tavily × MongoDB. × lastmile AI

Wingman.AI

Agenda

1. Problem
2. Solution
3. Use Cases
4. Code Explanation
5. Demo

Problem

85% of prospects and customers are dissatisfied with their on-the-phone experience, and feel that salespeople are unprepared for initial meetings

77% of executive buyers claim salespeople don't understand their issues, where they can help, and do not have relevant examples or case studies to share with them.

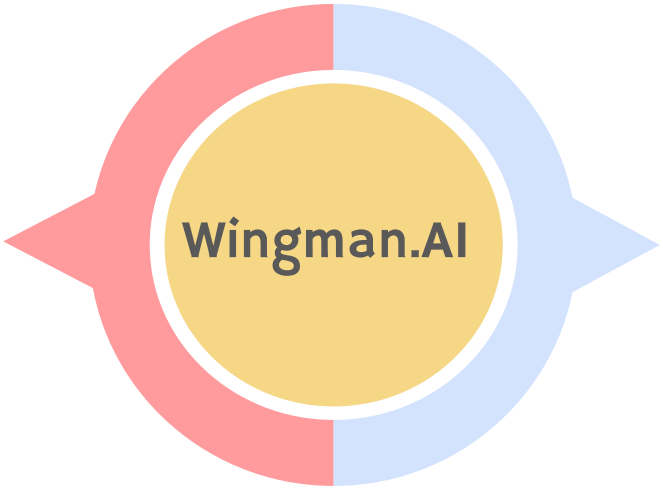
Sales leaders and managers say their **top challenge** is that customer needs and expectations keep changing

