AI VOID PROJECT LAB

Kanbots: AI Agents, Worktrees, & Dev Workflows

Master Kanbots: integrate AI agents like Claude/Codex, leverage git worktrees for isolated runs, and orchestrate multi-agent dev workflows with practical persona-based examples.

AI VOID

2026-05-24 22:46 IST

Contents

CHAPTER 01

Setting Up Your Kanbots Workshop: Tauri v2 and Svelte 5

Welcome to the Kanbots project, where we'll build an innovative desktop Kanban application designed to host and orchestrate multiple AI agents. This application will empower you to automate development tasks, from code generation to review, leveraging isolated Git worktrees for each agent's context.

In this first chapter, we lay the groundwork for Kanbots. We'll set up the core cross-platform desktop application using Tauri v2 for the backend and Rust, paired with a modern Svelte 5 frontend. By the end of this milestone, you will have a functional desktop application window displaying a basic Svelte interface, ready for further development. This foundational setup is crucial for enabling the local-first, privacy-conscious AI agent interactions that will define Kanbots.

Planning & Design: The Kanbots Architecture Core

Kanbots is designed as a desktop application to offer several key advantages: direct filesystem access, enhanced privacy for sensitive AI API keys, and a responsive, integrated user experience. To achieve this, we'll use a robust architecture:

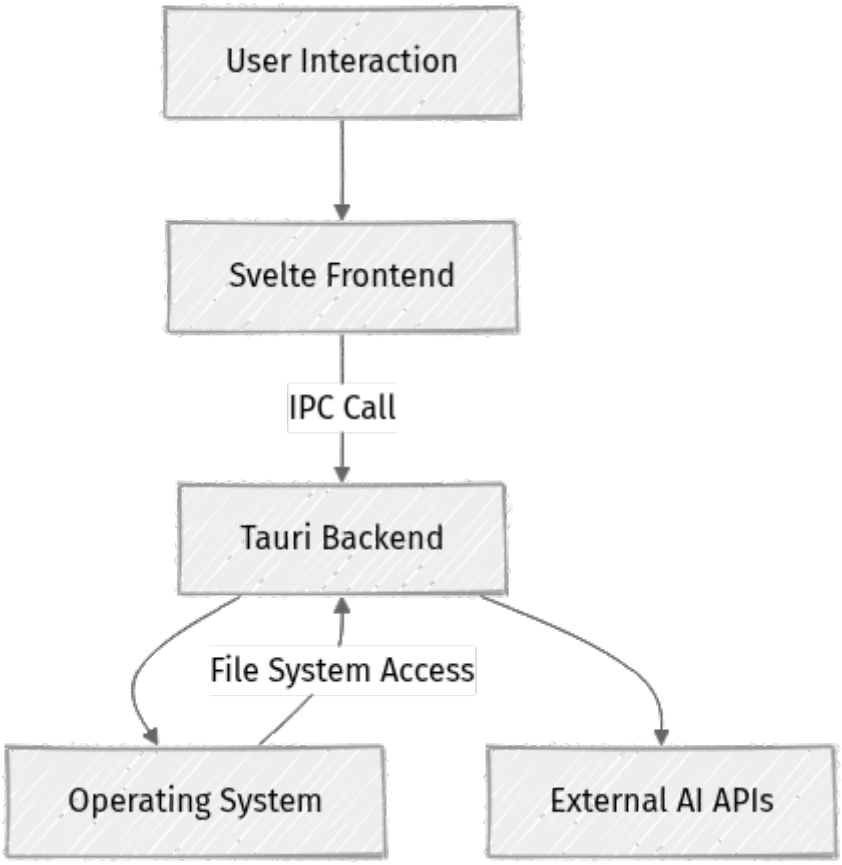
- **Tauri v2 (Rust Backend):** Tauri provides a framework for building cross-platform desktop applications using web technologies. Its Rust backend offers performance, strong type safety, and direct access to system-level functionalities like Git operations and secure API key storage. We chose Tauri v2 for its latest features and performance improvements.
- **Svelte 5 (Frontend):** Svelte is a modern JavaScript framework known for compiling code into tiny, vanilla JavaScript bundles, resulting in highly performant and reactive user interfaces. Svelte 5, while currently in beta, brings significant advancements in reactivity and component architecture, making it an excellent choice for a dynamic application like Kanbots.

- **Inter-Process Communication (IPC):** The Svelte frontend will communicate with the Rust backend via Tauri's IPC mechanism. This allows the UI to trigger backend logic (e.g., Git commands, AI agent orchestration) and receive updates, ensuring a seamless user experience.

Project Structure Overview

When we initialize our Tauri project, it will create a standard directory structure:

- `src-tauri/`: This directory houses all the Rust backend code, including the main application logic, API interactions, and system integrations (like Git worktree management).
- `src/`: This will contain our Svelte frontend code, including components, styling, and client-side logic.
- `package.json`: Manages frontend dependencies and scripts.
- `Cargo.toml`: Manages Rust dependencies and project metadata.



The user interacts with the Svelte frontend, which communicates with the Rust backend via IPC. The Rust backend then handles all system-level operations, including interacting with the operating system (e.g., file system for Git worktrees) and external AI APIs.

Step-by-Step Implementation: Building the Foundation

Let's get our development environment set up and create the initial Kanbots application.

Prerequisites

Before we begin, ensure you have the following tools installed and configured on your system. These versions are current as of 2026-05-24.

1. **Rust Toolchain:**
    - **What it is:** The Rust programming language and its package manager, Cargo, are essential for the Tauri backend.
    - **Why it exists:** Provides a high-performance, memory-safe language for system-level operations.
    - **Installation:** Follow the official instructions at rust-lang.org/tools/install. This typically involves running `curl --proto '=https' --tlsv1.2 -sSf <https://sh.rustup.rs> | sh`.
    - **Verification:**

```
rustup update
rustc --version
cargo --version
```

```
    You should see output similar to `rustc 1.78.0 (286c47847 2024-04-29)` and
`cargo 1.78.0 (c406a0a 2024-04-29)` or newer.
```

1. **Node.js and npm/Yarn:**
    - **What it is:** A JavaScript runtime and package manager (npm or Yarn) for managing our Svelte frontend dependencies.
    - **Why it exists:** Enables development and building of JavaScript-based web applications.
    - **Installation:** Download the LTS version from nodejs.org.
    - **Verification:**

```
        node -v
        npm -v
        # or yarn -v if you prefer Yarn
```

```
    You should see versions like `v20.12.2` for Node.js and `10.5.0` for npm
or newer.
```

1. **Git:**
    - **What it is:** The distributed version control system. Critical for managing worktrees later.
    - **Why it exists:** Tracks code changes, enables collaboration, and is fundamental to Kanbots' agent isolation.
    - **Installation:** Follow instructions at git-scm.com/downloads.
    - **Verification:**

```
        git --version
```

```
    You should see something like `git version 2.44.0` or newer.
```

1. **Code Editor:**
    - **Recommendation:** VS Code with Rust Analyzer and Svelte extensions.

1. Install Tauri CLI

We'll use the Tauri CLI to create and manage our project. We're targeting Tauri v2 for its advancements.

```
cargo install tauri-cli --version 2.0.0-beta.19
```

- **What it does:** Installs the Tauri command-line interface as a Rust binary.
- **Why this version:** `2.0.0-beta.19` is the latest stable beta as of 2026-05-24. While Tauri v1.x is stable, v2 offers significant improvements we'll leverage.

2. Create Your Kanbots Project

Now, let's create a new Tauri project using the Svelte TypeScript template.

Navigate to your desired projects directory in your terminal and run:

```
cargo tauri dev --template svelte-ts kanbots
```

- **`cargo tauri dev`**: This command is typically used to run the application in development mode. When used with `--template`, it initializes a new project.
- **`--template svelte-ts`**: Specifies that we want to use the Svelte and TypeScript template. This template is compatible with Svelte 5, though Svelte 5 itself is currently in beta.
- **`kanbots`**: This is the name of our project directory.

The CLI will prompt you for a few details. You can accept the defaults for now:

```
? What is your app name? › Kanbots
? What is the window title? › Kanbots
? Where should your frontend files be located, relative to the project root? ›
src
? What is your frontend dev command? › npm run dev
? What is your frontend build command? › npm run build
? What is the path to your frontend dist folder? › ../dist
```

This process will set up the necessary files and install frontend dependencies. It might take a few minutes.

