Langchain Feature:

Group Chat with AI Task Extraction: Using Gemini API Directly:

1. Overall Architecture

Your implementation will consist of these major components:

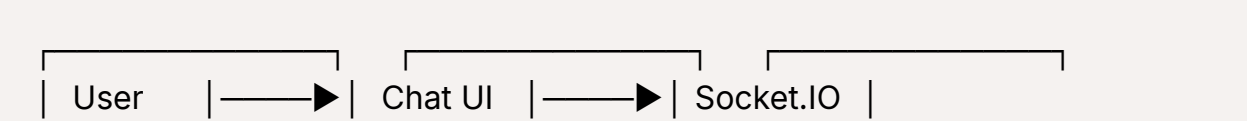
A. Backend Components

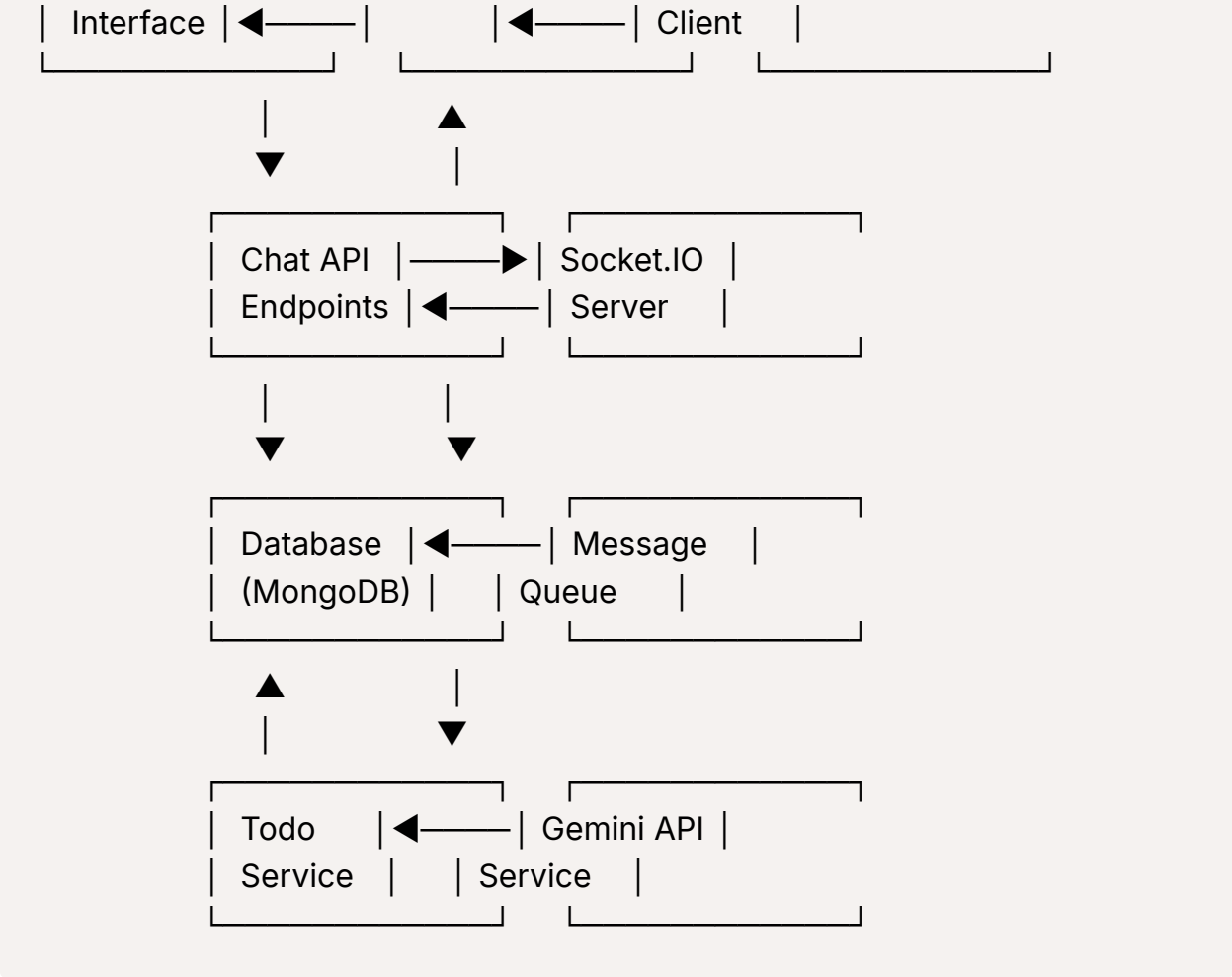
1. **Chat Data Models**: MongoDB schemas for storing conversations and messages
2. **Chat API Endpoints**: Express routes for sending/retrieving messages
3. **Socket.IO Integration**: Real-time message delivery system
4. **Message Queue**: Background processing system for AI analysis
5. **Gemini AI Service**: Direct integration with Google's Gemini API
6. **Task Creation Service**: Converting AI outputs to actual todo items

B. Frontend Components

1. **Chat UI**: Interface for group messaging
2. **Socket Client**: Real-time connection manager
3. **AI Task Review UI**: Interface for approving/rejecting AI-generated tasks
4. **API Services**: JavaScript modules for communicating with backend

2. Data Flow Architecture





3. Detailed Implementation Workflow

Stage 1: Database Models Setup

1. **Create Chat Models**:
   - `Chat`: Represents a group conversation
   - `ChatMessage`: Individual messages with sender, content, timestamps
   - Add AI processing metadata fields to track extraction status
2. **Update Todo Model**:
   - Add AI-related fields: `aiGenerated`, `verified`, `needsVerification`
   - Add metadata to track confidence scores and verification status

Stage 2: Backend API Development

1. **Chat Service Layer**:
   - `createGroupChat`: Initialize chat for a group
   - `addMessage`: Add new message to a conversation
   - `getGroupMessages`: Retrieve messages with pagination
2. **API Endpoints**:
   - `GET /api/chat/group/:groupId`: Get group messages
   - `POST /api/chat/group/:groupId/messages`: Send a message
   - Update existing todo endpoints to handle AI-generated tasks
3. **Socket.IO Events**:
   - `chat:send_message`: Client sends a message
   - `chat:message`: Server broadcasts message to group
   - `chat:system_message`: System/AI notifications
   - `tasks:ai_generated`: Notify about new AI tasks

Stage 3: Gemini API Integration

1. **Message Queue System**:
   - Background service monitoring for new messages
   - Groups messages by conversation for efficient processing
   - Implements time-based or volume-based triggers
2. **Gemini API Service**:
   - Direct integration with Google's Generative AI API
   - Uses your existing Gemini API key from env.js
   - Two-step approach:
     a. Task detection request (determines if messages contain tasks)
     b. Task extraction request (extracts detailed task information)
3. **Processing Workflow**:

- Collect recent messages (10-20) from a conversation
- Format conversation into a structured prompt for Gemini
- Send detection request to Gemini API
- If tasks detected, send extraction request to Gemini API
- Parse JSON response and validate task information
- Create todo items from extracted information
- Notify users about new tasks

Stage 4: Frontend Implementation

1. **Chat UI Components**:
   - Chat panel with message list and input
   - Message bubbles with sender information
   - System message styling for AI notifications
2. **Socket Integration**:
   - Connect to socket server on page load
   - Join group-specific rooms
   - Handle real-time message events
   - Update UI when messages arrive
3. **AI Task Review Interface**:
   - Display AI-generated tasks in review panel
   - Provide approve/reject actions
   - Show confidence scores and extracted metadata
   - Allow bulk approval option

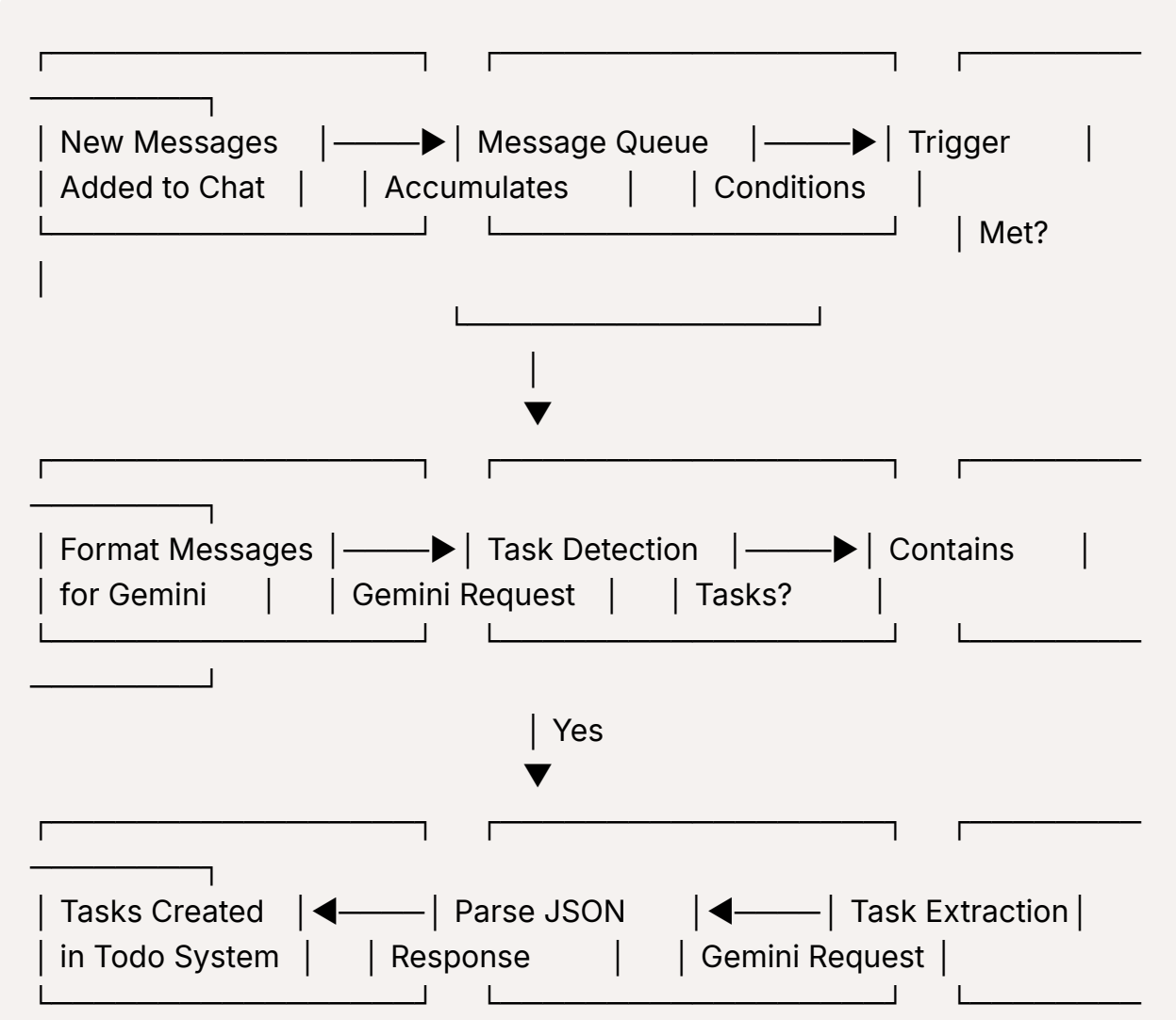
Stage 5: Integration with Existing App

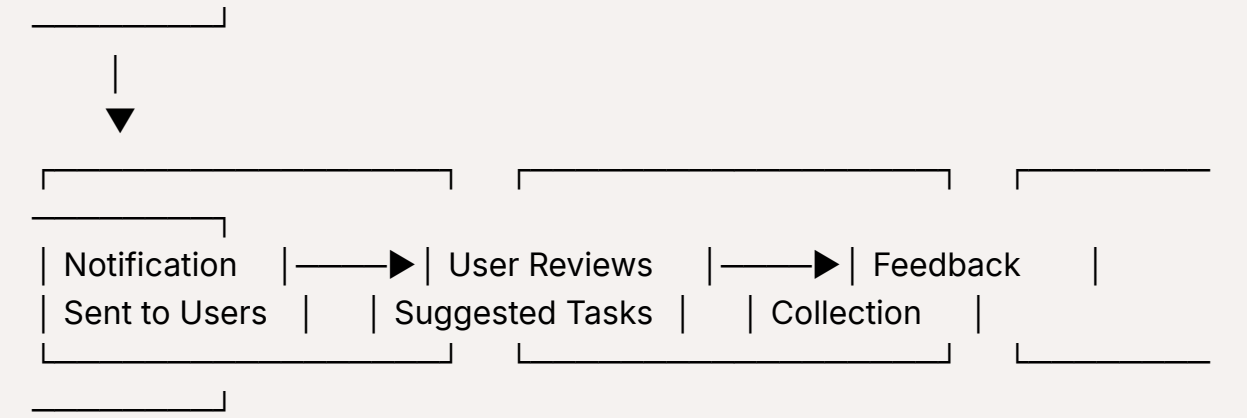
1. **Add Chat Tab to Group Pages**:
   - Implement tabbed interface (Todos, Chat, Members)

- Ensure proper routing and state management

2. **Update Todo List View**:
   - Filter to separate AI-generated tasks
   - Display verification status indicators
   - Show only verified tasks in main list
3. **Notification Integration**:
   - Use existing notification system for AI task alerts
   - Add new notification types for chat events

4. Gemini API Processing Flow





5. Specific Components Breakdown

a. Message Queue Service

Purpose: Buffer messages and trigger AI processing at appropriate times

Key Functions:

- `addMessage(message)`: Add message to processing queue
- `processQueue()`: Process accumulated messages
- `processGroupMessages(groupId)`: Process messages for a specific group

Triggering Logic:

- Time-based: Process after X minutes of inactivity
- Volume-based: Process after Y messages
- Combination approach: Process when either condition is met

b. Gemini AI Service

Purpose: Analyze conversations and extract actionable tasks using Gemini API

API Integration:

```
import { GoogleGenerativeAI } from "@google/generative-ai";
import { GEMINI_API_KEY } from '../../lib/env';

// Initialize the Gemini API
```

```
const genAI = new GoogleGenerativeAI(GEMINI_API_KEY);
const model = genAI.getGenerativeModel({
  model: "gemini-pro",
  generationConfig: {
    temperature: 0.2,
    topP: 0.8,
    topK: 40,
    maxOutputTokens: 1024,
  }
});
```

Two-Stage Process:

1. **Task Detection**:
   - **Purpose**: Quickly determine if a conversation contains tasks
   - **Request Format**:

     ```
     You are an AI assistant specialized in identifying tasks in conversation
     s.

