🎯 HYBRID RAG PROJECT - COMPLETE INTERVIEW PREPARATION GUIDE

For: Forestrat.AI Python Backend Developer (3-5 Years) Role
Date: October 31, 2025

📋 TABLE OF CONTENTS

🏗️ PROJECT OVERVIEW & ARCHITECTURE

What is Hybrid RAG?

CLEAR, CONCISE ANSWER (30 seconds): "Hybrid RAG is a document Q&A system that solves a critical problem with conventional RAG: handling tables in PDFs. Conventional RAG flattens tables into text, losing accuracy. Hybrid RAG digitizes tables into PostgreSQL, uses SQL for structured queries, and combines with vector search for unstructured text - giving 100% accurate results for tabular data while maintaining semantic search for text."

Project Architecture:

```
PDF Upload → PDF Processor (pdfplumber + Gemini)
             ├─→ Extract Text → Chunk → Embed (Gemini) → Store (Pinecone)
             └─→ Extract Tables → Schema Inference (Gemini) → Store
(PostgreSQL)

Query → Manager Agent (LangGraph)
        ├─→ Classify: text/table/both
        ├─→ Route to:
        │   ├─→ RAG Agent (Pinecone + Gemini)
        │   └─→ Table Agent (PostgreSQL + Gemini SQL generation)
        └─→ Combiner Agent → Final Response
```

Tech Stack:

- **Backend:** FastAPI (async/await)
- **LLM:** Google Gemini (gemini-1.5-flash, gemini-2.0-flash)
- **Vector DB:** Pinecone (768-dim embeddings)
- **Relational DB:** PostgreSQL (Supabase)
- **Orchestration:** LangGraph (state machine)
- **Frontend:** Streamlit

💻 CORE PYTHON QUESTIONS

1. Python's Memory Management & GIL

Q: How does Python's Global Interpreter Lock (GIL) affect concurrency?

ANSWER (Show Code):

```python
# From orchestrator.py - we use async/await to bypass GIL
class Orchestrator:
    def process_query(self, query: str, pdf_uuid: Optional[str] = None):
        # Single-threaded event loop - no GIL contention
        # When one request awaits I/O, event loop switches to another
        if self.use_manager:
            response = self.manager_agent.process_query(query, pdf_uuid)
            return response
```

Key Points:

- GIL allows only ONE thread to execute Python bytecode at a time
- Our FastAPI uses async/await (single thread + event loop) - NO GIL problem
- We scale with multiple workers (uvicorn --workers 4)
- GIL released during I/O operations (database, Gemini API calls)
- Python 3.13+ has optional GIL (PEP 703) but we don't need it

Real-World Impact: "In our chat.py, when 1000 concurrent requests hit /answer endpoint, FastAPI handles them efficiently because:

1. async def endpoints don't block - they await I/O
2. While one request awaits Pinecone query, event loop handles another
3. GIL doesn't matter because we're I/O bound, not CPU bound"

---

2. How FastAPI Handles Concurrency

Q: Is FastAPI single or multi-threaded? How does it achieve concurrency?

ANSWER (Point to Code):

```python
# From chat.py - all endpoints are async
@router.post("/answer", response_model=AnswerResponse)
```

```
async def answer_question(request: QueryRequest, fastapi_request: 
Request):
    # This is an async function - runs in event loop
    query = request.query.strip()

    # Delegating to orchestrator (also async internally)
    result = orchestrator.process_query(query, pdf_uuid)

    return {
        "answer": result.get("answer"),
        "success": True
    }
```

Technical Explanation:

- FastAPI is **single-threaded by default** with async/await
- Uses **event loop** (Starlette/asyncio) for concurrency
- One worker = one event loop = single thread handling many requests
- When async function hits `await`, event loop switches to another request

Scaling Strategy:

```
# From app.py
uvicorn app:app --host 0.0.0.0 --port 8000 --workers 4
```

- 4 workers = 4 separate processes = 4 event loops
- Each worker handles concurrent requests via async/await
- True parallelism across CPU cores (separate processes, separate GILs)

---

3. Asyncio Deep Dive

Q: What is asyncio? When would you use async vs threads?