10+ Months for a new sales rep to become fully productive

Solution

CUSTOMERS

Need Answers Fast



SALES REP

Unprepared

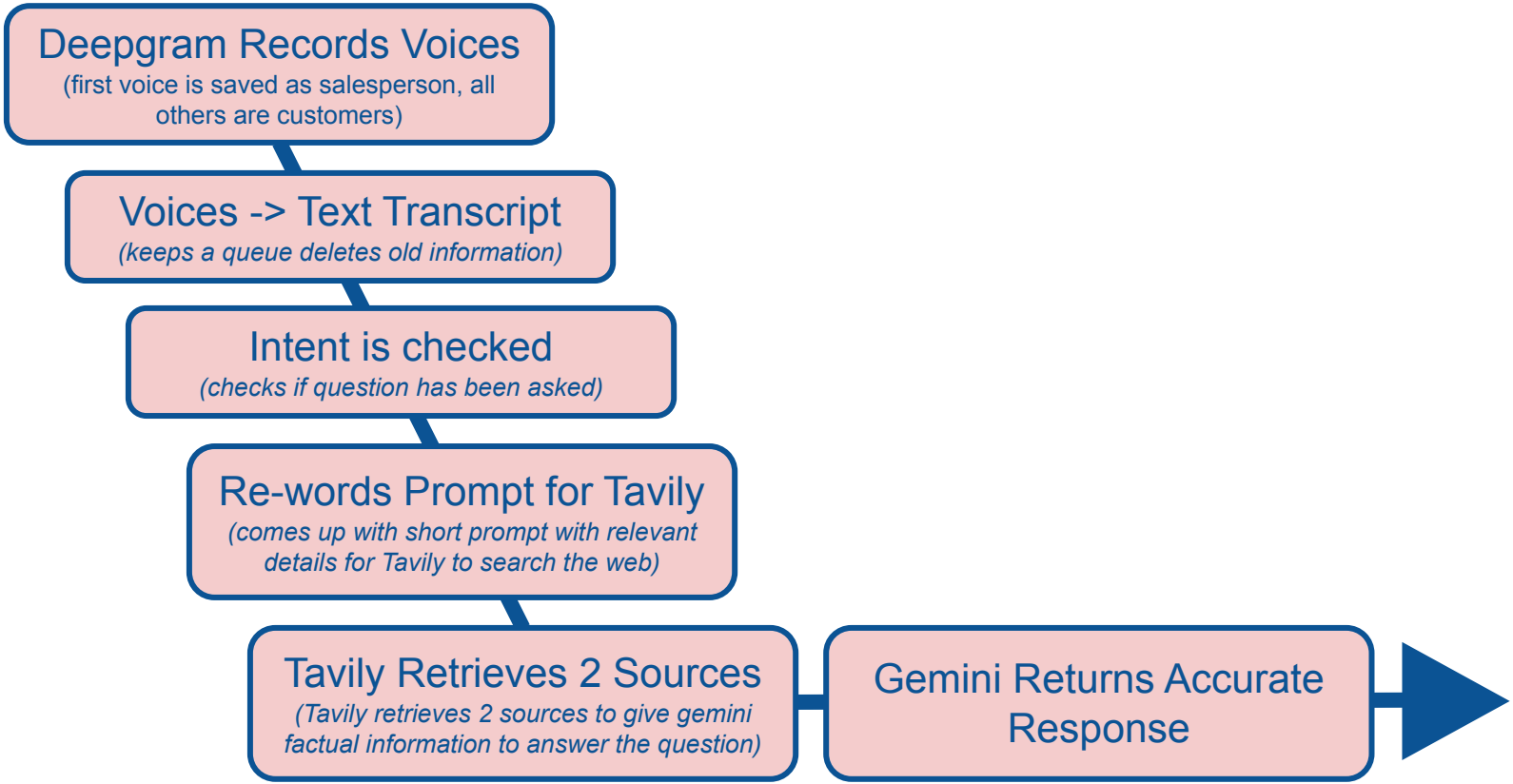
Real World Application

Pharmacists

Sales Representatives

Presentations with your internal team

Process Flow Diagram



*For the Intuitive Techies, it's like a CPU's five-stage pipeline (IF→ID→EX→MEM→WB) but each stage hands off to a different thread so the whole process runs in **parallel**.*

Not everything needs an answer!

```python
async def classify_intent(raw_text: str, conversation_context: str = "") -> str:
    """
    Classify user input using Groq LLaMA model into:
    - "Factual": question requires a fact-based answer
    - "Product-Opinion": question asks about opinions on a product, service, or topic
    - Returns None for personal/self-opinion questions (User-Opinion)
    Prints the detected type.

    NOTE: conversation_context is ignored; only the raw_text is used.
    """

    try:
        # --- Ensure it's a question ---
        lower_text = raw_text.lower().strip()
        question_words = [
            "what", "where", "when", "why", "how", "who", "which",
            "do", "does", "is", "are", "can", "could", "would", "will"
        ]
        if not (lower_text.endswith("?") or any(lower_text.startswith(w + " ") for w in question_words)):
            return None  # ignore statements

        print("\n-> Agent thinking about question...")

        # --- Prompt only uses the raw question ---
        prompt = f"""
You are a context-aware question classifier. Classify the question into one of three categories:

1. Factual - asks for factual information or knowledge.
2. Product-Opinion - asks about opinions regarding a product, service, or topic.
3. User-Opinion - asks about the user's personal feelings, thoughts, or state.
   For User-Opinion questions, respond with "None".

Rules:
- Respond **exactly** with "Factual", "Product-Opinion", or "None".

Examples:

Factual:
- What is the price of the iPhone 17?
- Where can I buy a Samsung Galaxy S25?

Product-Opinion:
- How do people feel about the new iPhone 17?
- Are users satisfied with the Xbox Series X?

User-Opinion (return None):
- How are you feeling today?
- Do you like yourself?

QUESTION: "{raw_text}"
"""
        completion = await asyncio.to_thread(
            groq_client.chat.completions.create,
            model="moonshotai/kimi-k2-instruct",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.0,
            max_tokens=30
        )

        result = completion.choices[0].message.content.strip().strip('"').strip()
```

- ➔ Ignore if…
  - Is it a statement?
  - Is it a personal question?
- ➔ Answer if…
  - Is it a Product-Opinion question?
  - Is it a Factual question?

```python
models_to_test = [
    #"allam-2-7b",
    #"meta-llama/llama-4-maverick-17b-128e-instruct",
    #"meta-llama/llama-guard-4-12b",
    #"meta-llama/llama-4-scout-17b-16e-instruct",
    "moonshotai/kimi-k2-instruct-0905",
    "llama-3.1-8b-instant",
    #"openai/gpt-oss-safeguard-20b",
    #"qwen/qwen3-32b",
    #"meta-llama/llama-prompt-guard-2-22m",
    #"llama-3.3-70b-versatile",
    #"groq/compound-mini",
    #"meta-llama/llama-prompt-guard-2-86m",
    #"openai/gpt-oss-120b",
    "moonshotai/kimi-k2-instruct",
    #"groq/compound",
    #"openai/gpt-oss-20b"
```

People Talk… unique.

```
# Query Restructuring Function (uses Groq)
async def restructure_query(context_text, raw_query):
    """
    Rewrites ambiguous user questions into standalone search queries using context.
    Uses Groq's Llama model for fast query rewriting.
    """

    prompt = f"""
You are a query rewriting engine.

Your task is to take the ambiguous user question and rewrite it into a fully
standalone web search query by resolving all missing details using the context.

Use the structured JSON object below as STRICT INPUT:

{{
  "context": {context_text!r},
  "raw_question": {raw_query!r}
}}

MANDATORY RULES:
1. The rewritten query MUST be fully understandable without the context.
2. You can use "context" to resolve vague references. but other times the previous thing said might not at all matter
   Example: "the 17" → "the iPhone 17" because context discusses an iPhone.
3. Replace any pronouns, short references, or implicit subjects with explicit nouns from context.
4. Do NOT preserve ambiguity — resolve it using if needed context.
5. Do NOT change the meaning of the question.
6. Output one single rewritten query.
7. Max 370 characters.
8. Output ONLY the rewritten query. No JSON. No quotes. No explanations.

Now produce the rewritten standalone search query:
"""

    try:
        completion = await asyncio.to_thread(
            groq_client.chat.completions.create,
            model="llama-3.3-70b-versatile",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.0,
            max_tokens=100
        )

        result = completion.choices[0].message.content.strip()
```

