Implementation Plan: 'Claude' - Color Syntax Highlighting

Overall Goal: To implement the "Claude" feature, enabling the generation of reMarkable notebooks where source code (or other text) is rendered as editable ink with color syntax highlighting, leveraging the reMarkable Pro's color capabilities.

Core Technology Choices (Informed by Research Document):

- **Generation Engine:** drawj2d (for converting .hcl scripts to .rm/.rmdoc files with colored, editable ink).
- **File Analysis & Verification:** rmscene (for inspecting generated files and potentially for packaging/metadata tasks if drawj2d output needs augmentation).
- **Syntax-to-HCL Translation:** A new Python-based component (the "Compiler").

Key Development Component:

- **Syntax-to-HCL Compiler:** A Python module responsible for:
  1. Lexical analysis of input source code.
  2. Mapping code tokens to a predefined color scheme compatible with drawj2d and reMarkable Pro.
  3. Generating a .hcl script with appropriate drawj2d commands for text rendering, color changes, and layout management.

Phase 1: drawj2d Environment Setup and Basic Invocation

Objective: Establish the ability to run drawj2d from the InkLink Python environment and render a simple .hcl file.

- **Tasks:**
  1. **drawj2d Accessibility:**
     - Ensure the drawj2d.jar executable is available and can be called by the InkLink application (e.g., included in the project, or path configurable).
  2. **Python Wrapper for drawj2d:**
     - Develop or enhance a Python utility (potentially within src/inklink/utils/hcl_render.py if it's suitable, or a new module) to:
       - Take a path to a .hcl file as input.
       - Invoke drawj2d via subprocess.
       - Specify output format (e.g., -Trmdoc).
       - Capture output file path, standard output, and errors from drawj2d.
  3. **Basic .hcl Test Script:**
     - Create a minimal .hcl file that draws a few lines of text in black using font LinesMono (as recommended by the research for code).
     - Example:
       ```
       # test_basic.hcl
       font LinesMono 3.0
       m 10 10
       pen black
       ```

```
text {Hello reMarkable!}
m 10 20
text {This is drawj2d.}
```

4. **Initial Test:**
   - Use the Python wrapper to execute drawj2d with test_basic.hcl.
   - Verify a .rmdoc file is created.
   - Manually transfer to a reMarkable device (or use rmapi if already integrated for uploads) and confirm it opens and text is visible and editable.

Phase 2: "Syntax-to-HCL Compiler" - Core Logic (Monocolor Text)

Objective: Develop the initial version of the compiler to convert source code into a monocolor, inked representation using drawj2d.

- **Tasks:**
  1. **Lexical Analysis Integration:**
     - Choose and integrate a Python lexical analysis library (e.g., Pygments is highly recommended in the research for its broad language support).
     - Create a component that takes source code text and a language identifier (e.g., "python", "java") as input and outputs a stream of tokens.
  2. **Basic HCL Generation Logic:**
     - Develop the core of the compiler to iterate through tokens from the lexer.
     - **Font and Initial Setup:** Start the .hcl script by setting a monospaced font (e.g., font LinesMono 3.0) and an initial pen color (e.g., pen black).
     - **Text Rendering:** For each token, generate a text {token_string} command.
     - **Layout Management (Basic):**
       - Implement logic to handle newlines in the source by advancing the Y-coordinate (m or linerel commands).
       - Handle basic spacing between tokens on the same line. For now, simple concatenation or a fixed space might suffice, to be refined later.
       - The src/inklink/services/ink_generation_service.py might provide a structural basis for this, but the output target is now .hcl commands instead of direct rmscene Point objects.
  3. **End-to-End Test (Monocolor):**
     - Input a simple source code snippet (e.g., a few lines of Python).
     - The compiler generates a .hcl file.
     - Use the Phase 1 Python wrapper to process this .hcl with drawj2d.
     - Verify the output .rmdoc on a reMarkable device: text should appear as ink, respecting line breaks.

Phase 3: "Syntax-to-HCL Compiler" - Color Syntax Highlighting

Objective: Enhance the compiler to apply colors to tokens based on their type, achieving syntax highlighting.

- **Tasks:**
    1. **Define Token-to-Color Mapping:**
        - Create a configurable mapping from token types (e.g., pygments.token.Keyword, Comment, String, Name.Function) to drawj2d-supported reMarkable Pro color names (referencing Table 1 from the research document, e.g., blue, grey, green, black).
        - This configuration could be a Python dictionary or a loadable JSON/YAML file.
    2. **Colorized HCL Generation:**
        - Modify the HCL generation logic:
            - Before rendering a token, check if its type requires a color change based on the mapping.
            - If so, emit the appropriate pen <color_name> command.
            - Optimize by only emitting pen commands when the color *actually changes* from the previous token.
    3. **Testing with Colors:**
        - Test with various source code examples from different languages.
        - Verify on a reMarkable Pro (or an emulator that supports its color palette) that syntax highlighting is applied correctly.
        - Use rmscene to programmatically inspect the generated .rm files (if drawj2d outputs .rm directly for pages) to check color properties of strokes, if feasible and necessary for debugging.

