ChatGPT

Implementation Plan: Integrating Claude Code into InkLink

Use Cases Suited for Claude Code in InkLink

- **Handwritten Pseudocode to Code Generation:** Leverage Claude Code's strength in code synthesis to convert rough input (e.g. pseudocode sketched on the reMarkable or voice-transcribed instructions) into actual code. InkLink's persona "Lilly" can send transcribed text (via Claude Vision) to Claude Code to produce initial Python function implementations or UI skeletons [1]. This supports an iterative **Ink-to-Code** workflow: the user sketches a flow or writes pseudo-code by hand, Claude generates baseline code, then further handwritten annotations/voice notes can prompt refinements by Claude [1].
- **AI-Assisted Code Review & Debugging:** Use Claude to analyze existing code for issues, improvements, or explanations. When the user tags or requests a review of a code snippet, an agent can feed the snippet (or even an image of code via Claude Vision) into Claude Code and ask for feedback. Claude can highlight bugs, suggest fixes, or explain the code's logic in simple terms [2]. In debugging scenarios, if an error occurs (e.g. stack trace or failing test), an agent can prompt Claude: *"Here's the error and relevant code – what likely went wrong and how to fix it?"* [2]. This turns Claude into a cloud-based "senior developer" that offers best-practice recommendations and error explanations beyond what local models might manage.
- **Technical Summaries & Documentation:** Claude Code can generate high-level summaries of technical content for the user. For example, after a coding session, an agent could ask Claude to **summarize the day's coding decisions** or produce documentation (function docstrings, design rationale) based on code diffs or note transcripts. It could also draft **commit messages** or PR descriptions by analyzing changes [1]. These tasks benefit from Claude's larger context window and training on best practices, yielding more coherent and context-aware documentation than a small local model might.
- **Researching Best Practices & Prototyping:** When the user poses a broad development question (e.g. *"What's the best library for X, and how should I structure Y?"*), Claude can be invoked to do a deep dive. InkLink can offload such questions to Claude Code, which can draw on its extensive training to suggest best practices or even outline a solution. The **AI Dev Team** concept can include an agent that, upon a request like *"Research best practices for using API Z and draft a usage example,"* uses Claude to gather insights and produce a brief report or code skeleton [3]. Because Claude has more comprehensive knowledge of tools/frameworks, it can propose solutions that local models might not know. The results would then be returned to the reMarkable (e.g. as an ink note or daily briefing) for the user to review.

Orchestrating Claude Code Alongside Local Agents

- **Intelligent Task Routing:** Implement a routing mechanism to decide when to engage Claude Code versus local models. InkLink's architecture already envisions a **hybrid AI processing** approach where local LLMs handle simple or privacy-sensitive tasks, and cloud models handle complex ones [4]. The

plan is to extend this: for any task classified as *code generation/review, large-text summarization, or requiring external knowledge*, automatically route the request to a Claude-backed agent [5]. This could be done via a central orchestrator or within the Multi-Connection Protocol (MCP) messaging layer.

