Revolutionizing Project Management with AI Agents and LangGraph

Yotam Braun
Follow

Published in
Towards AI
13 min read
Oct 10, 2024

Being A Project Manager Sucks, But Why: A day in the life of a project manager revolves around task statuses, time entries, scattered files, and never-ending chats. More often than not, they are seen standing over the shoulders of team members asking them about status updates, why time was not logged, and so on.

Why Does It Suck To Be A Project Manager? (It Doesn't Have To Be That Way
Life as a project manager can be hard. To succeed, you must navigate the challenges that come your way like a pro
kashyapvartika.medium.com

Improving project management with the **AI-driven project management system**, where **RAG** and **LangGraph** could come in handy. This tutorial demonstrates the key components of the system and explains how they can help solve challenges in project management.

. . .

Project Management Problems

Managing projects can sometimes feel overwhelming, like solving a complex puzzle under pressure. Here are the core challenges:

1. **Information Overload**: Project management involves a massive amount of emails, documents, and meeting notes, and finding the right information at the right time can be very challenging.

2. **Inefficient Task Creation**: Manually creating and prioritizing tasks is very time-consuming and may lead to errors.
3. **Team Coordination Issues**: Assigning tasks effectively depends heavily on the team members' skills a availability.

These difficulties can lead to missed deadlines, blown budgets, and burned-out teams.

. . .

AI Solution

Imagine a system that:

- **Automatically generates and prioritizes tasks** using AI insights and historical data.
- **Enhances team collaboration** by suggesting the best people for each task.
- **Adapts workflows on the fly**, handling changes smoothly.
- **Gives you practical insights** through clear, straightforward reports.

Now, let's explore how it all works by looking at the main agents in the system.

https://github.com/yotambraun/Project_Management_System_with_RAG/tree/main

. . .

https://github.com/yotambraun/Project_Management_System_with_RAG/tree/main

1. TaskAgent: Automating Task Creation

The Struggle with Manual Task Creation

In traditional project management, creating tasks often involves:

- **Brainstorming**: Coming up with tasks that cover all aspects of a project.
- **Estimating**: Guessing how long each task will take.
- **Skill Matching**: Figuring out what skills each task needs.

This isn't just time-consuming; it also depends a lot on the project manager's experience and memory.

The TaskAgent Solution

The `TaskAgent` automates this process by:

- **Leveraging AI**: Uses a language model to generate tasks based on project descriptions.
- **Utilizing Historical Data**: Considers similar tasks from past projects to ensure nothing is overlooked.
- **Incorporating Team Skills**: Aligns tasks with the available skills within the team.

How It Works

- **Input**: You provide a simple task description and the project ID.
- **Data Retrieval**: The agent gets similar tasks and project details using the `Retriever` class.
- **AI Generation**: It puts together a prompt and uses the AI model to generate detailed task information.
- **Output**: The result is a well-defined task with a title, estimated duration, required skills, and a description.

Code Walkthrough

Here's a simplified version of the `TaskAgent`:

```
class TaskAgent:
    """Automates task creation using AI and historical data."""
    def __init__(self, retriever: Retriever):
        self.llm = ChatGroq(
            groq_api_key=os.getenv("GROQ_API_KEY"),
            model="mixtral-8x7b-32768",
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2,
        )
        self.parser = PydanticOutputParser(pydantic_object=TaskOutput)
        self.retriever = retriever

    def create_task(self, description: str, project_id: int) -> dict:
        # Step 1: Retrieve similar tasks and project context
        similar_tasks = self.retriever.get_similar_tasks(description, project_id)
        project_context = self.retriever.get_project_context(project_id)
        team_skills = self.retriever.get_team_skills(project_id)

        # Step 2: Prepare the prompt for the AI model
        prompt = f"""
        You are a task creation assistant. Create a task based on the following description:
        '{description}'.

        Consider these similar tasks: {similar_tasks}.
        Project context: {project_context}.
        Available team skills: {team_skills}.

        Provide a title, estimated duration in hours, and required skills.
        """

        # Step 3: Generate the task using the AI model
        response = self.llm.generate_text(prompt)
        task_info = self.parser.parse(response)

        # Step 4: Return the task details
        return {
            'title': task_info.title,
            'description': description,
            'estimated_duration': task_info.estimated_duration,
            'required_skills': task_info.required_skills,
            'created_at': datetime.datetime.now().isoformat(),
            'status': 'New',
            'project_id': project_id}
```

