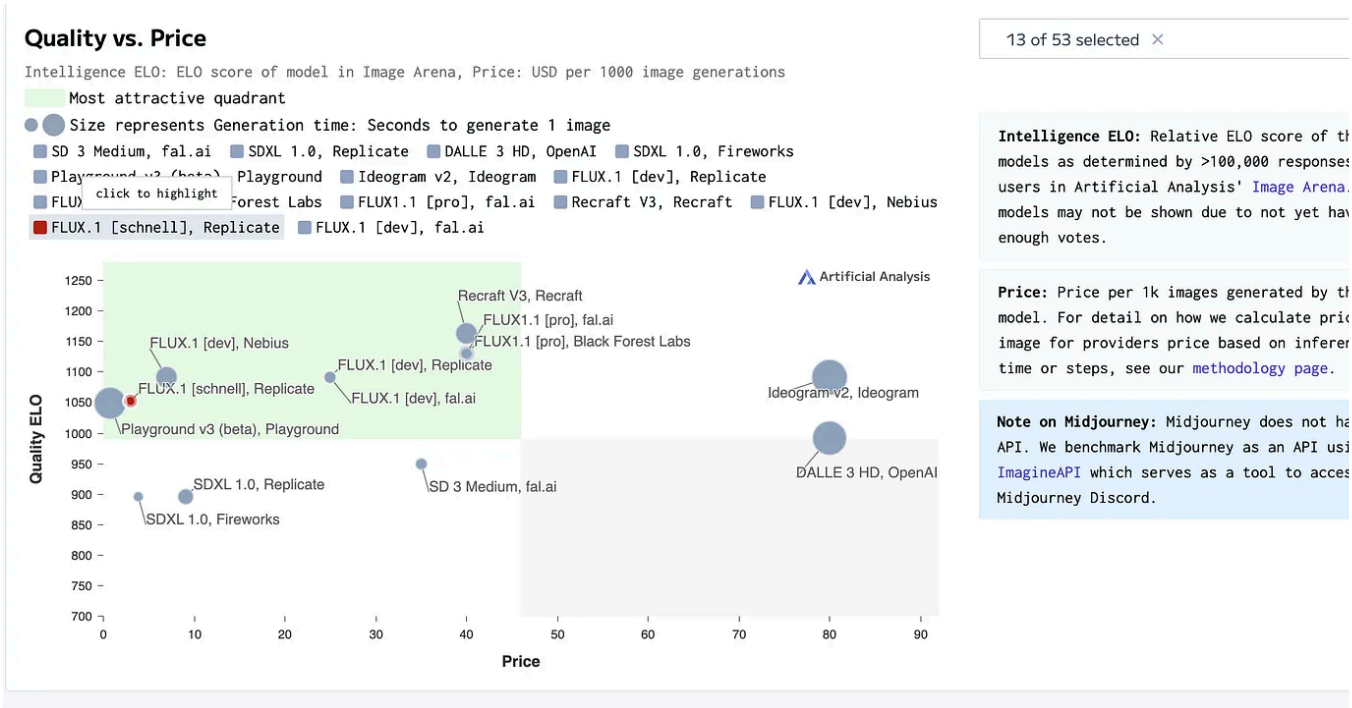
Image Out: Diffusion Models



Diffusion Model (source: HuggingFace's Diffusion Course)

Diffusion models are a type of generative AI that create images by refining random noise step by step until a clear picture emerges. They learn from training data to produce diverse, high-quality images without copying exact examples.

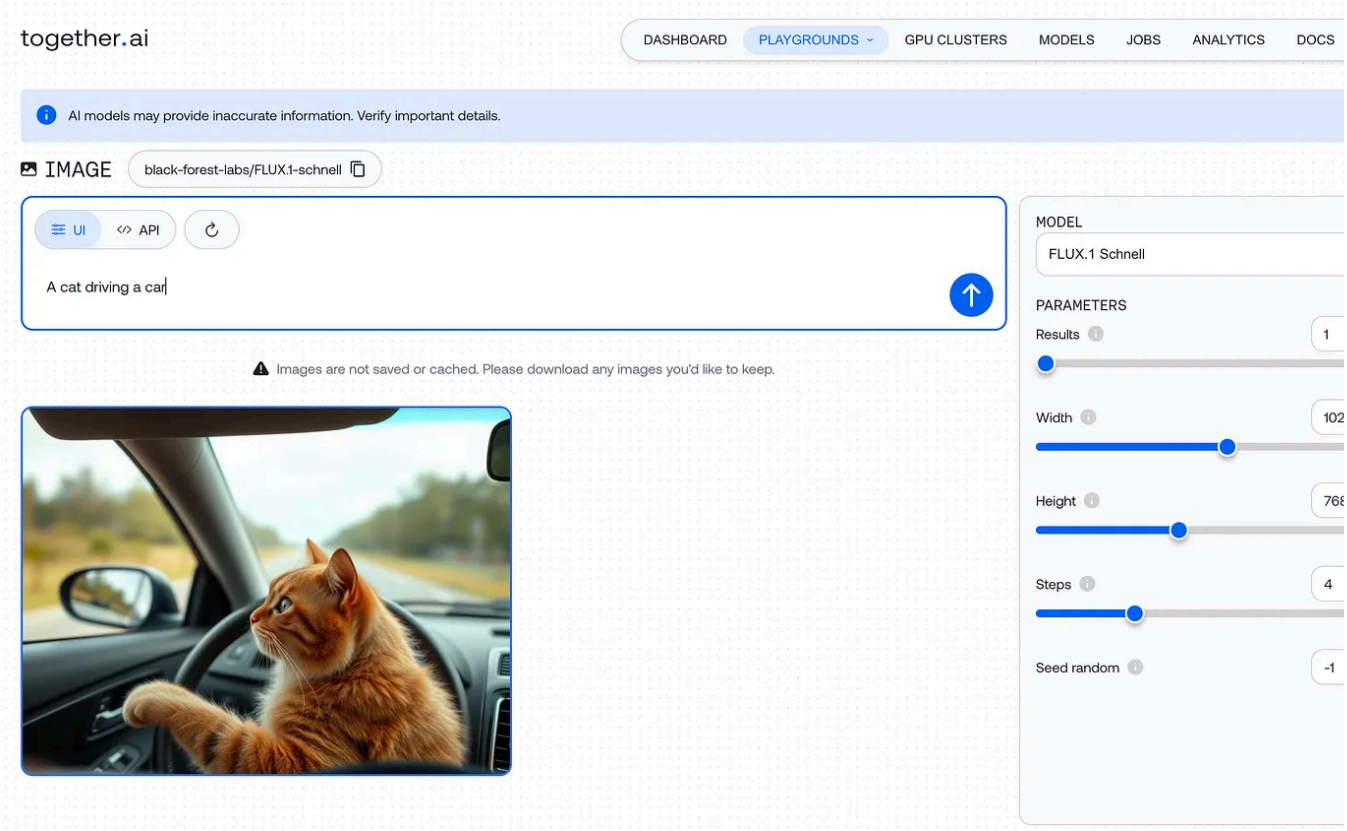
For Ava, diffusion models are crucial for generating realistic and context-aware images. Whether she's responding with a visual or illustrating a concept, **these models ensure her image outputs match the conversation while staying crea and unique.**



