How Event-Driven Architecture Let Me Build a Self-Learning Chatbot Without Breaking Everything

When I first started building SmartChat, I wasn't trying to build some huge "agent system." I just wanted a chatbot that could respond intelligently and not feel like it had memory loss every time I asked a follow-up question.

But once I started adding features, I ran into a problem that shows up in basically every real software project:

How do you keep adding new behavior without rewriting your entire system?

That's where event-driven architecture (EDA) ended up being the perfect solution. It let me evolve SmartChat step-by-step without tightly coupling everything together.

This post is the story of how I used the soorma-core framework to build SmartChat, and how event-driven design helped it grow into a self-learning, multimodal agent with observability built in.

What Even Is Event-Driven Architecture?

The easiest way to explain event-driven architecture is:

Instead of services directly calling each other, they emit events when something happens.

For example:

- A user sends a message
- The agent generates a response
- The user gives feedback

Other parts of the system can listen to those events and react however they want. The important part is that the original component doesn't need to know who's listening.

That's what makes event-driven systems feel like they can "evolve." You can keep plugging in new capabilities without rewriting the core.

The Project: SmartChat

SmartChat is a chatbot I built using the soorma-core platform framework. The goal was to create an agent that doesn't just respond, but gradually becomes smarter over time by learning useful facts and adapting based on feedback.

I built it in three phases, and each phase was basically me realizing "okay this works… but what if it could do more?"

Phase 1: The Minimal Chatbot

I started with the soorma-core example 01-hello-world:

https://github.com/soorma-ai/soorma-core/blob/dev/examples/01-hello-world/README.md

The example is a great starting point, but it's intentionally simple. So I used Cursor to extend it into a real chatbot by connecting it to an LLM for actual responses instead of static greetings. I also vibe-coded a frontend client to make it feel like a real product rather than just a backend demo.

At this point, SmartChat worked, but it had a big limitation:

It had no memory.

Every message felt like starting over.

Phase 2: Adding Episodic Memory So It Remembers Conversations

In Phase 2, I wanted SmartChat to remember what the user said earlier in the conversation so it could respond with actual context.

To do that, I followed soorma-core multi-agent design patterns:

https://github.com/soorma-ai/soorma-core/blob/dev/docs/DESIGN_PATTERNS.md

Then I integrated episodic memory using the 06-memory-episodic example:

https://github.com/soorma-ai/soorma-core/blob/dev/examples/06-memory-episodic/README.md

This made the chatbot instantly feel smarter. Instead of responding like every message was brand new, it could reference past exchanges and maintain conversational flow.

Phase 2 introduced a basic form of learning: the agent could use conversation history to fill in gaps and respond more naturally.

But there was still a major limitation.

This memory only existed within the same conversation session. If the user came back later, SmartChat didn’t really retain meaningful knowledge long-term.

Phase 3: Evolving the System With Choreography and Semantic Learning

This is where event-driven architecture became the main reason the project worked.

The limitation of Phase 2 was that SmartChat could only “learn” from raw conversation history. It wasn’t extracting long-term knowledge about the user or storing facts that could carry over into new conversations.

So in Phase 3, I used the choreography style of event-driven architecture to evolve the system without rewriting the existing agents.

Instead of modifying the core chatbot, I introduced a new agent called feedback_agent.

This agent silently listens to message exchanges. Whenever it detects something important, it extracts relevant facts and stores them into soorma-core semantic memory. This creates a long-term knowledge base that SmartChat can retrieve later, even across brand new conversations.

This was the turning point.

Now SmartChat didn’t just remember what happened five minutes ago. It could remember useful facts over time and use them to generate better responses later.

And the best part is that I didn’t have to tightly couple this logic into the main agent. I added a new agent that subscribed to events already flowing through the system.

That’s exactly what event-driven architecture is good at:
you can add new behaviors without breaking what already works.

Adding Real-Time Voice With Pipecat

After building the text chatbot, I wanted SmartChat to feel more natural and interactive, so I added real-time voice support using Daily Pipecat.

Pipecat made it possible to build a low-latency voice pipeline where SmartChat could listen, transcribe speech, generate responses, and speak back in real time.

The voice pipeline uses:

- Daily.co for low-latency audio transport
- Deepgram for fast speech-to-text
- OpenAI for text-to-speech

The key design decision was that both the text worker and voice worker share the same agent logic. That means anything SmartChat learns through voice is instantly available in text chat, and vice versa.

So instead of being “two separate bots,” it stays one unified brain across modalities.

Observability With Weave

Once the system became multi-agent and started learning, it became really important to understand what was happening internally.

So I integrated Weights and Biases Weave to trace and debug the agent’s decision-making.

With Weave, I can inspect:

- which semantic facts were retrieved
- what the final prompt looked like
- how the LLM responded
- how feedback affects behavior over time

Instead of guessing why the agent gave a certain answer, Weave makes the reasoning process visible. That was especially useful once feedback and semantic memory started influencing responses, because the system wasn’t just “responding,” it was adapting.

Why This Felt Like System “Evolution”

Looking back, each phase felt like a real upgrade:

Phase 1
A chatbot that responds.

Phase 2
A chatbot that remembers conversation history.

Phase 3
A chatbot that learns long-term knowledge.

Then Pipecat added voice interaction, and Weave added observability so the system could actually be inspected and debugged like a real product.

The biggest thing I learned from this build is that event-driven architecture makes experimentation and expansion way easier. You can keep adding new agents and new capabilities without rewriting everything.

Final Thoughts

SmartChat started as a simple hello-world chatbot, but it evolved into a multi-agent system with episodic memory, semantic learning, real-time voice interaction, and full observability.

EDA made the project scalable, Pipecat made it multimodal, and Weave made it debuggable.

If you want to see the full implementation and architecture details, the docs are here:

https://github.com/siyabhadoria/smartchat/blob/main/README.md
https://github.com/siyabhadoria/smartchat/blob/main/PROJECT_OVERVIEW.md