Orchestrating Agentic State Machines with LangGraph

Learnings from automating clinical documentation (doctor–patient conversation → SOAP note)

Rajeshwari Sah • Conf42

GitHub: rajeshwarisah/medical-scribe-agent

Why this problem matters

Documentation is a major productivity + burnout driver

≈ 2 hours

EHR + desk work per 1 hour patient time

≈ 48%

Physicians reporting ≥1 burnout symptom (2023)

$4.6B

Annual U.S. cost linked to burnout (turnover + hours)

> *“I became a doctor to care for people — not to be a data entry clerk.”*

Goal: shrink documentation time while keeping notes transparent, auditable, and correct.

What we’re building

A multi-agent pipeline that outputs structured SOAP notes

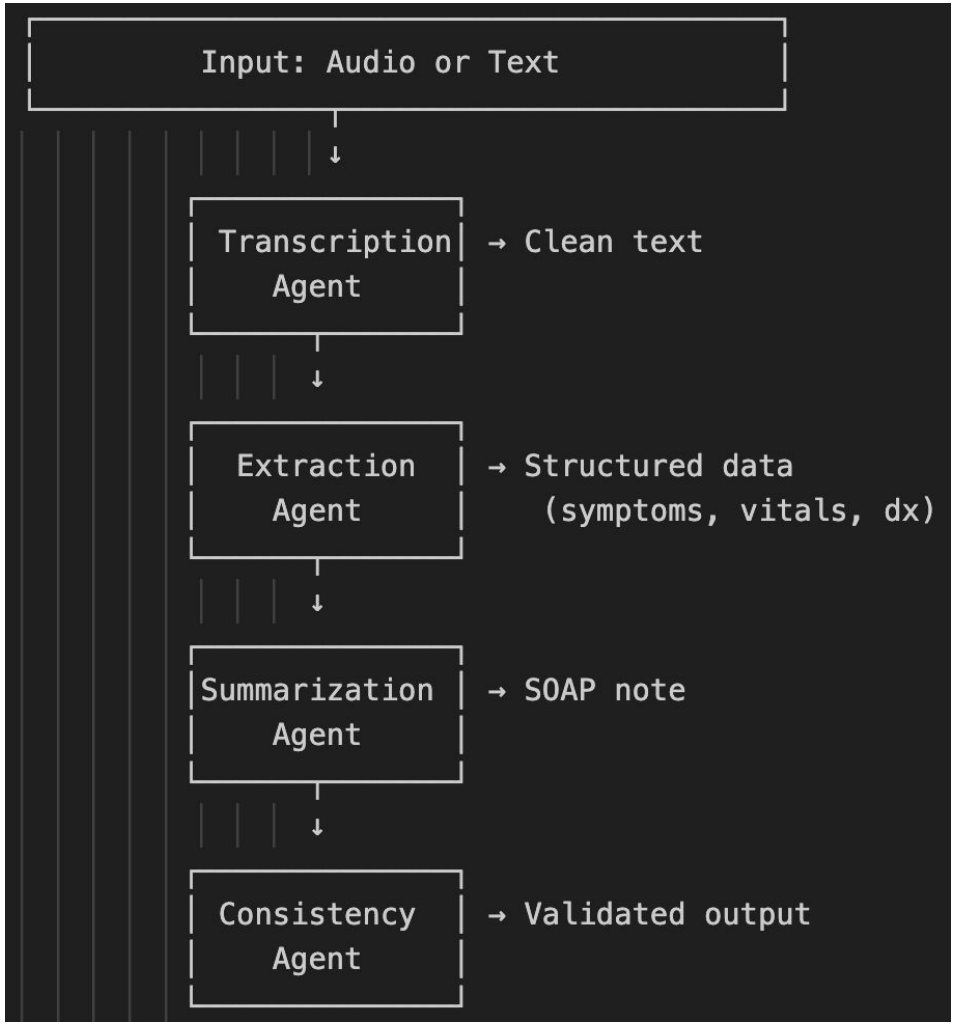
Outcome

A production-minded prototype that:

- Turns dialogue (audio/text) into a clean transcript
- Extracts clinical entities into typed JSON (Pydantic)
- Generates SOAP notes with medical style constraints
- Runs a consistency check to catch omissions & contradictions