3. Explore the Project Structure

Once the command completes, you'll have a `kanbots` directory. Let's look at the key parts:

```
kanbots/
├── src-tauri/             # Rust backend code
│   ├── Cargo.toml        # Rust dependencies and project metadata
│   ├── src/              # Main Rust source files
│   │   ├── main.rs       # Tauri application entry point
│   │   └── commands.rs   # IPC commands exposed to the frontend
│   └── ...
├── src/                  # Svelte frontend code
│   ├── App.svelte        # Main Svelte component
│   ├── main.ts           # Svelte application entry point
│   └── ...
├── package.json          # Frontend dependencies and scripts (npm/yarn)
├── tsconfig.json         # TypeScript configuration
└── ...
```

- **`src-tauri/src/main.rs`**: This is the heart of your Rust application. It initializes Tauri, defines the main window, and registers any custom commands the frontend can call.
- **`src/App.svelte`**: This is your main Svelte component. It's where the initial UI of your application is defined.

4. Run Your Kanbots Application

Navigate into your newly created `kanbots` directory:

```
cd kanbots
```

Now, start the application in development mode:

```
cargo tauri dev
```

- **What it does:** This command first builds the Rust backend, then starts the Svelte development server, and finally launches the Tauri desktop window, which loads your Svelte frontend.
- **Why this command:** It provides hot-reloading for your Svelte changes and detailed logging from the Rust backend, making development efficient.

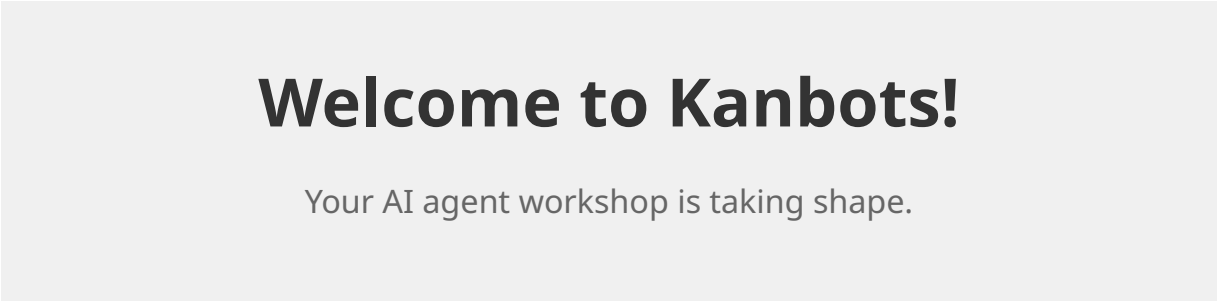
You should see a new desktop window appear, displaying the default Svelte starter page.

5. Customize the Frontend

Let's make a small change to confirm everything is working as expected.

Open `src/App.svelte` in your code editor. Replace the existing content with a simpler message:

File: `kanbots/src/App.svelte````svelte




```
- **What changed:** We removed the default Tauri/Svelte boilerplate and added a simple `<h1>` and `<p>` tag with a custom message and basic styling.
- **Why this change:** This confirms that the Svelte frontend is correctly integrated and that hot-reloading is functional.

Save the file. You should see the Tauri desktop window automatically update to display "Welcome to Kanbots!"

## Testing & Verification

At this stage, verification is straightforward:

1.  **Run the application:** Execute `cargo tauri dev` from the `kanbots` project root.
2.  **Observe the desktop window:** A new desktop application window should appear.
3.  **Check content:** The window should display "Welcome to Kanbots! Your AI agent workshop is taking shape."
4.  **Confirm hot-reloading:** Make a minor text change in `src/App.svelte` and save. The window content should update immediately without needing to restart `cargo tauri dev`.

If these steps are successful, your basic Kanbots workshop is correctly set up.

## Production Considerations

Even at this early stage, it's good to think about production:

- **Build Times:** Rust compilation can be lengthy, especially the first time. Tauri optimizes subsequent builds, but expect initial builds to take several minutes depending on your system. This is a common tradeoff for performance and safety.
- **Bundle Size:** Tauri applications are generally lightweight because they leverage the native WebView rather than bundling an entire browser runtime (like Electron). This contributes to smaller executable sizes, which is beneficial for deployment.
- **Cross-Platform Consistency:** Tauri handles much of the platform-specific
```

```
boilerplate. However, always test your application on your target operating
systems (Windows, macOS, Linux) as you add features to catch any platform-
specific UI or behavior quirks early.
- **Secure API Keys (Future):** We'll integrate AI agents later, requiring API
keys. For a desktop application, hardcoding keys is a major security risk.
We'll explore secure methods like OS-level secret management (e.g., Keychain
on macOS, Credential Manager on Windows) or environment variables in a later
chapter to keep these sensitive credentials out of your codebase.

## Common Issues & Solutions

1.  **Rust Toolchain Not Found / Not Up-to-Date:**
    - **Issue:** `cargo` commands fail, or `rustc` reports an old version.
    - **Solution:** Ensure Rust is correctly installed via `rustup` and
updated with `rustup update`. Restart your terminal after installation.
2.  **Node.js/npm Errors:**
    - **Issue:** Frontend dependencies fail to install, or `npm run dev`
doesn't work.
    - **Solution:** Verify Node.js and npm are installed and on your PATH.
Sometimes, reinstalling `node_modules` (`rm -rf node_modules && npm install`)
can resolve issues.
3.  **Tauri Build Errors (Missing Dependencies):**
    - **Issue:** `cargo tauri dev` fails with errors related to system
libraries (e.g., `webkit2gtk` on Linux, developer tools on macOS/Windows).
    - **Solution:** Tauri often requires specific system dependencies for the
WebView. Refer to the [Tauri prerequisites documentation](https://tauri.app/
v2/guides/getting-started/prerequisites) for your operating system and install
any missing packages.
4.  **Frontend Not Loading / Blank Window:**
    - **Issue:** The Tauri window appears but is blank or shows an error
related to the frontend.
    - **Solution:** Check the console in your browser's developer tools (you
can usually open this in the Tauri window with F12 or Cmd+Option+I). Look for
Svelte compilation errors or network errors if the frontend dev server isn't
running correctly. Ensure `npm run dev` (or `yarn dev`) runs successfully in
your `src` directory.

## Summary & Next Step

Congratulations! You've successfully set up the core Kanbots desktop
application using Tauri v2 and Svelte 5. You have a running desktop window,
and you've verified that frontend changes are reflected in real-time. This
robust foundation is critical for building a responsive, feature-rich
application.

In the next chapter, we'll dive into the fascinating world of Git worktrees.
You'll learn how to programmatically create and manage isolated Git
environments directly from your Rust backend, a core mechanism for enabling
our AI agents to work on tasks without interfering with each other's
codebases.

---
This page is AI-assisted and reviewed. It references official documentation
and recognized resources where relevant.

## References

- [Tauri Official Documentation](https://tauri.app/v2/guides/introduction)
- [Rust Programming Language](https://www.rust-lang.org/learn)
- [Node.js Official Website](https://nodejs.org/en/docs)
- [Svelte Official Documentation](https://svelte.dev/docs/introduction)
- [Git Official Documentation](https://git-scm.com/doc)
```

CHAPTER 02

Building the Core Kanban Board UI

In this chapter, we're laying the visual and interactive groundwork for Kanbots: a functional Kanban board. This isn't just about pretty pixels; it's about creating the canvas where our AI agents will operate. By the end of this milestone, you will have a desktop application with a fully interactive Kanban board, allowing you to add, edit, and move task cards between columns. This core UI is essential for managing the AI-driven development tasks we'll introduce later.

This chapter is critical because a robust, intuitive UI is the user's direct interface to the powerful AI orchestration we're building. Without a solid foundation here, managing complex multi-agent workflows would be cumbersome. When we finish, you'll be able to confirm that you can manage tasks efficiently, setting the stage for integrating AI automation.

Planning the Kanban Board Structure

Our Kanbots application will present a familiar Kanban interface: columns representing stages (e.g., "To Do," "In Progress," "Done") and cards representing individual tasks. Each card will eventually host one or more AI agents.

