🛠️ God Mode Technical Implementation

Core Components

1. Oracle Tracker (`src/features/god-mode/oracle-tracker/`)

Parser (`parser.ts`)

```
interface OracleLog {
  id: string; // Unique identifier for the log entry
  timestamp: number; // Unix timestamp of the event
  type: "input" | "output"; // Type of log entry
  content: string; // Raw content of the message
  metadata: {
    domain: string; // Domain of the chat (e.g.,
        'chat.openai.com')
    chatId: string; // Unique identifier for the chat
    contextWindow: number; // Current context window size
    messageId?: string; // Platform-specific message ID
    parentId?: string; // Parent message ID for threading
    role?: "user" | "assistant"; // Role of the message sender
  };
  raw: {
    html: string; // Original HTML content
    text: string; // Extracted text content
    markdown: string; // Converted markdown
  };
}

class OracleParser {
  private static instance: OracleParser;
  private observers: Set<MutationObserver>;
  private textareaObserver: MutationObserver;

  private constructor() {
    this.observers = new Set();
    this.textareaObserver = new
        MutationObserver(this.handleTextareaChanges);
  }

  static getInstance(): OracleParser {
    if (!OracleParser.instance) {
      OracleParser.instance = new OracleParser();
    }
    return OracleParser.instance;
  }
```

```
// Start tracking a specific chat container
startTracking(container: HTMLElement): void {
  const observer = new
       MutationObserver(this.handleChatChanges);
  observer.observe(container, {
    childList: true,
    subtree: true,
    characterData: true,
  });
  this.observers.add(observer);
}

// Handle changes in textareas (user input)
private handleTextareaChanges(mutations: MutationRecord[]):
       void {
  for (const mutation of mutations) {
    if (mutation.target instanceof HTMLTextAreaElement) {
      this.parseInput(mutation.target);
    }
  }
}

// Handle changes in chat messages (AI output)
private handleChatChanges(mutations: MutationRecord[]): void {
  for (const mutation of mutations) {
    if (this.isChatMessage(mutation.target)) {
      this.parseOutput(mutation.target);
    }
  }
}

// Parse user input from textarea
parseInput(textarea: HTMLTextAreaElement): OracleLog {
  return {
    id: generateUniqueId(),
    timestamp: Date.now(),
    type: "input",
    content: textarea.value,
    metadata: this.extractMetadata(),
    raw: {
      html: textarea.outerHTML,
      text: textarea.value,
      markdown: this.htmlToMarkdown(textarea),
    },
  };
}
```

```
  // Parse AI output from message element
  parseOutput(message: HTMLElement): OracleLog {
    return {
      id: generateUniqueId(),
      timestamp: Date.now(),
      type: "output",
      content: message.textContent || "",
      metadata: this.extractMetadata(),
      raw: {
        html: message.outerHTML,
        text: message.textContent || "",
        markdown: this.htmlToMarkdown(message),
      },
    };
  }

  // Convert HTML to markdown
  private htmlToMarkdown(element: HTMLElement): string {
    // Implementation of HTML to Markdown conversion
    // Handles code blocks, lists, links, etc.
  }
}
```

Storage (`storage.ts`)

```
interface StorageConfig {
  local: {
    maxSize: number; // Maximum size in bytes
    compression: boolean; // Whether to compress stored data
  };
  gist: {
    enabled: boolean; // Whether Gist sync is enabled
    updateInterval: number; // Sync interval in milliseconds
    maxSize: number; // Maximum size per Gist
  };
}

class OracleStorage {
  private static instance: OracleStorage;
  private config: StorageConfig;
  private queue: OracleLog[];
  private isSyncing: boolean;

  private constructor() {
    this.config = this.loadConfig();
    this.queue = [];
    this.isSyncing = false;
  }
```

```
static getInstance(): OracleStorage {
  if (!OracleStorage.instance) {
    OracleStorage.instance = new OracleStorage();
  }
  return OracleStorage.instance;
}

// Save a log entry
async saveLog(log: OracleLog): Promise<void> {
  this.queue.push(log);
  await this.processQueue();
}

// Process the queue of logs
private async processQueue(): Promise<void> {
  if (this.isSyncing) return;
  this.isSyncing = true;

  try {
    // Save to local storage
    await this.saveToLocal();

    // Sync to Gist if enabled
    if (this.config.gist.enabled) {
      await this.syncToGist();
    }
  } finally {
    this.isSyncing = false;
  }
}

// Save to local storage
private async saveToLocal(): Promise<void> {
  const logs = await this.getLocalLogs();
  logs.push(...this.queue);

  // Apply size limits and compression
  const processedLogs = this.processLogs(logs);

  await chrome.storage.local.set({
    oracle_logs: processedLogs,
  });

  this.queue = [];
}

// Sync to GitHub Gist
private async syncToGist(): Promise<void> {
```

```
    const logs = this.queue;
    if (logs.length === 0) return;

    const gistData = this.prepareGistData(logs);
    await this.updateGist(gistData);
  }

  // Retrieve full context for a chat
  async getFullContext(chatId: string): Promise<OracleLog[]> {
    const logs = await this.getLocalLogs();
    return logs.filter((log) => log.metadata.chatId === chatId);
  }
}
```