Some key points to keep in mind:

- **AI Model Initialization**: The `ChatGroq` model is configured with parameters like temperature and max tokens to control the output.
- **Prompt Engineering**: Writing the right prompt is really important because it gives the AI context and tells

what we want.

- **Parsing the Response**: This `PydanticOutputParser` ensures the AI's output is neat and easily converted into Python dictionary.

An Example from Real Life

Imagine we're working on an 'E-Commerce Website Development project, and we need to add a payment gateway.

Using the TaskAgent:

```
# Initialize the retriever and TaskAgent
retriever = Retriever(db_session)
task_agent = TaskAgent(retriever)

# Create a new task
new_task = task_agent.create_task("Implement payment gateway integration", project_id=1)
```

Expected Output:

```
{
  "title": "Develop Payment Gateway Integration",
  "description": "Implement payment gateway integration",
  "estimated_duration": 40,
  "required_skills": ["Python", "Django", "Payment APIs"],
  "created_at": "2024-10-10T15:30:00",
  "status": "New",
  "project_id": 1
}
```

Impact:

- **Saves Time**: Gets rid of having to write out task details by hand.
- **Improves Accuracy**: Less chance we'll miss important parts of the task.

. . .

2. PriorityAgent: Intelligent Task Prioritization

The Challenge of Prioritization

Determining which tasks need immediate attention is often subjective and influenced by external pressures. Th can lead to:

- **Critical Tasks Being Overlooked**: Important tasks may not get the attention they need.
- **Inefficient Use of Resources**: Teams may focus on less important tasks.

The PriorityAgent Solution

The `PriorityAgent` makes prioritizing tasks more objective by:

- **Analyzing Task Complexity**: Considers estimated duration and required skills.
- **Evaluating Project Context**: Looks at dependencies and project milestones.

- **Evaluating Project Context**: Looks at dependencies and project milestones.
- **Leveraging AI Insights**: Provides a priority level with clear reasoning.

How It Works

1. **Input**: Receives a task dictionary.
2. **Data Retrieval**: Gets project details and the priorities of similar tasks.
3. **AI Evaluation**: Uses the AI model to set a priority level.
4. **Output**: Gives you the priority and explains why.

Code Walkthrough

```
class PriorityAgent:
    """Assigns priorities to tasks using AI and project context."""
    def __init__(self, retriever: Retriever):
        self.llm = ChatGroq(...)
        self.parser = PydanticOutputParser(pydantic_object=PriorityOutput)
        self.retriever = retriever

    def assign_priority(self, task: dict) -> dict:
        # Step 1: Retrieve context
        project_context = self.retriever.get_project_context(task['project_id'])
        similar_priorities = self.retriever.get_similar_tasks_priorities(task['description'], task['project_id'])
        team_skills = self.retriever.get_team_skills(task['project_id'])

        # Step 2: Prepare the prompt
        prompt = f"""
        You are a task prioritization assistant. Assign a priority to the following task:
        '{task['title']}'.

        Task details: Duration - {task['estimated_duration']} hours, Skills - {', '.join(task['required_skills'])}.
        Project context: {project_context}.
        Similar tasks' priorities: {similar_priorities}.
        Team skills: {team_skills}.

        Provide the priority (High, Medium, or Low) and reasoning.
        """

        # Step 3: Get the AI's response
        response = self.llm.generate_text(prompt)
        priority_info = self.parser.parse(response)

        # Step 4: Return the priority details
        return priority_info.dict()
```

Important things to note:

- **Contextual Analysis**: The agent doesn't decide on its own; it looks at different factors.
- **AI Reasoning**: By explaining the reasoning, the decision is clear and can be shared with others.