ANSWER (Show Our Usage):

```
# From rag_agent.py - synchronous approach
class ChatbotAgent:
    def answer_question(self, question: str, pdf_uuid: str = None):
        # Blocking operations (but event loop can switch during I/O)
        results = self.vectorstore.similarity_search_with_score(question, 
k=5)
        response = self.llm.generate_content(prompt)
        return {"answer": response.text}
```

When to Use What:

Use Case	Technology	Example from Project
I/O-bound (API calls, DB queries)	**async/await**	All FastAPI endpoints
CPU-bound (heavy computation)	**multiprocessing**	Not needed (LLM does heavy work)
Legacy sync code	**ThreadPoolExecutor**	Could wrap sync DB operations

Real Example - If We Had CPU-Intensive Task:

```
import asyncio
from concurrent.futures import ProcessPoolExecutor

async def process_large_pdf(pdf_path):
    loop = asyncio.get_event_loop()
    with ProcessPoolExecutor() as pool:
        # Run CPU-intensive task in separate process (bypasses GIL)
        result = await loop.run_in_executor(pool, heavy_pdf_processing,
pdf_path)
    return result
```

---

4. Design Patterns in Hybrid RAG

Q: Describe a design pattern you've used in Python and why.

ANSWER (Show Multiple Patterns):

A. Singleton Pattern (Service Initialization)

```
# From __init__.py - single orchestrator instance per app
def create_app():
    app = FastAPI()

    # Initialize services once (Singleton pattern)
    chatbot_agent = ChatbotAgent()  # Single instance
    manager_agent = ManagerAgent(chatbot_agent=chatbot_agent)
    orchestrator = Orchestrator(
        chatbot_agent=chatbot_agent,
        manager_agent=manager_agent
    )

    # Store in app.state (accessible to all requests)
    app.state.orchestrator = orchestrator
    return app
```

Why: ML models and DB connections should be loaded once (memory efficient)

B. Strategy Pattern (Query Routing)

```python
# From orchestrator.py - different strategies for different query types
class Orchestrator:
    def process_query(self, query: str, pdf_uuid: Optional[str] = None):
        if self.use_manager:
            # Strategy 1: LangGraph-based routing
            return self.manager_agent.process_query(query, pdf_uuid)
        else:
            # Strategy 2: Direct RAG
            return self.chatbot_agent.answer_question(query, pdf_uuid)
```

Why: Flexible query processing - can switch strategies without changing client code

C. Factory Pattern (Dynamic Table Creation)

```python
# From pdf_processor.py - dynamic Pydantic models
def _create_pydantic_model(self, schema_info: TableSchema) -> type:
    type_mapping = {
        "string": (str, Field(default="")),
        "integer": (Optional[int], Field(default=None)),
        "float": (Optional[float], Field(default=None))
    }

    # Dynamically create model class
    DynamicModel = type(
        f"{schema_info.table_name}Model",
        (BaseTableModel,),
        model_attrs
    )
    return DynamicModel
```

Why: Tables have dynamic schemas - can't hardcode Pydantic models

D. State Machine Pattern (LangGraph)

```python
# From manager_agent.py - workflow as state machine
def _create_workflow(self) -> StateGraph:
    workflow = StateGraph(AgentState)

    # Define states (nodes)
    workflow.add_node("manager", self._manager_node)
    workflow.add_node("table", self._table_node)
    workflow.add_node("rag", self._rag_node)
    workflow.add_node("combiner", self._combiner_node)

    # Define transitions (edges)
    workflow.add_conditional_edges("manager", self._decide_route, {
        "table_only": "table",
        "rag_only": "rag",
        "both": "table"
    })
    return workflow.compile()
```

Why: Complex agent orchestration with conditional routing

⚡ CONCURRENCY & PERFORMANCE

5. Performance Profiling (NOT Debugging)

Q: How would you debug performance issues in Python?