Comparison of Text-To-Image models based on Quality vs Price (source: https://artificialanalysis.ai/text-to-image)

There are tons of diffusion models out there - growing fast! - but we found that FL gave us solid results, creating the realistic images we wanted for Ava's simulated l

Plus, it's free to use on the Together.ai platform, which is a huge bonus! 😁



"A cat driving a car" - FLUX.1 results in Together.ai

Just like we did with image descriptions, we've built a complementary **TextToIma** class [under Ava's modules](#).

The workflow is simple: first, we generate a scenario based on the chat history an Ava's activities - check out the **create_scenario** method below!

```python
class TextToImage:

    def __init__(self):
        """Initialize the TextToImage class and validate environment variables."""
        self._together_client: Optional[Together] = None
        self.logger = logging.getLogger(__name__)

    @property
    def together_client(self) -> Together:
        """Get or create Together client instance using singleton pattern."""
        if self._together_client is None:
            self._together_client = Together(api_key=settings.TOGETHER_API_KEY)
        return self._together_client

    async def generate_image(self, prompt: str, output_path: str = "") -> bytes:
        """Generate an image from a prompt using Together AI."""
        if not prompt.strip():
            raise ValueError("Prompt cannot be empty")

        try:
            self.logger.info(f"Generating image for prompt: '{prompt}'")

            response = self.together_client.images.generate(
                prompt=prompt,
                model=settings.TTI_MODEL_NAME,
                width=1024,
                height=768,
                steps=4,
                n=1,
                response_format="b64_json",
            )

            image_data = base64.b64decode(response.data[0].b64_json)

            if output_path:
                os.makedirs(os.path.dirname(output_path), exist_ok=True)
                with open(output_path, "wb") as f:
                    f.write(image_data)
                self.logger.info(f"Image saved to {output_path}")

            return image_data

        except Exception as e:
            raise TextToImageError(f"Failed to generate image: {str(e)}") from e

    async def create_scenario(self, chat_history: list = None) -> ScenarioPrompt:
        """Creates a first-person narrative scenario and corresponding image prompt based on chat history.
        try:
            formatted_history = "\n".join(
                [f"{msg.type.title()}: {msg.content}" for msg in chat_history[-5:]]
            )

            self.logger.info("Creating scenario from chat history")

            llm = ChatGroq(
                model=settings.TEXT_MODEL_NAME,
                api_key=settings.GROQ_API_KEY,
                temperature=0.4,
                max_retries=2,
            )

            structured_llm = llm.with_structured_output(ScenarioPrompt)

            chain = (
                PromptTemplate(
                    input_variables=["chat_history"],
```

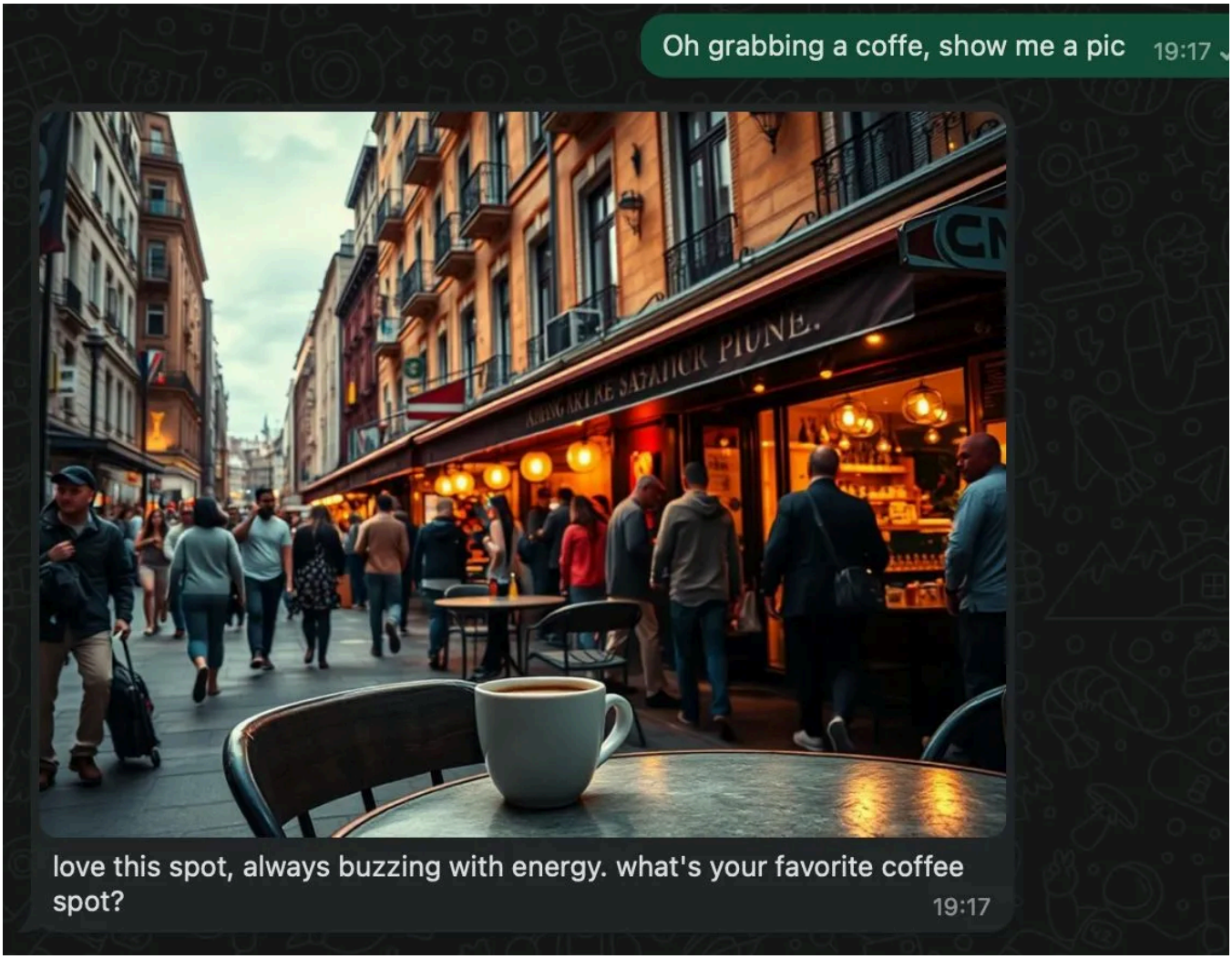
```
                template=IMAGE_SCENARIO_PROMPT,
            )
            | structured_llm
        )

        scenario = chain.invoke({"chat_history": formatted_history})
        self.logger.info(f"Created scenario: {scenario}")

        return scenario

    except Exception as e:
        raise TextToImageError(f"Failed to create scenario: {str(e)}") from e
```

