My Reflection on Building the LangGraph Q&A Assistant

📝 My Journey & What I Learned

When I First Ran the Code...

Honestly, I was excited to execute it, but **things didn't go as planned**. I quickly noticed:

1. **The output was a mess** - no structure, just raw text dumps
2. **It kept running forever** - got stuck in loops multiple times
3. **No web search happened** - even though I expected it to research
4. **Zero reflections** - the agent wasn't learning from mistakes
5. **Couldn't trace anything** - had no idea which node was executing when
6. **Lost all history** - restarted = everything gone

I was like, "Okay, this needs serious work!" 😅

🛠️ How I Fixed Each Issue

A. Structured JSON Output

The Problem I Faced: My evaluator node was returning plain text, sometimes markdown, sometimes just rambling. When I tried `json.loads()`, it crashed constantly.

What I Did:

```
def extract_json_from_text(text: str) -> dict:

    # I added multiple fallback strategies

    # First, try markdown code blocks

    json_match = re.search(r'```(?:json)?\s*(\{.*?\})\s*```', text, re.DOTALL)



    # Then try finding JSON anywhere

    json_match = re.search(r'\{[^{}]*(?:\{[^{}]*\}[^{}]*)*\}', text, re.DOTALL)
```

```
    # Finally, direct parse
    return json.loads(text)
```

Why I Did It This Way:

- I realized Ollama doesn't always follow format instructions perfectly
- I needed **defensive parsing** with graceful degradation
- If all parsing fails, I create a reasonable fallback based on content length and iteration

What I Learned:

> LLMs are unpredictable. Always have Plan B, C, and D for structured outputs!

B. Breaking the Infinite Loop

The Problem I Faced: My workflow would sometimes run 20+ iterations because there was no exit condition. I watched it regenerate answers forever! 😱

What I Did:

```
def should_continue(state: QAState) -> Literal["search", "reflect", "finalize"]:
    iteration_count = state.get("iteration_count", 0)
    max_iterations = state.get("max_iterations", 3)

    # FIRST CHECK - hard stop
    if iteration_count >= max_iterations:
        print("   ➡️ Max iterations reached → FINALIZE")
        return "finalize"
```

Why I Did It This Way:

- I put the max iterations check **at the very top** - no exceptions
- Made it user-configurable through the CLI
- Added clear logging so I can see WHY it's stopping

What I Learned:

> Always add circuit breakers in iterative systems. Otherwise, you're burning API credits for nothing!

C. Adding Web Search Tool

The Problem I Faced: My agent was answering everything from its training data, even for current events. When I asked "latest news about X", it had nothing.

What I Did:

```
def search_tool_node(state: QAState) -> QAState:

    search = TavilySearchResults(max_results=3, search_depth="advanced")



    try:

        results = search.invoke(question)

        # Format and add to context

        formatted_results = [f"{idx}. {result.get('content', '')}\nSource: {result.get('url', 'N/A')}"

                             for idx, result in enumerate(results, 1)]
```

Then I made it flow into generation:

```
# In generate_answer_node

if state.get("search_results"):

    context = f"\n\nSearch Results:\n{state['search_results']}\n\n"
```

Why I Did It This Way:

- The evaluator can set `needs_search: true` when it detects knowledge gaps
- Search results become **context** for the next generation
- Added error handling because APIs fail sometimes

What I Learned:

> Tool calls should be conditional, not mandatory. Let the agent decide when it needs external help!

D. Implementing Reflections

The Problem I Faced: Each iteration was independent - the agent kept making the same mistakes! No learning between attempts.

What I Did:

```
def reflection_node(state: QAState) -> QAState:
    evaluation = state.get("evaluation", {})
    reflections = state.get("reflections", [])

    # Build reflection from evaluation feedback
    new_reflection = f"Iteration {state.get('iteration_count', 0)}: "

    if evaluation.get("weaknesses"):
        new_reflection += f"Address: {', '.join(evaluation['weaknesses'])}. "

    if evaluation.get("suggestions"):
        new_reflection += f"Improve: {', '.join(evaluation['suggestions'])}"

    reflections.append(new_reflection)
```

Then inject back into generation:

```
reflection_context = "\n\nPrevious Reflections:\n" + "\n".join(
    f"- {r}" for r in state["reflections"]
)
```

Why I Did It This Way:

- Reflections **accumulate** across iterations
- They're built from the evaluator's structured feedback
- The generator sees ALL past reflections, so it learns

What I Learned:

> Memory isn't just storage - it's about making the agent AWARE of its own history!

E. Adding Tracing & LangSmith Integration 🎯

The Problem I Faced: I had **zero visibility** into what was happening. Nodes were executing in black boxes. I couldn't debug failures or optimize performance.