- **Claude as a Specialized Agent ("Cloud Coder"):** Introduce a dedicated Claude-based agent in the "AI Dev Team." This agent would be orchestrated like other agents (e.g. `LimitlessContextualInsightAgent` or `DailyBriefingAgent`), but its tool is the Claude Code CLI. For example, when the user adds a `#code` tag or a developer-oriented query is detected, the orchestrator can package the context and delegate it to this Claude agent. The agent then returns the output (code, review, or summary) to be placed back into the workflow (e.g. rendered as an `.rm` ink note or spoken via the pendant). This design keeps Claude's usage modular and in line with other agents' patterns.
- **MCP Integration for Delegation:** Use InkLink's MCP (Multi-Connection Protocol) to enable seamless inter-agent communication. Local agents can act as intermediaries that decide "this task is too complex – send to Claude." For instance, the **Lilly persona** can function as a manager: it parses a user's handwritten query, and if it recognizes a coding task, it formulates a Claude prompt and invokes the Claude agent, then relays the answer back. The MCP message could contain fields like `{"task": "code_review", "content": "<code or query>"}` which the Claude agent listens for. This asynchronous, message-based design means Claude can work in parallel while the rest of the system remains responsive [6].
- **Adapters and Modular Integration:** Extend the InkLink adapters to include cloud AI models. The codebase likely has an `inklink.adapters` module for interfacing with different AI services. A new **ClaudeAdapter** (or similar) should be implemented [7], encapsulating all calls to the Claude CLI. This adapter can expose high-level methods like `generate_code(prompt)`, `review_code(snippet, instruction)`, `summarize_text(text)`, etc., which internally handle constructing the CLI command and parsing the output. By adding this adapter, existing agents can call Claude without needing to know CLI details – they simply request the adapter for results. The adapter pattern also makes it easy to add future models or switch out Claude if needed [7].
- **Workflow Orchestration Examples:** For clarity, consider a few end-to-end flows:
- *Ink-to-Code:* The sync service detects a `#code` tag on a note -> triggers Lilly (local agent) to OCR the note via Claude Vision -> Lilly forms a prompt and calls ClaudeAdapter.generate_code(...) -> Claude returns code text -> InkLink converts that into an ink file and syncs to tablet for user.
- *Code Review:* User takes a photo of code and tags it `#review` -> InkLink uses Claude Vision to get text -> a CodeReviewAgent sends the code text to ClaudeAdapter.review_code(...) -> result is delivered as an annotated note or an audio summary through the pendant.
- *Async Research:* User writes "Lilly, research X..." -> a ResearchAgent invokes Claude asynchronously (perhaps in a background thread or task queue) -> when Claude's answer arrives, the agent creates a summary note or briefing (possibly linking into the daily briefing system). This result could also be cached in the knowledge graph for future reference.

CLI Integration Patterns for Claude in Agent Workflows

- **One-off Prompt Calls:** Utilize Claude Code's one-shot mode for straightforward interactions. Agents can invoke the Claude CLI with a single command using the `-p` flag to get a quick response [8]. For example, an agent may execute: `claude -p "Explain what the function below does: \n<code snippet>"` and capture Claude's output. One-shot usage is ideal for stateless requests

like summarizing text or answering a single question, and it keeps the integration simple (spawn process, get result, terminate).

- **Piping Content for Analysis:** When dealing with large inputs (like long code files, logs, or transcripts), the agent can pipe data into Claude Code. For instance, to have Claude analyze error logs or a lengthy file, the agent could run: `cat errors.log | claude -p "Analyze these errors and suggest fixes"` [9]. The Claude CLI will read the piped content as context. InkLink's adapter should handle creating a temp file or pipe stream as needed, then assembling the full prompt to avoid hitting shell limits.
- **Maintaining Session Context:** For multi-turn interactions or iterative refinement, take advantage of Claude Code's session resume features. The CLI supports continuing the last conversation with `claude -c` or resuming a named session with `-r <session-id>` [10]. InkLink could assign a session ID per ongoing "thread" of interaction (e.g. a particular coding task) so that follow-up prompts keep Claude's context. For example, after an initial code generation call, subsequent refinement prompts could use `claude -c -p "Now refine that code to use a different API"` to leverage Claude's memory of the prior output. The adapter can store session identifiers in the agent state, ensuring continuity when needed.
- **Invoking via Subprocess:** The Claude integration will likely be done by calling the CLI from Python (since InkLink's backend is Python). The plan is to use a subprocess call (or an async equivalent) to run the `claude` command. The environment should include the proper CLI configurations (e.g. `CLAUDE_COMMAND=claude` and the chosen `CLAUDE_MODEL` in the environment) so that the CLI knows which model to use [11]. The adapter can construct the command string or argument list and then spawn the process. It should also capture STDOUT/STDERR for the response or any error messages. Wrapping this in a higher-level function simplifies usage across agents.
- **Output Parsing and Post-Processing:** Once Claude returns a result, the integration layer should handle formatting it for InkLink's use. For code outputs, this might involve extracting just the code block (if Claude wraps it in markdown) and inserting it into an `.rm` file template. For prose answers, the text can be directly written into an ink page or converted to audio (text-to-speech) if delivered via the pendant. Ensuring that the output is clean (e.g., removing any extra CLI annotations or slash-commands Claude might output) will be part of this post-processing.
- **Error Handling & Timeouts:** The CLI call should include safeguards: a timeout so that a long-running Claude process doesn't hang the system, and retry logic for transient failures. If the CLI exits with an error (non-zero status), the adapter can catch this and either fall back (if possible) or return a friendly error message to the user/agent. Logging these events is also useful for debugging integration issues.