     [CONVERSATION]
     {formatted conversation text}
     [/CONVERSATION]

     Does this conversation contain any tasks, action items, or work that ne
     eds to be done?
     Respond with a JSON object:
     {
       "containsTasks": true/false,
       "confidence": 0-100,
       "reasoning": "Brief explanation"
     }
     ```

   - **Response Processing**:
     - Parse JSON response

- Extract containsTasks boolean and confidence score
- Proceed to extraction step if tasks detected with confidence > threshold

2. **Task Extraction**:
   - **Purpose**: Extract detailed task information
   - **Request Format**:

     ```
     You are an AI assistant specialized in extracting tasks from conversati
     ons.

     [CONVERSATION]
     {formatted conversation text}
     [/CONVERSATION]

     Extract all tasks mentioned in this conversation. For each task, provid
     e:
     1. title: A clear, concise task title
     2. description: Detailed explanation with context
     3. dueDate: Due date if mentioned (YYYY-MM-DD format, or null)
     4. assignee: Who should complete the task (name or username, or nul
     l)
     5. priority: Priority level (high, medium, low, or null)

     Format your response as a JSON array of task objects.
     Return ONLY the JSON array.
     ```

   - **Response Processing**:
     - Parse JSON array from response
     - Validate each task object has required fields
     - Normalize data (dates, priorities, etc.)
     - Map to todo schema for creation

c. Task Creation & Notification

Purpose: Create todo items and notify users

Workflow:

1. Create todo items with AI metadata flags
2. Set verification status to pending
3. Send socket notification to group members
4. Add system message to chat about created tasks
5. Update processing metrics for monitoring

Implementation Example:

```
// After extracting tasks from Gemini API response
for (const task of extractedTasks) {
  // Create todo with AI metadata
  const newTodo = await createTodo({
    title: task.title,
    description: task.description || '',
    dueDate: normalizeDate(task.dueDate),
    priority: normalizePriority(task.priority),
    assignee: await findUserByName(task.assignee, groupMembers),
    group: groupId,
    metadata: {
      aiGenerated: true,
      needsVerification: true,
      confidence: calculateConfidence(task),
      generatedAt: new Date()
    }
  });

  createdTasks.push(newTodo);
}

// Notify users about new tasks
```

```
socketService.notifyGroup(groupId, 'tasks:ai_generated', {
  taskCount: createdTasks.length,
  tasks: createdTasks.map(t => ({ _id: t._id, title: t.title }))
});
```

d. User Review Interface

Purpose: Allow users to verify AI-generated tasks

Features:

1. Separated display of unverified AI tasks
2. Approve/reject actions for each task
3. Bulk approve option for quick verification
4. Edit capabilities before approval
5. Feedback mechanism for improving extraction

6. Prompt Engineering for Gemini API

Task Detection Prompt

```
You are an AI assistant specialized in identifying tasks mentioned in conversat
ions.

[CONVERSATION]
[User1 - June 10, 2023, 10:15 AM]
We need to finish the website redesign by next Friday.

[User2 - June 10, 2023, 10:16 AM]
I'll handle the frontend part, can someone take care of the backend updates?

[User3 - June 10, 2023, 10:18 AM]
I can work on the backend. We also need to update the documentation.
[/CONVERSATION]
```

Does this conversation contain any tasks, action items, or work that needs to be done?
Look for:
1. Explicit task assignments ("John will create the design")
2. Implicit tasks ("We need to finish the report")
3. Deadlines or timeframes ("by tomorrow", "in two weeks")
4. Responsibilities ("I'll handle the backend")

Respond with a JSON object:
{
  "containsTasks": true/false,
  "confidence": 0-100,
  "reasoning": "Brief explanation of your decision"
}

Task Extraction Prompt

You are an AI assistant specialized in extracting actionable tasks from convers ations.

[CONVERSATION]
[User1 - June 10, 2023, 10:15 AM]
We need to finish the website redesign by next Friday.

[User2 - June 10, 2023, 10:16 AM]
I'll handle the frontend part, can someone take care of the backend updates?

[User3 - June 10, 2023, 10:18 AM]
I can work on the backend. We also need to update the documentation.
[/CONVERSATION]

Extract all tasks mentioned in this conversation. For each task, provide:

1. title: A clear, concise task title
2. description: Detailed explanation with context

```
3. dueDate: Due date if mentioned (YYYY-MM-DD format, or null)
4. assignee: Who should complete the task (name or username, or null)
5. priority: Priority level (high, medium, low, or null)

Format your response as a JSON array of task objects:
[
  {
    "title": "...",
    "description": "...",
    "dueDate": "YYYY-MM-DD",
    "assignee": "...",
    "priority": "..."
  }
]

Return ONLY the JSON array.
```

7. Format Conversation Function

Purpose: Convert chat messages to a format suitable for Gemini API

```
function formatConversation(messages) {
  return messages.map(msg =>
    `[${msg.sender.username} - ${new Date(msg.createdAt).toLocaleString()}]
     ${msg.content}
    `
  ).join('\\n\\n');
}
```

8. Parse Gemini Response Functions

Parse Task Detection Response

```
function parseTaskDetection(responseText) {
  try {
```

```
    // Extract JSON from the response
    const jsonMatch = responseText.match(/\\{[\\s\\S]*\\}/);
    if (!jsonMatch) {
      throw new Error("No valid JSON found in response");
    }

    const json = JSON.parse(jsonMatch[0]);
    return {
      containsTasks: Boolean(json.containsTasks),
      confidence: Number(json.confidence || 0),
      reasoning: String(json.reasoning || "")
    };
  } catch (error) {
    console.error("Failed to parse task detection response:", error);
    return {
      containsTasks: false,
      confidence: 0,
      reasoning: "Failed to parse response"
    };
  }
}
```