An Example from Real Life

Using the `new_task` from the previous example:

```
# Initialize the PriorityAgent
priority_agent = PriorityAgent(retriever)

# Assign priority to the new task
priority_info = priority_agent.assign_priority(new_task)
```

Expected Output:

```
{
  "priority": "High",
  "reasoning": "The payment gateway is critical for processing transactions, which is essential for the e-commerce platform's functionality."
}
```

Impact:

- **Objective Decision-Making**: Helps avoid bias when prioritizing tasks.
- **Clear Communication**: Gives reasons that you can share with your team.

3. SuggestionAgent: Providing Actionable Suggestions

The Need for Guidance

Even with well-defined tasks, team members may need guidance on the best approach, especially for complex tasks.

The SuggestionAgent Solution

The `SuggestionAgent` aids by:

- **Analyzing Similar Tasks**: Looks at how similar tasks were completed successfully in the past.
- **Recommending Resources**: Suggests tools, libraries, or tutorials that can help.
- **Providing Step-by-Step Guidance**: Offers actionable steps to complete the task efficiently.

How It Works

1. **Input**: Receives a task and project ID.
2. **Data Retrieval**: Fetches similar completed tasks and team skills.
3. **AI Generation**: Uses the AI model to generate suggestions and resources.
4. **Output**: Returns a structured set of suggestions.

Code Walkthrough

```
class SuggestionAgent:
    """Generates suggestions and resources for tasks."""
    def __init__(self, retriever: Retriever):
        self.llm = ChatGroq(...)
        self.parser = PydanticOutputParser(pydantic_object=SuggestionOutput)
        self.retriever = retriever

    def generate_suggestions(self, task: Dict[str, Any], project_id: int) -> Dict[str, Any]:
        # Step 1: Retrieve context
        project_context = self.retriever.get_project_context(project_id)
        similar_tasks = self.retriever.get_similar_completed_tasks(task['description'], project_id)
        team_skills = self.retriever.get_team_skills(project_id)

        # Step 2: Prepare the prompt
        prompt = f"""
        You are a project management assistant. Provide suggestions for the following task:
        '{task['title']}'.

        Task details: Duration - {task['estimated_duration']} hours, Skills - {', '.join(task['required_skills'])}.
        Project context: {project_context}.
        Similar completed tasks: {similar_tasks}.
        Team skills: {team_skills}.

        Provide actionable steps and recommend resources.
        """

        # Step 3: Generate suggestions
        response = self.llm.generate_text(prompt)
        suggestions = self.parser.parse(response)

        # Step 4: Return suggestions
        return suggestions.dict()
```

An Example from Real Life

```
# Initialize the SuggestionAgent
suggestion_agent = SuggestionAgent(retriever)

# Generate suggestions for the new task
suggestions = suggestion_agent.generate_suggestions(new_task, project_id=1)
```

Expected Output:

```
{
  "steps": [
    "Research payment gateway options (e.g., Stripe, PayPal).",
    "Set up developer accounts with the chosen gateway.",
    "Integrate the gateway API into the Django application.",
    "Implement secure payment processing and data handling.",
    "Test transactions in a sandbox environment."
  ],
  "resources": [
    "Stripe API Documentation",
    "Django Payments Library",
    "PCI Compliance Guidelines"
  ]
}
```

Impact:

- **Assists Team Members**: Helps team members by giving them a clear plan, so they're less confused.
- **Improves Efficiency**: Makes Things More Efficient: Helps us avoid common mistakes by learning from past tasks.

4. CollaborationAgent: Optimizing Team Formation

The Challenge of Team Assignment

Assigning the right people to tasks can be complex due to:

- **Skill Matching**: Ensuring team members have the necessary skills.
- **Availability**: Balancing workloads to prevent burnout.
- **Team Dynamics**: Considering how team members work together.

The CollaborationAgent Solution

The `CollaborationAgent` helps by:

- **Analyzing Team Members**: Looks at skills, current workload, and past collaborations.
- **Suggesting Optimal Assignments**: Recommends team members best suited for the task.
- **Proposing Communication Plans**: Suggests how the team should coordinate.