Privacy Considerations and Fallback Strategies

- **Selective Data Sharing:** Since Claude Code runs in the cloud, be deliberate about what data is sent. InkLink should **avoid transmitting highly sensitive content** (personal notes, credentials, etc.) unless necessary. For code files or notes that the user marks as private, default to using local LLMs. The system could have a user-setting or per-request flag indicating "cloud-allowed." This aligns with the project's hybrid design where local AI is used for privacy by default, and cloud is opted-in for heavy tasks [4]. Agents can prompt the user or check a config before sending data to Claude.
- **Anonymization and Minimization:** When sending prompts to Claude, the integration can scrub identifiable info. For instance, replace user names or specific project names with generic placeholders if possible, or only send the necessary excerpt of a note or code relevant to the query

(rather than whole notebooks or repos). Claude doesn't need the entirety of your knowledge base to, say, debug a function – providing just the function and error message will limit data exposure.

- **Offline Mode and Degraded Performance:** Plan for times when Claude is unavailable (network outage, Claude service downtime) or the user chooses to disable cloud usage. The system should detect this and **gracefully degrade**. For example, if a user asks for a code review while offline, InkLink could respond with a polite note: "Cloud assistant not reachable; using local model." Then attempt a simpler analysis via a local model (Ollama) as a fallback. Although the local result may be less comprehensive, it is better than nothing, and the user is informed of the context.
- **Caching Claude's Outputs:** To enhance resilience and privacy, consider caching the results of Claude calls for reuse. For example, if Claude produces a summary of a particular note or a piece of code, store that in the knowledge graph or a local cache. If the same content needs analysis later and hasn't changed, the system can reuse the cached summary instead of calling the cloud again. This reduces repeated data exposure and provides quick answers when offline.
- **User Transparency and Control:** Make Claude's involvement transparent to the user. The InkLink control center could show an indicator (like a cloud icon) whenever a request is being handled by Claude, so the user knows the data is leaving the local device. Also provide a master switch for Claude integration. For instance, a user might disable "Cloud AI" during sensitive work sessions. In such cases, the agents should respect this and not invoke Claude, possibly queuing the requests or informing the user of limited functionality.
- **Security Measures:** Use the Claude API/CLI in a secure manner – ensure API keys or tokens for the Claude CLI are stored safely (in `.env` as done now) and not exposed. Since the Claude Code tool can theoretically execute code (it's agentic), be mindful if you ever allow it to run commands. In the current plan, we restrict Claude to providing answers/text which we then handle. If in future the integration allowed Claude to, for example, directly commit code or run tests, that should be behind strict user approval due to obvious security implications. For now, treat Claude as a read-only advisor whose suggestions are applied by InkLink or the user manually.

Prompting Strategies and Example Workflows for Claude

Using the right prompt patterns will guide Claude Code to produce helpful results. Below are examples of how to prompt Claude effectively for InkLink's purposes:

- **Handwritten Note to Code Conversion:** After using Claude Vision to get text from a note, prompt Claude with context and clear instructions. *Example: "You are a coding assistant. The user wrote the following pseudocode by hand:* `<transcribed text>`*. Convert this into a well-structured Python function. Include comments explaining each step. Ensure the code is idiomatic and handles errors."* This prompt gives Claude the role, the input, and the expectation (Python code with comments). In testing, include any specific style preferences (function name conventions, etc.) so that the generated code fits the user's style.
- **Code Review and Improvement:** Supply the code and ask for analysis in one prompt. *Example: "Analyze the following Python function and identify any bugs or improvements:\n* `python\n<code snippet>\n` *\nProvide suggestions and a brief explanation for each."* By wrapping the code in triple backticks, we ensure Claude treats it as code. The instruction asks for both bugs and improvements, guiding Claude to do a mini code review. When integrated, the agent can take Claude's response and format it into an ink annotation or a list of review comments for the user [2].
- **Debugging Assistance:** Provide Claude with the error message and relevant code context. *Example: "The code below is producing a runtime error.\n* `python\n<offending code>\n` *\nError:* `<error`

`trace>` *\nWhat is the likely cause of this error and how can it be fixed?"* Claude will explain the error and suggest a fix. The agent can then present this as a suggested patch or explanation note. If the code is long, consider summarizing or pointing Claude to the likely area (to stay within context limits). Claude's strength is explaining and reasoning, so prompts that explicitly ask "why" and "how to fix" will yield useful diagnostic advice [2].