Parse Task Extraction Response

```
function parseTaskExtraction(responseText) {
  try {
    // Extract JSON array from response
    const jsonMatch = responseText.match(/\\[[\\s\\S]*\\]/);
    if (!jsonMatch) {
      throw new Error("No valid JSON array found in response");
    }

    const tasks = JSON.parse(jsonMatch[0]);

    // Validate and normalize each task
```

```
    return tasks.map(task => ({
      title: String(task.title || "").trim(),
      description: String(task.description || "").trim(),
      dueDate: task.dueDate && /^\\d{4}-\\d{2}-\\d{2}$/.test(task.dueDate)
        ? task.dueDate
        : null,
      assignee: String(task.assignee || "").trim() || null,
      priority: ["high", "medium", "low"].includes(task.priority)
        ? task.priority
        : "medium",
    })).filter(task => task.title); // Filter out tasks without titles
  } catch (error) {
    console.error("Failed to parse task extraction response:", error);
    return [];
  }
}
```

9. Implementation Timeline

Phase 1: Basic Chat Functionality (1-2 weeks)

1. Implement chat data models
2. Create basic chat API endpoints
3. Add real-time socket communication
4. Build frontend chat UI components

Phase 2: Gemini API Integration (1-2 weeks)

1. Set up message queue service
2. Implement Gemini API service
3. Create prompt templates and response parsers
4. Add processing metrics and monitoring

Phase 3: Task Integration (1-2 weeks)

1. Modify todo service to handle AI-generated tasks
2. Implement task verification workflow
3. Create AI task review interface
4. Add notification system for new AI tasks

Phase 4: Refinement & Optimization (1 week)

1. Add feedback collection for task accuracy
2. Implement prompt optimization based on feedback
3. Add rate limiting and cost control measures
4. Optimize performance for larger conversations

10. Optimization Strategies

Rate Limiting and Cost Management

```
class GeminiRateLimiter {
  constructor() {
    this.requestCount = 0;
    this.lastResetTime = Date.now();
    this.maxRequestsPerMinute = 60; // Adjust based on your plan
  }

  async checkLimit() {
    // Reset counter if a minute has passed
    if (Date.now() - this.lastResetTime > 60000) {
      this.requestCount = 0;
      this.lastResetTime = Date.now();
    }

    // Check if we're over limit
    if (this.requestCount >= this.maxRequestsPerMinute) {
      // Wait until reset
      const waitTime = 60000 - (Date.now() - this.lastResetTime);
```

```
      await new Promise(resolve => setTimeout(resolve, waitTime));
      // Recursive call after waiting
      return this.checkLimit();
    }

    // Increment counter and proceed
    this.requestCount++;
    return true;
  }
}
```

Dynamic Temperature Settings

```
function getOptimalTemperature(task) {
  // Lower temperature for more deterministic results in task extraction
  if (task === 'task_extraction') {
    return 0.1;
  }

  // Slightly higher for detection which is more of a classification task
  if (task === 'task_detection') {
    return 0.2;
  }

  return 0.3; // Default
}
```

Response Validation and Error Handling

```
function validateTaskObject(task) {
  // Required fields
  if (!task.title || task.title.trim().length < 3) {
    return false;
  }
```

```
  // Validate date format if present
  if (task.dueDate && !/^\\d{4}-\\d{2}-\\d{2}$/.test(task.dueDate)) {
    task.dueDate = null;
  }

  // Validate priority
  if (task.priority && !["high", "medium", "low"].includes(task.priority)) {
    task.priority = "medium";
  }

  return true;
}
```

This comprehensive workflow provides a detailed structure for implementing a group chat system with AI task extraction using the Gemini API directly. It builds on your existing todo app structure while adding the new chat and AI extraction capabilities.

Group Chat with AI Task Extraction: Using LangChain & LangGraph

1. Overall Architecture

Your implementation will consist of these major components:

A. Backend Components

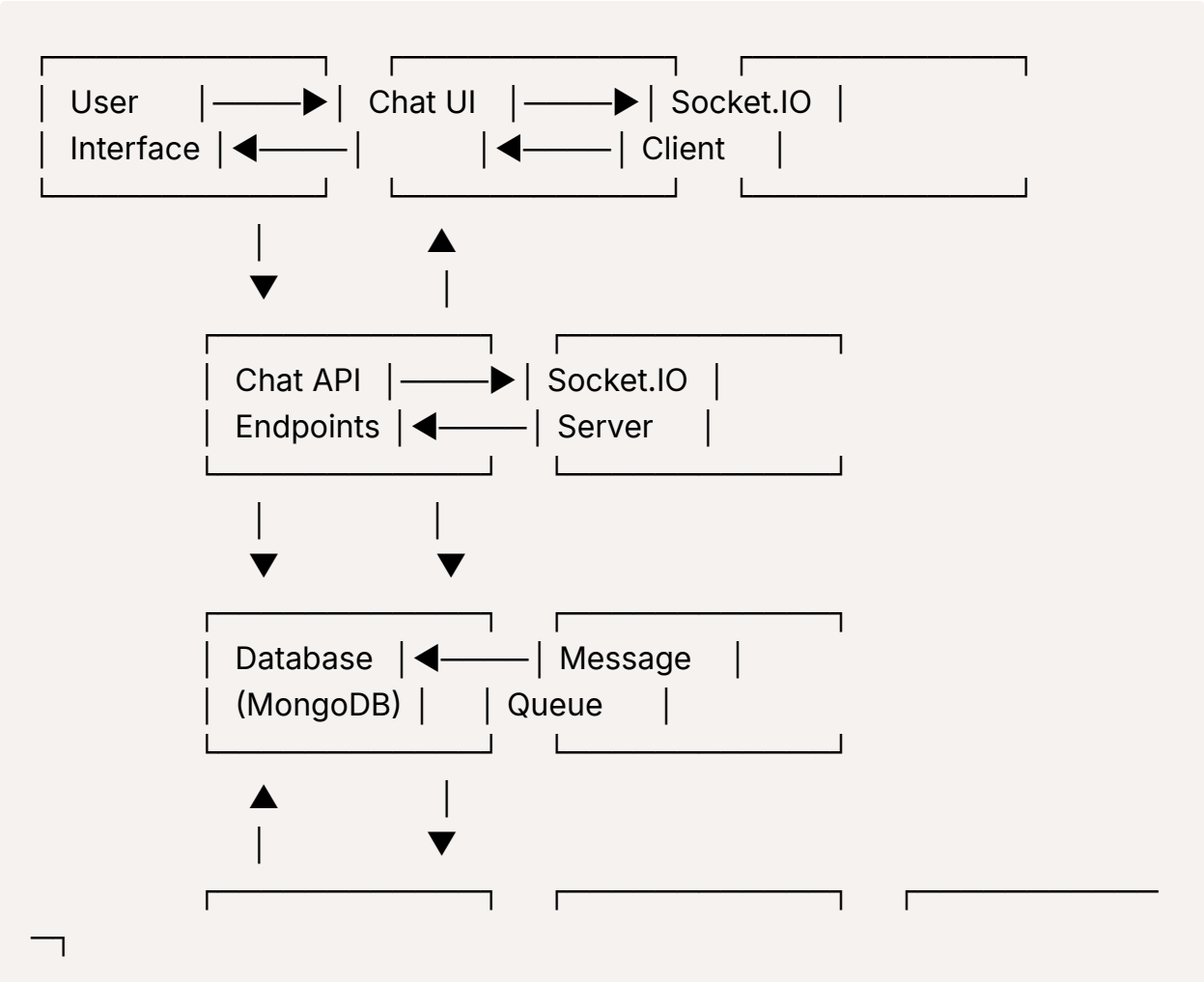
1. **Chat Data Models**: MongoDB schemas for storing conversations and messages
2. **Chat API Endpoints**: Express routes for sending/retrieving messages
3. **Socket.IO Integration**: Real-time message delivery system