How It Works

1. **Input**: Receives a task and project ID.
2. **Data Retrieval**: Fetches available team members and their skills.
3. **AI Analysis**: Uses the AI model to suggest team formation and communication strategies.
4. **Output**: Returns detailed collaboration suggestions.

Code Walkthrough

```
class CollaborationAgent:
    """Suggests team formation and communication plans for tasks."""
    def __init__(self, retriever: Retriever):
        self.llm = ChatGroq(...)
        self.parser = PydanticOutputParser(pydantic_object=CollaborationOutput)
        self.retriever = retriever

    def suggest_collaboration(self, task: Dict[str, Any], project_id: int) -> Dict[str, Any]:
        # Step 1: Retrieve data
        available_team_members = self.retriever.get_available_team_members(project_id)
        project_context = self.retriever.get_project_context(project_id)
        similar_collaborations = self.retriever.get_similar_collaborations(task['description'], project_id)

        # Step 2: Prepare the prompt
        prompt = f"""
        You are a collaboration assistant. Suggest team members for the following task:
        '{task['title']}'.

        Task details: Duration - {task['estimated_duration']} hours, Skills - {', '.join(task['required_skills'])}.
        Available team members: {available_team_members}.
        Project context: {project_context}.
        Similar collaborations: {similar_collaborations}.

        Provide a team formation and communication plan.
        """

        # Step 3: Generate suggestions
        response = self.llm.generate_text(prompt)
        collaboration_info = self.parser.parse(response)

        # Step 4: Return collaboration details
        return collaboration_info.dict()
```

An Example from Real Life

```
# Initialize the CollaborationAgent
collaboration_agent = CollaborationAgent(retriever)

# Suggest collaboration for the new task
collaboration_info = collaboration_agent.suggest_collaboration(new_task, project_id=1)
```

Expected Output:

```
{
  "team_formation": [
    {"member_name": "Alice Smith", "role": "Lead Developer"},
    {"member_name": "Bob Johnson", "role": "Backend Developer"},
    {"member_name": "Carol Martinez", "role": "QA Specialist"}
  ],
  "communication_plan": "Daily stand-up meetings and a dedicated Slack channel."
}
```

Impact:

- **Optimizes Resource Allocation**: Uses resources better by making sure the right people get the right tasks.

- **Enhances Team Dynamics**: Improves teamwork by looking at how people have worked together befo

5. ReportAgent: Generating Reports

The Importance of Reporting

Regular reports are crucial for:

- **Stakeholder Communication**: Keeping everyone informed about progress.
- **Identifying Risks**: Finding potential issues.
- **Strategic Planning**: Making decisions based on data.

The ReportAgent Solution

The `ReportAgent` automates report generation by:

- **Compiling Project Data**: Aggregates information on tasks, team performance, and milestones.
- **Analyzing Metrics**: Evaluates key performance indicators.
- **Providing Insights**: Highlights risks and offers recommendations.

How It Works

1. **Input**: Receives a list of tasks.
2. **Data Retrieval**: Fetches project details and similar tasks' priorities.
3. **AI Generation**: Uses the AI model to generate a detailed report.
4. **Output**: Returns a structured report.

Code Walkthrough

```
class ReportAgent:
    """Generates comprehensive project reports using AI."""
    def __init__(self, retriever: Retriever):
        self.llm = ChatGroq(...)
        self.parser = PydanticOutputParser(pydantic_object=ReportOutput)
        self.retriever = retriever

    def generate_report(self, tasks: List[Dict[str, Any]]) -> Dict[str, Any]:
        if not tasks:
            return {
                "summary": "No tasks available for report generation.",
                "key_metrics": {},
                "risks": ["No tasks to analyze risks."],
                "recommendations": ["Start by adding tasks to the project."]
            }

        # Step 1: Retrieve context
        project_id = tasks[0]['project_id']
        project_context = self.retriever.get_project_context(project_id)
        similar_projects = self.retriever.get_similar_projects(project_id)

        # Step 2: Prepare the prompt
        prompt = f"""
        You are an AI assistant generating a project report.

        Project context: {project_context}.
        Tasks: {tasks}.
        Similar projects: {similar_projects}.

        Provide a summary, key metrics, risks, and recommendations.
        """

        # Step 3: Generate the report
        response = self.llm.generate_text(prompt)
        report_info = self.parser.parse(response)

        # Step 4: Return the report
        return report_info.dict()
```