CRITICAL: Don't talk about pdb (that's debugging). Talk about **profiling**.

ANSWER (Show Tools):

```python
# 1. cProfile - Function-level profiling
import cProfile
import pstats

def profile_endpoint():
    profiler = cProfile.Profile()
    profiler.enable()

    # Run the code
    orchestrator.process_query("Test question")

    profiler.disable()
    stats = pstats.Stats(profiler)
    stats.sort_stats('cumulative')
    stats.print_stats(10)  # Top 10 slowest functions
```

Tools I Would Use:

Tool	Use Case	Example
cProfile	Function timing	`cProfile.run('orchestrator.process_query(query)')`
line_profiler	Line-by-line timing	`@profile` decorator on slow functions
memory_profiler	Memory usage	Find memory leaks in PDF processing
py-spy	Production profiling	No code changes, attach to running process

Tool	Use Case	Example
FastAPI Middleware	Request timing	Track endpoint latency

Real-World Example in Our Project:

```python
# Add to __init__.py for production monitoring
import time
from fastapi import Request

@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)
    logger.info(f"{request.url.path} took {process_time:.2f}s")
    return response
```

---

6. Real-World Scalability Issues

Q: Real-world performance and scalability issues with FastAPI?

ANSWER (Based on Our Project):

Issue 1: Slow PDF Upload Processing

```python
# PROBLEM: Synchronous PDF processing blocks event loop
def process_pdf_upload(file):
    pdf_processor.extract_and_store_content(temp_file_path)  # BLOCKS!
```

SOLUTION: Use background tasks

```python
from fastapi import BackgroundTasks

@router.post("/uploadpdf")
async def upload_pdf(file: UploadFile, background_tasks: BackgroundTasks):
    # Save file immediately
    temp_path = save_temp_file(file)

    # Process in background
    background_tasks.add_task(process_pdf_in_background, temp_path)

    return {"status": "processing", "message": "PDF upload started"}

def process_pdf_in_background(path):
    # Heavy processing doesn't block API
    pdf_processor.extract_and_store_content(path)
```

Issue 2: Database Connection Pool Exhaustion

```python
# PROBLEM: Each request creates new connection
def _execute_sql_query(self, sql_query):
    conn = psycopg2.connect(...)  # New connection each time!
```

SOLUTION: Use connection pooling

```python
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

engine = create_engine(
    database_url,
    poolclass=QueuePool,
    pool_size=20,          # Max 20 connections
    max_overflow=10,       # Allow 10 extra if needed
    pool_timeout=30,       # Wait 30s for available connection
    pool_pre_ping=True     # Test connection before use
)
```

Issue 3: Slow LLM API Calls

```python
# PROBLEM: Sequential API calls
table_response = table_agent.process_query(query)  # Wait...
rag_response = rag_agent.process_query(query)      # Wait...
```

SOLUTION: Parallel API calls

```python
import asyncio

async def process_query_parallel(query):
    # Run both agents concurrently
    table_task = asyncio.create_task(table_agent.process_async(query))
    rag_task = asyncio.create_task(rag_agent.process_async(query))

    # Wait for both to complete
    table_response, rag_response = await asyncio.gather(table_task,
rag_task)
    return combiner_agent.combine(table_response, rag_response)
```

🚀 FASTAPI DEEP-DIVE

7. Pydantic Validation

Q: How does FastAPI leverage Pydantic?

ANSWER (Show Our Code):

```
# From models.py
class QueryRequest(BaseModel):
    query: str                      # Type validation
    pdf_uuid: Optional[str] = None  # Optional field

# From chat.py
@router.post("/answer", response_model=AnswerResponse)
async def answer_question(request: QueryRequest, fastapi_request:
Request):
    # FastAPI automatically:
    # 1. Validates incoming JSON matches QueryRequest
    # 2. Raises 422 if validation fails
    # 3. Converts JSON to QueryRequest object
    query = request.query.strip()  # Access validated field
```