Core Components and Data Flow

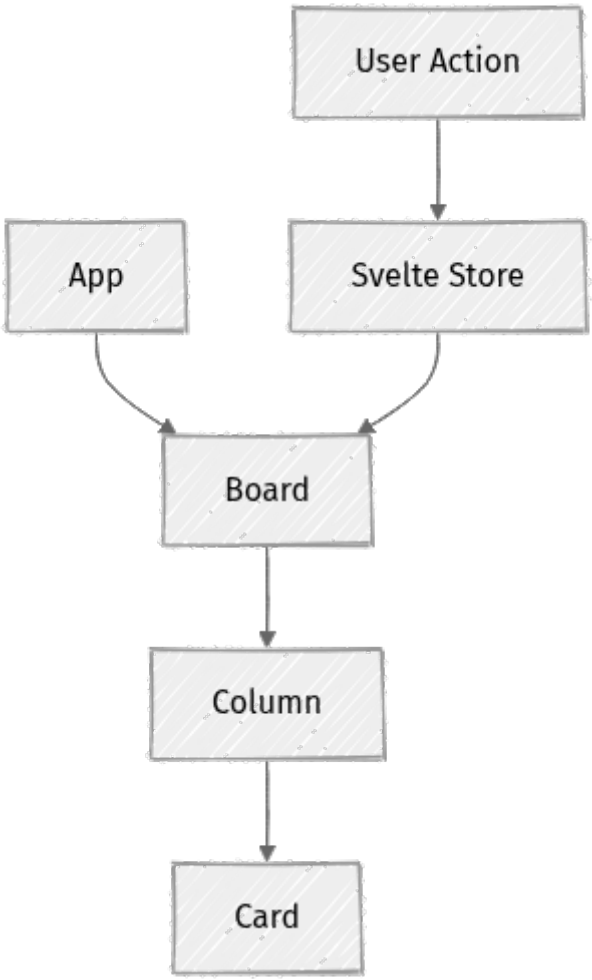
We'll use Svelte 5's reactivity model and component-based architecture to build the UI. The state of our Kanban board (columns and cards) will be managed using Svelte stores, providing a centralized, reactive data source.

The primary components we'll build are:

- `Board.svelte`: The top-level component that orchestrates columns.
- `Column.svelte`: Represents a single Kanban column, managing its title and the list of cards within it.

- `Card.svelte`: Displays an individual task card, including its title and description.

Data will flow from the `Board` component, which holds the overall state, down to `Column` and then `Card` components via props. User interactions (e.g., adding a card, dragging a card) will trigger updates to the Svelte store, which in turn will reactively update the UI.



📌 **Key Idea:** A clear component hierarchy and centralized state management are crucial for building complex, interactive UIs that are easy to reason about and maintain.

Project File Structure

Within our existing Tauri project (initialized in Chapter 1), we'll focus on the `src-tauri/src/main.rs` (minimal for now) and `src/` directory for the Svelte frontend.

```
kanbots/
├── src-tauri/
│   ├── src/
│   │   └── main.rs
│   └── Cargo.toml
└── src/
    ├── App.svelte         // Main Svelte application
    ├── main.ts            // Svelte entry point
    ├── lib/
    │   ├── components/
    │   │   ├── Board.svelte
    │   │   ├── Column.svelte
    │   │   └── Card.svelte
    │   └── stores.ts      // Svelte stores for state management
    └── index.css          // Global styles
```

Step-by-Step Implementation

Let's build out our Kanban board incrementally. We'll start with the data model, then the components, and finally, the drag-and-drop functionality.

1. Define Data Models and Svelte Store

First, we need to define the structure for our cards and columns. We'll use TypeScript for type safety and create a Svelte writable store to manage the board's state.

Create `src/lib/stores.ts`:

```ts
// src/lib/stores.ts
import { writable } from 'svelte/store';
import { v4 as uuidv4 } from 'uuid'; // For unique IDs

// Define the shape of a Card
export interface Card {
  id: string;
  title: string;
  description: string;
  columnId: string; // To link cards to columns
}

// Define the shape of a Column
export interface Column {
  id: string;
  title: string;
}

// Initial board data
const initialColumns: Column[] = [
  { id: 'todo', title: 'To Do' },
  { id: 'in-progress', title: 'In Progress' },
  { id: 'done', title: 'Done' },
];
```

```
const initialCards: Card[] = [
  { id: uuidv4(), title: 'Set up Tauri + Svelte', description:
'Initialize the project and basic structure.', columnId: 'todo' },
  { id: uuidv4(), title: 'Design Kanban UI', description: 'Sketch out the
layout for columns and cards.', columnId: 'in-progress' },
  { id: uuidv4(), title: 'Implement Card Drag & Drop', description: 'Enable
moving cards between columns.', columnId: 'in-progress' },
  { id: uuidv4(), title: 'Write Chapter 1', description: 'Initial project
setup documentation.', columnId: 'done' },
];

// Svelte writable stores
export const columns = writable<Column[]>(initialColumns);
export const cards = writable<Card[]>(initialCards);

// Function to add a new card
export function addCard(columnId: string, title: string, description: string)
{
  cards.update(currentCards => [
    ...currentCards,
    { id: uuidv4(), title, description, columnId }
  ]);
}

// Function to update a card
export function updateCard(cardId: string, newTitle: string, newDescription: s
tring) {
  cards.update(currentCards =>
    currentCards.map(card =>
      card.id === cardId ? { ...card, title: newTitle, description: newDescrip
tion } : card
    )
  );
}

// Function to move a card between columns
export function moveCard(cardId: string, targetColumnId: string) {
  cards.update(currentCards =>
    currentCards.map(card =>
      card.id === cardId ? { ...card, columnId: targetColumnId } : card
    )
  );
}
```

Explanation:

- `writable` from `svelte/store` creates reactive stores. When `cards` or `columns` are updated, any components subscribing to them will automatically re-render.
- `Card` and `Column` interfaces define our data types.
- `uuidv4` from the `uuid` library (install it: `npm install uuid @types/uuid`) ensures unique IDs for cards, which is crucial for identifying and manipulating them.
- `initialColumns` and `initialCards` provide some starting data.

- `addCard`, `updateCard`, and `moveCard` are helper functions to modify the `cards` store, encapsulating the update logic.

2. Create the Card Component

This component will display a single task card and make it draggable.

Create `src/lib/components/Card.svelte`:

```
<!-- src/lib/components/Card.svelte -->
<script lang="ts">
  import { cards, updateCard } from '$lib/stores';
  import type { Card } from '$lib/stores';

  export let card: Card;

  let isEditing = false;
  let editedTitle = card.title;
  let editedDescription = card.description;

  function handleSave() {
    updateCard(card.id, editedTitle, editedDescription);
    isEditing = false;
  }

  function handleDragStart(event: DragEvent) {
    if (event.dataTransfer) {
      event.dataTransfer.setData('text/plain', card.id);
      event.dataTransfer.effectAllowed = 'move';
    }
  }
</script>

<div
  class="card"
  draggable="true"
  on:dragstart={handleDragStart}
  on:dblclick={() => (isEditing = true)}
>
  {#if isEditing}
    <input type="text" bind:value={editedTitle} class="card-input-title" />
    <textarea bind:value={editedDescription} class="card-input-description"></
textarea>
    <button on:click={handleSave} class="card-save-button">Save</button>
    <button on:click={() => (isEditing = false)} class="card-cancel-
button">Cancel</button>
  {:else}
    <h3>{card.title}</h3>
    <p>{card.description}</p>
  {/if}
</div>

<style>
  .card {
    background-color: var(--card-bg);
    border-radius: 8px;
    padding: 12px;
    margin-bottom: 10px;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
```

```
    cursor: grab;
    transition: transform 0.1s ease-in-out;
  }
  .card:active {
    cursor: grabbing;
  }
  .card h3 {
    margin-top: 0;
    margin-bottom: 8px;
    color: var(--text-color);
  }
  .card p {
    font-size: 0.9em;
    color: var(--text-color-light);
    margin-bottom: 0;
  }
  .card-input-title, .card-input-description {
    width: calc(100% - 16px);
    padding: 8px;
    margin-bottom: 8px;
    border: 1px solid var(--border-color);
    border-radius: 4px;
    background-color: var(--input-bg);
    color: var(--text-color);
  }
  .card-save-button, .card-cancel-button {
    padding: 6px 10px;
    margin-right: 5px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 0.85em;
  }
  .card-save-button {
    background-color: var(--primary-color);
    color: white;
  }
  .card-cancel-button {
    background-color: var(--secondary-color);
    color: var(--text-color);
  }
</style>
```

Explanation:

- `export let card: Card;` declares a prop that expects a `Card` object.
- `draggable="true"` makes the `div` element draggable.
- `on:dragstart={handleDragStart}` attaches an event listener. Inside `handleDragStart`, `event.dataTransfer.setData('text/plain', card.id)` stores the card's ID, which we'll use to identify the dragged card.
- `on:dblclick` toggles an editing state, allowing users to update card details.
- Svelte's `bind:value` is used for two-way data binding with input fields.

- The `style` block provides basic styling. We'll define CSS variables soon.

3. Create the `Column` Component

This component will render a column title, display its cards, and handle dropping cards.

