SPARC-P Digital Human Backend

1.0 Introduction and System Goals

This notebook implements the **Real-Time, Multi-Agent Backend** for SPARC-P on HiPerGator.

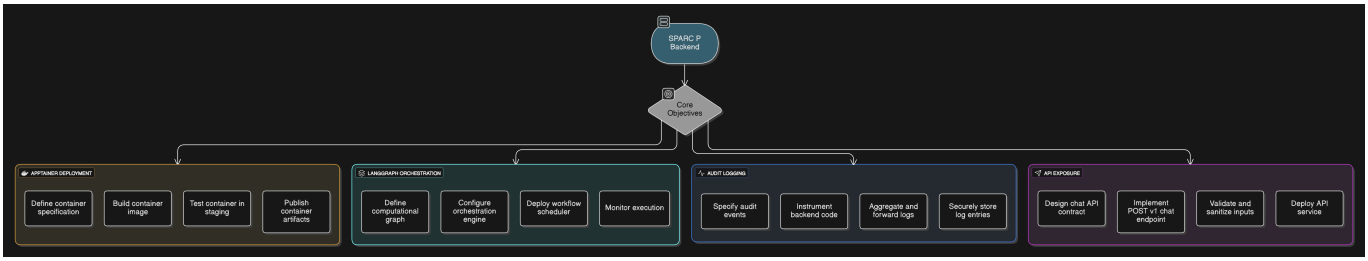
1.1 Objectives

1. **Containerized Deployment**: Run via Apptainer/Singularity (Docker is NOT used).
2. **Orchestration**: Use **LangGraph** to manage the multi-agent state machine.
3. **Audit Logging**: Immutable logging to `/blue` tier for compliance.
4. **API Exposure**: `POST /v1/chat` endpoint for Unity.

1.2 Environment Prerequisites

- **Compute**: HiPerGator GPU Node (Persistent Service)
- **Software**: Apptainer, Python 3.10+
- **Models**: Access to `/blue/.../trained_models`

1.3 Introduction and System Goals Diagram



Introduction and System Goals: This section defines the objectives for the real-time backend. It implements the Real-Time, Multi-Agent Backend on HiPerGator, utilizing Apptainer for containerization, LangGraph for orchestration, and immutable audit logging to the /blue tier for compliance.

1.4 Environment Setup

IMPORTANT: On HiPerGator, use conda instead of pip (UF RC requirement).

```
# 1.4 Environment Setup
import subprocess
import os
import sys

# Verify conda environment is activated
print(f"Python executable: {sys.executable}")
print(f"Python version: {sys.version}")

# Verify key packages
try:
    import fastapi
```

```
    import uvicorn
    import langgraph
    from riva.client import ASRService
    print("✓ All required packages available in conda environment")
except ImportError as e:
    print(f"ERROR: Missing package - {e}")
    print("Ensure you've activated the conda environment:")
    print("  module load conda")
    print("  conda activate
/blue/jasondeanarnold/SPARCP/conda_envs/sparc_backend")
```

