Assessment Task => Support Ticket Resolution Agent with Multi-Step Review Loop

---

🧭 Overview

You are tasked with building a **support ticket resolution agent** using the **LangGraph framework**. This assistant will receive a ticket (with a subject and description), classify it, retrieve relevant context, draft a response, and validate it through an automated reviewer. If the review fails, it should refine the context and retry.

This assessment tests your understanding of **graph-based orchestration, prompt engineering, modular architecture**, and **robust design practices** using LangGraph. It also evaluates your **awareness of your own code**, which is a core part of our technical expectations.

⚙ Objective

Build an AI-powered support agent using LangGraph that:

1. Accepts a support ticket (`subject`, `description`)
2. Classifies it into one of several predefined categories (e.g., Billing, Technical, Security, General)
3. Based on classification, retrieves context using RAG
4. Drafts an initial response based on the context
5. Reviews the draft using a policy/reviewer LLM
6. If the review fails, refines the context and re-drafts
7. Repeats the draft-review cycle a maximum of **2 times**

🗂️ Components Overview

Step	Description
🏷️ Classification	Classify the ticket based on subject + description
📚 Retrieval (RAG)	Fetch relevant info for that category (e.g., from docs or vector store)
📝 Draft Generator	Generate an answer draft using context + ticket
✅ Draft Reviewer	LLM checks policy compliance and quality
🔁 Retry Logic	If review fails, re-fetch context based on reviewer feedback and re-draft
📁 Escalation CSV	Optional: Add failed tickets to a CSV log file for a human review

---

📌 What We Are Expecting

This section explains — in detail — the desired behavior and responsibilities of the support agent you are building using **LangGraph**.

Your implementation must orchestrate a multi-step process that mirrors how real-world support assistants operate, with dynamic feedback loops and decision-making built into the graph. Below is a breakdown of each required component and how it should behave in practice.

📨 1. Input Ticket Handling

The system receives an input support ticket, containing:

- **subject**: A short string describing the topic (e.g. “Login failure on mobile”)
- **description**: A longer free-text explanation of the issue

This is the starting node of your graph. This data flows into the rest of the system and influences all downstream decisions.

🧠 2. Classification Step

This is the **first node** after receiving input.
Use a classification mechanism (e.g., LLMNode or classifier) to determine **which category** the support ticket belongs to.

Suggested categories:

- Billing
- Technical
- Security
- General

The classification result must be:

- Reliable, even with vague or multi-intent descriptions
- Passed downstream to influence RAG behavior

🔍 3. Context Retrieval (RAG)

Based on the classification output, fetch contextual information from a relevant source (e.g., documents, vector database, static index, or mocked RAG).

Each category should map to a specific knowledge source, which your retrieval system must be aware of and query selectively.

This step should:

- Be modular, with one node per category or a routing RAG layer

- Incorporate subject and description into the retrieval prompt/query
- Output a list of relevant docs or data snippets to feed into the response generation

---

✍️ 4. Draft Generation

Using the input ticket and retrieved context, compose an initial draft response to the user. This is your first synthesis point, combining:

- Original user query
- Retrieved knowledge
- Possibly, category-specific prompt logic

It should be:

- Clear and customer-facing
- Grounded in the provided context
- Flexible to retry (see below)

✅ 5. Review + Policy Check

The draft response must now pass through a review node, typically another LLM or tool node simulating a quality assurance check.

The reviewer's job:

- Evaluate whether the response is accurate, helpful, and compliant with support guidelines (e.g., don't overpromise, don't provide refunds without approval, don't advise on sensitive security issues).

Return either:

- ✅ Approved: Continue to output
- ❌ Rejected: Provide feedback for revision

🔁 6. Feedback-Driven Retry (Max: 2 Attempts)

If the reviewer rejects the draft:

1. Use the reviewer's feedback to refine the retrieval step (e.g., using more specific queries or category adjustments).
2. Re-run the draft generator with the new context + reviewer feedback.
3. Send the new draft back to the reviewer.
4. Repeat this process up to a maximum of 2 total review cycles.

⚠ This introduces a loop in your graph. It must terminate after 2 failed reviews.

📤 7. Final Output or Escalation

If the draft passes review at any stage → output the response as final.
If it fails both attempts (This is optional):

Trigger an escalation path where the system generates a message that a human support agent should review this case.

This message should:

- Include the original ticket
- Include all failed drafts
- Include reviewer feedback

Log this into a CSV file (`escalation_log.csv`) for manual triage.

🔐 Technical Constraints

✅ You must use LangGraph (https://langchain-ai.github.io/langgraph/)
✅ You must use the LangGraph CLI for local dev server — not bare checkpointing.
✅ You can use any LLM or embeddings provider.

📦 Deliverables

Your submission must include:

1. ✅ A well-structured codebase with clear modularity and comments.
2. ✅ A detailed `README.md` that explains:
    - Setup instructions (LangGraph dev environment)
    - How to run the agent and test it
    - What design and architectural decisions you made
3. ✅ A demonstration video (MP4 or link):
    - Show all major flows (happy path, retries, escalation)
    - Explain the rationale behind your architecture
    - Walk through the code with confidence
4. ✅ A Git repository with:
    - All source code
    - The README
    - The recorded demo
    - If escalation CSV is implemented, include sample file

🔗 The Git repo should be public or access should be granted to our reviewer (@ahmed-z0). explicitly.

🧪 Evaluation Criteria

Area	Criteria
✅ Functional Accuracy	Agent performs all core steps as described
🔄 Retry Logic	Clear control over loop with feedback incorporated
🧠 Prompt Engineering	Inputs and prompts are purposeful and well-structured

💡 Architectural Thinking	Nodes are modular and reusable; flow is efficient
💬 Code Awareness	Video shows you understand why you wrote what you wrote
🔍 Logging & Traceability	System state is visible and traceable across steps
🧹 Code Quality	Clean, well-commented, and logically organized codebase
🧰 Tool Use	Correct use of retrieval, classification, tools, and memory where needed
💣 Edge Handling	Handles failed retrievals, model outputs, or API errors gracefully
🔒 Responsible AI Use	Thoughtful use of LLMs; no obvious misuse or blind prompting

---

🚫 Disallowed Practices

❌ Do not bypass LangGraph’s CLI dev server or checkpointing architecture
❌ Do not use autogenerated code blindly without understanding it
❌ Do not use ChatGPT/Copilot/etc. to generate full implementations without verification
❌ Do not submit code you can’t explain confidently

📅 Timeline & Submission

- 🕐 **Duration:** 7 Days from receipt
- 📥 **Submission:** GitHub/GitLab link + recorded demo video
- 📁 **Format:** Source + README + Video

🧠 Closing Notes

This task mirrors real-world agent orchestration at production-grade companies using LangGraph. While the implementation is important, we most heavily evaluate how well you understand and explain your design choices, code structure, and failure handling. Be bold with tools, but thoughtful in execution.

Use AI responsibly, document rigorously, and design like this graph is going into production