An Example from Real Life

```
# Initialize the ReportAgent
report_agent = ReportAgent(retriever)

# Generate a report for the tasks
report = report_agent.generate_report([new_task])
```

Expected Output:

```
{
  "summary": "The project 'E-Commerce Website Development' is progressing well with the addition of the 'Develop Payment Gateway Integr
ation' task.",
  "key_metrics": {
    "total_tasks": 15,
    "completed_tasks": 10,
    "pending_tasks": 5,
    "high_priority_tasks": 2
  },
  "risks": [
    "Potential delays due to third-party API dependencies.",
    "Team members' workload may increase during integration testing."
  ],
  "recommendations": [
    "Begin integration as early as possible to account for unexpected issues.",
    "Consider allocating additional resources to high-priority tasks."
  ]
}
```


Impact:

- **Keeps Everyone Updated**: Clearly shows how the project is doing.
- **Proactive Risk Management**: Finds problems early so they don't get bigger.
- **Informs Strategic Decisions**: Helps with big decisions by giving suggestions to keep things moving smoothly.

. . .

6. The Retriever Class: Data Retrieval

Data Retrieval

The **Retriever** class is a crucial component that interacts with the database and the vector store to fetch releva information needed by the agents.

Key Functions of the Retriever

- **get_similar_tasks**: Retrieves tasks similar to a given description.
- **get_project_context**: Fetches detailed information about the project.
- **get_team_skills**: Retrieves the skills of team members involved in the project.
- **get_available_team_members**: Lists team members available for assignment.
- **get_similar_completed_tasks**: Finds similar tasks that have been completed.
- **get_similar_projects**: Finds similar projects to those that you are working on.

