Example 2: Building a Full-Stack Web Application

This example demonstrates complex multi-model orchestration for building a complete web application with frontend and backend.

Project Goal

Build a **Real-time Chat Application** with:
- React frontend with TypeScript
- FastAPI backend with WebSockets
- User authentication
- Real-time messaging
- Message history with search
- File attachments

Orchestration Flow

```
                    ┌─────────────────────┐
                    │   User Request      │
                    │ "Build chat app"    │
                    └──────────┬──────────┘
                               │
                               ▼
                ┌─────────────────────────┐
                │   orchestrate_task()    │
                │   Task Analysis         │
                └────────────┬────────────┘
                             │
          ┌──────────────────┼──────────────────┐
          │                  │                  │
          ▼                  ▼                  ▼
┌──────────────────┐ ┌──────────────────┐ ┌──────────────────┐
│   ARCHITECTURE   │ │ │   FRONTEND     │ │ │   BACKEND          │
│   (ChatGPT)      │ │ │   (Claude)     │ │ │   (Claude)         │
│                  │ │ │                │ │ │                    │
│  - System design │ │ │  - React setup │ │ │  - FastAPI setup   │
│  - Tech stack    │ │ │  - Components  │ │ │  - WebSocket       │
│  - Data flow     │ │ │  - State mgmt  │ │ │  - Auth            │
└──────────────────┘ └──────────────────┘ └──────────────────┘
          │                  │                  │
          └──────────────────┼──────────────────┘
                             │
                             ▼
              ┌────────────────────────────────┐
              │   INTEGRATION & REVIEW         │
              │   (Gemini + Kimi)              │
              │                                │
              │  - WebSocket integration       │
              │  - Security review             │
              │  - Performance analysis        │
              └────────────────────────────────┘
```

Phase 1: System Architecture (ChatGPT)

Initial Analysis

```
@ai-orchestrator analyze_task("
Build a real-time chat application with:
- React frontend (TypeScript)
- FastAPI backend
- WebSocket for real-time messaging
- User authentication (JWT)
- Chat rooms and direct messages
- Message history with search
- File attachments (images, documents)
- PostgreSQL database
")
```

Architecture Design

```
@ai-orchestrator route_to_model("
Design the complete architecture for a real-time chat application:

Tech Stack:
- Frontend: React, TypeScript, TailwindCSS
- Backend: FastAPI, Python
- Database: PostgreSQL with SQLAlchemy
- Real-time: WebSockets
- File storage: S3-compatible storage

Provide:
1. System architecture diagram (describe in text)
2. Frontend component hierarchy
3. Backend API structure
4. WebSocket message protocol design
5. Database schema
6. Authentication flow
7. File upload flow
", "openai")
```

Phase 2: Backend Implementation (Claude)

2.1 Project Setup

```
@ai-orchestrator route_to_model("
Create the FastAPI project structure for a chat application:

1. Project setup with proper folder structure
2. Configuration management (Pydantic settings)
3. Database connection and session management
4. SQLAlchemy model base
5. Alembic migrations setup

Include:
- requirements.txt
- .env.example
- docker-compose.yml for PostgreSQL
", "anthropic")
```

2.2 Data Models

```
@ai-orchestrator route_to_model("
Implement SQLAlchemy models for the chat application:

Models:
1. User: id, email, username, hashed_password, avatar_url, status, created_at
2. ChatRoom: id, name, type (group/direct), created_by, created_at
3. ChatRoomMember: room_id, user_id, role, joined_at
4. Message: id, room_id, sender_id, content, message_type, created_at, updated_at
5. Attachment: id, message_id, file_url, file_type, file_name, file_size

Include:
- Proper relationships and foreign keys
- Indexes for common queries
- Pydantic schemas for all models
", "anthropic")
```

2.3 WebSocket Implementation

```
@ai-orchestrator route_to_model("
Implement WebSocket handling for real-time chat in FastAPI:

1. WebSocket connection manager class
2. Connection authentication
3. Message broadcasting to room members
4. Typing indicators
5. Online/offline status updates
6. Reconnection handling

Message types to support:
- text_message
- typing_start / typing_stop
- user_joined / user_left
- message_read
- file_upload_started / file_upload_complete

Include error handling and graceful disconnection.
", "anthropic")
```

2.4 REST API Endpoints

```
@ai-orchestrator route_to_model("
Implement REST API endpoints for the chat application:

Auth endpoints:
- POST /auth/register
- POST /auth/login
- POST /auth/refresh
- GET /auth/me

User endpoints:
- GET /users/search?q={query}
- GET /users/{id}
- PATCH /users/me

Chat room endpoints:
- GET /rooms (list user's rooms)
- POST /rooms (create room)
- GET /rooms/{id}
- GET /rooms/{id}/messages (paginated)
- POST /rooms/{id}/messages
- POST /rooms/{id}/members (add member)

File endpoints:
- POST /files/upload
- GET /files/{id}

Include proper authentication, authorization, and pagination.
", "anthropic")
```

Phase 3: Frontend Implementation (Claude)