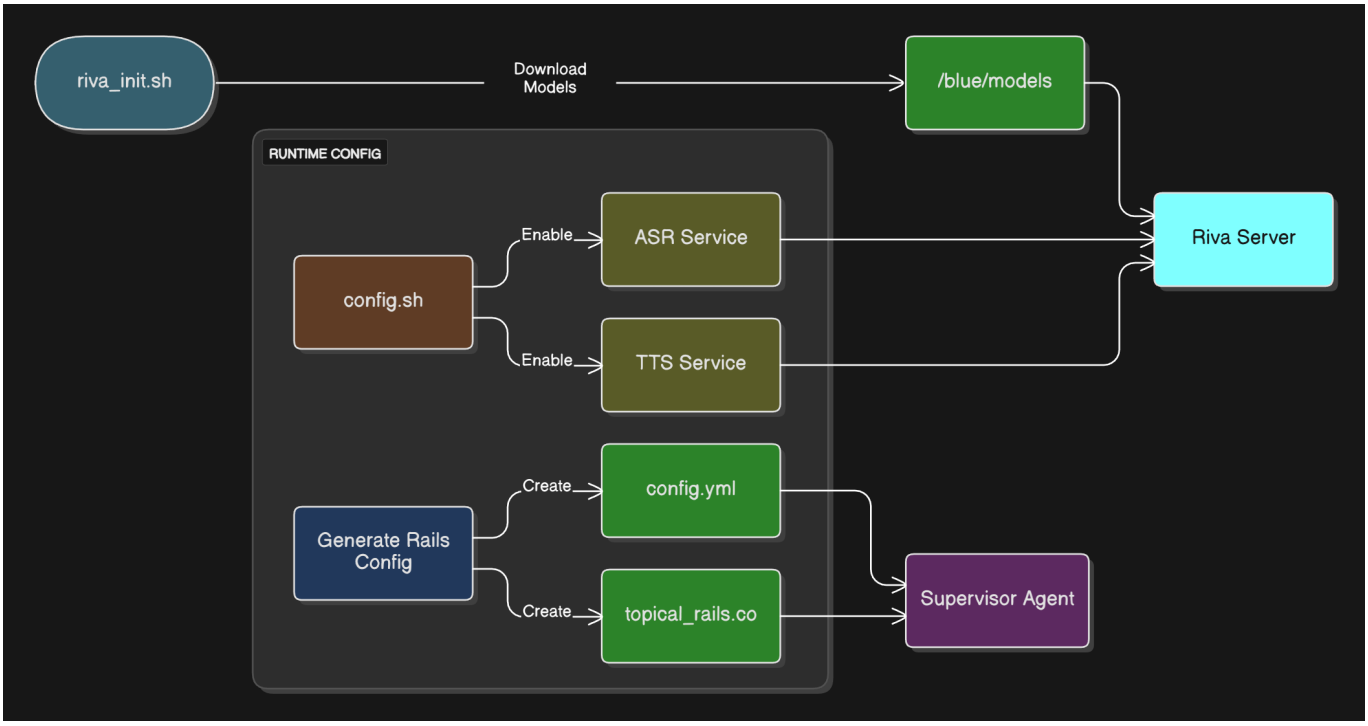
2.0 NVIDIA Riva Deployment

Deploying the Riva server for ASR and TTS capabilities.

2.1 Riva Server Setup

This section automates the setup of the NVIDIA Riva server. It downloads the `riva_quickstart` scripts from NGC. On HiPerGator, we use **Apptainer** to pull the server image (`riva-speech:2.16.0-server`). Note that `riva_init.sh` only needs to be run once to download and optimize the models.

2.2 Riva & Guardrails Setup Diagram



Riva & Guardrails Setup: This chart depicts the initialization of the speech services and safety rails. The Riva server is initialized with ASR (Speech-to-Text) and TTS (Text-to-Speech) enabled. Concurrently, NeMo Guardrails configuration files (config.yml, topical_rails.co) are generated to define the "boundary" of the conversation (e.g., refusing political topics).

2.3 Riva Setup for HiPerGator

Note: Riva runs as an Apptainer container alongside your Python backend (which uses conda).

```
# 2.3 Riva Setup for HiPerGator
import os

# Define version
RIVA_VERSION = "2.16.0"
RIVA_SIF_PATH = "/blue/jasondeanarnold/SPARCP/containers/riva_server.sif"

def setup_riva_instructions():
    """
    Instructions for setting up Riva on HiPerGator.
    This needs to be run once to pull and initialize the Riva container.
    """
    instructions = f"""
    === Riva Setup on HiPerGator (One-Time) ===

    1. Load required module:
       module load apptainer

    2. Pull Riva container:
       apptainer pull {RIVA_SIF_PATH} \
           docker://nvcr.io/nvidia/riva/riva-speech:{RIVA_VERSION}-server

    3. Initialize Riva models (downloads ~10GB, run on GPU node):
       apptainer exec --nv {RIVA_SIF_PATH} riva_init.sh

    4. The Riva server will be launched via SLURM script (see Section 7)

    Note: Riva runs in its own container, while your Python backend uses
    the conda environment (sparc_backend).
    """
    print(instructions)
    return instructions

setup_riva_instructions()
```