Repo + demo commands

- `streamlit run streamlit_app.py`
- `python src/evaluation/run_eval.py`



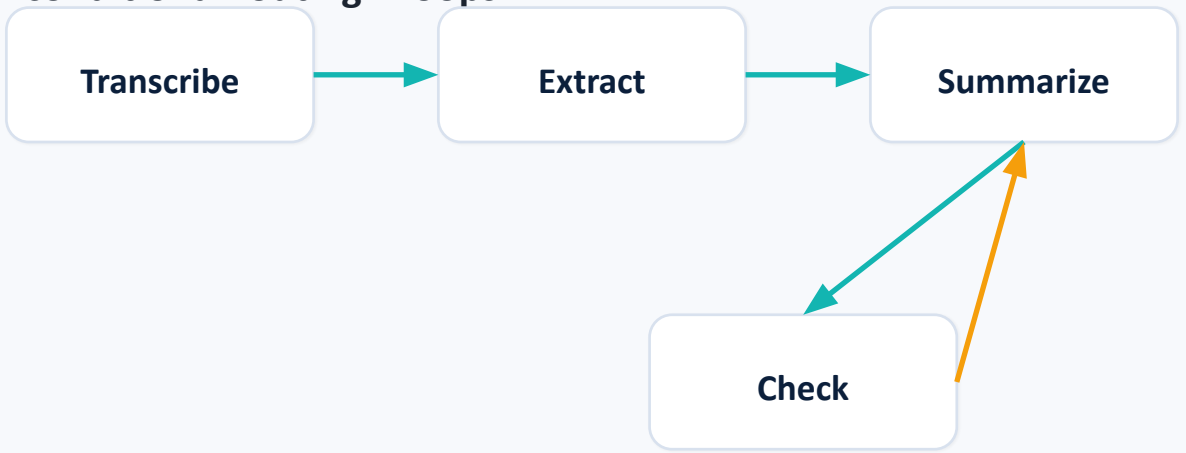
Why LangGraph

State-aware orchestration beats brittle linear chains

Linear chain (fragile)

- No shared memory
- Hard to branch
- No “try again” loops
- Weak audit trail

State machine graph (robust) conditional routing + loops



LangGraph gives you:

- Shared state (one object all agents read/write)
- Conditional edges (stop on error, branch by input type)
- Cycles (self-reflection / revision loops)
- Traceability (logs per node → audit trail)

Shared state = the “clipboard”

A typed data structure that accumulates everything

MedicalScribeState (high-level)

State schema (simplified)

```python
class MedicalScribeState(BaseModel):
    # input
    input_text: str | None
    audio_path: str | None

    # intermediate
    transcript: str | None
    encounter: ClinicalEncounter | None

    # outputs
    soap_note: SoapNote | None
    consistency_report: ConsistencyReport | None

    # audit + errors
    logs: list[dict] = Field(default_factory=list)
    error: str | None
```

Audit trail (per node)

```python
def add_log(self, agent: str, action: str, details: Dict = {}):
    self.logs.append({
        "agent": agent,
        "action": action,
        "timestamp": datetime.now().isoformat(),
        **details
    })
```

Design principle:

Nodes are pure functions:
state in → state out.

This makes testing + debugging straightforward,
and gives you a clean audit trail.

Data contracts with Pydantic

Use schemas to force structure + catch LLM mistakes

Example: ClinicalEncounter schema

```python
from pydantic import BaseModel, Field
from typing import List, Optional

class Symptom(BaseModel):
    name: str = Field(description="Symptom name")
    duration: Optional[str] = Field(description="How long")
    severity: Optional[str] = Field(description="Mild/Moderate/Severe")

class VitalSigns(BaseModel):
    blood_pressure: Optional[str] = None
    heart_rate: Optional[int] = None
    temperature: Optional[float] = None
    oxygen_saturation: Optional[int] = None

class ClinicalEncounter(BaseModel):
    chief_complaint: str
    symptoms: List[Symptom] = Field(default_factory=list)
    vitals: Optional[VitalSigns] = None
    diagnoses: List[str] = Field(default_factory=list)
    medications: List[str] = Field(default_factory=list)

    # Automatic validation!
    # LLM output must match this schema
```

Why this matters

- Schema becomes part of the prompt
- Validation errors are actionable
- Typed objects simplify downstream logic
- Easy to unit-test agent outputs

Pro tip

Field descriptions + examples boost extraction quality more than model size.

The universal agent node pattern

Repeatable structure for reliability + debuggability

```python
def agent_node(state: MedicalScribeState) -> MedicalScribeState:
    """
    Pure function: state in → state out
    Never raises exceptions, always returns state
    """
    # 1. Log start
    state.add_log("agent_name", "started")

    try:
        # 2. Validate inputs
        if not state.required_field:
            state.error = "Missing required field"
            return state

        # 3. Call LLM
        result = call_llm_with_structured_output(
            model=get_llm_client(),
            system_prompt="You are an expert...",
            user_message=f"Process this: {state.required_field}",
            response_model=OutputSchema
        )

        # 4. Update state
        state.output_field = result
        state.add_log("agent_name", "completed", {
            "input_length": len(state.required_field),
            "output_length": len(str(result))
        })

    except Exception as e:
        # 5. Handle errors gracefully
        state.error = f"Agent failed: {str(e)}"
        state.add_log("agent_name", "error", {"message": state.error})

    # 6. Always return state
    return state
```

8 steps

- Log start
- Validate required inputs
- Call LLM (structured)
- Update state
- Log success metadata
- Catch validation errors
- Catch all other errors
- Always return state

This template is the secret sauce.