3.1 Project Setup

```
@ai-orchestrator route_to_model("
Create React TypeScript project setup for a chat application:

1. Vite project with TypeScript
2. TailwindCSS configuration
3. Project folder structure
4. React Router setup
5. Axios configuration for API calls
6. WebSocket client setup
7. Context for auth and chat state

Include:
- tsconfig.json
- Environment configuration
- Type definitions
", "anthropic")
```

3.2 Core Components

```
@ai-orchestrator route_to_model("
Implement React components for the chat application:

1. Layout components:
   - MainLayout (sidebar + content)
   - Sidebar (room list)
   - ChatArea (messages + input)

2. Chat components:
   - RoomList (list of chat rooms)
   - RoomItem (single room preview)
   - MessageList (scrollable messages)
   - Message (single message with avatar)
   - MessageInput (text input + file attachment)
   - TypingIndicator

3. User components:
   - UserAvatar
   - UserStatus (online/offline)
   - UserSearch

Use TypeScript interfaces for all props.
Include proper styling with TailwindCSS.
", "anthropic")
```

3.3 State Management

```
@ai-orchestrator route_to_model("
Implement React state management for the chat application:

1. AuthContext:
   - User state
   - Login/logout functions
   - Token management

2. ChatContext:
   - Active room
   - Messages by room
   - Online users
   - Typing indicators

3. WebSocket hooks:
   - useWebSocket (connection management)
   - useMessages (message subscription)
   - useTyping (typing indicators)

4. API hooks (using React Query or similar):
   - useRooms
   - useMessages
   - useUser

Include proper TypeScript types and error handling.
", "anthropic")
```

Phase 4: Integration Analysis (Gemini)

WebSocket Integration Analysis

```
@ai-orchestrator route_to_model("
Analyze the WebSocket integration between React frontend and FastAPI backend:

Given:
- Backend WebSocket endpoint at /ws/{room_id}
- Frontend WebSocket client using native WebSocket API
- JWT authentication

Analyze:
1. Connection lifecycle management
2. Message serialization/deserialization
3. Error recovery strategies
4. Scalability considerations (multiple server instances)
5. Memory management for message history
6. Race conditions in message ordering

Provide:
- Potential issues and solutions
- Best practices recommendations
- Performance optimization suggestions
", "gemini")
```

Phase 5: Security & Code Review (Kimi)

Backend Security Review

```
@ai-orchestrator route_to_model("
Review the chat application backend for security:

[Paste auth.py, websocket.py, and main routes]

Review for:
1. Authentication vulnerabilities
2. WebSocket security (connection hijacking, message spoofing)
3. Input validation and sanitization
4. SQL injection risks
5. File upload security
6. Rate limiting needs
7. CORS configuration
8. Information disclosure in errors

", "moonshot")
```

Frontend Security Review

```
@ai-orchestrator route_to_model("
Review the React chat application for security:

[Paste relevant components]

Review for:
1. XSS vulnerabilities in message rendering
2. Token storage security
3. Secure WebSocket handling
4. File upload validation
5. Proper error handling (no sensitive data exposure)

", "moonshot")
```

Project Structure

```
chat-app/
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── main.py
│   │   ├── config.py
│   │   ├── database.py
│   │   ├── models/
│   │   │   ├── user.py
│   │   │   ├── chat_room.py
│   │   │   ├── message.py
│   │   │   └── attachment.py
│   │   ├── schemas/
│   │   ├── routers/
│   │   │   ├── auth.py
│   │   │   ├── users.py
│   │   │   ├── rooms.py
│   │   │   └── files.py
│   │   ├── websocket/
│   │   │   ├── manager.py
│   │   │   └── handlers.py
│   │   └── services/
│   ├── alembic/
│   ├── requirements.txt
│   └── docker-compose.yml
│
└── frontend/
    ├── src/
    │   ├── components/
    │   │   ├── layout/
    │   │   ├── chat/
    │   │   └── user/
    │   ├── contexts/
    │   │   ├── AuthContext.tsx
    │   │   └── ChatContext.tsx
    │   ├── hooks/
    │   │   ├── useWebSocket.ts
    │   │   └── useMessages.ts
    │   ├── pages/
    │   ├── services/
    │   ├── types/
    │   └── App.tsx
    ├── package.json
    └── vite.config.ts
```

Model Usage Summary

Phase	Tasks	Model	Token Estimate
Architecture	System design, diagrams	ChatGPT	~2000
Backend Setup	Project structure, config	Claude	~1500
Backend Models	SQLAlchemy, Pydantic	Claude	~2000
Backend WebSocket	Real-time handling	Claude	~2500
Backend API	REST endpoints	Claude	~3000
Frontend Setup	React, TypeScript	Claude	~1500
Frontend Components	UI components	Claude	~3000
Frontend State	Context, hooks	Claude	~2000
Integration	Analysis, optimization	Gemini	~1500
Security Review	Backend + Frontend	Kimi	~2000

Total estimated tokens: ~21,000

Tips for Complex Projects

1. **Phase your work**: Complete architecture before implementation
2. **Share context**: Include architecture decisions when coding
3. **Review incrementally**: Review each component, not the entire codebase
4. **Document as you go**: Ask models to include inline documentation
5. **Test between phases**: Validate each phase before moving forward