Reading Images, Drawing Dreams: VLMs meet Diffusion Models

Lesson 5: Ava learns to see

MIGUEL OTERO PEDRIDO
MAR 05, 2025

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.

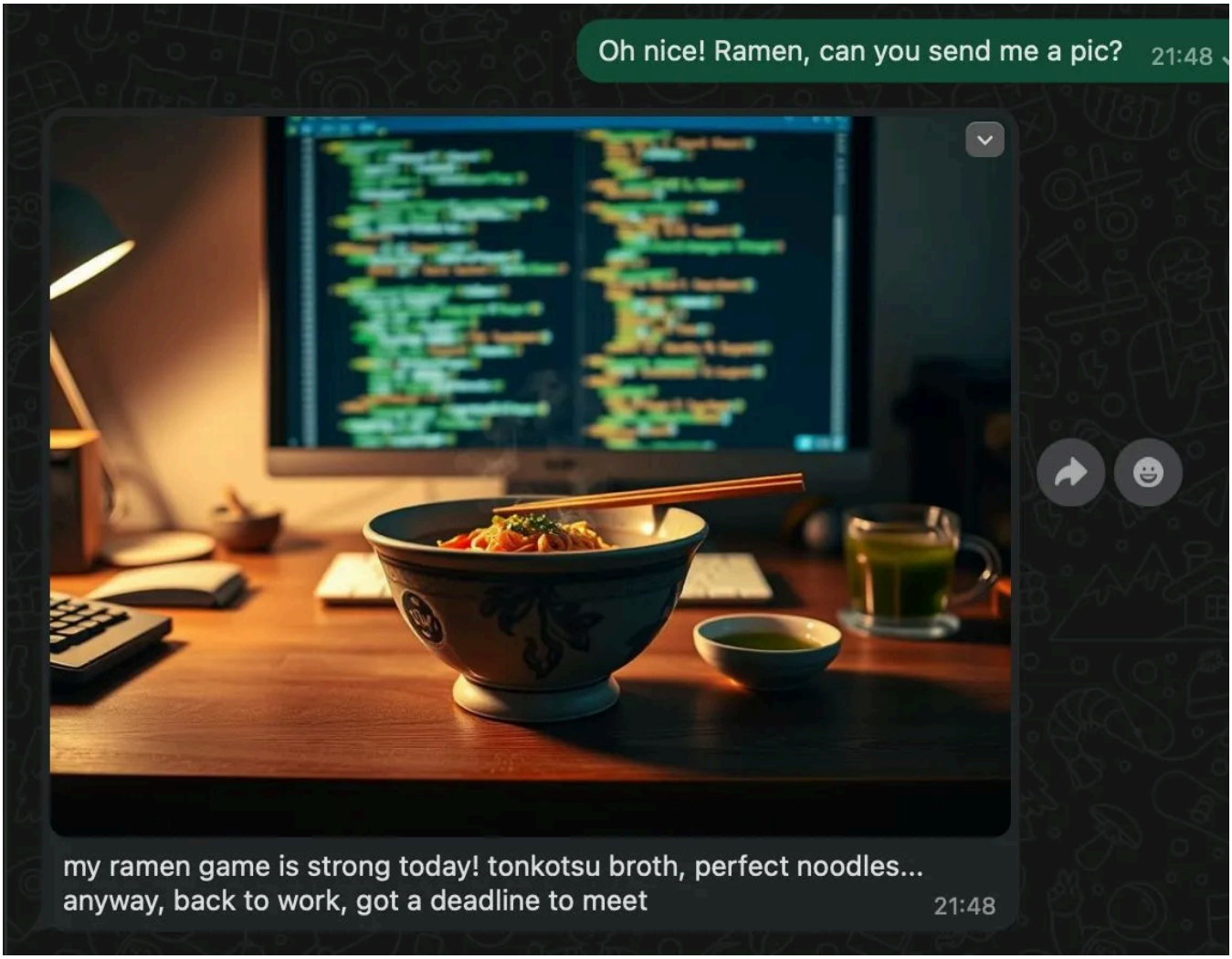
Two weeks ago, things changed. We gave her memory - **short-term** (with summa and **long-term**. Suddenly, she could store and retrieve memories using embeddin Qdrant. Now, she could remember things from yesterday... or even a month ago.