Create `src/lib/components/Column.svelte`:

```
<!-- src/lib/components/Column.svelte -->
<script lang="ts">
  import { cards, moveCard, addCard } from '$lib/stores';
  import type { Column, Card } from '$lib/stores';
  import CardComponent from './Card.svelte';

  export let column: Column;

  // Filter cards belonging to this column
  $: columnCards = $cards.filter(card => card.columnId === column.id);

  let showAddCardForm = false;
  let newCardTitle = '';
  let newCardDescription = '';

  function handleAddCard() {
    if (newCardTitle.trim()) {
      addCard(column.id, newCardTitle, newCardDescription);
      newCardTitle = '';
      newCardDescription = '';
      showAddCardForm = false;
    }
  }

  function handleDragOver(event: DragEvent) {
    event.preventDefault(); // Crucial to allow dropping
    if (event.dataTransfer) {
      event.dataTransfer.dropEffect = 'move';
    }
  }

  function handleDrop(event: DragEvent) {
    event.preventDefault();
    const cardId = event.dataTransfer?.getData('text/plain');
    if (cardId) {
      moveCard(cardId, column.id);
    }
  }
</script>

<div
  class="column"
  on:dragover={handleDragOver}
  on:drop={handleDrop}
>
  <h2>{column.title}</h2>
  <div class="cards-container">
    {#each columnCards as card (card.id)}
      <CardComponent card={card} />
    {/each}
```

```
  </div>

  {#if showAddCardForm}
    <div class="add-card-form">
      <input type="text" bind:value={newCardTitle} placeholder="Card title"
class="add-card-input" />
      <textarea bind:value={newCardDescription} placeholder="Description
(optional)" class="add-card-textarea"></textarea>
      <button on:click={handleAddCard} class="add-card-button">Add</button>
      <button on:click={() => showAddCardForm = false} class="cancel-card-
button">Cancel</button>
    </div>
  {:else}
    <button on:click={() => showAddCardForm = true} class="add-card-toggle-
button">+ Add Card</button>
  {/if}
</div>

<style>
  .column {
    flex: 1;
    min-width: 280px;
    max-width: 320px;
    background-color: var(--column-bg);
    border-radius: 10px;
    padding: 15px;
    margin: 0 10px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
    display: flex;
    flex-direction: column;
    height: fit-content; /* Allow column to grow with content */
    min-height: 200px; /* Minimum height for drag-and-drop target */
  }
  .column h2 {
    margin-top: 0;
    margin-bottom: 20px;
    color: var(--text-color);
    text-align: center;
  }
  .cards-container {
    flex-grow: 1;
  }
  .add-card-toggle-button {
    background-color: var(--secondary-color);
    color: var(--text-color);
    border: none;
    padding: 10px 15px;
    border-radius: 6px;
    cursor: pointer;
    margin-top: 15px;
    width: 100%;
    font-size: 1em;
    transition: background-color 0.2s ease;
  }
  .add-card-toggle-button:hover {
    background-color: var(--secondary-hover);
  }
  .add-card-form {
    display: flex;
    flex-direction: column;
    margin-top: 15px;
  }
```

```
  .add-card-input, .add-card-textarea {
    width: calc(100% - 16px);
    padding: 8px;
    margin-bottom: 8px;
    border: 1px solid var(--border-color);
    border-radius: 4px;
    background-color: var(--input-bg);
    color: var(--text-color);
  }
  .add-card-button, .cancel-card-button {
    padding: 8px 12px;
    margin-top: 5px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 0.9em;
  }
  .add-card-button {
    background-color: var(--primary-color);
    color: white;
  }
  .cancel-card-button {
    background-color: var(--secondary-color);
    color: var(--text-color);
  }
</style>
```

Explanation:

- `export let column: Column;` expects a `Column` object.
- `$: columnCards = $cards.filter(card => card.columnId === column.id);` is a Svelte reactive declaration. It automatically re-runs whenever the `$cards` store changes, filtering cards relevant to this column.
- `on:dragover={handleDragOver}` is vital. `event.preventDefault()` allows a drop to occur.
- `on:drop={handleDrop}` retrieves the `cardId` from `event.dataTransfer` and calls `moveCard` from our store to update the card's column.
- An "Add Card" button and form are included to create new cards directly within a column.

4. Create the Board Component

This component will arrange the columns and provide overall structure.

Create `src/lib/components/Board.svelte`:

```
<!-- src/lib/components/Board.svelte -->
<script lang="ts">
  import { columns } from '$lib/stores';
  import ColumnComponent from './Column.svelte';
</script>
```

```
<div class="board">
  {#each $columns as column (column.id)}
    <ColumnComponent column={column} />
  {/each}
</div>

<style>
  .board {
    display: flex;
    flex-wrap: nowrap; /* Prevent columns from wrapping */
    padding: 20px;
    gap: 20px; /* Space between columns */
    overflow-x: auto; /* Allow horizontal scrolling if many columns */
    height: 100vh; /* Take full viewport height */
    align-items: flex-start; /* Align columns to the top */
    background-color: var(--board-bg);
  }
</style>
```

Explanation:

- `{#each $columns as column (column.id)}` iterates over the `columns` store. The `$` prefix automatically subscribes to the store. `(column.id)` is a Svelte key, which helps optimize list rendering.
- Each `ColumnComponent` receives its respective `column` data as a prop.
- The `board` class uses Flexbox to lay out columns horizontally.

5. Update `App.svelte` and Global Styles

Now, let's integrate the `Board` component into our main `App.svelte` and add global styles.

Update `src/App.svelte`:

```
<!-- src/App.svelte -->
<script lang="ts">
  import Board from './lib/components/Board.svelte';
</script>

<main>
  <Board />
</main>

<style>
  :global(body) {
    margin: 0;
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto,
Helvetica, Arial, sans-serif, 'Apple Color Emoji', 'Segoe UI Emoji', 'Segoe UI
Symbol';
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    background-color: var(--board-bg); /* Ensure body matches board background
*/
```

```
  }

  :global(*) {
    box-sizing: border-box;
  }

  main {
    display: flex;
    min-height: 100vh; /* Ensure main takes full height */
    width: 100vw; /* Ensure main takes full width */
  }
</style>
```

Add `src/index.css` (create this file if it doesn't exist):

```
/* src/index.css */
:root {
  /* Color Palette */
  --primary-color: #4CAF50; /* Green for success/primary actions */
  --secondary-color: #FFC107; /* Amber for warnings/secondary actions */
  --accent-color: #2196F3; /* Blue for info/highlights */
  --danger-color: #F44336; /* Red for errors/destructive actions */

  /* Theming - Dark Mode Inspired */
  --background-color: #1a1a1a; /* Overall app background */
  --text-color: #e0e0e0; /* Primary text color */
  --text-color-light: #b0b0b0; /* Secondary text color */
  --border-color: #333; /* Borders and dividers */
  --shadow-color: rgba(0, 0, 0, 0.3); /* Shadow color */

  /* Kanban Specific */
  --board-bg: #282c34; /* Dark background for the entire board area */
  --column-bg: #3c4048; /* Slightly lighter dark for columns */
  --card-bg: #555c66; /* Even lighter dark for cards */
  --input-bg: #444; /* Input field background */

  /* Hover States */
  --primary-hover: #45a049;
  --secondary-hover: #ffca28;
}

body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Helvetic
a, Arial, sans-serif, 'Apple Color Emoji', 'Segoe UI Emoji', 'Segoe UI
Symbol';
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  background-color: var(--background-color);
  color: var(--text-color);
}

/* Scrollbar styling for a cleaner look */
::-webkit-scrollbar {
  width: 8px;
  height: 8px;
}

::-webkit-scrollbar-track {
```

```
  background: var(--background-color);
}

::-webkit-scrollbar-thumb {
  background: var(--column-bg);
  border-radius: 4px;
}

::-webkit-scrollbar-thumb:hover {
  background: var(--card-bg);
}
```

Explanation:

- The `:global(body)` and `:global(*)` selectors apply styles globally, necessary for `index.css` to be effective within Svelte's scoped styling.
- `main` is set to `flex` to ensure the `Board` component fills the available space.
- `index.css` defines a set of CSS variables (`:root`) for a consistent dark theme, making it easy to adjust colors later. These variables are then used throughout the components.

6. Install `uuid` and TypeScript Types

If you haven't already, install the `uuid` library and its TypeScript types:

```
npm install uuid
npm install -D @types/uuid
```

This ensures `uuidv4()` is available and correctly typed in `stores.ts`.

Testing & Verification

Now that the core UI is in place, let's run the application and verify its functionality.