Advanced Pydantic Example from Our Project:

```
# From pdf_processor.py - Dynamic model with custom validators
def _create_pydantic_model(self, schema_info: TableSchema):
    from pydantic import BaseModel, Field, field_validator

    # Dynamic field annotations
    annotations = {
        "revenue": Optional[float],
        "profit_margin": Optional[float]
    }

    # Custom validator for currency parsing
    @field_validator('revenue', mode='before')
    @classmethod
    def validate_currency(cls, v):
        if isinstance(v, str):
            # "$1,234.56" → 1234.56
            cleaned = re.sub(r'[\$€£¥₹,]', '', v)
            return float(cleaned) if cleaned else None
        return v

    return create_model('DynamicTable', **annotations)
```

Benefits:

1. **Auto validation** - FastAPI raises 422 for invalid input
2. **Auto docs** - Swagger UI shows required fields
3. **Type safety** - IDE autocomplete works
4. **Data parsing** - Converts JSON types automatically

---

8. Dependency Injection

Q: Explain dependency injection in FastAPI.

ANSWER (How We Use It):

```python
# From chat.py - get orchestrator from app state
@router.post("/answer")
async def answer_question(request: QueryRequest, fastapi_request: Request):
    # Dependency: orchestrator from app.state
    orchestrator = getattr(fastapi_request.app.state, 'orchestrator', None)

    if orchestrator is None:
        raise HTTPException(status_code=503, detail="Service unavailable")

    result = orchestrator.process_query(request.query, request.pdf_uuid)
    return result
```

Better Approach with Dependency Function:

```python
# Create dependency
def get_orchestrator(request: Request):
    orchestrator = request.app.state.orchestrator
    if not orchestrator:
        raise HTTPException(status_code=503, detail="Service unavailable")
    return orchestrator

# Use dependency
@router.post("/answer")
async def answer_question(
    request: QueryRequest,
    orchestrator: Orchestrator = Depends(get_orchestrator)
):
    # orchestrator automatically injected
    result = orchestrator.process_query(request.query, request.pdf_uuid)
    return result
```

Why It's Useful:

- **Testability:** Mock orchestrator in tests
- **Reusability:** Same dependency in multiple endpoints

- **Clean code:** No repetitive app.state access

---

9. Background Tasks & Async Endpoints

Q: How do you handle background tasks in FastAPI?

ANSWER (Current vs Improved):

Current Approach:

```
# From upload_pdf.py - blocks until complete
async def process_pdf_upload(file: UploadFile):
    processing_result = pdf_processor.extract_and_store_content(temp_path)
    text_chunks_stored = embedding_service.store_text_embeddings(...)
    return {"success": True, "tables_stored": ...}
```

Improved with Background Tasks:

```
from fastapi import BackgroundTasks

@router.post("/uploadpdf")
async def upload_pdf(
    file: UploadFile,
    background_tasks: BackgroundTasks
):
    # Save file
    temp_path = await save_file(file)
    task_id = str(uuid.uuid4())

    # Process in background
    background_tasks.add_task(
        process_pdf_background,
        task_id,
        temp_path
    )

    return {
        "task_id": task_id,
        "status": "processing",
        "message": "PDF processing started"
    }

def process_pdf_background(task_id, path):
    try:
        # Heavy processing
        result = pdf_processor.extract_and_store_content(path)
        # Store result in cache/DB
        redis.set(f"task:{task_id}", json.dumps(result))
    except Exception as e:
        redis.set(f"task:{task_id}", json.dumps({"error": str(e)}))
```

Status Check Endpoint:

```python
@router.get("/uploadstatus/{task_id}")
async def check_upload_status(task_id: str):
    result = redis.get(f"task:{task_id}")
    if not result:
        return {"status": "processing"}
    return {"status": "complete", "result": json.loads(result)}
```

---

10. API Versioning

Q: How do you version APIs in FastAPI?