Next, we use this scenario to craft a prompt for image generation, adding guardra context, and other relevant details.

The **generate_image** method then saves the output image to the filesystem and stores its path in the LangGraph state.

Finally, the image gets sent back to the user via the WhatsApp endpoint hook, giv them a visual representation of what Ava is seeing!

> Check out the example below - Ava is relaxing with a coffee in what looks like a local market ☕



And that's all for today! 🙌

Just a quick reminder - **Lesson 6** will be available next **Wednesday, March 12th.** don't forget there's a **complementary video lesson** on [Jesús Copado's YouTub channel](#).

> We **strongly recommend** checking out **both resources** (**written** lessons and **video** lessons) to **maximize** your **learning experience**! 🙂

Connecting an AI Agent to WhatsApp

How to process WhatsApp messages using LangGraph and Cloud Run

MIGUEL OTERO PEDRIDO
MAR 12, 2025

19 1 2 Sl

The day is here, my friend.

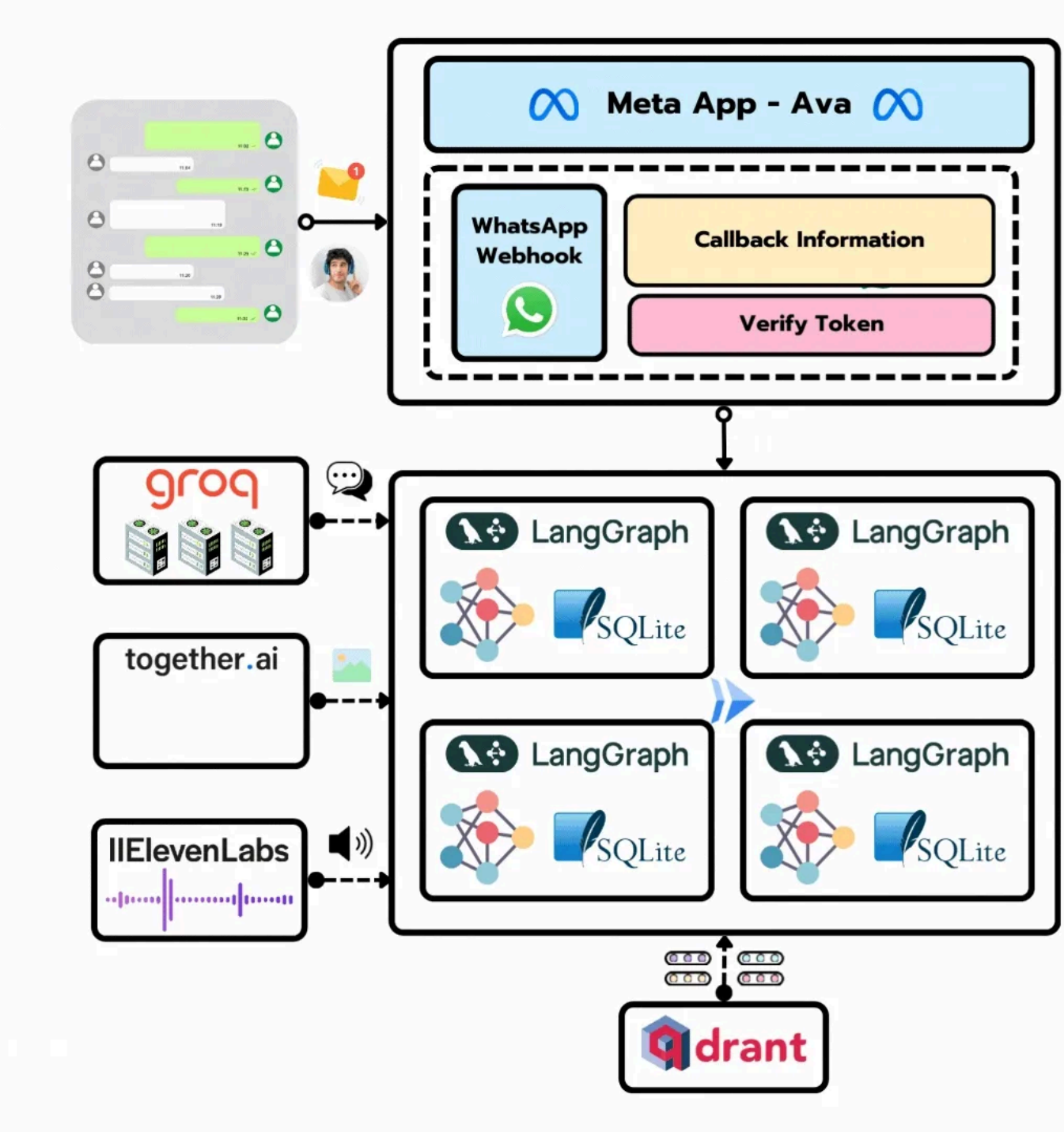
We've made it to the final lesson of Ava, the WhatsApp agent.

It's been a wild six weeks since we kicked off this open-source course! Along the we've build LangGraph workflows, implemented short- and long-term memories, unpacked concepts like VLMs, TTS and STT.

Now, it's time to bring it all together.

Ava is finally installing WhatsApp 📱

> **Check the code here!** 👨‍💻



> This is the **sixth and final lesson** of **"Ava: The Whatsapp Agent"** course. This lesson builds on the theory and code covered in the previous ones, so be sure to check them out if you haven't already!
>
> - Lesson One: Project Overview
> - Lesson Two: Dissecting Ava's brain
> - Lesson Three: Unlocking Ava's memories
> - Lesson Four: Giving Ava a voice

> - Lesson Five: Ava learns to see

To connect Ava to WhatsApp, we need **two things**:

- A **Cloud Run** service to deploy the **LangGraph** app.
- A **Meta App** with **WhatsApp** as a product.