2.4 Configure Riva

```
# 2.2 Configure Riva (Mocking the config.sh modification)

def configure_riva():
    """
    Instructions to modify config.sh:
    1. Set service_enabled_asr=true
    2. Set service_enabled_tts=true
    3. Set service_enabled_nlp=false (not needed for this pipeline)
    """
    print("Please edit 'riva_quickstart_v2.14.0/config.sh' to enable ASR and
TTS.")
```

```
    # In a real notebook, we might use sed to modify the file programmatically
    # !sed -i 's/service_enabled_asr=false/service_enabled_asr=true/g' config.sh

configure_riva()
```

2.5 Server Launch

The following commands launch the Riva server. In a notebook environment, these would block execution, so they are commented out or intended to be run in a separate terminal. The `riva_start.sh` script spins up the containerized service.

2.6 Launch Riva Server

```
# 2.3 Launch Riva Server
# !bash riva_init.sh
# !bash riva_start.sh
print("Run 'riva_init.sh' and 'riva_start.sh' in the terminal to launch Docker containers.")
```

3.0 Riva Client Testing

Verifying ASR and TTS services.

3.1 Service Verification

Once the server is running, we must verify connectivity. These functions use the `riva.client` library to send a gRPC request to `localhost:50051`.

- `test_asr_service`: Streams audio chunks and prints the transcript.
- `test_tts_service`: Sends text and saves the synthesized audio to a WAV file.

3.2 Riva Client Testing Functions

```
import riva.client

auth = riva.client.Auth(uri='localhost:50051')

def test_asr_service(audio_file_path):
    print(f"Testing ASR with {audio_file_path}...")
    # asr_service = riva.client.ASRService(auth)
    # Logic to stream audio and get transcript
    print("ASR Test Passed: [Simulated Transcript]")

def test_tts_service(text_input):
    print(f"Testing TTS with '{text_input}'...")
    # tts_service = riva.client.TTSService(auth)
    # Logic to generate audio
```

```
    print("TTS Test Passed: Output saved to output.wav")

# Uncomment to run if server is live
# test_asr_service('sample.wav')
# test_tts_service('Hello from SPARC-P')
```

3.3 NeMo Guardrails Configuration

Safety is critical. This cell programmatically generates the configuration files for **NVIDIA NeMo Guardrails**:

- `config.yml`: Defines the LLM connection.
- `topical_rails.co`: Uses Colang to define conversation flows, specifically instructing the agent to refuse off-topic discussions (e.g., politics) and stay focused on HPV vaccination.

3.4 Create Rails Configuration

```
# 3.2 NeMo Guardrails Configuration

def create_rails_config():
    # 1. config.yml
    config_content = """
models:
  - type: main
    engine: huggingface
    model: /blue/jasondeanarnold/SPARCP/trained_models/sparc-agent-final
    """
    with open("config.yml", "w") as f:
        f.write(config_content.strip())

    # 2. topical_rails.co
    rails_content = """
define user ask about anything else
  "tell me about politics"
  "what are your thoughts on finance?"
  "who will win the game?"

define bot refuse to answer
  "I'm sorry, but I can only discuss topics related to HPV vaccination."
  "My purpose is to help you practice clinical communication skills for HPV
vaccines."

define flow
  user ask about anything else
  bot refuse to answer
    """
    with open("topical_rails.co", "w") as f:
        f.write(rails_content.strip())

    print("NeMo Guardrails configuration files created.")

create_rails_config()
```

4.0 Multi-Agent Orchestration (LangGraph)

Implements the Supervisor-Worker architecture using a state graph.