ANSWER (Best Practice):

```python
# Create versioned routers
from fastapi import APIRouter

# Version 1
v1_router = APIRouter(prefix="/api/v1", tags=["v1"])

@v1_router.post("/answer")
async def answer_v1(request: QueryRequest):
    # Legacy behavior
    return orchestrator.process_query(request.query)

# Version 2 - with PDF UUID support
v2_router = APIRouter(prefix="/api/v2", tags=["v2"])

@v2_router.post("/answer")
async def answer_v2(request: QueryRequest):
    # New behavior with UUID filtering
    return orchestrator.process_query(request.query, request.pdf_uuid)

# Register both versions
app.include_router(v1_router)
app.include_router(v2_router)
```

URL Structure:

- `POST /api/v1/answer` - Legacy clients
- `POST /api/v2/answer` - New clients with UUID support

Benefits:

- Backward compatibility

- Clear deprecation path
- Separate documentation per version

🧠 LANGGRAPH & AGENT ARCHITECTURE

11. Why LangGraph? What Problem Does It Solve?

ANSWER:

Without LangGraph (Simple If-Else):

```
def process_query(query):
    if "table" in query or "count" in query:
        return table_agent.process(query)
    else:
        return rag_agent.process(query)
```

Problems:

- Hard to maintain complex routing logic
- Can't handle queries needing both agents
- No state management
- Difficult to debug
- Can't visualize flow

With LangGraph (State Machine):

```
# From manager_agent.py
class AgentState(BaseModel):
    query: str
    needs_table: bool = False
    needs_rag: bool = False
    table_response: str = ""
    rag_response: str = ""

workflow = StateGraph(AgentState)
workflow.add_node("manager", analyze_query)      # Decide routing
workflow.add_node("table", query_database)        # SQL agent
workflow.add_node("rag", query_vectors)           # RAG agent
workflow.add_node("combiner", merge_responses)    # Combine results

workflow.add_conditional_edges("manager", decide_route, {
    "table_only": "table",
    "rag_only": "rag",
    "both": "table"  # Then to RAG, then to combiner
})
```

Benefits:

1. **Visual workflow** - can export as graph
2. **State tracking** - every node updates state
3. **Complex routing** - conditional edges
4. **Debuggable** - see state at each step
5. **Testable** - test individual nodes

---

12. Manager Agent Deep-Dive

Q: How does the Manager Agent decide routing?

ANSWER (Show Code & Logic):

```python
# From manager_agent.py - Manager Node
def _manager_node(self, state: AgentState):
    # Load available table schemas
    schema_info = self._load_table_schema(state.pdf_uuid)

    system_prompt = f"""
    AVAILABLE TABLES: {schema_info}

    Analyze query: "{state.query}"

    Return JSON:
    {{
        "status": "rag" | "table" | "both",
        "table_sub_query": "SQL-answerable question",
        "rag_sub_query": "Knowledge-based question"
    }}
    """

    # Use Gemini to analyze
    response = self.llm.invoke([SystemMessage(system_prompt)])
    result = json.loads(response.content)

    # Update state
    if result["status"] == "table":
        state.needs_table = True
        state.table_sub_query = result["table_sub_query"]
    elif result["status"] == "rag":
        state.needs_rag = True
        state.rag_sub_query = result["rag_sub_query"]
    elif result["status"] == "both":
        state.needs_table = True
        state.needs_rag = True
        state.table_sub_query = result["table_sub_query"]
        state.rag_sub_query = result["rag_sub_query"]

    return state
```

Example Decision:

```
Query: "How many times did Brazil win and what leagues exist in Europe?"

Manager Analysis:
{
    "status": "both",
    "table_sub_query": "How many times did Brazil win according to the
data?",
    "rag_sub_query": "What are the major football leagues in Europe?"
}

Workflow:
1. Manager → decides "both"
2. Table Agent → executes SQL → "Brazil won 5 times"
3. RAG Agent → searches Pinecone → "European leagues include..."
4. Combiner → merges → "Brazil won 5 times. European leagues include..."
```

💾 DATABASE & DATA PIPELINE

13. PostgreSQL vs DuckDB - When to Use What?

Q: Difference between relational (PostgreSQL) and analytical (DuckDB) workloads.