2. Monk Mode (`src/features/god-mode/monk-mode/`)

Core (`index.ts`)

```
class MonkMode {
  private static instance: MonkMode;
  private isEnabled: boolean;
  private parser: OracleParser;
  private storage: OracleStorage;

  private constructor() {
    this.isEnabled = false;
    this.parser = OracleParser.getInstance();
    this.storage = OracleStorage.getInstance();
  }

  static getInstance(): MonkMode {
    if (!MonkMode.instance) {
      MonkMode.instance = new MonkMode();
    }
    return MonkMode.instance;
  }

  // Enable Monk Mode
  async enable(): Promise<void> {
    this.isEnabled = true;
    await this.initializeTracking();
  }

  // Initialize tracking for all chat containers
  private async initializeTracking(): Promise<void> {
    const containers = document.querySelectorAll(".chat-
        container");
    containers.forEach((container) => {
```

```
      this.parser.startTracking(container);
    });
  }

  // Handle keyboard shortcuts
  handleShortcut(event: KeyboardEvent): void {
    if (event.key === "g" && (event.metaKey || event.ctrlKey)) {
      this.showOracleViewer();
    }
  }
}
```

3. BFG Mode (`src/features/god-mode/bfg/`)

Context Injection (`index.ts`)

```
class BFGContextInjector {
  private static instance: BFGContextInjector;
  private storage: OracleStorage;

  private constructor() {
    this.storage = OracleStorage.getInstance();
  }

  static getInstance(): BFGContextInjector {
    if (!BFGContextInjector.instance) {
      BFGContextInjector.instance = new BFGContextInjector();
    }
    return BFGContextInjector.instance;
  }

  // Inject context into a chat
  async injectContext(chatId: string, context: OracleLog[]):
        Promise<void> {
    // Implementation for injecting context back into the chat
    // This will vary based on the platform's API
  }

  // Monitor for deleted messages
  async monitorDeletions(): Promise<void> {
    // Implementation for detecting and restoring deleted
        messages
  }
}
```

Implementation Notes

1. Performance Considerations

- Use Web Workers for heavy processing tasks
- Implement efficient data structures for log storage
- Optimize DOM observation to minimize performance impact
- Use compression for large log entries

2. Security Measures

- Encrypt sensitive data before storage
- Implement rate limiting for Gist sync
- Validate all input data
- Use secure storage mechanisms

3. Error Handling

- Implement robust error recovery
- Maintain backup copies of critical data
- Provide detailed error logging
- Handle network failures gracefully

4. Browser Compatibility

- Support Chrome, Firefox, and Edge
- Handle different DOM structures
- Adapt to platform-specific APIs
- Maintain consistent behavior across browsers

Testing Strategy

1. Unit Tests

- Test parser functionality
- Test storage operations
- Test context injection
- Test error handling

2. Integration Tests

- Test end-to-end logging
- Test Gist synchronization
- Test context restoration
- Test UI interactions

3. Performance Tests

- Measure memory usage
- Test with large datasets
- Monitor DOM performance

- Test sync operations

Deployment Considerations

1. Versioning

- Implement semantic versioning
- Maintain backward compatibility
- Handle data migration
- Document breaking changes

2. Monitoring

- Track usage metrics
- Monitor error rates
- Measure performance
- Collect user feedback

3. Updates

- Implement automatic updates
- Handle data migration
- Preserve user settings
- Maintain backward compatibility