1. **Start the Tauri development server:** Navigate to your project root in the terminal and run:

```
    npm run tauri dev
```

```
This command will compile your Rust backend, start the Svelte development
server, and launch the Tauri desktop window.
```

1. **Verify Initial Board State:**
    - You should see a desktop window open with three columns: "To Do," "In Progress," and "Done."
    - Each column should contain the initial cards defined in `src/lib/stores.ts`.
2. **Test Card Creation:**
    - Click the "+ Add Card" button in any column.
    - Enter a title and an optional description.
    - Click "Add." A new card should appear at the bottom of that column.
3. **Test Card Editing:**
    - Double-click on any card.
    - Edit its title and/or description.
    - Click "Save." The card should update with the new information.
    - Click "Cancel" to discard changes.
4. **Test Drag and Drop:**
    - Click and hold on a card, then drag it to another column.
    - Release the mouse button. The card should move from its original column to the new column.
    - Try moving cards within the same column as well.

If all these steps work as described, you have successfully built the core Kanban board UI.

Production Considerations

While this is a foundational step, thinking about production early helps.

- **State Persistence**: Currently, our board state is lost when the application closes. For a real application, we would persist `cards` and `columns` to a local database (e.g., SQLite via Tauri's Rust backend) or a local file. This will be addressed in a later chapter.

- **Performance**: For very large boards with hundreds or thousands of cards, rendering all cards at once can become slow. Techniques like list virtualization (only rendering visible items) would be necessary. For our initial scope, this is not a concern.
- **Accessibility**: Drag-and-drop functionality should ideally also be accessible via keyboard navigation. This requires additional ARIA attributes and event handlers, which is beyond our current scope but important for a polished production app.
- **Error Handling**: The UI assumes successful operations. In a production scenario, failed updates to the store (e.g., if persistence fails) would need user feedback.

Common Issues & Solutions

- **Cards not draggable/droppable**:
  - **Issue**: `draggable="true"` is missing on the `Card` component's root element, or `event.preventDefault()` is missing in `handleDragOver` in the `Column` component.
  - **Solution**: Double-check these attributes and function calls. `event.preventDefault()` is critical for enabling drop targets.
- **Svelte reactivity issues**:
  - **Issue**: Changes to stores or props don't seem to update the UI.
  - **Solution**: Ensure you are correctly using `$storeName` to access store values and that updates are done via `storeName.update(...)` or `storeName.set(...)`. Svelte's reactivity model is powerful but requires adherence to its patterns.
- **Styling not applying**:
  - **Issue**: Your CSS variables or component styles aren't visible.
  - **Solution**: Verify `src/index.css` is being imported (check `src/main.ts` or `src/App.svelte` for an import statement like `import './index.css';`). Also, ensure `:global()` is used for base styles that should affect elements outside the Svelte component's scope.

Summary & Next Step

You've successfully built the interactive Kanban board for Kanbots. You can now create, edit, and move task cards, providing a solid foundation for task management. This milestone is crucial because it gives us a tangible interface to interact with.

What's ready:

- A cross-platform desktop application powered by Tauri and Svelte.
- A functional Kanban board with multiple columns.
- Interactive cards that can be added, edited, and moved via drag-and-drop.

In the next chapter, we will integrate Git worktrees, which will serve as isolated environments for our AI agents, allowing them to work on tasks without interfering with each other or the main codebase.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- Tauri Documentation
- Svelte Documentation
- Svelte Store Documentation
- MDN Web Docs: Drag and Drop API
- uuid npm package

CHAPTER 03

Mastering Git Worktrees for Isolated Agent Tasks

Isolated Development with Git Worktrees

Imagine a team of highly efficient AI developers, each working on a separate feature branch, but all within the same repository, without ever stepping on each other's toes. This is the power we're bringing to Kanbots in this chapter. We'll enable each Kanban card to spawn and manage its own isolated Git environment using **Git worktrees**.

This milestone is critical because AI agents, especially those generating code, need a clean, predictable workspace. Without isolation, concurrent agents could overwrite each other's changes, leading to chaos and unpredictable outcomes. Git worktrees provide this crucial sandboxing, allowing agents to operate in parallel, each with its own working directory and branch, while still sharing the underlying repository history and objects.

By the end of this chapter, your Kanbots application will have the foundational Rust backend logic and Svelte frontend integration to create, switch to, and remove Git worktrees directly from a Kanban card. This sets the stage for integrating AI agents in the next chapter, ensuring they have a stable and isolated environment to perform their tasks.

Understanding Git Worktrees: The Foundation of Isolation

At its core, a Git worktree allows you to have multiple working directories attached to the same Git repository. Each working directory can be on a different branch.

What is a Git Worktree?

A worktree is a separate working tree, distinct from your main working directory (the "main worktree"). It has its own checked-out branch, but it shares the same `.git` directory (or rather, points back to the main `.git`

directory) as the primary repository. This means all worktrees share the same commit history, objects, and refs, but can have different files currently checked out and modified.

Why Do Worktrees Exist?

Git introduced worktrees to solve common developer frustrations:

- **Parallel Development:** Work on multiple features or bug fixes simultaneously without constantly stashing and switching branches in a single working directory.
- **Context Switching:** Quickly jump between different branches for review or testing without affecting your current work.
- **Experimentation:** Try out a new idea on a separate branch in an isolated environment.

How Worktrees Solve Problems for Kanbots

For Kanbots, worktrees are essential for:

- **Agent Isolation:** Each AI agent, assigned to a Kanban card, gets its own dedicated workspace. This prevents agents from interfering with each other's generated code or state.
- **Context Management:** An agent can operate on a specific branch within its worktree, ensuring its actions are confined to a defined scope.
- **Parallel Execution:** Multiple agents can run concurrently on different cards, each managing its own worktree, without worrying about merge conflicts until their tasks are complete and ready for integration.
- **Clean Slate:** When an agent starts a task, it can be provided with a fresh worktree, reducing the risk of stale or unexpected files influencing its behavior.

Basic Git Worktree Commands

Here are the fundamental Git commands we'll be translating into our Rust backend:

- **Add a new worktree:**

```
git worktree add <path> <branch_name>
```

```
This creates a new worktree at `<path>` and checks out `<branch_name>` in it.
If `<branch_name>` doesn't exist, it will be created.
```

- **List existing worktrees:**

```
git worktree list
```

```
Shows all worktrees associated with the current repository.
```

- **Remove a worktree:**

```
git worktree remove <path>
```

```
Deletes the worktree at `<path>`. The branch associated with it remains unless
deleted separately. Requires the worktree to be clean (no uncommitted
changes).
```

- **Force remove a worktree:**

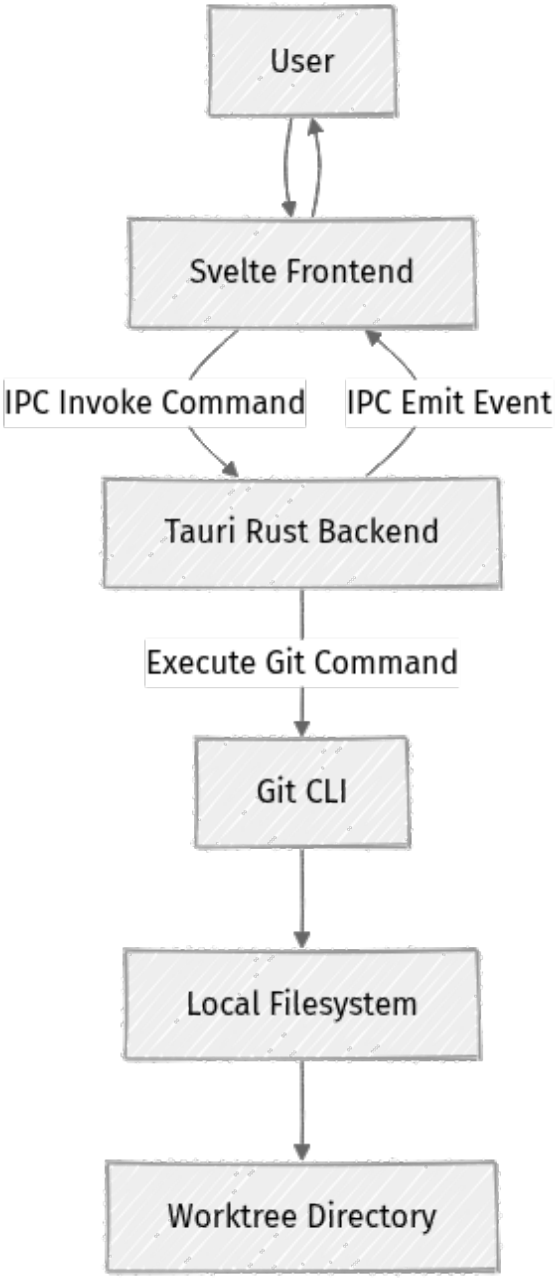
```
git worktree remove --force <path>
```

```
Deletes the worktree even if it has uncommitted changes. Use with caution.
```

Designing Worktree Integration for Kanbots

Integrating Git worktrees means our Rust backend will become the orchestrator for Git commands. The Svelte frontend will provide the UI elements to trigger these actions, communicating with Rust via Tauri's IPC layer.