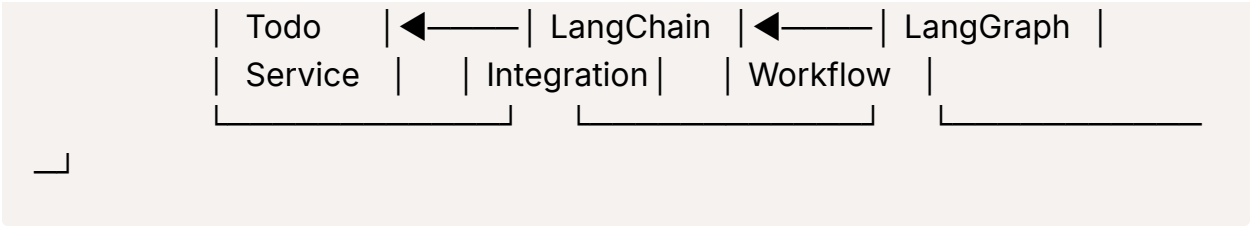
4. **Message Queue**: Background processing system for AI analysis
5. **LangChain Pipeline**: Structured AI processing using LangChain and LangGraph
6. **Task Creation Service**: Converting AI outputs to actual todo items

B. Frontend Components

1. **Chat UI**: Interface for group messaging
2. **Socket Client**: Real-time connection manager
3. **AI Task Review UI**: Interface for approving/rejecting AI-generated tasks
4. **API Services**: JavaScript modules for communicating with backend

2. Data Flow Architecture





3. Detailed Implementation Workflow

Stage 1: Database Models Setup

1. **Create Chat Models**:
   - `Chat`: Represents a group conversation
   - `ChatMessage`: Individual messages with sender, content, timestamps
   - Add AI processing metadata fields to track extraction status
2. **Update Todo Model**:
   - Add AI-related fields: `aiGenerated`, `verified`, `needsVerification`
   - Add metadata to track confidence scores and verification status

Stage 2: Backend API Development

1. **Chat Service Layer**:
   - `createGroupChat`: Initialize chat for a group
   - `addMessage`: Add new message to a conversation
   - `getGroupMessages`: Retrieve messages with pagination
2. **API Endpoints**:
   - `GET /api/chat/group/:groupId`: Get group messages
   - `POST /api/chat/group/:groupId/messages`: Send a message
   - Update existing todo endpoints to handle AI-generated tasks
3. **Socket.IO Events**:
   - `chat:send_message`: Client sends a message
   - `chat:message`: Server broadcasts message to group

- `chat:system_message`: System/AI notifications
- `tasks:ai_generated`: Notify about new AI tasks

Stage 3: LangChain Integration

1. **Message Queue System**:
   - Background service monitoring for new messages
   - Groups messages by conversation for efficient processing
   - Implements time-based or volume-based triggers
2. **LangChain & LangGraph Setup**:
   - Install required packages: `langchain`, `@langchain/openai`, `langgraph-js`, `zod`
   - Create model instances with your Gemini API key
   - Define processing chain with clear node responsibilities
   - Implement graph-based workflow for task extraction
3. **LangChain Processing Pipeline**:
   - Define chain components with specific responsibilities:
     - Context collection
     - Task detection
     - Task extraction
     - Validation
     - Task creation
   - Connect components in a directed graph using LangGraph
   - Handle success/failure paths and edge conditions

Stage 4: Frontend Implementation

1. **Chat UI Components**:
   - Chat panel with message list and input
   - Message bubbles with sender information

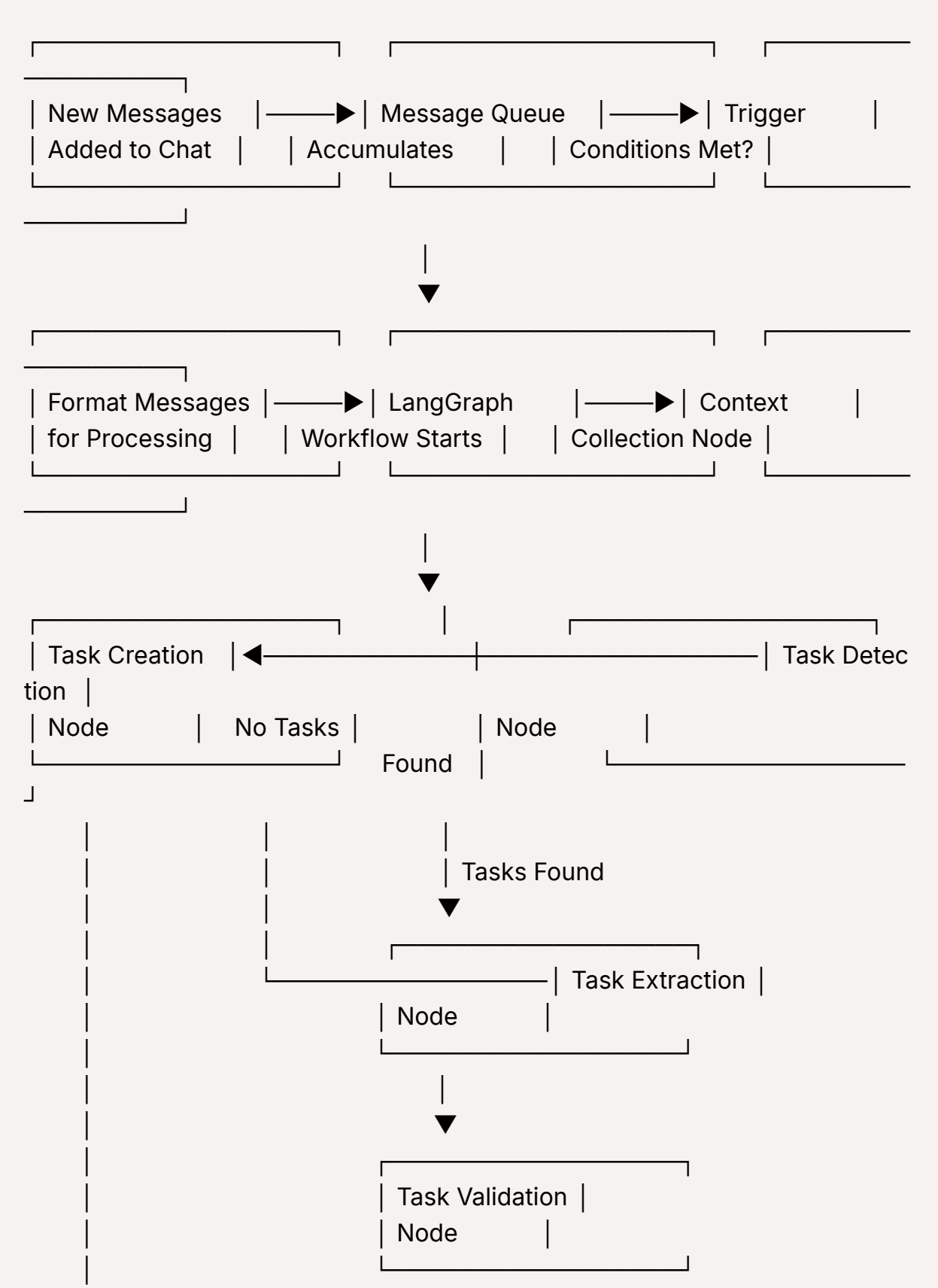
- System message styling for AI notifications

