Example 1: Building a REST API

This example demonstrates how to use the AI Orchestrator to build a REST API with proper task breakdown and model routing.

Project Goal

Build a **Task Management REST API** with:
- User authentication (JWT)
- CRUD operations for tasks
- Task assignment and status tracking
- PostgreSQL database

Workflow Diagram

Step-by-Step Commands

Step 1: Analyze the Project

```
@ai-orchestrator analyze_task("Build a task management REST API with:
- User authentication using JWT
- CRUD operations for tasks
- Task assignment between users
- Status tracking (todo, in_progress, done)
- PostgreSQL database
- FastAPI framework")
```

Expected Response:

```
{
  "task_type": "architecture",
  "primary_model": "openai",
  "subtasks": [
    {"description": "Design API architecture", "model": "openai"},
    {"description": "Design database schema", "model": "openai"},
    {"description": "Implement data models", "model": "anthropic"},
    {"description": "Implement authentication", "model": "anthropic"},
    {"description": "Implement CRUD endpoints", "model": "anthropic"},
    {"description": "Security review", "model": "moonshot"}
  ]
}
```

Step 2: Design the Architecture

```
@ai-orchestrator route_to_model("
Design a REST API architecture for a task management system:

Requirements:
- User registration and JWT authentication
- Tasks with title, description, status, assignee, due_date
- Task assignment between users
- Status workflow: todo -> in_progress -> done

Please provide:
1. API endpoint structure (RESTful)
2. Data models/schemas
3. Authentication flow
4. Project folder structure
", "openai")
```

Step 3: Implement Data Models

```
@ai-orchestrator route_to_model("
Implement SQLAlchemy models for a task management API:

Models needed:
1. User: id, email, hashed_password, full_name, created_at
2. Task: id, title, description, status, assignee_id, creator_id, due_date,
created_at, updated_at

Use:
- SQLAlchemy with PostgreSQL
- Pydantic for schemas
- Proper relationships between User and Task

Include:
- Model definitions
- Pydantic schemas for request/response
- Database session management
", "anthropic")
```

Step 4: Implement Authentication

```
@ai-orchestrator route_to_model("
Implement JWT authentication for FastAPI:

1. User registration endpoint
2. Login endpoint returning JWT token
3. JWT verification middleware
4. Password hashing with bcrypt
5. Current user dependency

Use python-jose for JWT and passlib for passwords.
Include proper error handling and validation.
", "anthropic")
```

Step 5: Implement CRUD Endpoints

```
@ai-orchestrator route_to_model("
Implement CRUD endpoints for tasks in FastAPI:

Endpoints:
- GET /tasks - List all tasks (with filters)
- POST /tasks - Create new task
- GET /tasks/{id} - Get task details
- PUT /tasks/{id} - Update task
- DELETE /tasks/{id} - Delete task
- POST /tasks/{id}/assign - Assign task to user
- PATCH /tasks/{id}/status - Update task status

Include:
- Authentication required for all endpoints
- Authorization checks (only creator/assignee can modify)
- Proper HTTP status codes
- Pagination for list endpoint
", "anthropic")
```

Step 6: Security Review

```
@ai-orchestrator route_to_model("
Review this authentication code for security vulnerabilities:

[Paste the auth.py code here]

Check for:
1. JWT security (secret key handling, algorithm, expiration)
2. Password hashing security
3. SQL injection vulnerabilities
4. Input validation
5. Error message information leakage
6. Rate limiting needs
7. CORS configuration
", "moonshot")
```

Model Routing Summary

Task	Model	Reason
API architecture design	ChatGPT	High-level system design
Database schema design	ChatGPT	Data modeling expertise
Data model implementation	Claude	Coding task
Authentication code	Claude	Coding task
CRUD endpoints	Claude	Coding task
Security review	Kimi	Code review specialization

Pro Tips

1. **Start broad, then narrow**: Use `orchestrate_task` first to get an overview, then `route_to_model` for specific implementations.
2. **Include context**: When implementing a feature, include the architecture decisions from earlier steps.
3. **Iterate on reviews**: After getting review feedback from Kimi, use Claude to implement the suggested fixes.
4. **Test as you go**: After each implementation step, test the code before moving on.

Complete Project Structure (After All Steps)

```
task_api/
├── app/
│   ├── __init__.py
│   ├── main.py           # FastAPI app entry point
│   ├── config.py         # Configuration settings
│   ├── database.py       # Database connection
│   ├── models/
│   │   ├── __init__.py
│   │   ├── user.py       # User SQLAlchemy model
│   │   └── task.py       # Task SQLAlchemy model
│   ├── schemas/
│   │   ├── __init__.py
│   │   ├── user.py       # User Pydantic schemas
│   │   └── task.py       # Task Pydantic schemas
│   ├── routers/
│   │   ├── __init__.py
│   │   ├── auth.py       # Auth endpoints
│   │   └── tasks.py      # Task CRUD endpoints
│   ├── services/
│   │   ├── __init__.py
│   │   ├── auth.py       # Auth business logic
│   │   └── task.py       # Task business logic
│   └── dependencies/
│       ├── __init__.py
│       └── auth.py       # Auth dependencies
├── tests/
│   ├── test_auth.py
│   └── test_tasks.py
├── requirements.txt
├── .env.example
└── README.md
```