Architecture Overview



1. **Svelte Frontend:** Provides buttons or actions on a Kanban card to create, delete, or interact with a worktree.
2. **Tauri Rust Backend:** This is where the core logic resides. It receives commands from the Svelte frontend via IPC (Inter-Process Communication).
3. **Git CLI:** The Rust backend will execute `git` commands using `std::process::Command`, interacting directly with the local Git installation.
4. **Local Filesystem:** Git creates and manages the actual worktree directories on the user's disk.

Data Flow for Creating a Worktree

1. User clicks "Create Worktree" on a Kanban card in the Svelte UI.
2. The Svelte frontend invokes a Rust command (e.g., `create_card_worktree`) via Tauri's IPC. It passes the `card_id` and potentially a `branch_name`.
3. The Rust backend receives the command.
4. It constructs and executes a `git worktree add` command, creating a new directory structure for the worktree.
5. If successful, the Rust backend might emit an event back to the Svelte frontend (e.g., `worktree_created`) to update the UI.
6. The Svelte frontend receives the event and updates the card's status or displays the worktree path.

Target Repository Structure

Kanbots will manage a dedicated Git repository where all agent worktrees will reside. This repository will be separate from the Kanbots application's own source code. We'll call this the "agent development repository" or `agent_dev_repo`.

A typical structure might look like this:

```
/path/to/kanbots-app/ # Your Kanbots desktop application
/path/to/agent_dev_repo/ # The Git repository managed by Kanbots
├── .git/
├── .worktrees/
│   ├── card_123_feature_x/ # Worktree for card 123
│   └── card_456_bug_fix/ # Worktree for card 456
├── src/
├── Cargo.toml
└── ... (main branch content)
```

Each worktree will be created within the `agent_dev_repo/.worktrees/` directory. The name of the worktree directory will incorporate the card ID and potentially the branch name for clarity.

Implementing Git Worktree Management (Rust Backend)

We'll start by adding the necessary Rust logic to the Tauri backend. We will leverage `std::process::Command` to execute Git CLI commands.

1. Update `Cargo.toml`

We don't need any new crates for `std::process::Command`, but it's good practice to ensure `tauri` is configured correctly.

```
# src-tauri/Cargo.toml
[package]
name = "kanbots"
version = "0.1.0"
description = "A Tauri + Svelte + AI agents Kanban app"
authors = ["You"]
license = ""
repository = ""
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[build-dependencies]
tauri-build = { version = "2.0.0-beta", features = [] }

[dependencies]
tauri = { version = "2.0.0-beta", features = ["shell-open"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.38.0", features = ["full"] } # Required for async
operations and process handling

[features]
# this feature is used for production builds to regenerate the Tauri command
schema
# do not remove it
custom-protocol = ["tauri/custom-protocol"]
```

Decision: We include `tokio` with `full` features for robust asynchronous process execution, which is ideal for running external commands like Git without blocking the main thread.

2. Define Git Command Utilities

Create a new module `src-tauri/src/git_commands.rs` to encapsulate our Git interactions.

```
// src-tauri/src/git_commands.rs
use std::path::{Path, PathBuf};
use std::process::Command;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio::process::Command as TokioCommand;

/// Executes a git command in the specified repository path.
async fn run_git_command(
    repo_path: &Path,
    args: &[&str],
) -> Result<String, String> {
```

```
    let repo_path_str = repo_path.to_str().ok_or_else(|| "Invalid repo path".t
o_string())?;

    let mut command = TokioCommand::new("git");
    command.current_dir(repo_path_str);
    command.args(args);

    let output = command.output().await.map_err(|e| format!
("Failed to execute git command: {}", e))?;

    if output.status.success() {
        Ok(String::from_utf8_lossy(&output.stdout).to_string())
    } else {
        Err(format!(
            "Git command failed: {}\nStderr: {}",
            String::from_utf8_lossy(&output.status.code().unwrap_or(-1).to_str
ing()),
            String::from_utf8_lossy(&output.stderr)
        ))
    }
}

/// Initializes a new git repository at the given path if it doesn't exist.
pub async fn init_repo_if_not_exists(repo_path: &Path) -> Result<String, Strin
g> {
    if !repo_path.exists() {
        tokio::fs::create_dir_all(repo_path)
            .await
            .map_err(|e| format!("Failed to create repo directory: {}", e))?;
        run_git_command(repo_path, &["init"]).await
    } else if !repo_path.join(".git").exists() {
        // If directory exists but is not a git repo, init it
        run_git_command(repo_path, &["init"]).await
    } else {
        Ok(format!("Repository already exists at {}", repo_path.display()))
    }
}

/// Creates a new worktree for a given card ID and branch name.
/// The worktree will be created within the `repo_path/.worktrees/` directory.
pub async fn create_worktree(
    repo_path: &Path,
    card_id: &str,
    branch_name: &str,
) -> Result<PathBuf, String> {
    let worktrees_dir = repo_path.join(".worktrees");
    tokio::fs::create_dir_all(&worktrees_dir)
        .await
        .map_err(|e| format!("Failed to create .worktrees directory: {}",
e))?;

    let worktree_path = worktrees_dir.join(format!("{}_{}", card_id, branch_na
me));

    if worktree_path.exists() {
        return Err(format!("Worktree already exists at {}", worktree_path.disp
lay()));
    }

    let result = run_git_command(
        repo_path,
        &[
```

```
            "worktree",
            "add",
            worktree_path.to_str().unwrap(), // Safe due to previous check
            branch_name,
        ],
    ).await?;

    // If branch_name didn't exist, git worktree add creates it and checks it
out.
    // We should also ensure the main branch exists and has an initial commit
for worktrees to work reliably.
    // For simplicity, we assume the main repo is initialized and has at least
one commit.

    Ok(worktree_path)
}

/// Removes a worktree for a given card ID and branch name.
pub async fn remove_worktree(
    repo_path: &Path,
    card_id: &str,
    branch_name: &str,
    force: bool,
) -> Result<String, String> {
    let worktrees_dir = repo_path.join(".worktrees");
    let worktree_path = worktrees_dir.join(format!("{}_{}", card_id, branch_na
me));

    if !worktree_path.exists() {
        return Err(format!("Worktree not found at {}", worktree_path.display()
));
    }

    let mut args = vec!["worktree", "remove"];
    if force {
        args.push("--force");
    }
    args.push(worktree_path.to_str().unwrap()); // Safe due to previous check

    run_git_command(repo_path, &args).await
}

/// Gets the path to the agent development repository.
/// For simplicity, we'll place it in a known location relative to the app
data directory.
pub fn get_agent_dev_repo_path(app_handle: &tauri::AppHandle) -> Result<PathBu
f, String> {
    let app_data_dir = app_handle.path().app_data_dir().ok_or("Failed to get
app data directory")?;
    let repo_path = app_data_dir.join("agent_dev_repo");
    Ok(repo_path)
}
```

Explanation:

- `run_git_command`: A helper function to execute any `git` command. It takes the repository path and a slice of arguments. It captures `stdout` and `stderr` and returns an `Ok` result on success or an `Err` with detailed error messages. We use `tokio::process::Command` for async execution.
- `init_repo_if_not_exists`: This function ensures our target `agent_dev_repo` is a valid Git repository. If the directory doesn't exist, it creates it and initializes a new Git repo. If the directory exists but isn't a repo, it initializes it.
- `create_worktree`: This is the core function for adding a worktree. It constructs the target path within `.worktrees/` and executes `git worktree add`. It returns the path to the newly created worktree.
- `remove_worktree`: This function deletes a worktree. It supports a `force` option, mapping directly to `git worktree remove --force`.
- `get_agent_dev_repo_path`: A utility to determine the consistent location for our `agent_dev_repo`. Placing it in the app's data directory (e.g., `~/Library/Application Support/com.kanbots.dev/agent_dev_repo` on macOS) is a robust solution for desktop apps.