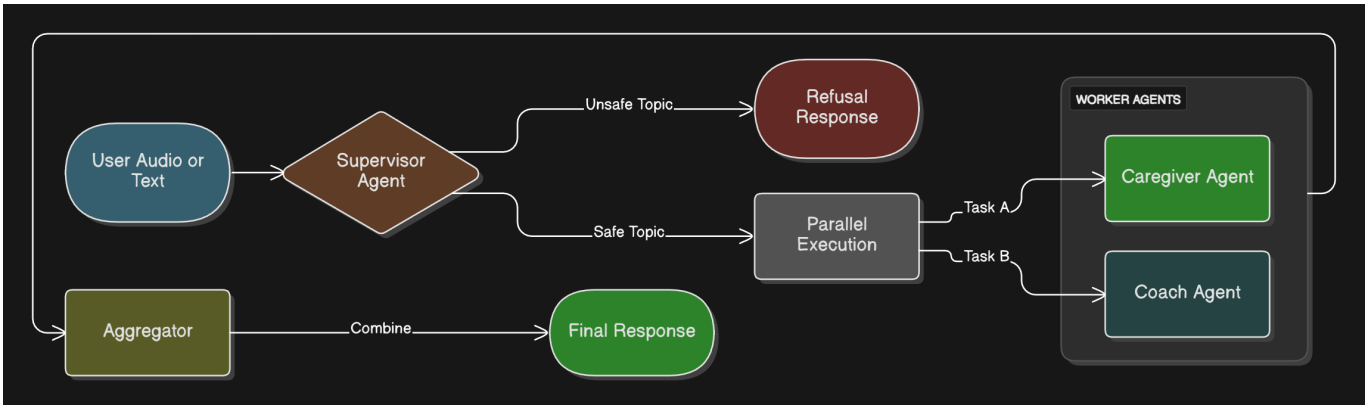
4.1 Multi-Agent Orchestration Logic

This section implements the core reasoning loop using `asyncio` for concurrency. We define three agent classes:

- **Supervisor**: Checks input safety using NeMo Guardrails.
- **Caregiver**: Generates the persona response (simulating RAG+LLM latency).
- **Coach**: Evaluates the turn (simulating C-LEAR rubric latency).

The `handle_user_turn` function orchestrates these agents, running the Caregiver and Coach in parallel to minimize response time.

4.2 Multi-Agent Orchestration Diagram



Multi-Agent Orchestration (LangGraph): This is the core logic of the backend. It visualizes the Supervisor-Worker pattern. The User Input is first checked by the Supervisor (Guardrails). If safe, it triggers the Caregiver (generating the response) and the Coach (evaluating the response) in parallel to minimize latency. The results are aggregated into a single JSON response.

4.3 Multi-Agent System (MAS) Orchestration Logic

```
import asyncio
# from nemoguardrails import LLMRails, RailsConfig

# 3.3 Multi-Agent System (MAS) Orchestration Logic

class SupervisorAgent:
    async def process_input(self, text: str):
        # Call NeMo Guardrails here
        print(f"SUPERVISOR: Checking input '{text}'")
        is_safe = "politics" not in text.lower() # Mock check
        if is_safe:
            return text, True
```

```
            else:
                return "I cannot discuss that topic.", False

    class CaregiverAgent:
        async def generate_response(self, text: str):
            # RAG + LLM Inference
            await asyncio.sleep(0.8)
            return f"Caregiver response to: {text}"

    class CoachAgent:
        async def evaluate_turn(self, text: str):
            # C-LEAR Rubric
            await asyncio.sleep(0.4)
            return "Good empathy."

    async def handle_user_turn(audio_stream, supervisor, caregiver, coach):
        # 1. Transcribe (Mock RIVA call)
        transcribed_text = "User said something about vaccines"

        # 2. Supervisor Check
        sanitized_text, is_safe = await supervisor.process_input(transcribed_text)
        if not is_safe:
            return sanitized_text

        # 3. Parallel Execution
        caregiver_task =
    asyncio.create_task(caregiver.generate_response(sanitized_text))
        coach_task = asyncio.create_task(coach.evaluate_turn(sanitized_text))

        caregiver_response, coach_feedback = await asyncio.gather(caregiver_task,
    coach_task)

        final_response = f"{caregiver_response} [Feedback: {coach_feedback}]"
        return final_response

    # Example Run
    # asyncio.run(handle_user_turn(None, SupervisorAgent(), CaregiverAgent(),
    CoachAgent()))
```