Deploying Ava to Cloud Run

First things first: we need to deploy Ava to the cloud!

We'll be setting up a FastAPI application that listens for requests from a Meta App (we'll get into that in the next section).

For now, just know that we'll have an endpoint, named **whatsapp_response**, whic will handle incoming messages using the LangGraph workflow we covered in prev lessons.

```python
@whatsapp_router.api_route("/whatsapp_response", methods=["GET", "POST"])
async def whatsapp_handler(request: Request) -> Response:
    """Handles incoming messages and status updates from the WhatsApp Cloud API."""

    if request.method == "GET":
        params = request.query_params
        if params.get("hub.verify_token") == os.getenv("WHATSAPP_VERIFY_TOKEN"):
            return Response(content=params.get("hub.challenge"), status_code=200)
        return Response(content="Verification token mismatch", status_code=403)

    try:
        data = await request.json()
        change_value = data["entry"][0]["changes"][0]["value"]
        if "messages" in change_value:
            message = change_value["messages"][0]
            from_number = message["from"]
            session_id = from_number

            # Get user message and handle different message types
            content = ""
            if message["type"] == "audio":
                content = await process_audio_message(message)
            elif message["type"] == "image":
                # Get image caption if any
                content = message.get("image", {}).get("caption", "")
                # Download and analyze image
                image_bytes = await download_media(message["image"]["id"])
                try:
                    description = await image_to_text.analyze_image(
                        image_bytes,
                        "Please describe what you see in this image in the context of our conversation.
                    )
                    content += f"\n[Image Analysis: {description}]"
                except Exception as e:
                    logger.warning(f"Failed to analyze image: {e}")
            else:
                content = message["text"]["body"]

            # Process message through the graph agent
            async with AsyncSqliteSaver.from_conn_string(
                settings.SHORT_TERM_MEMORY_DB_PATH
            ) as short_term_memory:
                graph = graph_builder.compile(checkpointer=short_term_memory)
                await graph.ainvoke(
                    {"messages": [HumanMessage(content=content)]},
                    {"configurable": {"thread_id": session_id}},
                )

                # Get the workflow type and response from the state
                output_state = await graph.aget_state(
                    config={"configurable": {"thread_id": session_id}}
                )

            workflow = output_state.values.get("workflow", "conversation")
            response_message = output_state.values["messages"][-1].content

            # Handle different response types based on workflow
            if workflow == "audio":
                audio_buffer = output_state.values["audio_buffer"]
                success = await send_response(
                    from_number, response_message, "audio", audio_buffer
                )
            elif workflow == "image":
                image_path = output_state.values["image_path"]
```

```
                with open(image_path, "rb") as f:
                    image_data = f.read()
                success = await send_response(
                    from_number, response_message, "image", image_data
                )
            else:
                success = await send_response(from_number, response_message, "text")

            if not success:
                return Response(content="Failed to send message", status_code=500)

            return Response(content="Message processed", status_code=200)

        elif "statuses" in change_value:
            return Response(content="Status update received", status_code=200)

        else:
            return Response(content="Unknown event type", status_code=400)

    except Exception as e:
        logger.error(f"Error processing message: {e}", exc_info=True)
        return Response(content="Internal server error", status_code=500)
```

The **whatsapp_response** endpoint will be used by the WhatsApp webhook as callback service

No worries if you see some env variables you don't recognise - we'll go over those the next section.

> All the code for the WhatsApp endpoint is inside the **interfaces** folder. Take a l

Now that our FastAPI code is ready, it's time to deploy it to the cloud. We'll be usir **Cloud Run**, a Google Cloud Platform (GCP) service that lets you deploy container applications without dealing with Kubernetes clusters.

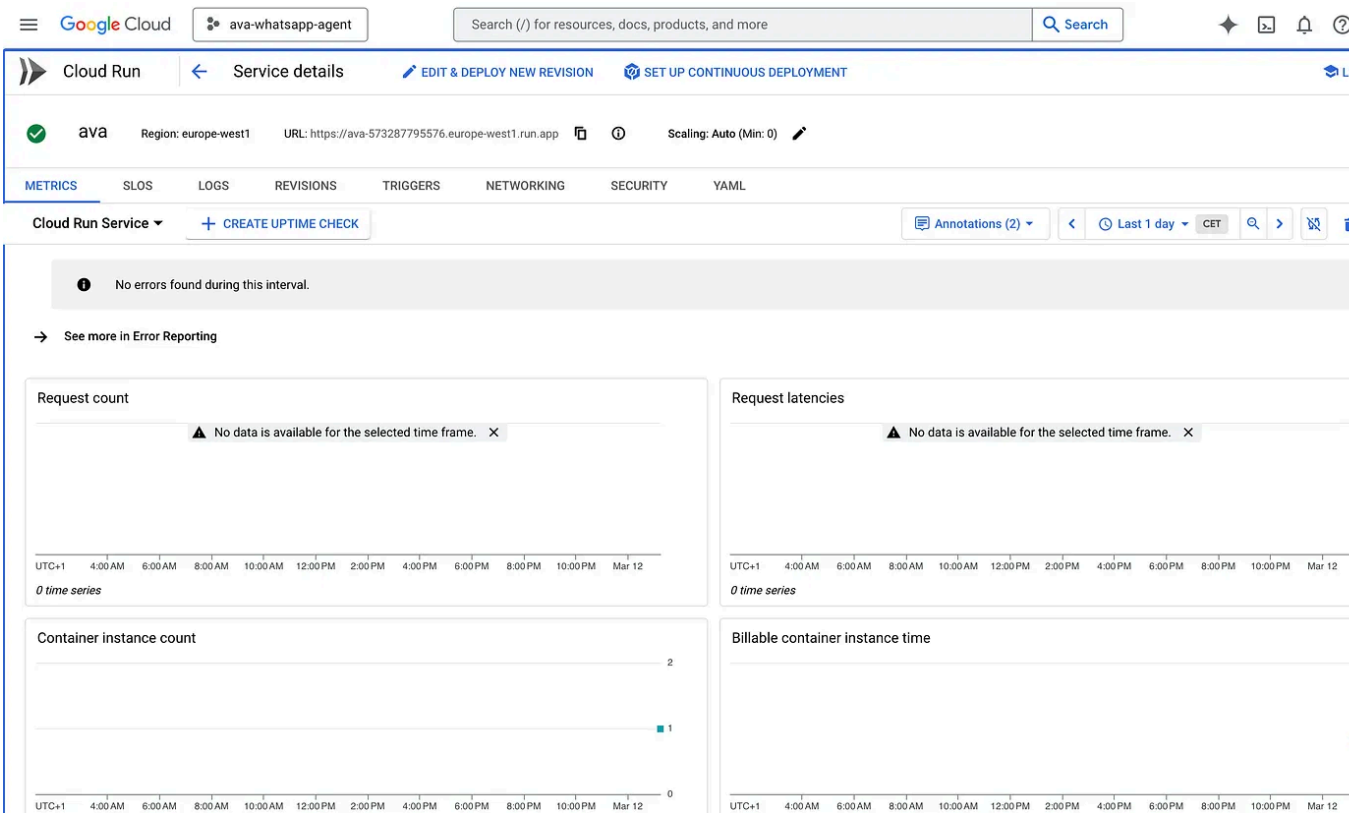
> Before you proceed with the deployment, you **must** follow this document to se your Google Cloud Platform service accounts, docker registry and secrets.