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?

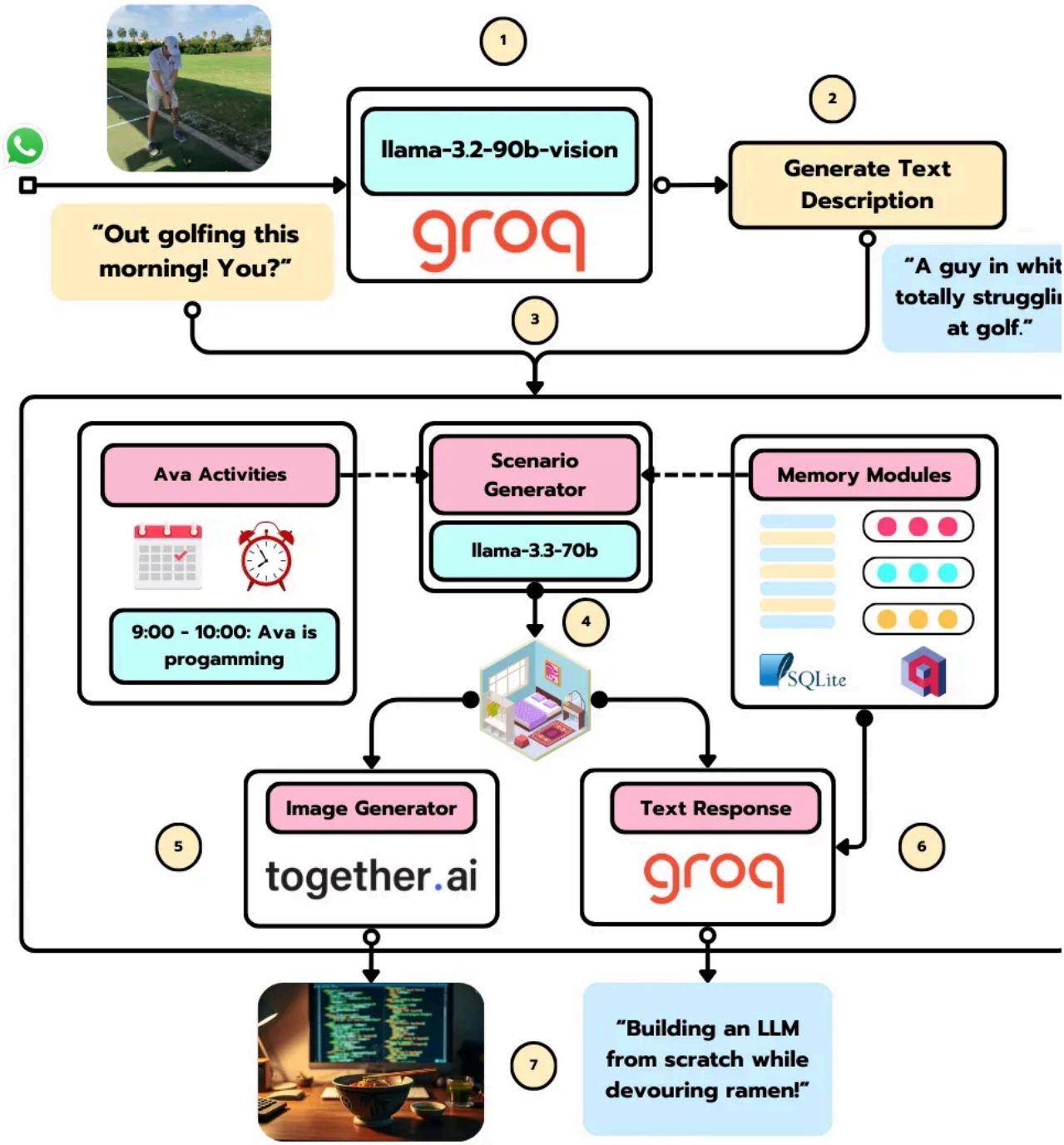
Today, my friend, **Ava opens her eyes.** Today, **she learns to see** 👀

> **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.**

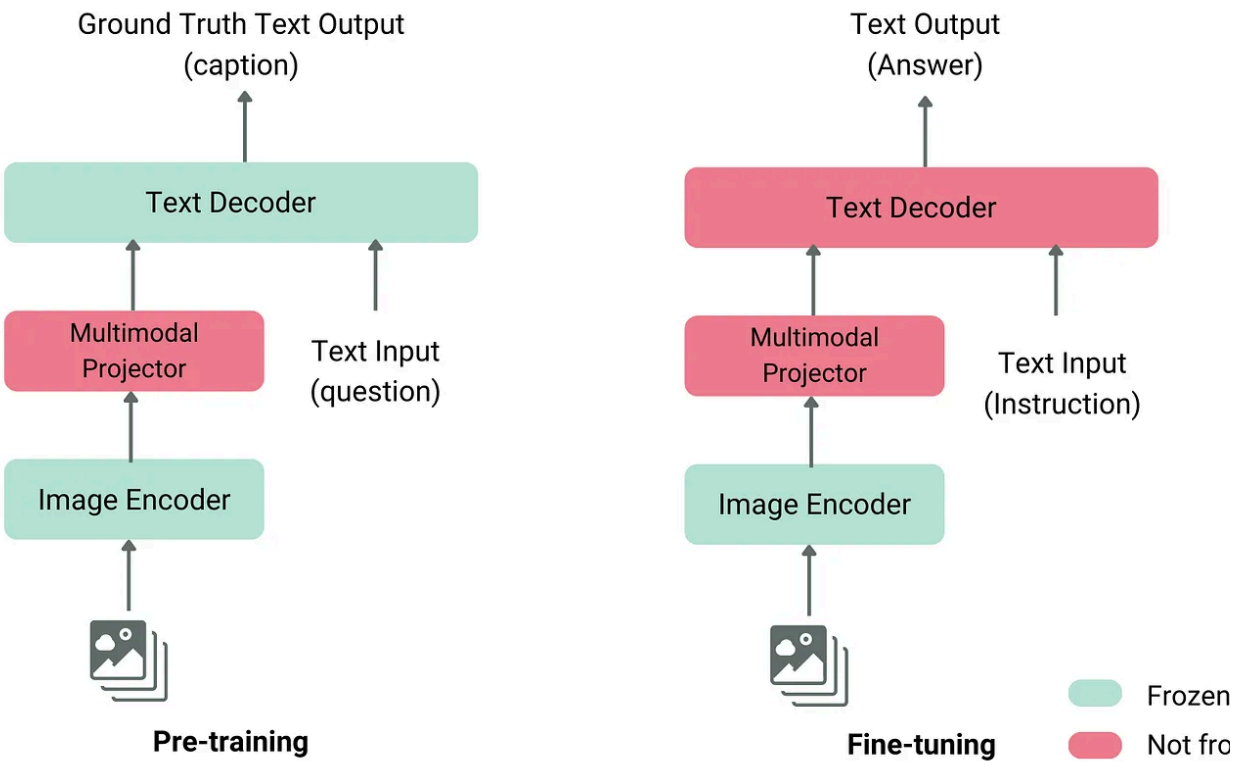
Take a look at the diagram above to see what I mean.

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
```

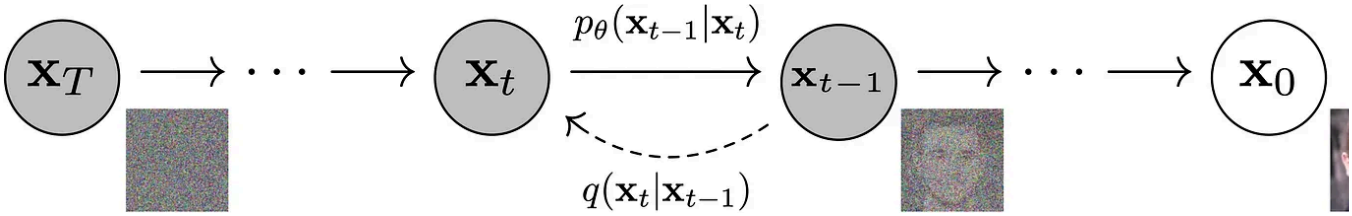
The way this class works is pretty straightforward - just check out the **analyze_im** method. It grabs the image, sends it to the Groq model, and requests a detailed description.