Agent 1 — Transcription

Audio/text → clean, speaker-attributed transcript

Responsibilities

- Accept either raw text or an audio file path
- Run ASR (Whisper / vendor API / mock) → transcript
- Normalize speaker turns + remove filler
- Write transcript back to shared state

```python
def transcription_node(state):
    if state.audio_path:
        text = asr(state.audio_path)
    else:
        text = state.input_text

    state.transcript = normalize_dialogue(text)
    return state
```

Example output

Doctor: What brings you in today?
Patient: I've had a dry cough for a week…

Doctor: Any fever?
Patient: Around 100°F a few days ago.

Tip: keep transcription deterministic; save creativity for later agents.

Agent 2 — Information extraction

Transcript → typed ClinicalEncounter

Prompt + schema drive extraction

```
CLINICAL_ENCOUNTER_SCHEMA = """
Extract the following from the conversation:
- chief_complaint: Main reason for visit (string)
- symptoms: List of {name, duration, severity}
- vitals: {blood_pressure, heart_rate, temperature, oxygen_saturation}
- diagnoses: List of diagnoses mentioned
- medications: List of medications prescribed or discussed
"""
```

Node implementation

```python
def extraction_node(state: MedicalScribeState) -> MedicalScribeState:
    state.add_log("extraction", "started")

    if not state.transcript:
        state.error = "No transcript to extract from"
        return state

    try:
        encounter = call_llm_with_structured_output(
            model=get_llm_client(),
            system_prompt="You are a medical information extraction specialist...",
            user_message=f"Extract clinical information:\n\n{state.transcript}",
            response_model=ClinicalEncounter
        )

        state.encounter = encounter
        state.add_log("extraction", "completed", {
            "num_symptoms": len(encounter.symptoms),
            "num_diagnoses": len(encounter.diagnoses)
        })
    except Exception as e:
        state.error = f"Extraction failed: {str(e)}"
        state.add_log("extraction", "error", {"message": state.error})

    return state
```

ClinicalEncounter fields (examples)

- chief_complaint
- symptoms: name / duration / severity
- vital_signs: BP, temp, SpO_2…
- diagnoses (+ optional ICD-10)
- medications + test orders
- follow_up instructions

Output is typed JSON → safer downstream summarization.

Agent 3 — Summarization to SOAP

ClinicalEncounter → readable clinician note

Node implementation

```python
def summarization_node(state: MedicalScribeState) -> MedicalScribeState:
    state.add_log("summarization", "started")

    if not state.encounter:
        state.error = "No encounter data to summarize"
        return state

    try:
        # Build prompt with encounter data
        prompt = create_soap_note_prompt(state.encounter)

        # Call LLM to generate SOAP note
        soap_note = call_llm_with_structured_output(
            model=get_llm_client(),
            system_prompt="You are an expert medical scribe...",
            user_message=prompt,
            response_model=SoapNote
        )

        state.soap_note = soap_note
        state.add_log("summarization", "completed")

    except Exception as e:
        state.error = f"Summarization failed: {str(e)}"
        state.add_log("summarization", "error", {"message": state.error})

    return state
```

SOAP format

S — Subjective

O — Objective

A — Assessment

P — Plan

Constraints we enforce:

- No new facts
- Use extracted entities
- Clinician tone
- Include follow-up

Tip: temperature=0 for note generation

Agent 4 — Consistency & self-reflection

Catch hallucinations, omissions, contradictions

Node implementation

```python
def consistency_check_node(state: MedicalScribeState) -> MedicalScribeState:
    state.add_log("consistency", "started")

    if not state.encounter or not state.soap_note:
        state.error = "Missing encounter or SOAP note"
        return state

    try:
        # Build comparison prompt
        prompt = create_consistency_check_prompt(
            state.encounter,
            state.soap_note
        )

        # LLM validates consistency
        report = call_llm_with_structured_output(
            model=get_llm_client(),
            system_prompt="You are a medical QA specialist...",
            user_message=prompt,
            response_model=ConsistencyReport
        )

        state.consistency_report = report

        # If issues found, optionally revise
        if not report.is_consistent and len(report.issues) > 3:
            state.add_log("consistency", "issues_found", {
                "num_issues": len(report.issues)
            })
            # Could trigger revision agent here

        state.add_log("consistency", "completed")

    except Exception as e:
        state.error = f"Consistency check failed: {str(e)}"
        state.add_log("consistency", "error", {"message": state.error})

    return state
```

What it checks

- Every symptom has a plan
- Vitals are reflected correctly
- No invented meds/tests
- Assessment matches extracted diagnoses

Routing option

If issues > threshold →
revise note OR flag for review.