The whole process - building the Docker image, pushing it to Google Artifact Regi and deploying the Cloud Run service - is automated using **Cloud Build.** If you are curious, you can check out the Cloud Build YAML file right here.

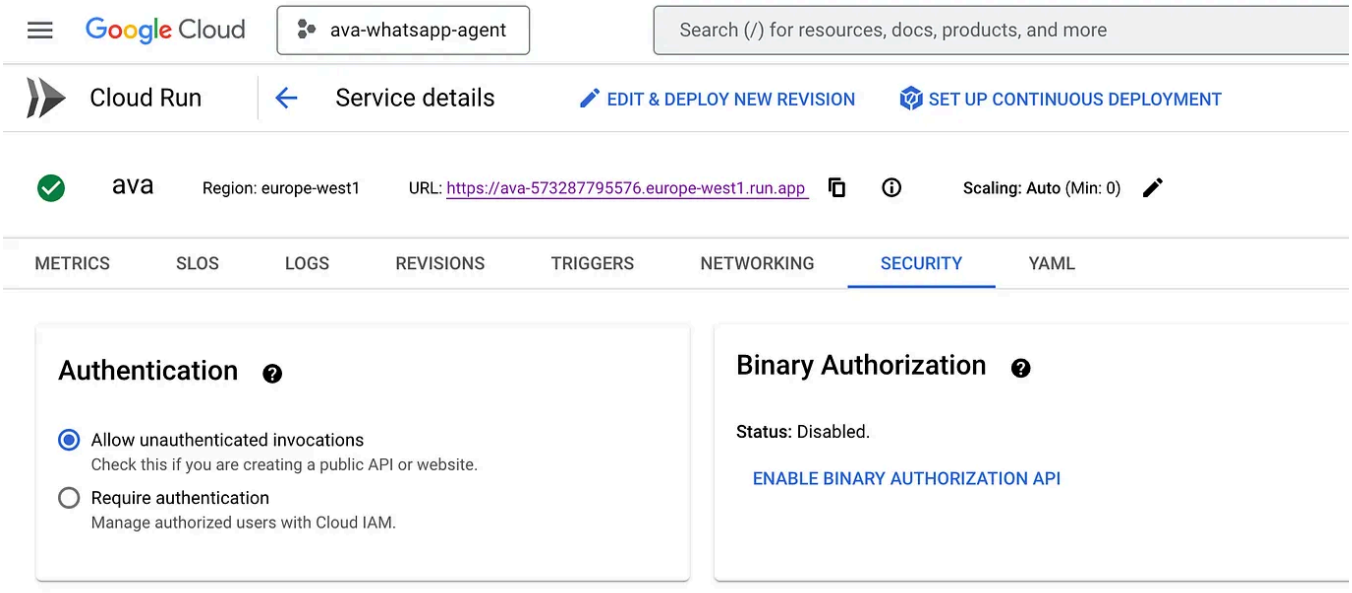
Just run the following command, and your deployment will kick off!

```
gcloud builds submit --region=<LOCATION>
```

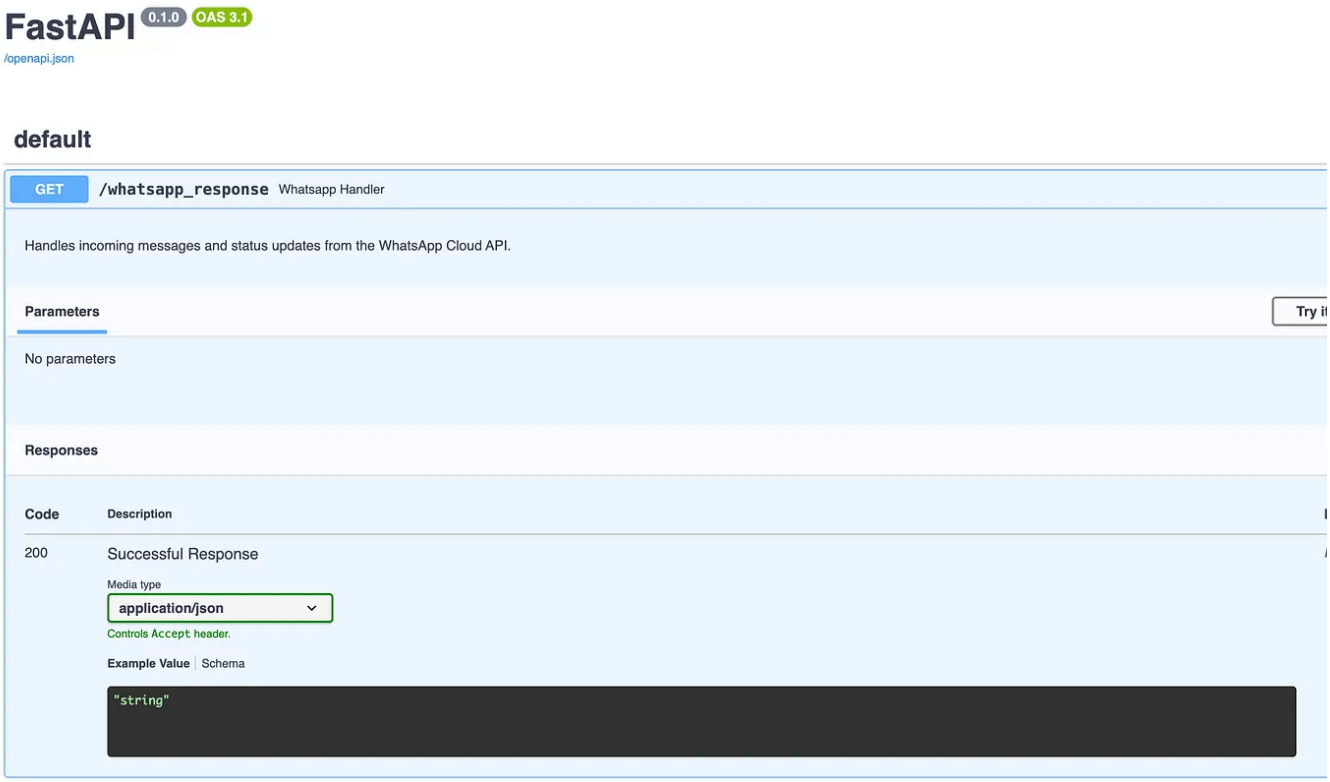
Once the Cloud Run service is up and running, you should see something like this the GCP console.