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

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

Ava can be ruthless...

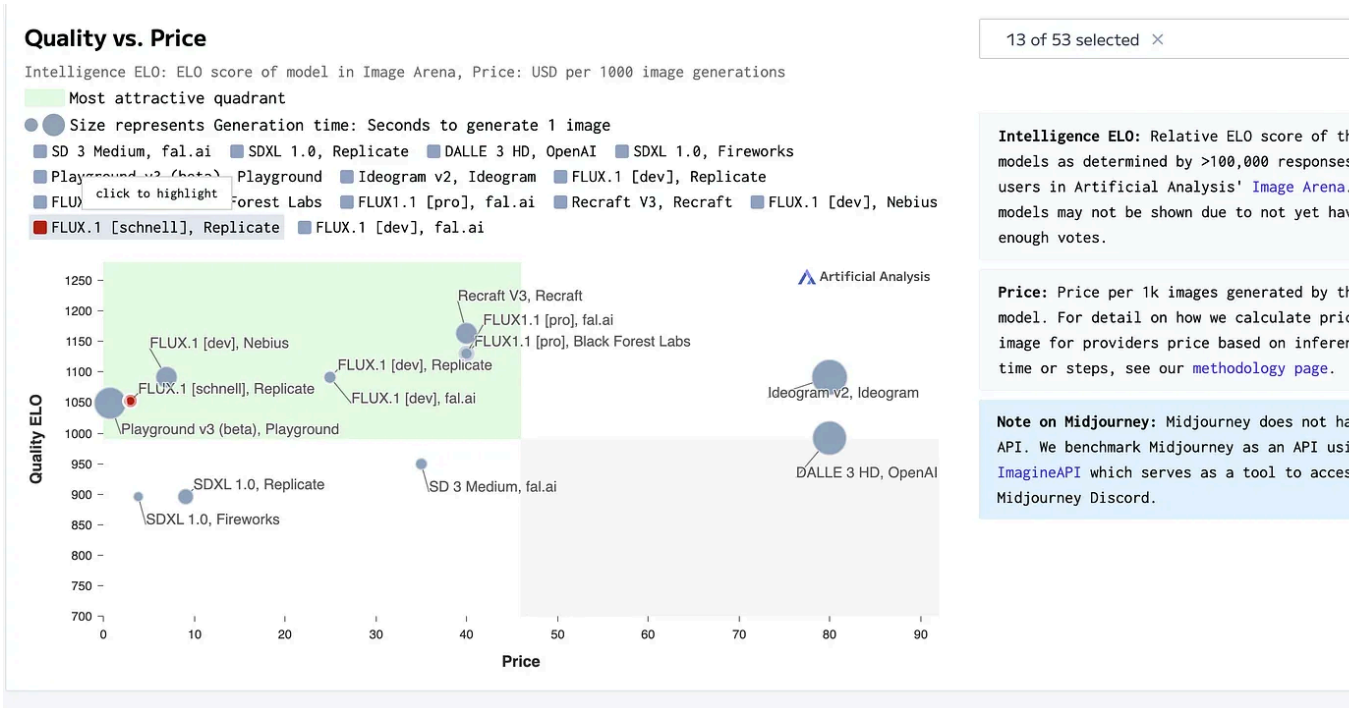
Image Out: Diffusion Models



Diffusion Model (source: HuggingFace's Diffusion Course)

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

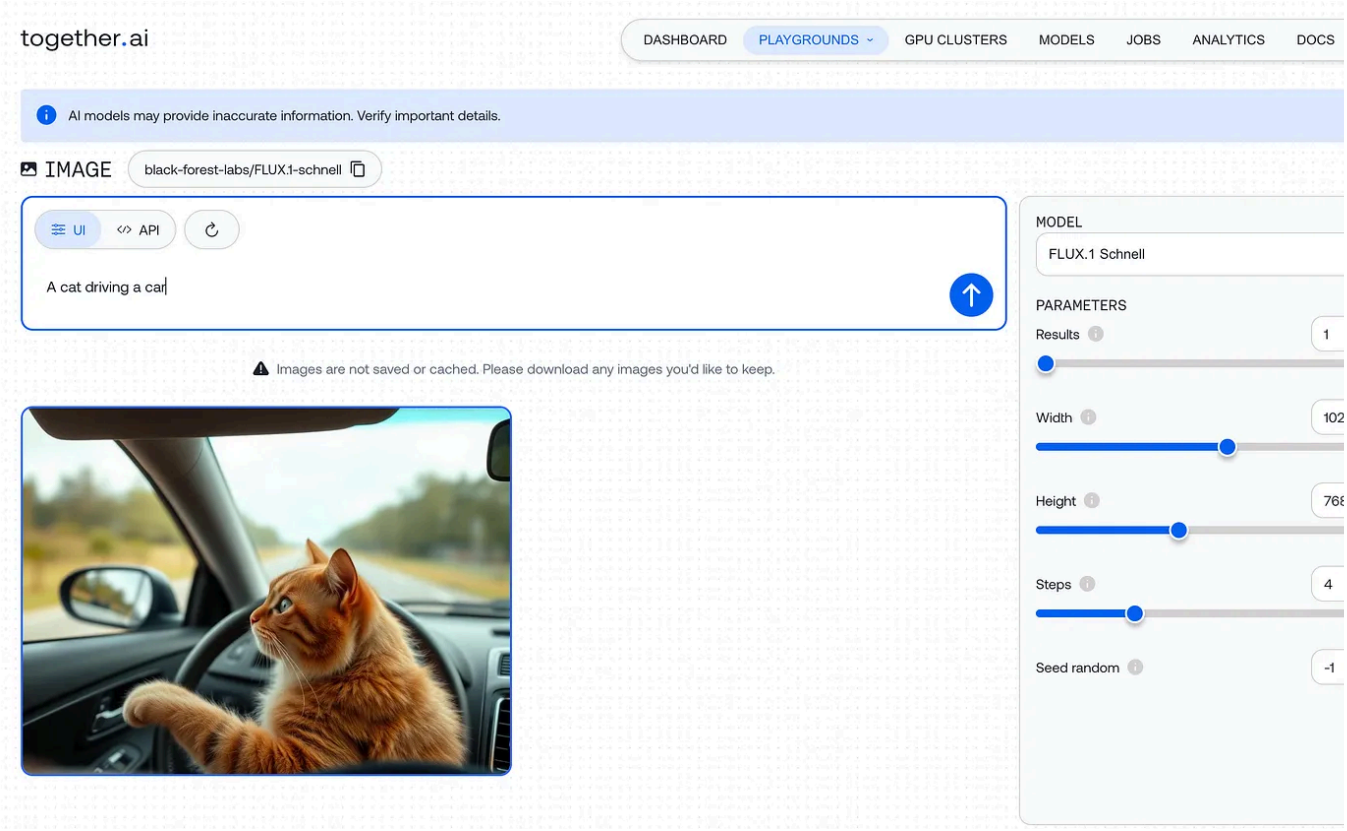
For Ava, diffusion models are crucial for generating realistic and context-aware