What I Did:

First, I added **real-time console tracing**:

```
# Stream execution to see each step

for step_output in workflow.stream(initial_state):

    for node_name, node_state in step_output.items:

        print(f"\n🔵 Executed: {node_name}")

        memory.log_state(node_name, node_state)
```

Then I integrated **LangSmith for production observability**:

```
# Disable LangSmith tracing by default (enable only if explicitly set)

if os.getenv("LANGCHAIN_TRACING_V2", "").lower() == "true":

    print("✅ LangSmith tracing enabled")

else:

    os.environ["LANGCHAIN_TRACING_V2"] = "false"
```

And detailed router logging:

```
print(f"\n🔀 ROUTER: Iteration {iteration_count}/{max_iterations}")
```

print(" ➡️ Answer acceptable → FINALIZE")

Why I Added LangSmith:

- **Console logging is great for development**, but I needed **production-grade observability**
- LangSmith gives me:
    - Complete trace of every LLM call with inputs/outputs
    - Token usage and cost tracking per run
    - Latency metrics for each node
    - Ability to replay and debug failed runs
    - Visual graph of execution flow
- Made it **opt-in by default** - only activates when explicitly enabled via environment variable
- This way local development stays fast, but production has full visibility

What I Learned:

Two-tier observability is key:

- **Console logs** = Quick feedback during development
- **LangSmith** = Deep insights for production debugging and optimization

The Power of LangSmith I Discovered:

- I can see EXACTLY which prompts produced low scores
- Token usage helps me optimize costs (found I was wasting tokens on redundant context!)
- Can compare different runs side-by-side to see what works better
- Team members can view traces without accessing my terminal

Pro Tip I Learned: Always make tracing opt-in with environment variables:

Development: fast, no tracing

python main.py

Production: full observability

LANGCHAIN_TRACING_V2=true python main.py

This saved me from accidentally sending 1000s of traces during testing! 😅

F. Building Persistent Memory

The Problem I Faced: Every time I restarted, **all my Q&A history was gone**. I couldn't compare sessions or track improvements.

What I Did: Built a complete `MemoryManager` class:

```
class MemoryManager:

    def create_session(self, question: str) -> str:

        session_id = datetime.now().strftime("%Y%m%d_%H%M%S")

        # Create structured session data



    def log_state(self, node_name: str, state: Dict):

        # Log every state transition with metadata



    def save_final_answer(self, answer: str, full_state: Dict):

        # Save complete final state



    def get_session_history(self, limit: int = 10):

        # Browse past sessions
```

Why I Did It This Way:

- JSON files are simple, human-readable, and don't need a database
- Each session gets a unique timestamp-based ID
- Metadata tracks iterations, search usage, reflection count
- Can export individual sessions for sharing

What I Learned:

> Persistent memory transforms a script into an application. History = learning!

💪 Challenges I Overcame

1. **Ollama's JSON inconsistency** - Solved with multi-strategy parsing
2. **Conditional routing complexity** - Drew it on paper first, then coded
3. **State management** - Used TypedDict strictly to catch bugs early
4. **Error handling** - Added try-catch everywhere tools are called
5. **User experience** - Made the CLI actually enjoyable to use
6. **LangSmith setup** - Learned about environment-based configuration for tracing

---

🎯 What I'm Proud Of

1. **It actually works end-to-end** - Not just a demo!
2. **Production-ready patterns** - Error handling, logging, persistence
3. **Smart fallbacks** - Degrades gracefully when things fail
4. **Interactive UX** - Real application, not just a script
5. **Complete observability** - Can debug any issue with console logs OR LangSmith
6. **Professional tracing** - Two-tier observability (dev vs prod)

🚀 What I'd Do Next

If I had more time, I'd add:

- **Async execution** - Speed up with `asyncio`
- **Better evaluation** - Multi-criteria scoring (accuracy, clarity, completeness)
- **RAG integration** - Load documents for domain-specific questions
- **Streaming output** - Show answer generation in real-time
- **Unit tests** - Test each node independently
- **Config file** - YAML/JSON for all parameters
- **LangSmith datasets** - Create test cases to measure improvement over time
- **Custom LangSmith runs** - Add tags and metadata for better filtering

---

🎓 Key Takeaways

Building this taught me:

✅ **Agentic workflows need guardrails** - Max iterations, timeouts, validation
✅ **Observability from day one** - Saved me hours of debugging
✅ **Graceful degradation > Perfection** - Fallbacks are your friend
✅ **State machines are powerful** - But need careful planning
✅ **Memory = Intelligence** - Reflections make agents actually learn
✅ **LangSmith is a game-changer** - Production LLM apps need professional tracing
✅ **Opt-in tracing is smart** - Fast dev loops, deep prod insights

---

🏁 Final Thoughts

I went from a **broken workflow with 6 critical issues** to a **fully functional agentic Q&A system** that:

- Generates structured outputs reliably ✅
- Never gets stuck in infinite loops ✅
- Calls external tools when needed ✅
- Learns from its mistakes through reflection ✅
- Provides complete traceability ✅
- Remembers everything across sessions ✅
- **Has professional-grade observability with LangSmith** ✅

The best part? I didn't just fix bugs - I learned how to build robust LLM systems that actually work in production. **Adding LangSmith was the cherry on top** - now I can confidently deploy this knowing I'll be able to debug any issue that comes up!

This is the kind of code I'd be proud to ship to real users! 💪

📊 Before vs After LangSmith

Before:

- "Something broke... let me add 50 print statements" 😓
- "Why did this run cost $2? No idea!" 💸
- "Can't reproduce that weird bug..." 🤷

After:

- "Let me check the LangSmith trace... ah, that prompt was too long!" 🎯
- "This node uses 3K tokens, I can optimize that" 💡
- "Here's the exact trace link showing the bug" 🔗

Now I understand why they say: "Code is easy. Making it observable is hard. But observability is what separates toys from production systems." 😅

LangSmith = My new superpower! 🚀