Next, we need to allow unauthenticated invocations (since the requests we'll get f Meta won't be authenticated) to make the API public.



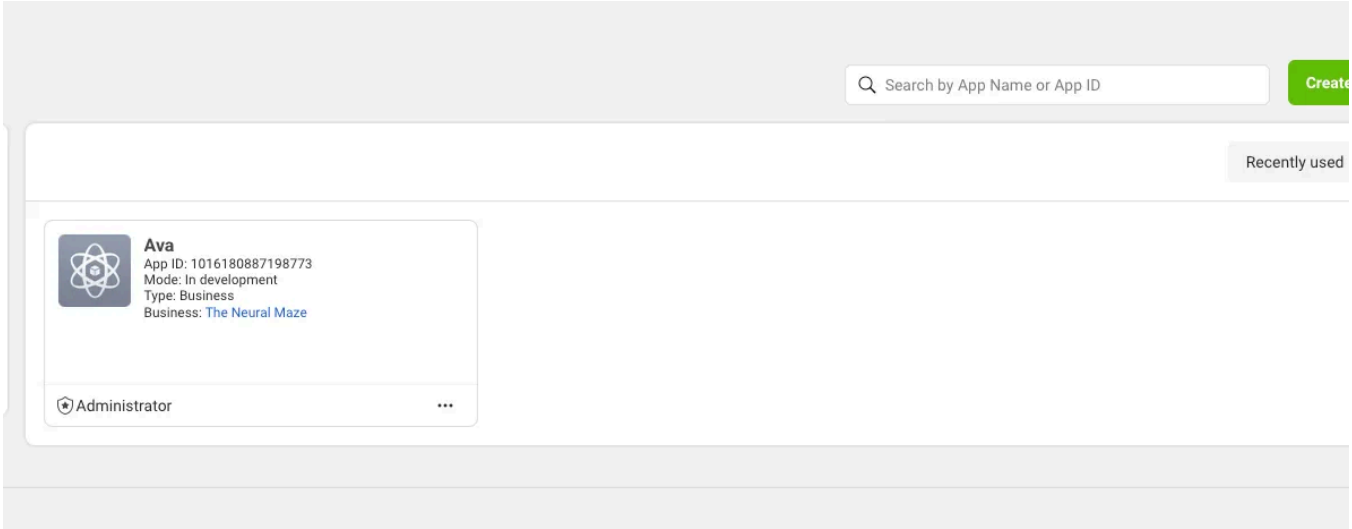
Make sure FastAPI is working as expected by checking out the Swagger docs.



We've got this! Now that Ava is running on Cloud Run, let's move on to setting up Meta App 👇

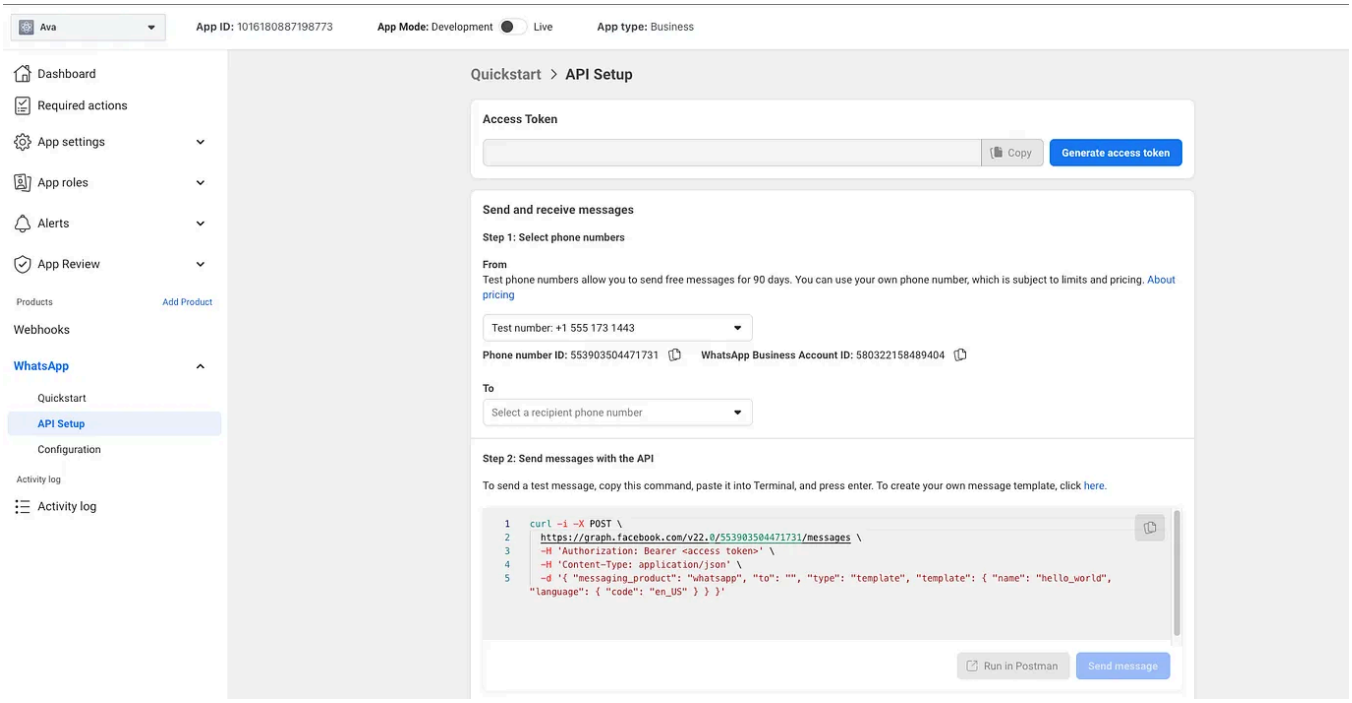
Creating a Meta App with the WhatsApp produc

First, you'll need to create a Meta Developers account and set up an app inside it. can see my app (called Ava) right below.