- **Summarizing Technical Discussions or Decisions:** When the user has a long note or design discussion, an agent can ask Claude for a summary focusing on key decisions. *Example: "Summarize the following discussion about our project's architecture. Focus on what decisions were made and why: \n<project meeting notes or design text>"* Claude will output a concise summary highlighting decisions and rationale. The agent might prepend a system-style note like "You are an AI documentation assistant" to encourage a formal tone, and instruct for bullet points if that's easier to read. The result can be placed on the reMarkable as a summary page or added to the knowledge graph for later reference.
- **Research and Best Practices Inquiry:** For broader questions, frame the prompt so Claude provides structured information. *Example: "You are an AI research assistant. The user asks: 'What are the best practices for implementing authentication in a Flask API?' Provide a brief overview of recommended approaches, tools or libraries, and potential pitfalls."* This prompt sets Claude's role and asks for an overview, which should result in a clear, paragraph or bullet-point answer. The agent can further instruct Claude to keep the answer concise (e.g. "limit to one page") to fit it into an ink note nicely. If the user wants more detail, the agent could follow up with the same session ID to drill deeper on a specific point.
- **Iterative Refinement Workflow:** It's effective to use Claude's memory for step-by-step improvement. For instance, after getting an initial code draft, the agent can present it to the user for feedback (perhaps the user scribbles notes like "make it faster" or "use recursion"). The agent then incorporates that feedback into a follow-up Claude prompt: *"Refine the previous code according to these notes:* `<user notes>`*. Provide the updated code."* Because we use Claude's session continuity, it "remembers" the earlier code it produced, making the refinement more accurate. The plan should include developing prompt templates for such follow-ups (ensuring the prompt includes something like "previous code" or uses the `-c` flag so Claude has the context). This iterative loop continues until the user is satisfied.
- **Prompt Tone and Controls:** In all cases, prompts should be clear about the expected format (e.g. "reply in markdown," "only give the code block," or "list steps 1-3"). Claude Code may also respond to special phrases to think harder (Anthropic's "think step-by-step" triggers) – the agent could exploit this if needed by appending "Think through the problem step by step before answering." However, avoid overly long prompts; keep them focused. The **Claude.md** memory file in the repo can be used to store project guidelines or coding style notes that Claude should follow [12] [13]. By maintaining a consistent prompt style and leveraging Claude's ability to understand the project context (via the CLAUDE.md or prior messages), the integration will yield more relevant and personalized outputs.

Recommended Abstractions and Wrappers for Claude Integration

- **Claude Adapter Module:** Create a dedicated module (e.g. `inklink/adapters/claude_adapter.py`) to encapsulate all interactions with the Claude CLI. This module can define a class (say, `ClaudeClient`) with methods corresponding to high-level actions: `generate_code_from_text(text)`, `review_code(code)`, `ask_best_practices(query)`, etc. Internally, these methods build the appropriate prompt and call the CLI. Centralizing this logic makes maintenance easier and allows tweaking prompt formats or CLI flags in one place. It also

aligns with the project's modular design – similar to how local LLMs are integrated – and will make future **cloud model swaps** (e.g., if using a different API) straightforward [7].