Phase 4: "Syntax-to-HCL Compiler" - Advanced Layout and Refinements

Objective: Improve the visual fidelity of the rendered code, especially indentation and spacing.

- **Tasks:**
    1. **Precise Layout Management:**
        - Implement more sophisticated logic for positioning tokens. This is critical for accurate indentation.
        - Since font LinesMono is monospaced, calculate the width of each character (or assume a fixed width) and the height of lines.
        - Use m x_coord y_coord (absolute move) or linerel dx dy (relative move) commands in the .hcl script to precisely place each token.
        - Handle leading spaces/tabs for indentation by calculating the required horizontal offset.
    2. **Whitespace Handling:**
        - Decide how to handle multiple whitespace characters (e.g., render them, or consolidate and calculate equivalent space).
    3. **Tab Character Expansion:**
        - Implement logic to expand tab characters (\t) into a configurable number of

spaces (e.g., 4 or 8).

4. **Configuration Options:**
   - Allow users to configure:
     - Font size.
     - Tab width.
     - Color scheme.
5. **Performance for Large Files (Consideration):**
   - The research notes potential performance issues. While full optimization might be later, keep an eye on the size of generated .hcl files and drawj2d processing time.
   - Consider if pagination (splitting very long source files into multiple reMarkable pages within one .rmdoc) is necessary. drawj2d might handle single long pages well with reMarkable's continuous scroll.

Phase 5: Full Notebook Integration and rmscene Utilization

Objective: Integrate the drawj2d-based generation into the broader InkLink services and ensure fully compliant and feature-rich reMarkable notebooks.

- **Tasks:**
  1. **Service Integration:**
     - Create a new service (e.g., SyntaxHighlightingService) or adapt the existing InkGenerationService to manage the "Syntax-to-HCL Compiler" and drawj2d pipeline.
     - This service should accept source code text, language, and any formatting options.
     - It will output a path to the generated .rmdoc file.
  2. **.rmdoc Packaging and Metadata:**
     - drawj2d can output .rmdoc files directly. Evaluate if this output is sufficient or if InkLink needs to further process it.
     - **If drawj2d output is sufficient:** Ensure it includes necessary metadata (e.g., for title, last modified).
     - **If drawj2d output needs augmentation (e.g., for custom metadata, cover pages, or if drawj2d only outputs single .rm page files):**
       - Use rmscene.scene_stream.TaggedBlockWriter to construct the full .rmdoc structure, incorporating the page(s) generated by drawj2d. This leverages your existing rmscene expertise for creating the notebook shell, while drawj2d handles the complex ink generation.
       - This aligns with the original plan's Step 1 ("Implement complete rmscene file serialization") but focuses it on packaging rather than stroke generation for this specific feature.
  3. **Multi-Page Document Support (if needed for long code):**
     - If pagination is implemented in Phase 4, ensure the .rmdoc correctly structures multiple pages.
  4. **Verification with rmscene:**

        - Use rmscene.read_tree to parse the generated .rmdoc files as a final verification step. Check for structural integrity, correct page count, and presence of expected metadata and content blocks.

Phase 6: User Interface and Cloud Integration

Objective: Expose the "Claude" feature to users via the web UI and enable reMarkable cloud synchronization.

- **Tasks:**
    1. **Web UI Enhancements (web/index.html, web/main.js):**
        - Add UI elements for:
            - Pasting or uploading source code.
            - Selecting the programming language.
            - (Optional) Configuring syntax highlighting options (font size, color scheme if made user-configurable).
        - JavaScript logic to send the code and options to the backend InkLink server.
        - Display status and provide a download link for the generated .rmdoc file.
    2. **Backend Endpoint:**
        - Create a new Flask endpoint in src/inklink/server.py (or modify an existing one) to receive syntax highlighting requests.
        - This endpoint will call the SyntaxHighlightingService.
    3. **reMarkable Cloud API Integration:**
        - Integrate with the rmapi_adapter.py (or similar) to allow uploading the generated syntax-highlighted .rmdoc files directly to the user's reMarkable cloud account. This is from your original plan (Step 5).

Ongoing Considerations:

- **Extensible Character Patterns (for non-code text):** The original plan to support more character patterns for generating *handwriting-style ink from plain text* (using rmscene's Line and Point objects directly) remains a valid, separate feature. The drawj2d approach is specifically for pre-formatted, syntax-highlighted content.
- **Error Handling:** Implement robust error handling throughout the pipeline (lexer, HCL generator, drawj2d invocation).
- **Testing:** Rigorous testing at each phase, including on-device testing with a reMarkable Pro if possible, is crucial. Your existing test infrastructure (tests/) should be expanded.

This revised plan heavily leverages the insights from your research document, prioritizing drawj2d for the generation of colored ink as recommended. It breaks down the development of the "Syntax-to-HCL Compiler" into manageable stages and clarifies the role of rmscene in this specific context.