Code Walkthrough

Below is the **Retriever** class from `retriever.py`:

```
from typing import List, Dict, Any
from sqlalchemy.orm import Session
from backend.ai_engine.rag.vector_store import vector_store
from backend.database import crud
from backend.database import models
import json
```

```
class Retriever:
    def __init__(self, db: Session):
        self.db = db

    def get_similar_tasks(self, description: str, project_id: int, k: int = 3) -> List[Dict[str, Any]]:
        query = f"Project ID: {project_id} | Task: {description}"
        similar_docs = vector_store.similarity_search(query, k=k)
        return [{"title": doc.metadata["title"], "description": doc.page_content} for doc in similar_docs]

    def get_project_context(self, project_id: int) -> Dict[str, Any]:
        project = crud.get_project(self.db, project_id)
        if not project:
            print(f"No project found for id: {project_id}")
            return {}
        context = {
            "name": project.name,
            "description": project.description,
            "start_date": str(project.start_date),
            "end_date": str(project.end_date),
            "status": project.status,
            "team_members": [member.name for member in project.team_members]
        }
        print(f"Project context: {context}")
        return context


    def get_project_team_members(self, project_id: int):
        return self.db.query(models.TeamMember).join(models.Project.team_members).filter(models.Project.id == project_id).all()

    def get_similar_tasks_priorities(self, description: str, project_id: int, k: int = 3) -> List[Dict[str, Any]]:
        query = f"Project ID: {project_id} | Task: {description}"
        similar_docs = vector_store.similarity_search(query, k=k)
        return [{"title": doc.metadata["title"], "priority": doc.metadata.get("priority", "Unknown")} for doc in similar_docs]

    def get_available_team_members(self, project_id: int) -> List[Dict[str, Any]]:
        team_members = crud.get_project_team_members(self.db, project_id)
        return [{"name": tm.name, "skills": tm.skills, "role": tm.role} for tm in team_members]

    def get_similar_collaborations(self, task_description: str, project_id: int, k: int = 3) -> List[Dict[str, Any]]:
        query = f"Project ID: {project_id} | Collaboration for: {task_description}"
        similar_docs = vector_store.similarity_search(query, k=k)
        return [{"task": doc.metadata["task"], "collaboration": doc.page_content} for doc in similar_docs]

    def get_project_tasks(self, project_id: int) -> List[Dict[str, Any]]:
        tasks = crud.get_project_tasks(self.db, project_id)
        return [{"title": task.title, "status": task.status, "priority": task.priority} for task in tasks]

    def get_team_performance(self, project_id: int) -> Dict[str, Any]:
        tasks = crud.get_project_tasks(self.db, project_id)
        completed_tasks = sum(1 for task in tasks if task.status == "Completed")
        total_tasks = len(tasks)
        return {
            "task_completion_rate": completed_tasks / total_tasks if total_tasks > 0 else 0,
            "total_tasks": total_tasks,
            "completed_tasks": completed_tasks
        }

    def get_similar_projects(self, project_id: int, k: int = 3) -> List[Dict[str, Any]]:
        project = crud.get_project(self.db, project_id)
        if not project:
            return []

        query = f"Project: {project.name} | Description: {project.description}"
        similar_docs = vector_store.similarity_search(query, k=k)

        return [
            {
                "name": doc.metadata.get("name", "Unnamed Project"),
                "description": doc.page_content
            }
            for doc in similar_docs
```

```
        ]

    def get_similar_completed_tasks(self, task_description: str, project_id: int, k: int = 3) -> List[Dict[str, Any]]:
        query = f"Project ID: {project_id} | Completed Task: {task_description}"
        similar_docs = vector_store.similarity_search(query, k=k)
        return [{"title": doc.metadata["title"], "description": doc.page_content} for doc in similar_docs]

    def get_team_skills(self, project_id: int) -> Dict[str, List[str]]:
        team_members = self.get_project_team_members(project_id)
        return {tm.name: json.loads(tm.skills) if tm.skills else [] for tm in team_members}

    def get_available_team_members(self, project_id: int) -> List[Dict[str, Any]]:
        team_members = crud.get_project_team_members(self.db, project_id)
        return [{"name": tm.name, "skills": tm.skills.split(',')} for tm in team_members]

    def get_related_information(self, question: str, k: int = 3) -> List[Dict[str, Any]]:
        similar_docs = vector_store.similarity_search(question, k=k)
        related_info = [{"title": doc.metadata.get("title", "Unknown"), "content": doc.page_content} for doc in similar_docs]
        print(f"Related information: {related_info}")
        return related_info
```

So, here's how it works:

- **Similarity Search**: Uses the vector store to find similar documents based on embeddings.
- **Database Interaction**: Interfaces with the database to fetch project and team member data.
- **Data Formatting**: Structures the data in a way that's easy for agents to consume.

. . .

7. Orchestrating the Workflow with LangGraph

The Need for Dynamic Workflows

Static workflows can't adapt to the changing nature of projects. A dynamic workflow allows for:

- **Real-Time Adjustments**: You can make changes quickly as the project evolves.
- **State Management**: Keeps track of the current state and transitions.

LangGraph Solution

`LangGraph` enables us to define a stateful workflow where each node represents a step in the process, and edge define the transitions.

Workflow Definition

```
def create_workflow():
    try:
        workflow = StateGraph(Dict)

        # Initialize agents
        task_agent = TaskAgent(retriever)
        priority_agent = PriorityAgent(retriever)
        suggestion_agent = SuggestionAgent(retriever)
        collaboration_agent = CollaborationAgent(retriever)
        report_agent = ReportAgent(retriever)

        # Define nodes
        workflow.add_node("create_task", lambda state: create_task_node(state, task_agent))
        workflow.add_node("assign_priority", lambda state: assign_priority_node(state, priority_agent))
        workflow.add_node("generate_suggestions", lambda state: generate_suggestions_node(state, suggestion_agent))
        workflow.add_node("suggest_collaboration", lambda state: suggest_collaboration_node(state, collaboration_agent))
        workflow.add_node("generate_report", lambda state: generate_report_node(state, report_agent))

        # Define edges
        workflow.add_edge("create_task", "assign_priority")
        workflow.add_edge("assign_priority", "generate_suggestions")
        workflow.add_edge("generate_suggestions", "suggest_collaboration")
        workflow.add_edge("suggest_collaboration", "generate_report")
        workflow.add_edge("generate_report", END)

        workflow.set_entry_point("create_task")
        return workflow.compile()
    except Exception as e:
        print(f"Error creating workflow: {e}")
        return None

workflow = create_workflow()
```