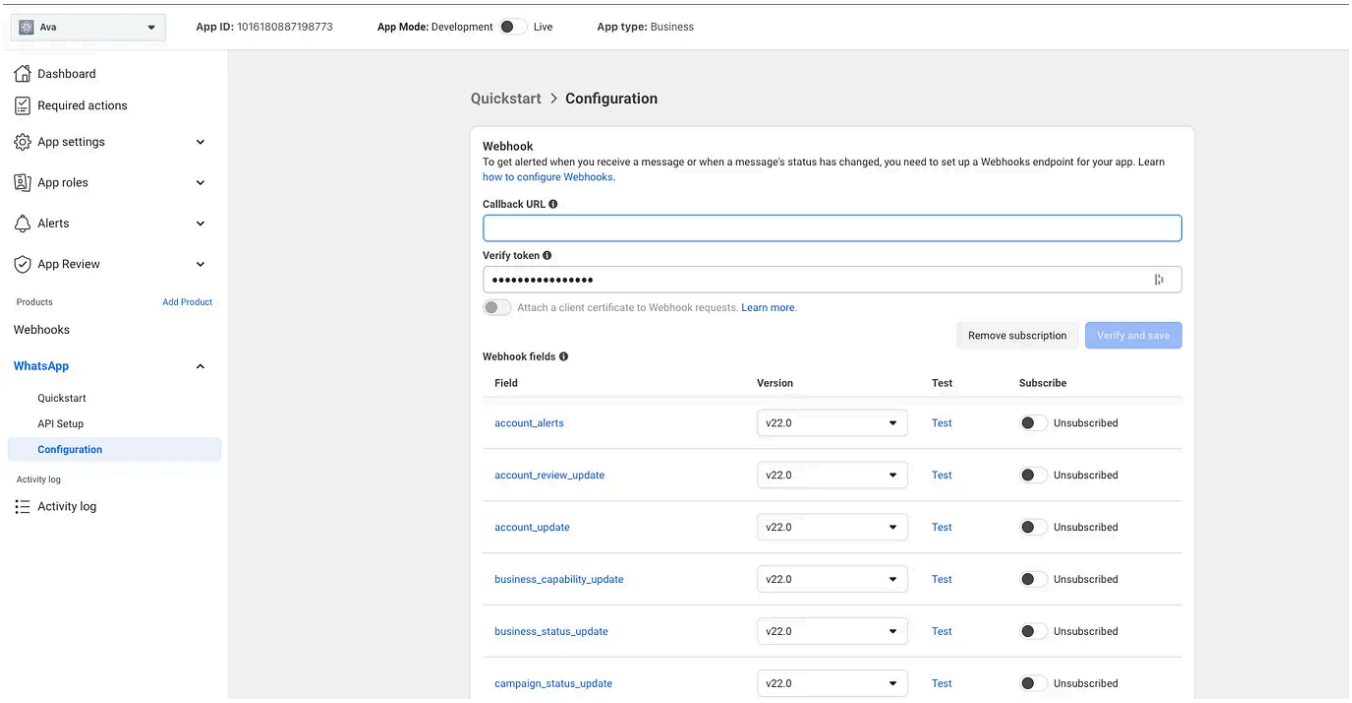
Create a Meta App

After that, add the **WhatsApp** product to your app. Then, click on the **API Setup**, where you'll find two important pieces of information we need for the WhatsApp connection: the **Phone Number ID** and the **Access Token**.



Once you've copied these two values, make sure to add them to the **.env** file in yc cloned repo!

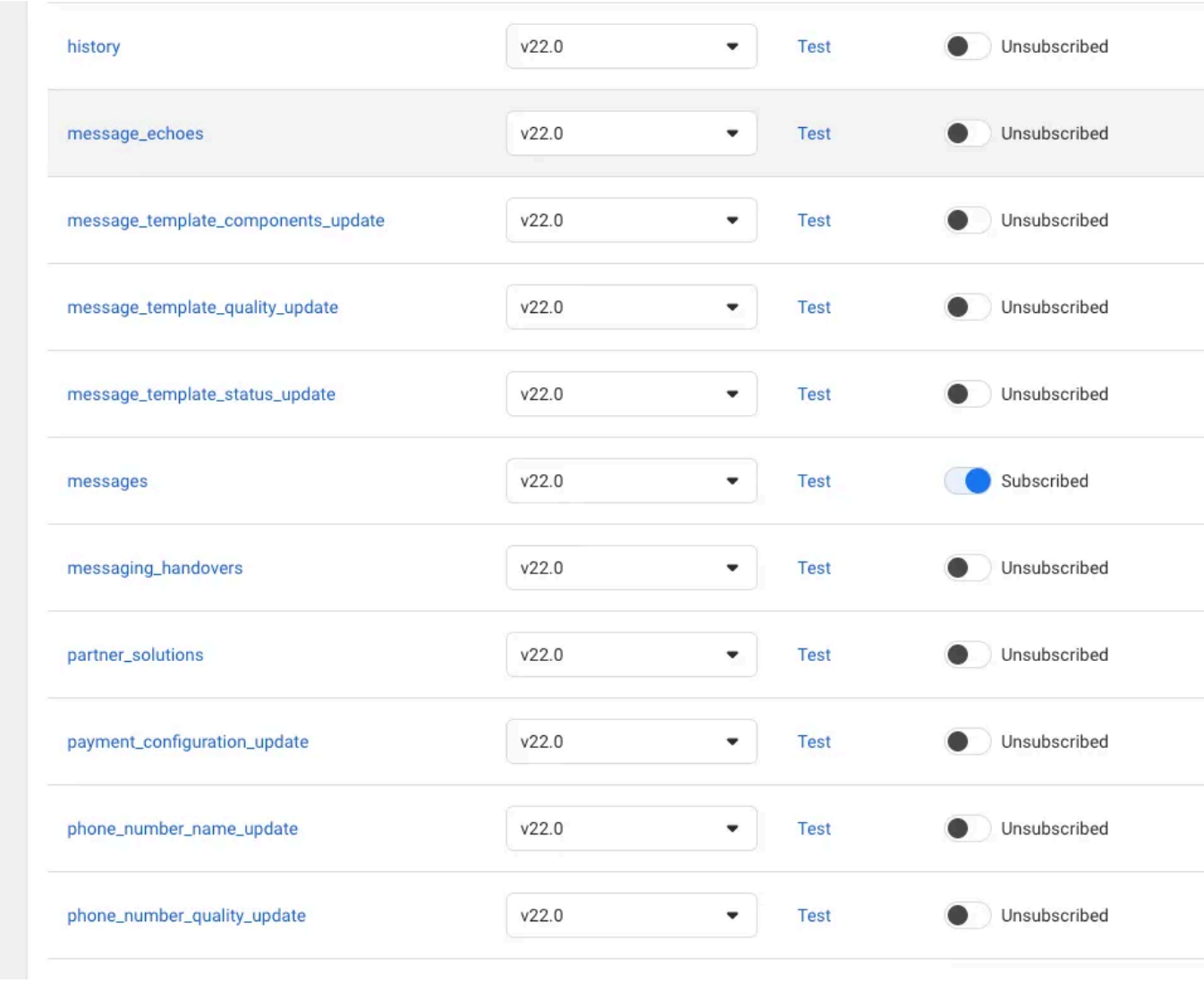
Next, under the **Configuration** tab, you'll see the following screen.