3. Integrate Commands into `src-tauri/src/main.rs`

Now, expose these Git commands as Tauri IPC commands.

```
// src-tauri/src/main.rs
// ... existing imports ...
mod git_commands; // Import the new module

// Tauri command to initialize the agent development repository
#[tauri::command]
async fn setup_agent_dev_repo(app_handle: tauri::AppHandle) -> Result<String,
String> {
    let repo_path = git_commands::get_agent_dev_repo_path(&app_handle)?;
    git_commands::init_repo_if_not_exists(&repo_path).await
}

// Tauri command to create a new worktree for a card
#[tauri::command]
async fn create_card_worktree(
    app_handle: tauri::AppHandle,
    card_id: String,
    branch_name: String,
) -> Result<String, String> {
    let repo_path = git_commands::get_agent_dev_repo_path(&app_handle)?;
    let worktree_path = git_commands::create_worktree(&repo_path, &card_id, &b
ranch_name).await?;
    Ok(format!("Worktree created at {}", worktree_path.display()))
}
```

```
// Tauri command to remove a worktree for a card
#[tauri::command]
async fn remove_card_worktree(
    app_handle: tauri::AppHandle,
    card_id: String,
    branch_name: String,
    force: bool,
) -> Result<String, String> {
    let repo_path = git_commands::get_agent_dev_repo_path(&app_handle)?;
    let result = git_commands::remove_worktree(&repo_path, &card_id, &branch_n
ame, force).await?;
    Ok(result)
}


fn main() {
    tauri::Builder::new()
        .setup(|app| {
            // Optional: Initialize the repo on app startup.
            // For this chapter, we'll trigger it from the UI for clarity.
            // let handle = app.handle();
            // tauri::async_runtime::spawn(async move {
            //     match setup_agent_dev_repo(handle).await {
            //         Ok(msg) => println!("Repo setup: {}", msg),
            //         Err(e) => eprintln!("Failed to setup repo: {}", e),
            //     }
            // });
            Ok(())
        })
        .invoke_handler(tauri::generate_handler![
            // ... existing handlers ...
            setup_agent_dev_repo,
            create_card_worktree,
            remove_card_worktree,
        ])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Explanation:

- We import our new `git_commands` module.
- `setup_agent_dev_repo`: An IPC command to ensure the `agent_dev_repo` is ready. This is a good first step for the user.
- `create_card_worktree`: Takes `card_id` and `branch_name` and calls our `git_commands::create_worktree` function.
- `remove_card_worktree`: Takes `card_id`, `branch_name`, and a `force` boolean, then calls `git_commands::remove_worktree`.
- These new commands are added to `tauri::generate_handler!`.

4. Create an Initial Commit in the Agent Dev Repo (Manual Step)

For `git worktree add` to work reliably, the main repository (`agent_dev_repo`) should have at least one commit. This is a common pitfall. Before testing, run these commands once in the `agent_dev_repo` directory (which will be created when you first run `setup_agent_dev_repo` via Kanbots):

```
# Navigate to your agent_dev_repo after it's created by Kanbots
# Example: cd ~/Library/Application\ Support/com.kanbots.dev/agent_dev_repo/
(macOS)
git add .
git commit -m "Initial commit for agent development repository"
```

You can also automate this in `init_repo_if_not_exists` by checking if there are any commits, but for now, we'll keep it a manual setup step to highlight the requirement.

Integrating with the Svelte Frontend

Now, let's add UI elements to our Svelte frontend to trigger these Rust commands. We'll add buttons to a Kanban card component.

1. Update `src/lib/tauri.ts`

If you have a `tauri.ts` file for IPC calls, add the new command definitions.

```
// src/lib/tauri.ts
import { invoke } from '@tauri-apps/api/core';

export const tauriCommands = {
    // ... existing commands ...
    setupAgentDevRepo: async (): Promise<string> => {
        return invoke('setup_agent_dev_repo');
    },
    createCardWorktree: async (cardId: string, branchName: string): Promise<st
ring> => {
        return invoke('create_card_worktree', { cardId, branchName });
    },
    removeCardWorktree: async (cardId: string, branchName: string, force: bool
ean = false): Promise<string> => {
        return invoke('remove_card_worktree', { cardId, branchName, force });
    },
};
```

2. Add Buttons to a Kanban Card Component

Let's assume you have a `KanbanCard.svelte` component. We'll add buttons to it.

```
<!-- src/lib/components/KanbanCard.svelte -->
<script lang="ts">
    import { tauriCommands } from '$lib/tauri';
    import { createEventDispatcher } from 'svelte';
    import { writable } from 'svelte/store';

    export let card: { id: string; title: string; description: string;
worktreePath?: string; };

    const dispatch = createEventDispatcher();
    const worktreeStatus = writable(card.worktreePath ? 'created' : 'none');

    async function handleSetupRepo() {
        try {
            const result = await tauriCommands.setupAgentDevRepo();
            console.log('Repo setup:', result);
            alert(`Agent Dev Repo Setup: ${result}`);
        } catch (error) {
            console.error('Error setting up repo:', error);
            alert(`Error setting up repo: ${error}`);
        }
    }

    async function handleCreateWorktree() {
        const branchName = `feature-${card.id}`; // Simple branch naming
convention
        try {
            const result = await tauriCommands.createCardWorktree(card.id,
branchName);
            console.log('Worktree created:', result);
            alert(`Worktree created: ${result}`);
            card.worktreePath = result.split('at ')[1]; // Extract path from
message
            worktreeStatus.set('created');
            dispatch('cardUpdate', card); // Notify parent component of card
update
        } catch (error) {
            console.error('Error creating worktree:', error);
            alert(`Error creating worktree: ${error}`);
        }
    }

    async function handleRemoveWorktree() {
        const branchName = `feature-${card.id}`;
        if (!confirm('Are you sure you want to remove this worktree?'))
return;
        try {
            const result = await tauriCommands.removeCardWorktree(card.id,
branchName, true); // Force remove for simplicity in tutorial
            console.log('Worktree removed:', result);
            alert(`Worktree removed: ${result}`);
            card.worktreePath = undefined;
            worktreeStatus.set('none');
            dispatch('cardUpdate', card);
        } catch (error) {
```

```
            console.error('Error removing worktree:', error);
            alert(`Error removing worktree: ${error}`);
        }
    }
</script>

<div class="kanban-card">
    <h3>{card.title}</h3>
    <p>{card.description}</p>
    {#if $worktreeStatus === 'none'}
        <button on:click={handleCreateWorktree}>Create Worktree</button>
    {:else}
        <p>Worktree: {$worktreeStatus} ({card.worktreePath})</p>
        <button on:click={handleRemoveWorktree}>Remove Worktree</button>
    {/if}
    <button on:click={handleSetupRepo}>Setup Agent Repo (First Time)</button>
</div>

<style>
    .kanban-card {
        background-color: #f0f0f0;
        border: 1px solid #ccc;
        padding: 10px;
        margin-bottom: 10px;
        border-radius: 5px;
    }
    button {
        margin-right: 5px;
        padding: 5px 10px;
        cursor: pointer;
    }
</style>
```

Explanation:

- We import `tauriCommands` and `createEventDispatcher` from Svelte.
- `worktreeStatus` is a Svelte store to reactively update the UI based on worktree presence.
- `handleSetupRepo`: Calls the Rust command to initialize the `agent_dev_repo`. This button should ideally be in a global settings area, but for this chapter, it's placed on the card for easy access during testing.
- `handleCreateWorktree`: Generates a branch name (e.g., `feature-card_id`) and calls `createCardWorktree`. It then updates the card's `worktreePath` and `worktreeStatus`.
- `handleRemoveWorktree`: Calls `removeCardWorktree` with `force: true` for simpler testing.
- The UI conditionally renders "Create Worktree" or "Remove Worktree" based on `worktreeStatus`.

Testing & Verification: Ensuring Worktree Isolation

Now it's time to verify that our worktree management works as expected.

Steps to Verify

1. **Start Kanbots:**

```
    npm run tauri dev
```

1. **Initialize Agent Dev Repo:**
    - In the Kanbots UI, click the "Setup Agent Repo (First Time)" button on any card.
    - You should see an alert "Agent Dev Repo Setup: Repository already exists..." or "Repository initialized...".
    - **Crucial Manual Step:** Navigate to the `agent_dev_repo` directory (e.g., `~/Library/Application Support/com.kanbots.dev/agent_dev_repo/` on macOS, or similar in `AppData/Roaming` on Windows, or `~/.config` on Linux). If this is the first time, run:

```
        # Example path - adjust for your OS
        cd "$(tauri info | grep 'App data directory' | awk '{print $NF}')/
agent_dev_repo"
        git add .
        git commit -m "Initial commit for agent development repository"
```