Node Functions

```
def create_task_node(state: Dict[str, Any], task_agent: TaskAgent) -> Dict[str, Any]:
    new_task = task_agent.create_task(state['input_description'], state['project_id'])
    state['tasks'].append(new_task)
    return state

def assign_priority_node(state: Dict[str, Any], priority_agent: PriorityAgent) -> Dict[str, Any]:
    for task in state['tasks']:
        priority_info = priority_agent.assign_priority(task)
        task.update(priority_info)
    return state

def generate_suggestions_node(state: Dict[str, Any], suggestion_agent: SuggestionAgent) -> Dict[str, Any]:
    for task in state['tasks']:
        task['suggestions'] = suggestion_agent.generate_suggestions(task, state['project_id'])
    return state

def suggest_collaboration_node(state: Dict[str, Any], collaboration_agent: CollaborationAgent) -> Dict[str, Any]:
    for task in state['tasks']:
        task['collaboration'] = collaboration_agent.suggest_collaboration(task, state['project_id'])
    return state

def generate_report_node(state: Dict[str, Any], report_agent: ReportAgent) -> Dict[str, Any]:
    state['report'] = report_agent.generate_report(state['tasks'])
    return state
```

Workflow Execution

```
# Define initial state
state = {
    'input_description': "Implement payment gateway integration",
    'project_id': 1,
    'tasks': []
}

# Execute the workflow
workflow(state)
```

Here's what happens:

- **State Updated**: The `state` now includes task details, priorities, suggestions, collaboration info, and the report.
- **Flexible Workflow**: We can tweak the workflow by adding or changing steps.

. . .

Real-World Impact

Implementing this system made a big difference:

- **Efficiency Boost**: Automated processes reduced time spent on administrative tasks by over 50%.
- **Enhanced Collaboration**: Teams said they communicated better and understood their roles more clea
- **Improved Decision-Making**: Using data insights helped us make more strategic choices and cut down on project risks.
- **Greater Flexibility**: The dynamic workflow let us adjust smoothly as project needs changed.

. . .

Conclusion

Using AI agents and a dynamic workflow with `LangGraph` changed how I manage projects. With routine tasks automated and smart insights available, I could focus more on strategic planning and leading.

Key Takeaways:

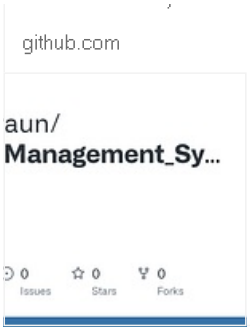
- **Use AI for Routine Tasks**: Save time for bigger-picture thinking.
- **Make Decisions with Data**: Align choices with project goals.
- **Encourage Collaboration**: Keep team members on the same page and communicating well.
- **Stay Flexible**: Be ready to adjust workflows as the project changes.

. . .

GitHub Repository:

GitHub - yotambraun/Project_Management_System_with_RAG

Contribute to yotambraun/Project_Management_System_with_RAG development by creating an account on GitHub.



Thanks for reading!

. . .

If you enjoyed this post, please give it a clap. Feel free to **follow** me on **Medium** for more articles!

- **LinkedIn**

References

LangGraph

LangChain

ChatGroq

langchain_ollama

ollama

groq

Rags

NLP

Langchain

Langgraph

Agents

3

Follow

Written by Yotam Braun

29 Followers

Data Scientist specializing in time series forecasting | Master's in Statistics https://www.linkedin.com/in/yotam-braun-35a005183/ https://github.com/yotambraun