- **Unified LLM Interface:** If not already present, define a generic interface or base class for language model agents. Both local models (via Ollama) and Claude can implement this interface. For example, an abstract class `BaseLLM` with a method `ask(prompt)`. The local version might call an Ollama API, while the Claude version calls the CLI. Agents can then be coded against `BaseLLM` without caring which implementation is behind it. This abstraction is useful for fallback: if Claude is unavailable, the system could instantiate a local LLM implementation instead, using the same interface to continue operation (albeit with simpler results).
- **Configuration & Environment:** Use the existing configuration system (.env or config files) to manage Claude settings. The adapter should read environment variables like `CLAUDE_COMMAND` and `CLAUDE_MODEL` (already described in the README [11]) to know how to invoke the CLI and which model variant to use. This makes it easy to upgrade models (e.g. switching to a newer Claude version) or adjust parameters (like temperature, if the CLI exposes it via flags or config). Document these settings in the project README/CLAUDE.md so users can customize their Claude integration without digging into code.
- **Async and Callback Structure:** Considering some Claude operations might be long-running (especially if analyzing a lot of code or doing intensive tasks), design the wrapper to handle asynchronous calls. For instance, the `ClaudeAdapter` could offer both a synchronous call (for quick queries) and an asynchronous variant that runs in a background thread or task queue. The asynchronous approach would be used by agents that need to free up (e.g., the UI shouldn't freeze while waiting on a long Claude answer). InkLink can then notify the user when the result is ready (for example, by adding a notification on the reMarkable or playing a sound via the pendant). This follows the **asynchronous collaboration** vision of the project [14], where AI agents work independently and report back when done.
- **Error Handling and Retries in Wrapper:** Implement robust error handling in the Claude adapter. If the CLI returns an error or times out, the adapter can have a retry mechanism (perhaps try one more time, or switch to a smaller query if possible). If it fails definitively, the adapter should return a structured error that the calling agent can recognize – allowing the agent to inform the user gracefully (e.g., "Claude is currently unavailable, please try again later"). Logging these failures is important for debugging; consider logging Claude's stderr outputs or exit codes to a file.
- **Integrating with Knowledge Graph & Memory:** Claude's outputs shouldn't exist in isolation. The plan recommends that any significant result (like a code snippet generated or a summary created) be fed back into InkLink's knowledge repositories. For example, after Claude generates code, an agent could automatically document that in the knowledge graph or attach it to the related note's metadata (e.g., "Code generated on 2025-05-17 by Claude for requirement X"). This way, even if the Claude integration is swapped out or the user looks back later, there's a trace of what Claude contributed. It also allows local agents to learn – e.g., the ProactiveProjectTrackerAgent could monitor Claude's outputs to update project status.
- **Extensibility for Other Cloud Models:** While we focus on Claude now, keep the design open for other cloud AI services. The adapter approach can be generalized: maybe have a `CloudAIManager` that can call Claude, OpenAI, or others based on a setting or task type. The **Model Routing** logic mentioned earlier could live here – deciding at runtime which API to call for a given prompt [5]. For now, Claude Code is the primary addition, but if, say, GPT-4 or a future model offers something Claude doesn't, adding it would be as simple as writing a new adapter and plugging it into this manager. Ensuring the codebase's structure (perhaps via factory patterns or dependency injection) supports this swap will "future-proof" InkLink's AI integration.

By following this implementation plan, InkLink will augment its AI Dev Team with Claude Code's powerful cloud-based capabilities. This enables advanced code generation, thorough code reviews, and rich technical insights that complement the on-device experience. The result is a smarter, more capable InkLink system that uses local AI for efficiency and privacy, and calls on Claude as a cloud expert for the heavy lifting when needed – fulfilling the vision of a **"truly personal assistant & development team"** that grows with the user [15].

---

[1] [2] [3] [4] [5] [6] [7] [14] [15] Can you review the project to help guide the next....docx
file://file-KJx6VQxCUPM8pXHM9Wouy2

[8] [9] [10] CLI usage and controls - Anthropic
https://docs.anthropic.com/en/docs/claude-code/cli-usage

[11] GitHub - rmulligan/remarkable-ink-link: Bring AI to your reMarkable tablet. InkLink syncs notes, transcribes handwriting, shares web content as ink, extracts entities, augments research, and integrates tasks with calendars—bridging analog thinking with modern AI workflows.
https://github.com/rmulligan/remarkable-ink-link

[12] [13] Anthropic Claude Code CLI: Prompts & Tool Definitions
https://aiengineerguide.com/blog/claude-code-prompt/