AI Companion Implementation Summary

Features Implemented

1. Markdown Rendering

- Integrated MarkdownUI library for rendering markdown content
- Created a custom MarkdownView component with:
  - Loading states
  - Error handling
  - GitHub-style theme
  - Support for inline images and links
  - Proper styling and layout

2. Code Syntax Highlighting

- Integrated Highlightr library for syntax highlighting
- Created a custom CodeBlockView component with:
  - Language detection and display
  - Syntax highlighting based on language
  - Copy button with feedback
  - Proper styling with scrolling for long code blocks
- Created a custom HighlightrSyntaxHighlighter for MarkdownUI integration
- Added support for multiple themes (Xcode, GitHub, Monokai, etc.)

3. Conversation Summarization

- Implemented TokenCounter service for estimating token usage
- Created ConversationSummary model for storing summaries
- Enhanced ConversationManager with:
  - AI-powered title generation
  - AI-powered conversation summarization
  - Token counting and context window management
  - Automatic detection of when summarization is needed
  - Methods for including summaries in conversation context
- Updated ChatViewModel to use the enhanced ConversationManager
- Added UI elements to display token usage and trigger summarization

Implementation Details

Markdown Rendering

The MarkdownView component uses MarkdownUI to render markdown content with proper styling and formatting. It includes loading states and error handling to ensure a smooth user experience. The component is used for both user and assistant messages, providing a consistent look and feel.

```
struct MarkdownView: View {
```

```
    let content: String
    @State private var isLoading = true
    @State private var error: Error? = nil

    var body: some View {
        Group {
            if isLoading {
                ProgressView()
                    .onAppear {
                        // Simulate a short loading time for better UX
                        DispatchQueue.main.asyncAfter(deadline: .now() + 0.1) {
                            isLoading = false
                        }
                    }
            } else if let error = error {
                // Error view
                // ...
            } else {
                Markdown(content)
                    .markdownTheme(.gitHub)
                    .markdownCodeSyntaxHighlighter(.highlightr(theme: .xcode))
                    .markdownImageProvider(.asset())
                    // ...
            }
        }
        // ...
    }
    // ...
}
```

Code Syntax Highlighting

The CodeBlockView component uses Highlightr to provide syntax highlighting for code blocks. It includes a header with the language name and a copy button, and properly handles scrolling for long code blocks.

```
struct CodeBlockView: View {
    let code: String
    let language: String?
    @State private var isCopied = false

    private let highlightr = Highlightr()

    var body: some View {
        VStack(alignment: .leading, spacing: 0) {
            // Code block header
```

```
            // ...

            // Code content with syntax highlighting
            ScrollView(.horizontal, showsIndicators: false) {
                if let highlightedCode = highlightedCode {
                    highlightedCode
                        .padding(12)
                } else {
                    Text(code)
                        .font(.system(.body, design: .monospaced))
                        .padding(12)
                }
            }
            // ...
        }
        // ...
    }
    // ...
}
```

Conversation Summarization

The TokenCounter service provides methods for estimating token usage in messages and conversations. The ConversationManager uses this service to determine when a conversation needs to be summarized and to ensure that the context window fits within the model's limits.

```
class TokenCounter {
    // ...

    func estimateTokenCount(for text: String) -> Int {
        // Simple estimation based on character count
        return Int(ceil(Double(text.count) / averageCharsPerToken))
    }

    func estimateTokenCount(for message: Message) -> Int {
        // Add overhead for message metadata (role, etc.)
        let roleOverhead = 4 // Approximate token overhead for role
        return roleOverhead + estimateTokenCount(for: message.content)
    }

    func estimateTokenCount(for messages: [Message]) -> Int {
        // Sum the token counts for all messages
        return messages.reduce(0) { $0 + estimateTokenCount(for: $1) }
    }
```

```
    // ...
}
```

The ConversationManager includes methods for generating AI-powered summaries and titles, and for including these summaries in the conversation context.

```
func summarizeConversationWithAI(_ conversation: Conversation) async throws -> ConversationS
    // Get messages to summarize
    let messagesToSummarize = getMessagesToSummarize(conversation)

    // Create a system prompt for summarization
    let systemPrompt = """
    You are a helpful assistant that summarizes conversations. Create a concise summary of t
    Focus on the key points, questions, and answers. The summary should be informative enoug
    but brief enough to save tokens. Use bullet points for clarity.
    """

    // Format the conversation for the AI
    // ...

    // Send the request to the AI router
    let response = try await AIRouter.shared.routeMessage(messages: messages, options: optio

    // Create a summary object
    let summary = ConversationSummary(
        conversationId: conversation.id,
        content: summaryContent,
        summarizedMessageIds: messagesToSummarize.map { $0.id }
    )

    // Save the summary
    saveSummary(summary)

    return summary
}
```

The ChatViewModel includes methods for checking if a conversation needs summarization and for manually triggering summarization.

```
var conversationNeedsSummarization: Bool {
    return conversationManager.conversationNeedsSummarization(conversationManager.currentCon
}

func summarizeConversation() async {
    isGenerating = true

    do {
        let summary = try await conversationManager.summarizeConversationWithAI(conversation
```

```
        await MainActor.run {
            // Show a notification that summarization was successful
            // ...
        }
    } catch {
        await MainActor.run {
            errorMessage = "Failed to summarize conversation: \(error.localizedDescription)"
            showError = true
            isGenerating = false
        }
    }
}
```

The ChatView includes UI elements for displaying token usage and triggering summarization.

```
// Token usage indicator
HStack {
    Text("Tokens: \(viewModel.currentConversationTokenCount) / \(viewModel.currentModelToken
        .font(.caption)
        .foregroundColor(viewModel.conversationNeedsSummarization ? .orange : .secondary)

    if viewModel.conversationNeedsSummarization {
        Button(action: {
            Task {
                await viewModel.summarizeConversation()
            }
        }) {
            Label("Summarize", systemImage: "text.append")
                .font(.caption)
        }
        .buttonStyle(.bordered)
        .controlSize(.small)
    }
}
```

Conclusion

The implementation provides a robust solution for rendering markdown content, highlighting code syntax, and managing conversation context through summarization. The code is well-structured, follows best practices, and provides a good user experience.