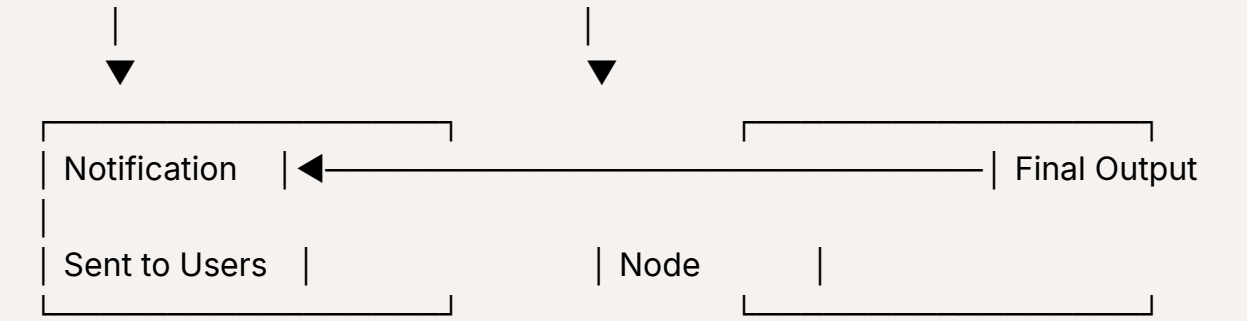
2. **Socket Integration**:
   - Connect to socket server on page load
   - Join group-specific rooms
   - Handle real-time message events
   - Update UI when messages arrive
3. **AI Task Review Interface**:
   - Display AI-generated tasks in review panel
   - Provide approve/reject actions
   - Show confidence scores and extracted metadata
   - Allow bulk approval option

Stage 5: Integration with Existing App

1. **Add Chat Tab to Group Pages**:
   - Implement tabbed interface (Todos, Chat, Members)
   - Ensure proper routing and state management
2. **Update Todo List View**:
   - Filter to separate AI-generated tasks
   - Display verification status indicators
   - Show only verified tasks in main list
3. **Notification Integration**:
   - Use existing notification system for AI task alerts
   - Add new notification types for chat events

4. LangChain & LangGraph Processing Flow


New Messages Added to Chat
Message Queue Accumulates
Trigger Conditions Met?
Format Messages for Processing
LangGraph Workflow Starts
Context Collection Node
Task Creation Node
No Tasks
Task Detection Node
Found
Tasks Found
Task Extraction Node
Task Validation Node



5. Specific Components Breakdown

a. Message Queue Service

Purpose: Buffer messages and trigger AI processing at appropriate times

Key Functions:

- `addMessage(message)`: Add message to processing queue
- `processQueue()`: Process accumulated messages
- `processGroupMessages(groupId)`: Process messages for a specific group

Triggering Logic:

- Time-based: Process after X minutes of inactivity
- Volume-based: Process after Y messages
- Combination approach: Process when either condition is met

b. LangChain Integration

Purpose: Create a structured AI pipeline for processing conversations

Setup:

```
import { ChatGoogleGenerativeAI } from "@langchain/google-genai";
import { PromptTemplate } from "@langchain/core/prompts";
import { RunnableSequence } from "@langchain/core/runnables";
import { JsonOutputParser } from "@langchain/core/output_parsers";
import { z } from "zod";
import { GEMINI_API_KEY } from '../../lib/env';
```

```
// Initialize models with different settings for different tasks
const taskDetectionModel = new ChatGoogleGenerativeAI({
  apiKey: GEMINI_API_KEY,
  modelName: "gemini-pro",
  temperature: 0.1, // Lower for more deterministic detection
});

const taskExtractionModel = new ChatGoogleGenerativeAI({
  apiKey: GEMINI_API_KEY,
  modelName: "gemini-pro",
  temperature: 0.2, // Slightly higher for creative but accurate extraction
});
```

Chain Components:

1. **Task Detection Chain**:

```
// Schema definition
const taskDetectionSchema = z.object({
  containsTasks: z.boolean(),
  confidence: z.number().min(0).max(100),
  reasoning: z.string(),
});

// Prompt template
const taskDetectionPrompt = PromptTemplate.fromTemplate(`
  You are an AI assistant specialized in identifying tasks mentioned in conv
ersations.

  [CONVERSATION]
  {conversation}
  [/CONVERSATION]

  Does this conversation contain any tasks, action items, or work that need
s to be done?
```

```
  Look for:
  1. Explicit task assignments ("John will create the design")
  2. Implicit tasks ("We need to finish the report")
  3. Deadlines or timeframes ("by tomorrow", "in two weeks")
  4. Responsibilities ("I'll handle the backend")

  Respond with a JSON object:
  {
    "containsTasks": true/false,
    "confidence": 0-100,
    "reasoning": "Brief explanation of your decision"
  }
`);

// Chain definition
const taskDetectionChain = RunnableSequence.from([
  taskDetectionPrompt,
  taskDetectionModel,
  new JsonOutputParser(),
]);
```