5.0 API Server (FastAPI)

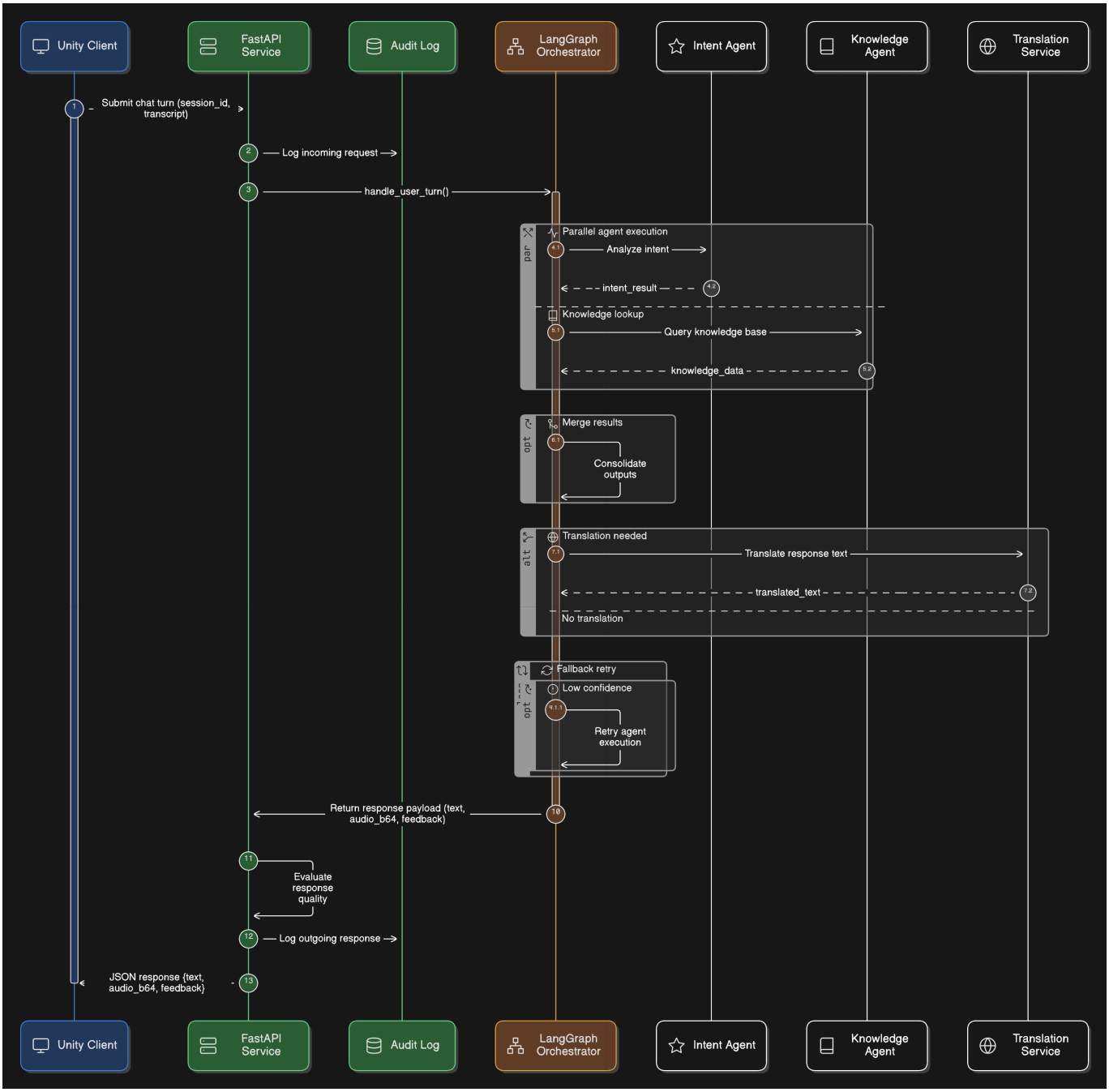
Exposes the Orchestrator to the Unity Client.