- ➔ Takes prior context to help answer question.
- ➔ Utilizes Groq as Gemini struggled to make strong queries to pass on

Tavily prevents hallucination… and angry customers!

```python
def tavily_search_tool(query: str):
    try:
        response = tavily_client.search(query=query, search_depth="basic", max_results=2)
        results = response.get('results', [])
        if not results: return "No results found."
        return "\n".join([f"- {re.sub(r'\s+', ' ', r['content']).strip()} ({r['url']})" for r in results])
    except: return "Search failed."
```

- ➔ The restructured query is sent to Tavily to pull high-quality factual information from the web.
- ➔ Only returns Tavily's top two results to keep Gemini's processing quick for real time!

Gemini Saves the day!

```python
async def run_agent_logic(current_text, speaker_role):
    text = re.sub(r'(?i)\b(um|uh|so|you know)\b', '', current_text)

    if speaker_role == "Om":
        parts = re.split(r'(?i)hey[\W_]+google', text, maxsplit=1)
        if len(parts) > 1:
            query = parts[1].strip()
        else: return
    else:
        query = text

    print(flush=True)

    classification = await classify_intent(query)

    if "Personal" in classification:
        print(f"   [Ignored - Personal/Statement]", flush=True)
        FULL_TRANSCRIPT.append(f"[{speaker_role}]: {current_text}")
        return

    full_context = get_full_context(current_text, speaker_role)
    search_query = await restructure_query(full_context, query)

    print(f"-> Agent rewording the question: {search_query}", flush=True)

    search_context = await asyncio.to_thread(tavily_search_tool, search_query)

    print(f"-> Agent answering question...", flush=True)

    prompt = f"""
    [TRANSCRIPT CONTEXT]
    {full_context}

    [USER QUERY]
    {query}

    [SEARCH RESULTS]
    {search_context}

    [INSTRUCTION]
    Answer the user's question using the search results.
    Keep it spoken-style, helpful, and under 2 sentences.
    """

    try:
        response = await asyncio.to_thread(
            gemini_client.models.generate_content,
            model="gemini-2.0-flash",
            contents=prompt,
            config=types.GenerateContentConfig(temperature=0.0)
        )

        answer = response.text.strip()
```

- Gemini utilizes its reasoning with Tavily's factual results to produce accurate answers
- Keeps responses to two concise sentences, enabling sales reps to quickly grasp the key information needed to address customer questions.

DeepGram and Microphone Logic

```python
# --- 1. Audio Streaming Infrastructure ---
class MicrophoneStream:
    def __init__(self, rate=16000, chunk=1024):
        self.rate = rate
        self.chunk = chunk
        self.p = pyaudio.PyAudio()
        self.queue = queue.Queue()
        self.stream = None

    def callback(self, in_data, frame_count, time_info, status):
        self.queue.put(in_data)
        return (None, pyaudio.paContinue)

    def start(self):
        self.stream = self.p.open(
            format=pyaudio.paInt16,
            channels=1,
            rate=self.rate,
            input=True,
            frames_per_buffer=self.chunk,
            stream_callback=self.callback
        )
        return self

    def stop(self):
        if self.stream:
            self.stream.stop_stream()
            self.stream.close()
        self.p.terminate()

    def get_data(self):
        try:
            return self.queue.get(timeout=0.1)
        except queue.Empty:
            return None
```

```python
def start_deepgram_socket(mic_queue, loop):
    url = "wss://api.deepgram.com/v1/listen?encoding=linear16&sample_rate=16000&channels=1&mod
    headers = {"Authorization": f"Token {DEEPGRAM_KEY}"}

    state = {
        "current_speaker": None,
        "current_buffer": [],
        "last_speech_time": time.time(),
        "has_processed_turn": False
    }

    def watcher_logic():
        while True:
            time.sleep(0.1)
            if not state["current_buffer"]: continue

            silence = time.time() - state["last_speech_time"]

            if silence > 1.5 and not state["has_processed_turn"]:
                full_text = " ".join(state["current_buffer"]).strip()
                if not full_text: continue

                clean_text = full_text.lower()
                role = "Om" if state["current_speaker"] == 0 else "Customer"
                should_trigger = False

                if state["current_speaker"] == 0:
                    if re.search(r'hey[\W_]+google', clean_text):
                        parts = re.split(r'hey[\W_]+google', clean_text, maxsplit=1)
                        if len(parts) > 1 and len(parts[1].strip()) > 2:
                            should_trigger = True

                elif state["current_speaker"] == 1:
                    should_trigger = True

                if should_trigger:
                    asyncio.run_coroutine_threadsafe(run_agent_logic(full_text, role), loop)
                    state["has_processed_turn"] = True
                    if state["current_speaker"] == 0:
                        state["current_buffer"] = []

                elif state["current_speaker"] == 1:
                    state["has_processed_turn"] = True

    threading.Thread(target=watcher_logic, daemon=True).start()
```