ANSWER:

Feature	PostgreSQL (OLTP)	DuckDB (OLAP)
Use Case	Transactional (CRUD)	Analytical (aggregations)
Write Pattern	Frequent small writes	Bulk loads
Read Pattern	Row-oriented (specific records)	Column-oriented (full scans)
Query Type	`SELECT * WHERE id=5`	`SELECT SUM(revenue) GROUP BY year`
Our Project	✅ Store table rows	❌ Not needed (small data)

Why We Use PostgreSQL:

```python
# From table_agent.py
def _execute_sql_query(self, sql_query):
    conn = psycopg2.connect(
        host=os.getenv('DATABASE_HOST'),
        database=os.getenv('DATABASE_NAME')
    )
    cursor.execute(sql_query)
    results = cursor.fetchall()
```

Use Cases:

- Store PDF table data (row-by-row inserts)
- Query specific rows (World Cup match by year)
- ACID transactions (data consistency)

When I'd Use DuckDB:

- Analyzing millions of rows locally
- Ad-hoc analytics without server
- Querying Parquet files directly
- Data science notebooks

---

14. Schema Evolution & Management

Q: How do you handle schema evolution in data systems?

ANSWER (Show Our Approach):

```python
# From pdf_processor.py - Dynamic schema inference with Gemini
def _query_gemini_for_schema(self, table_data, pdf_uuid):
    prompt = """
    Analyze table and provide schema:
    {
        "table_name": "pdf_abc123_financial_summary",
        "table_schema": {
            "year": "integer",
            "revenue": "currency",
            "profit_margin": "percentage"
        }
    }
    """

    schema_data = gemini.generate_content(prompt)

    # Store schema for future reference
    self.schemas[table_name] = {
        "schema": schema_data["table_schema"],
        "pdf_uuid": pdf_uuid,
        "created_at": timestamp
    }
    self._save_schemas()  # Save to table_schema.json
```

Schema Storage:

```json
// src/backend/utils/table_schema.json
{
    "pdf_abc123_world_cup": {
        "schema": {
            "year": "integer",
            "home_team": "string",
            "away_team": "string",
            "home_score": "integer",
            "winner": "string"
        },
        "description": "FIFA World Cup match results",
        "pdf_uuid": "abc123",
        "created_at": "2025-10-31T10:00:00"
    }
}
```

Handling Schema Changes:

```python
# If PDF uploaded with similar but different table
def detect_schema_drift(old_schema, new_data):
    # Gemini compares schemas
    if has_new_columns(old_schema, new_data):
        # Add columns with ALTER TABLE
        alter_table_add_columns(new_columns)
    if has_type_conflicts(old_schema, new_data):
        # Create new table version
        create_table_v2(new_schema)
```

🎤 INTERVIEW INTRODUCTION STRATEGY

The Perfect Introduction (2 Minutes Max)

OPENING (30 seconds): "I'm a Python backend developer with 3-5 years of experience building scalable data systems. I've worked at [IIT/Walmart], and I'm currently freelancing on data-intensive projects. Due to personal reasons, I relocated, which is why I'm seeking new opportunities now."

PROJECT SHOWCASE (60 seconds): "Recently, I built a Hybrid RAG document Q&A system that solves a critical problem. Let me show you..."

[SHARE DEPLOYED LINK + ARCHITECTURE DIAGRAM]

"Conventional RAG flattens tables in PDFs, losing accuracy. My system:

1. Digitizes tables into PostgreSQL with Gemini-powered schema inference
2. Uses Pinecone for text embeddings
3. Routes queries intelligently via LangGraph - table queries get SQL, text queries get RAG
4. Achieves 100% accuracy on tabular data

For example, asking 'How many World Cup matches ended in a draw?' - conventional RAG guesses, Hybrid RAG queries the database and returns exact count: 14."