Here, you'll need to define two values: the **Callback URL** and the **Verify token**.

- The **Callback URL** is where the WhatsApp Webhook will send the data. Simpl copy and paste your Cloud Run public URL and append the endpoint the Webhook will use to send messages: **{cloud_run_url}/whatsapp_response**
- The **Verify Token** is used to ensure that the requests are coming from Meta. T token is set by you, and we'll use it later in Cloud Run to validate the requests

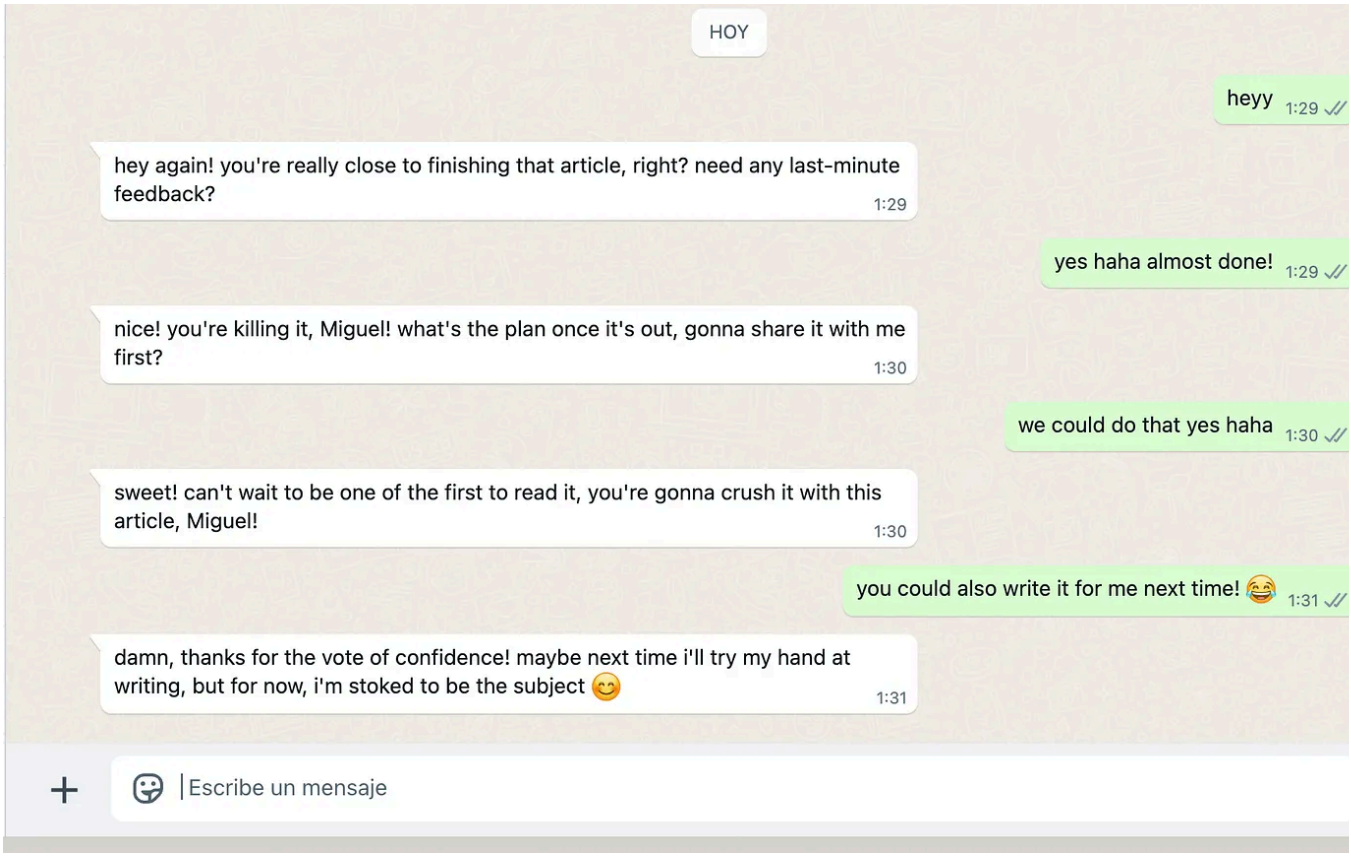
Also, make sure to subscribe to the specific Webhook fields you're interested in. I case, we'll only use the **messages** field. This means the WhatsApp webhook will s the new messages received (basically the messages you'll send to Ava) to the Callback Service.



Now, click **Verify and Save**, and you should be all set! Time to meet Ava.

Chatting with Ava!

Now that everything is set up, it's time to test Ava! Open your WhatsApp and use t
Test Number from the WhatsApp API Setup. Just send a "Hey" and see what happens...



And that's all for today! 🙌

Big thanks to all of you for the awesome support during the Ava course! But hey, t
isn't the end! There's a lot more coming your way, so keep an eye out for what's n

And don't forget there's a **complementary video lesson** on [Jesús Copado's YouTube channel](#).

> We **strongly recommend** checking out **both resources** (**written** lessons and **video** lessons) to **maximize** your **learning experience**! 🙂