2. **Task Extraction Chain**:

```
// Schema definition
const taskSchema = z.object({
  title: z.string(),
  description: z.string(),
  dueDate: z.string().nullable(),
  assignee: z.string().nullable(),
  priority: z.enum(["high", "medium", "low"]).nullable(),
});

const tasksSchema = z.array(taskSchema);

// Prompt template
```

```
const taskExtractionPrompt = PromptTemplate.fromTemplate(`
  You are an AI assistant specialized in extracting actionable tasks from co
nversations.

  [CONVERSATION]
  {conversation}
  [/CONVERSATION]

  Extract all tasks mentioned in this conversation. For each task, provide:

  1. title: A clear, concise task title
  2. description: Detailed explanation with context
  3. dueDate: Due date if mentioned (YYYY-MM-DD format, or null)
  4. assignee: Who should complete the task (name or username, or null)
  5. priority: Priority level (high, medium, low, or null)

  Format your response as a JSON array of task objects.
  Return ONLY the JSON array.
`);

// Chain definition
const taskExtractionChain = RunnableSequence.from([
  taskExtractionPrompt,
  taskExtractionModel,
  new JsonOutputParser(),
]);
```

c. LangGraph Workflow

Purpose: Orchestrate the flow between different processing steps

Implementation:

```
import { StateGraph, END } from "langgraph-js";

// Define the state interface
```

```
const state = {
  conversation: null,
  containsTasks: false,
  confidence: 0,
  tasks: [],
  validatedTasks: [],
  createdTasks: [],
  errors: []
};

// Define the nodes
const nodes = {
  // Collect context and format conversation
  collectContext: async (state) => {
    try {
      const formattedConversation = formatConversation(state.messages);
      return {
        ...state,
        conversation: formattedConversation
      };
    } catch (error) {
      return {
        ...state,
        errors: [...state.errors, `Context error: ${error.message}`]
      };
    }
  },

  // Detect if conversation contains tasks
  detectTasks: async (state) => {
    try {
      const result = await taskDetectionChain.invoke({
        conversation: state.conversation
      });

      return {
```

```
      ...state,
      containsTasks: result.containsTasks,
      confidence: result.confidence,
      reasoning: result.reasoning
    };
  } catch (error) {
    return {
      ...state,
      errors: [...state.errors, `Detection error: ${error.message}`]
    };
  }
},

// Extract detailed task information
extractTasks: async (state) => {
  try {
    const result = await taskExtractionChain.invoke({
      conversation: state.conversation
    });

    return {
      ...state,
      tasks: Array.isArray(result) ? result : []
    };
  } catch (error) {
    return {
      ...state,
      errors: [...state.errors, `Extraction error: ${error.message}`]
    };
  }
},

// Validate extracted tasks
validateTasks: async (state) => {
  try {
    const validatedTasks = state.tasks.filter(task => {
```

```
      // Basic validation
      return task.title && task.title.trim().length >= 3;
    }).map(task => ({
      ...task,
      // Normalize data
      dueDate: normalizeDate(task.dueDate),
      priority: normalizePriority(task.priority)
    }));

    return {
      ...state,
      validatedTasks
    };
  } catch (error) {
    return {
      ...state,
      errors: [...state.errors, `Validation error: ${error.message}`]
    };
  }
},

// Create tasks in the system
createTasks: async (state) => {
  try {
    const createdTasks = [];

    for (const task of state.validatedTasks) {
      const newTask = await todoService.createTodo({
        title: task.title,
        description: task.description || '',
        dueDate: task.dueDate,
        assignee: await findUserByName(task.assignee, state.groupMembers),
        priority: task.priority || 'medium',
        status: 'pending',
        group: state.groupId,
        aiGenerated: true,
```

```
        needsVerification: true
      });

      createdTasks.push(newTask);
    }

    return {
      ...state,
      createdTasks
    };
  } catch (error) {
    return {
      ...state,
      errors: [...state.errors, `Creation error: ${error.message}`]
    };
  }
},

// Notify users about created tasks
notifyUsers: async (state) => {
  try {
    if (state.createdTasks.length > 0) {
      // Notify via socket
      socketService.notifyGroup(
        state.groupId,
        'tasks:ai_generated',
        {
          taskCount: state.createdTasks.length,
          tasks: state.createdTasks
        }
      );

      // Add system message to chat
      addSystemMessage(
        state.groupId,
        `I've created ${state.createdTasks.length} new task${state.createdTask
```

```
s.length > 1 ? 's' : ''} based on your conversation:\\n${state.createdTasks.map
(t => `• ${t.title}`).join('\\n')}`
        );
      }

      return state;
    } catch (error) {
      return {
        ...state,
        errors: [...state.errors, `Notification error: ${error.message}`]
      };
    }
  }
};

// Define the edges (conditional routing)
const edges = {
  collectContext: (state) => {
    if (state.errors.length > 0) return "END";
    if (!state.conversation) return "END";
    return "detectTasks";
  },

  detectTasks: (state) => {
    if (state.errors.length > 0) return "END";
    if (!state.containsTasks || state.confidence < 60) return "END";
    return "extractTasks";
  },

  extractTasks: (state) => {
    if (state.errors.length > 0) return "END";
    if (!state.tasks.length) return "END";
    return "validateTasks";
  },

  validateTasks: (state) => {
```

```
    if (state.errors.length > 0) return "END";
    if (!state.validatedTasks.length) return "END";
    return "createTasks";
  },

  createTasks: (state) => {
    if (state.createdTasks.length > 0) return "notifyUsers";
    return "END";
  },

  notifyUsers: (state) => "END"
};

// Create the graph
const taskExtractionGraph = new StateGraph({
  channels: {
    conversation: null,
    containsTasks: false,
    confidence: 0,
    tasks: [],
    validatedTasks: [],
    createdTasks: [],
    errors: []
  },
  nodes,
  edges
});

// Compile the graph
const runTaskExtraction = taskExtractionGraph.compile();
```

d. Task Creation & Notification

Purpose: Create todo items and notify users based on LangChain extraction

Implementation:

```
// Function to process a group's messages
async function processGroupMessages(groupId) {
 try {
  // Get recent messages
  const messages = await ChatMessage.find({ groupId })
   .populate('sender', 'username email')
   .sort({ createdAt: -1 })
   .limit(20);

  if (messages.length < 3) {
   return { status: 'insufficient_context' };
  }

  // Get group members for assignee matching
  const group = await Group.findById(groupId).populate('members.user');

  // Run the LangGraph workflow
  const result = await runTaskExtraction({
   messages: messages.reverse(),
   groupId,
   groupMembers: group.members
  });

  console.log('Task extraction result:', result);

  return {
   status: result.errors.length > 0 ? 'error' : 'success',
   taskCount: result.createdTasks.length,
   tasks: result.createdTasks,
   errors: result.errors
  };
 } catch (error) {
  console.error('Error processing group messages:', error);
  return {
   status: 'error',
```

```
      error: error.message
    };
  }
}
```

e. User Review Interface

Purpose: Allow users to verify AI-generated tasks

Features:

1. Separated display of unverified AI tasks
2. Approve/reject actions for each task
3. Bulk approve option for quick verification
4. Edit capabilities before approval
5. Feedback mechanism for improving extraction

6. Benefits of LangChain & LangGraph Approach

1. **Structured Processing Pipeline**:
   - Clear separation of concerns with dedicated components
   - Each processing step has well-defined inputs and outputs
   - Easy to reason about the flow of data
2. **Conditional Routing**:
   - Handles edge cases automatically
   - Skips unnecessary processing
   - Graceful error handling at each step
3. **Maintainability**:
   - Each component can be tested in isolation
   - Easy to add new processing steps
   - Simplified debugging with clear state transitions