Wiring the LangGraph workflow

Nodes + conditional edges → executable state machine

Graph construction

```python
from langgraph.graph import StateGraph, END

def create_medical_scribe_graph():
    workflow = StateGraph(MedicalScribeState)

    # Add nodes
    workflow.add_node("transcribe", transcription_node)
    workflow.add_node("extract", extraction_node)
    workflow.add_node("summarize", summarization_node)
    workflow.add_node("check_consistency", consistency_check_node)

    # Set entry point
    workflow.set_entry_point("transcribe")

    # Add conditional edges
    workflow.add_conditional_edges(
        "transcribe",
        should_continue_after_transcription,
        {
            "extract": "extract",
            "end": END
        }
    )

    workflow.add_conditional_edges(
        "extract",
        should_continue_after_extraction,
        {
            "summarize": "summarize",
            "end": END
        }
    )

    workflow.add_conditional_edges(
        "summarize",
        should_continue_after_summarization,
        {
            "check_consistency": "check_consistency",
            "end": END
        }
    )

    workflow.add_edge("check_consistency", END)

    return workflow.compile()
```

Conditional routing

```python
def should_continue_after_extraction(state: MedicalScribeState):
    if state.error:
        return "end"
    if state.encounter:
        return "summarize"
    return "end"
```

Key idea

After each node, a small router decides:

- continue?
- which node next?
- or END?

Debuggable by design

Optional: retrieval grounding

Map diagnoses to ICD-10/SNOMED and cite evidence

Where grounding fits

Implementation sketch

Grounding node (pseudo)

```python
def coding_node(state: MedicalScribeState) -> MedicalScribeState:
    if not state.encounter or not state.encounter.diagnoses:
        return state

    # 1) retrieve relevant ontology entries
    hits = icd10_or_snomed_search([d.condition for d in state.encounter.diagnoses])

    # 2) attach best codes + short evidence
    state.encounter = attach_codes(state.encounter, hits)
    state.add_log("coding", "completed", {"n_hits": len(hits)})
    return
```

Measuring quality

Extraction accuracy + summarization fidelity + readability

Entity-level F1 (structured extraction)

```python
def calculate_entity_f1(predicted: ClinicalEncounter,
                        gold: ClinicalEncounter) -> float:
    """
    Precision = TP / (TP + FP)
    Recall = TP / (TP + FN)
    F1 = 2 * (Precision * Recall) / (Precision + Recall)
    """
    pred_entities = extract_all_entities(predicted)
    gold_entities = extract_all_entities(gold)

    tp = len(pred_entities & gold_entities)  # Intersection
    fp = len(pred_entities - gold_entities)  # Predicted but not in gold
    fn = len(gold_entities - pred_entities)  # In gold but not predicted

    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    return f1
```

- Precision/Recall/F1 on symptoms, vitals, meds…
- Best for “did we extract the right facts?”

Summarization metrics

- ROUGE-L (overlap vs reference SOAP)
- Readability (e.g., Flesch-Kincaid)
- Consistency issues per note

Practical tip:

Track metrics per agent — then improve the weakest link.

Suggested eval command:

```
python src/evaluation/run_eval.py
```

Production considerations

Trust, privacy, and observability matter more than clever prompts

Safety & compliance

- Treat all inputs as PHI; minimize data retention
- Human-in-the-loop review before charting
- Clear disclaimers + intended-use boundaries
- Prompt injection defenses (tool + retrieval hardening)

Reliability

- Schema validation + retries on failure
- Temperature=0 for extraction + summaries
- Fallback paths on missing audio/transcript
- Consistency checks to catch omissions

Observability

- Per-node logs (inputs/outputs + timing)
- Store run metadata for audits
- Dashboards for validation errors
- Golden test set + regression metrics

Cost & latency

- Cache transcription + retrieval results
- Batch evaluation offline
- Choose models per node (cheap for routing)
- Stream partial results to UI

Quick start

Run the Streamlit demo and inspect the state + logs

1) Install + configure

Setup

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
cp .env.example .env  # add API key
```

2) Launch demo UI

What to look for in the UI

- Transcript, extracted JSON, SOAP note, and consistency report
- Execution log (per-agent timestamps + actions)
- Failure mode: inspect `state.error`

Tip: record your screen while running one sample so viewers see the full state flow.

Key takeaways

1) Use a shared, typed state as your contract.
2) Build agents as pure functions with logs + errors.
3) Orchestrate with conditional routing + reflection loops.
4) Measure quality per agent — then iterate surgically.

Repo: rajeshwarisah/medical-scribe-agent

Thank you! — Questions / follow-ups welcome