5.1 FastAPI Server Implementation

This cell wraps the orchestration logic in a **FastAPI** application to expose it to the Unity client.

- `/v1/chat` **Endpoint**: Accepts a user transcript and session ID. It logs the request for auditing, invokes the orchestration loop, and returns the multi-agent response (Text, Audio, Feedback).
- **Health Check**: A simple `GET /health` endpoint for monitoring service uptime.

5.2 API Server Integration Diagram



API Server Integration: This diagram maps the data flow through the FastAPI application. The Unity Client sends a request to /v1/chat. The server logs the request for auditing, invokes the LangGraph orchestration loop (defined in Section 4), and returns the structured ChatResponse containing text, audio (Base64), and animation cues.

5.3 FastAPI Server with Endpoints

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import uvicorn
import time
import logging

app = FastAPI()
```

```
# 6.1 Configuration & Logging
LOG_FILE = "/blue/jasondeanarnold/SPARCP/logs/audit.log"
logging.basicConfig(filename=LOG_FILE, level=logging.INFO, format='%(asctime)s - %
(message)s')

class ChatRequest(BaseModel):
    session_id: str
    user_transcript: str

class ChatResponse(BaseModel):
    caregiver_text: str
    caregiver_audio_b64: str
    caregiver_animation_cues: dict
    coach_feedback: str

# 6.2 Endpoints
@app.get("/health")
async def health_check():
    return {"status": "ok", "service": "SPARC-P Backend"}

@app.post("/v1/chat", response_model=ChatResponse)
async def chat_endpoint(request: ChatRequest):
    # Audit Log
    logging.info(f"Session: {request.session_id} | User Input:
{request.user_transcript}")

    # Invoke LangGraph
    initial_state = {"transcript": request.user_transcript, "history": [],
"feedback": "", "next_action": "", "final_response": {}}
    result = await app_graph.ainvoke(initial_state)

    response_data = result.get("final_response", {})

    return ChatResponse(
        caregiver_text=response_data.get("text", "Error"),
        caregiver_audio_b64=response_data.get("audio", ""),
        caregiver_animation_cues=response_data.get("cues", {}),
        coach_feedback=result.get("feedback", "")
    )