images. Whether she's responding with a visual or illustrating a concept, **these models ensure her image outputs match the conversation while staying crea and unique.**



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"],
```

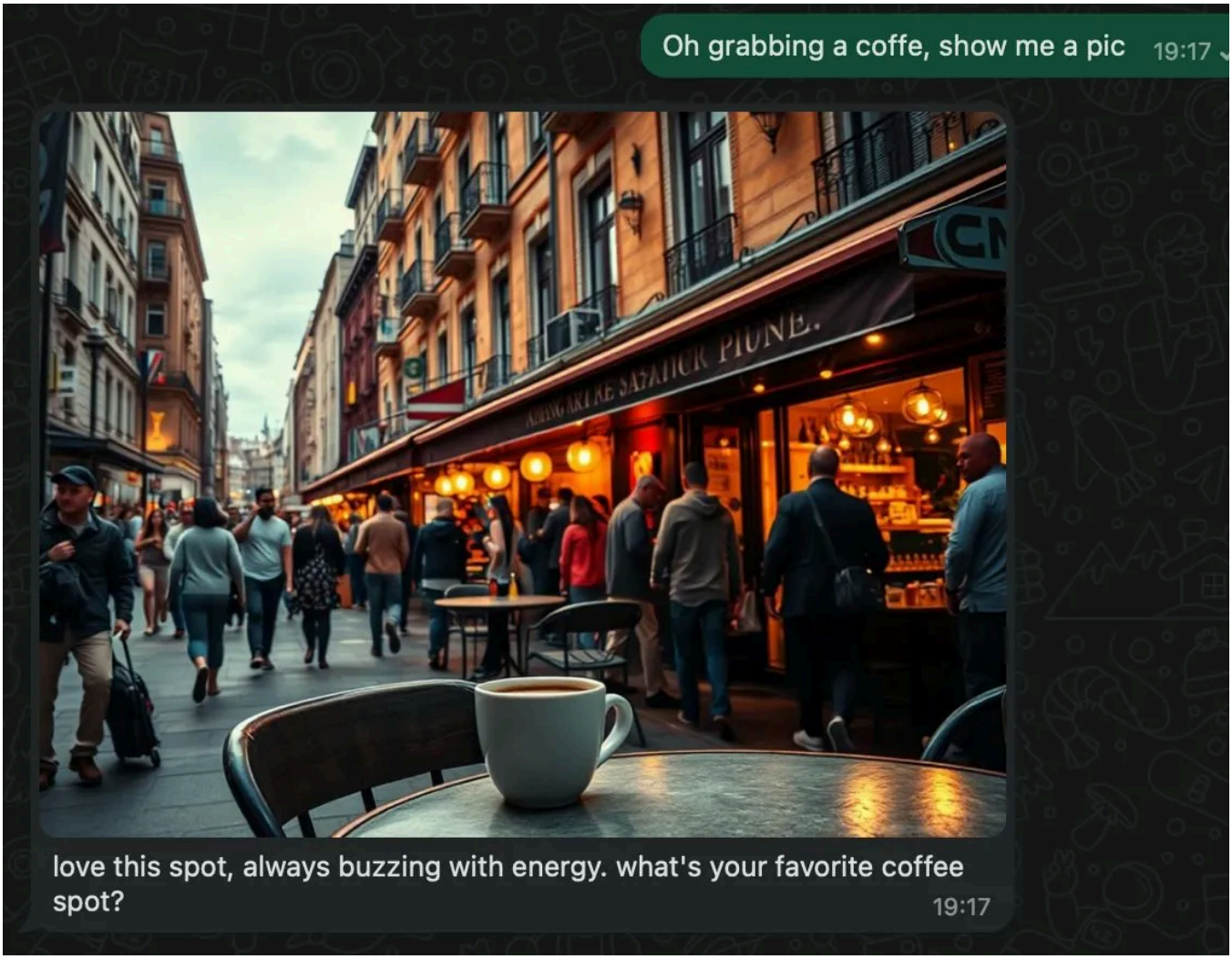
```
                template=IMAGE_SCENARIO_PROMPT,
            )
            | structured_llm
        )

        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**! 🙂

Thanks for reading The Neural Maze! Subscribe for free to receive new posts and support my work.

13 Likes • 2 Restacks

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...

Alex Razvant 5 Mar

Liked by Miguel Otero Pedrido

Great article man, given the complexities and different parts going underneath VLMs and Di Models, you guys managed to summarize it pretty well! 10/10 🔥

LIKE (1) REPLY

1 reply by Miguel Otero Pedrido

1 more comment...