ARCHITECTURE HIGHLIGHT (30 seconds): "The architecture uses:

- FastAPI async/await for concurrency
- LangGraph state machine for agent orchestration

- Multi-agent system with Manager, Table, RAG, and Combiner agents
- PostgreSQL + Pinecone dual storage
- Gemini for LLM and embeddings"

[SHOW CODE ON SCREEN]

🚨 COMMON PITFALLS TO AVOID

❌ WHAT NOT TO DO (From interview_imp.txt)

1. **Too Verbose**
   - ❌ Rambling about theory
   - ✅ "Let me show you in my code..."
2. **Not Showing Code**
   - ❌ "I know how Pydantic works..."
   - ✅ "Here's my models.py - BaseModel validates QueryRequest..."
3. **Saying "Uh" or "I Think"**
   - ❌ "Uh, I think GIL is, um..."
   - ✅ "GIL prevents parallel execution. Let me show how we handle it..."
4. **Debugging Instead of Profiling**
   - ❌ "I use pdb to debug..."
   - ✅ "I use cProfile for performance, line_profiler for bottlenecks..."
5. **Generic Answers**
   - ❌ "FastAPI is fast and supports async..."
   - ✅ "In our chat.py, async def answer_question uses await..."

📝 QUICK REFERENCE CHEAT SHEET

30-Second Answers with Code Location

Question	Answer Location in Code	Key Point
FastAPI concurrency	`chat.py:83` `async def`	Event loop, single-threaded, async/await
GIL impact	`orchestrator.py:35`	We use async (I/O bound) - GIL doesn't matter
Pydantic usage	`models.py:13` `QueryRequest`	Auto validation, type safety
Design pattern	`__init__.py:79` Singleton	Services initialized once

Question	Answer Location in Code	Key Point
LangGraph routing	`manager_agent.py:76` workflow	State machine with conditional edges
Performance profiling	Add middleware in `__init__.py`	cProfile, line_profiler, not pdb
Database choice	`table_agent.py:239` psycopg2	PostgreSQL for OLTP, not OLAP
API versioning	Router with prefix `/api/v1`	Separate routers per version

---

🎯 FINAL PRE-INTERVIEW CHECKLIST

30 Minutes Before Interview

- [ ] Open deployed app in browser (demo ready)
- [ ] Open VS Code with codebase (code ready to show)
- [ ] Have architecture.png visible
- [ ] Test screen share (can you navigate code smoothly?)
- [ ] Practice Ctrl+Shift+F (search "BaseModel", "async def", "workflow")

During Interview - Remember

1. **Always show code** - "Let me show you..." (Ctrl+Shift+F)
2. **30-60 seconds per answer** - concise, then ask "Want more detail?"
3. **Look at camera** - not just screen
4. **Use project examples** - every answer ties to your code
5. **Ask for thinking time** - "Let me collect my thoughts..."

🚀 ADVANCED TOPICS

If They Ask About Production Deployment

Current Setup:

- Backend: Uvicorn + Gunicorn (4 workers)
- Database: Supabase (managed PostgreSQL)
- Vector DB: Pinecone (serverless)
- Frontend: Streamlit

Production Improvements:

```
# Load balancing with Gunicorn
gunicorn app:app \
    --workers 4 \
    --worker-class uvicorn.workers.UvicornWorker \
    --bind 0.0.0.0:8000 \
    --timeout 120 \
    --access-logfile logs/access.log
```

Monitoring:

```python
# Add Prometheus metrics
from prometheus_client import Counter, Histogram

request_count = Counter('app_requests_total', 'Total requests')
request_duration = Histogram('app_request_duration_seconds', 'Request
duration')

@app.middleware("http")
async def metrics_middleware(request, call_next):
    request_count.inc()
    with request_duration.time():
        response = await call_next(request)
    return response
```

🎬 CONCLUSION

You now have:

1. ✅ Complete project understanding
2. ✅ Answers to all interview questions
3. ✅ Code examples for every topic
4. ✅ Clear, concise explanations (30-60 seconds)
5. ✅ Strategy to showcase your work

Remember: "The moment you show deployed application, the moment you show code, that's what I'm looking at." - Your Mentor

---

GOOD LUCK! 🚀

Practice these answers, navigate your code confidently, and you'll demonstrate complete, practical knowledge of this project.