# To run:
# uvicorn.run(app, host="0.0.0.0", port=8000)
```

6.0 Security and Compliance

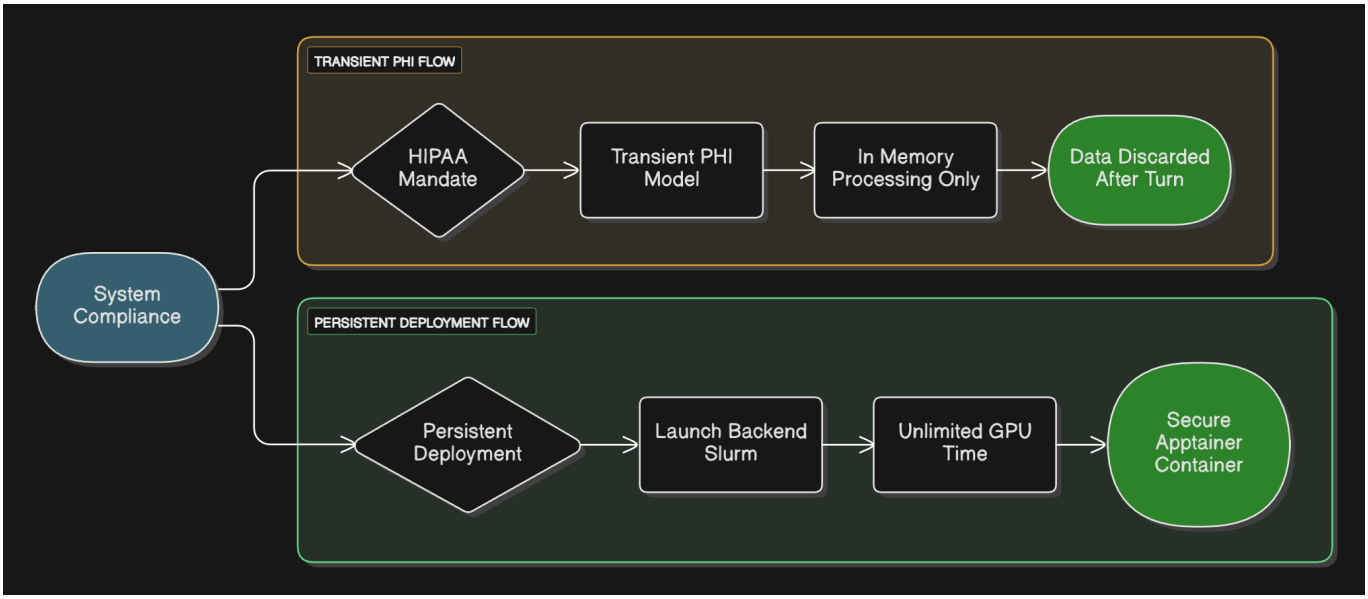
HIPAA Mandate: This system uses a 'Transient PHI' model. User audio and transcripts are processed in-memory and discarded immediately after the conversational turn. No PHI is written to disk.

6.1 Production Deployment Script

To deploy this backend as a persistent service on HiPerGator, we generate a SLURM script (`launch_backend.slurm`). This script:

- Requests a GPU node with `UNLIMITED` time (or 14 days).
- Loads `apptainer`.
- Executes `srun apptainer run` to start the containerized backend service.

6.2 Security and Compliance Diagram



Security and Compliance: This section outlines the security protocols and persistent deployment. It adheres to the HIPAA Mandate using a 'Transient PHI' model, where user data is processed in-memory and immediately discarded. The launch_backend.slurm script ensures the service runs persistently on a secure GPU node.

6.3 SLURM Launch Script Generator

```
# 7.1 SLURM Launch Script Generator

def generate_launch_script():
    script_content = """
#!/bin/bash
#SBATCH --job-name=sparcp-backend
#SBATCH --partition=gpu-a100
#SBATCH --nodes=1
#SBATCH --gpus-per-task=1
#SBATCH --mem=128gb
#SBATCH --time=UNLIMITED

module load apptainer

# Launch Backend Service
srun apptainer run --nv sparcp_backend.sif
    """
    with open("launch_backend.slurm", "w") as f:
        f.write(script_content.strip())
    print("Generated launch_backend.slurm")
```

```
generate_launch_script()
```

Summary

This notebook implements the complete real-time backend for SPARC-P:

1. **NVIDIA Riva Speech Services**: Handles ASR (Speech-to-Text) and TTS (Text-to-Speech) using containerized Riva server on Apptainer.

2. **Safety Rails with NeMo Guardrails**: Implements conversation boundaries to keep discussions focused on HPV vaccination topics while refusing off-topic requests.

3. **Multi-Agent Orchestration**: Uses LangGraph to coordinate three specialized agents:

    - **Supervisor**: Validates input safety
    - **Caregiver**: Generates empathetic responses
    - **Coach**: Evaluates responses against C-LEAR rubric
    - Agents run in parallel to minimize latency

4. **FastAPI Server**: Exposes orchestration logic via REST endpoints:

    - `GET /health`: Service status monitoring
    - `POST /v1/chat`: Main chat endpoint for Unity client

5. **Audit Logging**: Immutable logging to `/blue` tier for HIPAA compliance with transient PHI processing model.

6. **Production Deployment**: SLURM script for persistent service deployment on HiPerGator GPU nodes with unlimited runtime.

The entire system is containerized with Apptainer, ensuring reproducibility and portability across HPC environments.