4. **Scalability**:
    - Asynchronous processing throughout
    - Can be distributed across multiple workers
    - Backpressure handling with queue system
5. **Advanced Features**:
    - Retry logic built into processing nodes
    - State persistence for long-running processes
    - Detailed logging and monitoring capabilities

7. Helper Functions

Format Conversation

```
function formatConversation(messages) {
  return messages.map(msg ⇒
    `[${msg.sender.username} - ${new Date(msg.createdAt).toLocaleString()}]
     ${msg.content}
    `
  ).join('\\n\\n');
}
```

Normalize Date

```
function normalizeDate(dateString) {
  if (!dateString) return null;

  // Check if it's already a valid ISO date
  if (/^\\d{4}-\\d{2}-\\d{2}$/.test(dateString)) {
    return dateString;
  }

  try {
```

```
  // Try to parse relative dates like "tomorrow" or "next Friday"
  const now = new Date();

  // Simple relative date parsing
  if (/tomorrow/i.test(dateString)) {
    const tomorrow = new Date(now);
    tomorrow.setDate(tomorrow.getDate() + 1);
    return tomorrow.toISOString().split('T')[0];
  }

  if (/next week/i.test(dateString)) {
    const nextWeek = new Date(now);
    nextWeek.setDate(nextWeek.getDate() + 7);
    return nextWeek.toISOString().split('T')[0];
  }

  // Add more relative date parsing logic as needed

  // Last resort: try to parse with Date
  const parsedDate = new Date(dateString);
  if (!isNaN(parsedDate.getTime())) {
    return parsedDate.toISOString().split('T')[0];
  }
 } catch (error) {
  console.error('Error parsing date:', error);
 }

 return null;
}
```

Normalize Priority

```
function normalizePriority(priority) {
 if (!priority) return 'medium';
```

```
  const p = priority.toLowerCase().trim();

  if (p === 'high' || p === 'urgent' || p === 'important') return 'high';
  if (p === 'low' || p === 'minor') return 'low';

  return 'medium';
}
```

Find User By Name

```
async function findUserByName(name, groupMembers) {
  if (!name || !groupMembers) return null;

  // Normalize name for comparison
  const normalizedName = name.toLowerCase().trim();

  // Look through group members
  for (const member of groupMembers) {
    if (member.user && typeof member.user === 'object') {
      const username = member.user.username?.toLowerCase() || '';

      if (username.includes(normalizedName) || normalizedName.includes(user
name)) {
        return member.user._id;
      }
    }
  }

  return null;
}
```

8. Implementation Timeline

Phase 1: Basic Chat Functionality (1-2 weeks)

1. Implement chat data models
2. Create basic chat API endpoints
3. Add real-time socket communication
4. Build frontend chat UI components

Phase 2: LangChain & LangGraph Setup (1-2 weeks)

1. Install required packages
2. Set up model instances and prompt templates
3. Create individual chain components
4. Build LangGraph workflow

Phase 3: Task Integration (1-2 weeks)

1. Modify todo service to handle AI-generated tasks
2. Implement task verification workflow
3. Create AI task review interface
4. Add notification system for new AI tasks

Phase 4: Refinement & Optimization (1 week)

1. Add feedback collection for task accuracy
2. Implement prompt optimization based on feedback
3. Add rate limiting and cost control measures
4. Optimize performance for larger conversations

9. Testing and Quality Assurance

Unit Testing LangChain Components

```
describe('TaskDetectionChain', () => {
  test('Should detect tasks in conversation', async () => {
    const conversation = [
```

```
      '[User1] We need to finish the website by next Friday.',
      '[User2] I\\'ll work on the design tomorrow.'
    ].join('\\n\\n');

    const result = await taskDetectionChain.invoke({ conversation });

    expect(result.containsTasks).toBe(true);
    expect(result.confidence).toBeGreaterThan(70);
  });

  test('Should not detect tasks in casual conversation', async () => {
    const conversation = [
      '[User1] How was your weekend?',
      '[User2] It was great, we went hiking.'
    ].join('\\n\\n');

    const result = await taskDetectionChain.invoke({ conversation });

    expect(result.containsTasks).toBe(false);
  });
});
```

Testing LangGraph Workflow

```
describe('TaskExtractionGraph', () => {
  test('Should extract tasks from conversation', async () => {
    const messages = [
      {
        sender: { username: 'User1' },
        content: 'We need to finish the website by next Friday.',
        createdAt: new Date()
      },
      {
        sender: { username: 'User2' },
        content: 'I\\'ll work on the design tomorrow.',
```

```
      createdAt: new Date()
    }
  ];

  const result = await runTaskExtraction({
    messages,
    groupId: 'test-group',
    groupMembers: []
  });

  expect(result.errors.length).toBe(0);
  expect(result.tasks.length).toBeGreaterThan(0);
  expect(result.tasks[0]).toHaveProperty('title');
  expect(result.tasks[0]).toHaveProperty('dueDate');
 });
});
```

10. Monitoring and Optimization

Performance Monitoring

```
class LangChainPerformanceMonitor {
 constructor() {
  this.stats = {
   detectionRequests: 0,
   extractionRequests: 0,
   successfulExtractions: 0,
   failedExtractions: 0,
   avgProcessingTimeMs: 0,
   totalProcessingTimeMs: 0,
   tasksCreated: 0
  };
 }

 recordDetection(startTime, success) {
```

```
    const duration = Date.now() - startTime;
    this.stats.detectionRequests++;
    this.updateAvgTime(duration);

    return {
      duration,
      success
    };
  }

  recordExtraction(startTime, taskCount, success) {
    const duration = Date.now() - startTime;
    this.stats.extractionRequests++;

    if (success) {
      this.stats.successfulExtractions++;
      this.stats.tasksCreated += taskCount;
    } else {
      this.stats.failedExtractions++;
    }

    this.updateAvgTime(duration);

    return {
      duration,
      taskCount,
      success
    };
  }

  updateAvgTime(duration) {
    const totalTime = this.stats.avgProcessingTimeMs * this.stats.totalProcessi
ngTimeMs;
    this.stats.totalProcessingTimeMs++;
    this.stats.avgProcessingTimeMs = (totalTime + duration) / this.stats.totalPro
cessingTimeMs;
```

```
  }

  getStats() {
    return {
      ...this.stats,
      successRate: this.stats.extractionRequests > 0
        ? (this.stats.successfulExtractions / this.stats.extractionRequests) * 100
        : 0
    };
  }
}

const performanceMonitor = new LangChainPerformanceMonitor();
export default performanceMonitor;
```

This comprehensive workflow provides a detailed structure for implementing a group chat system with AI task extraction using LangChain and LangGraph. It builds upon your existing todo app structure while adding sophisticated AI processing capabilities with a structured, maintainable architecture.