```
    This ensures the main repo has a commit, which is required for adding
worktrees reliably.
```

1. **Create a New Kanban Card:** Add a new card to your board (e.g., "Implement User Login").
2. **Create Worktree for the Card:**
    - Click the "Create Worktree" button on your new card.
    - You should see an alert indicating the worktree was created, along with its path.
    - The button should change to "Remove Worktree".

3. **Verify Worktree on Filesystem:**
    - Open your terminal and navigate to the `agent_dev_repo` directory.
    - Run `git worktree list`. You should see your new worktree listed, pointing to the path like `agent_dev_repo/.worktrees/card_123_feature-card_123`.
    - Navigate into the new worktree directory: `cd .worktrees/card_123_feature-card_123`.
    - Run `git branch`. It should show you are on `feature-card_123`.
4. **Test Isolation:**
    - Inside the worktree directory (`agent_dev_repo/.worktrees/card_123_feature-card_123`), create a new file: `echo "Hello from agent" > agent_output.txt`.
    - Commit this change:

```
        git add .
        git commit -m "Agent generated file"
```

```
- Now, navigate back to the main `agent_dev_repo` directory: `cd ../..`.
- Check if `agent_output.txt` exists here: `ls -l agent_output.txt`. It should
*not* exist.
- Verify the current branch: `git branch`. You should be on `main` (or
whatever your default branch is). This demonstrates the isolation.
```

1. **Remove Worktree:**
    - Back in the Kanbots UI, click "Remove Worktree" on the card.
    - Confirm the action. You should see an alert confirming removal.
    - The button should revert to "Create Worktree".
2. **Verify Removal:**
    - In your terminal, run `git worktree list` again. The worktree should no longer be listed.
    - Check the filesystem: `ls -l .worktrees/card_123_feature-card_123`. The directory should be gone.

If all these steps pass, your Kanbots application is successfully managing Git worktrees!

Production Considerations: Robust Worktree Management

While our current implementation is functional, a production-ready system requires more robustness.

- **Error Handling:** The current error messages are basic. In production, we'd parse Git's stderr more carefully to provide user-friendly feedback (e.g., "Branch already exists," "Worktree not empty," "Git command not found").
- **Disk Space Management:** Each worktree consumes disk space. If users create many cards/worktrees, this can add up. Consider:
    - A warning or limit on the number of active worktrees.
    - Automatic cleanup of old/stale worktrees (e.g., after a card is archived).
- **Concurrency and Locking:** While Git is generally safe with worktrees, if multiple agents try to modify the same underlying repository objects (e.g., fetching new remotes) at the exact same time, there could be edge cases. For now, our commands are simple enough not to cause issues, but complex workflows might need more careful synchronization.
- **Security:** AI agents will execute commands within their worktrees. While worktrees provide filesystem isolation for code changes, the agent itself runs with the permissions of the Kanbots application. Ensure agents are not given capabilities that could compromise the host system (e.g., running arbitrary executables outside their designated worktree). This will be a more critical concern in later chapters when agents are actually introduced.
- **User Feedback:** Beyond simple alerts, the UI should provide real-time feedback on worktree status (e.g., "Creating worktree...", "Worktree ready", "Error creating worktree").
- **Branch Management:** Our current implementation uses a simple `feature-card_id` branch name. A more advanced system might allow users to specify branch names, or integrate with remote repositories for pushing agent-generated branches.

Common Issues & Solutions

1. **"Repository does not have any commits yet" or "fatal: working tree is not a git repository"**:
    - **Issue:** `git worktree add` requires the main repository (`agent_dev_repo`) to have at least one commit.
    - **Solution:** Ensure you've followed the manual step to `git add .` and `git commit` in your `agent_dev_repo` after it's initialized.
2. **"Worktree already exists at "**:
    - **Issue:** You tried to create a worktree at a path that already exists. This can happen if a previous `remove_worktree` failed to delete the directory.
    - **Solution:** Manually delete the offending directory (`rm -rf <path>`) and then try creating the worktree again. Our `create_worktree` function checks for this.
3. **"fatal: 'worktree' is not a git command"**:
    - **Issue:** Your Git installation might be too old, or Git is not properly installed/in your system's PATH. Git worktrees were introduced in Git 2.5.
    - **Solution:** Update Git to a recent version (e.g., Git 2.45.0 as of 2026-05-24). Ensure `git` is accessible from your terminal.
4. **Permissions Errors**:
    - **Issue:** Kanbots (and thus the Rust backend) doesn't have the necessary permissions to create directories or execute Git commands in the target location.
    - **Solution:** Check the permissions of the parent directory where `agent_dev_repo` is created. Ensure the user running Kanbots has write access. On Linux, placing it in `~/.config/kanbots/agent_dev_repo` is usually safe.

Summary & Next Steps

In this chapter, we've laid a crucial foundation for Kanbots: the ability to manage isolated Git worktrees. You've learned:

- The fundamental concept of Git worktrees and why they are invaluable for parallel, isolated development.
- How to design the interaction between the Svelte frontend and Rust backend for Git operations.
- Implemented Rust backend logic using `std::process::Command` to create and remove worktrees.
- Integrated these functionalities into the Svelte UI.
- Verified the worktree creation and isolation through practical terminal commands.

Your Kanbots application can now dynamically provision Git environments for each card, a prerequisite for intelligent agent execution. This architectural decision significantly reduces complexity for agent orchestration and ensures a clean, controlled environment for code generation and review.

Next, we'll build upon this by integrating our first AI agent (e.g., Claude Code or Codex) into a Kanban card, enabling it to perform tasks within its newly managed worktree.

References

- Git Worktree Documentation
- Tauri Documentation
- Svelte 5 Documentation
- Tokio Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Integrating Your First AI Agent: Claude Code or Codex

This chapter marks a pivotal moment for Kanbots. We're moving beyond a static Kanban board and injecting intelligence by integrating our first AI agent. You'll learn how to connect an AI model like Claude Code or a modern OpenAI equivalent (e.g., GPT-4o) to a Kanban card. This enables the agent to perform specific tasks, such as generating code, within its dedicated git worktree. By the end of this milestone, your Kanbots application will be able to dispatch a task to an AI agent, have that agent generate content (like a simple code file), and observe the results directly within the isolated worktree associated with your Kanban card. This lays the foundation for powerful, automated development workflows.

Project Overview: Bringing AI to Your Kanban Board

The goal of this chapter is to empower individual Kanban cards with AI capabilities. Imagine a card titled "Implement User Login." Instead of manually writing the initial boilerplate, you could prompt an AI agent directly from that card. The agent would then operate in its isolated environment, generate the code, and even commit it, ready for review. This significantly accelerates the initial development phase and allows developers to focus on higher-level logic.

Why This Matters

Integrating AI agents directly into your development workflow, especially in a local-first desktop application like Kanbots, offers several key advantages:

- **Accelerated Prototyping:** Quickly generate boilerplate, test cases, or initial code structures, reducing manual setup time.
- **Contextual AI Assistance:** Agents operate within the specific context of a Kanban card and its associated worktree, ensuring outputs are highly relevant.

- **Isolated Experimentation:** Git worktrees provide a safe sandbox for AI-generated code. Changes are isolated, easy to review, and can be discarded without affecting the main codebase.
- **Privacy and Control:** Running AI orchestration locally gives you more control over your data and interactions compared to purely cloud-based solutions.

By the end of this chapter, you will have a Kanbots application where a user can select an AI model, provide a prompt to a Kanban card, and witness the AI agent generate code directly into the card's dedicated git worktree.

Tech Stack Deep Dive

For this chapter, we're building on our existing Tauri (Rust + Svelte) foundation and introducing new components:

- **Rust (`reqwest`, `dotenv`, `tokio`):** The backend orchestrator will use `reqwest` for making HTTP requests to external AI APIs, `dotenv` for securely loading API keys, and `tokio` for asynchronous operations.
- **AI Agent APIs (Claude Code or OpenAI):** We'll integrate with either Anthropic's Claude API (using the latest Opus model) or OpenAI's API (using GPT-4o as a capable code-generation model, superseding older Codex models). These services provide the actual intelligence.
- **Git:** Our existing `git_worktree` module is crucial, as it provides the isolated environment for AI agents to operate within.
- **Svelte:** The frontend will provide the UI elements to trigger AI tasks and display their status.

Milestones and Build Plan

To achieve our goal of integrating a functional AI agent, we'll follow these steps:

1. **Secure API Key Management:** Implement a robust way to load API keys without hardcoding them.
2. **Rust AI Client Module:** Create a dedicated Rust module to handle communication with external AI APIs, abstracting away the HTTP details.

3. **Rust Agent Orchestration:** Modify the main Rust backend to receive AI task requests, prepare the git worktree, invoke the AI client, and write the generated output to a file within the worktree.
4. **Svelte UI Integration:** Add interactive elements to the Kanban card for triggering AI tasks, providing prompts, selecting models, and displaying agent status.

Architecture: Orchestrating the AI Interaction

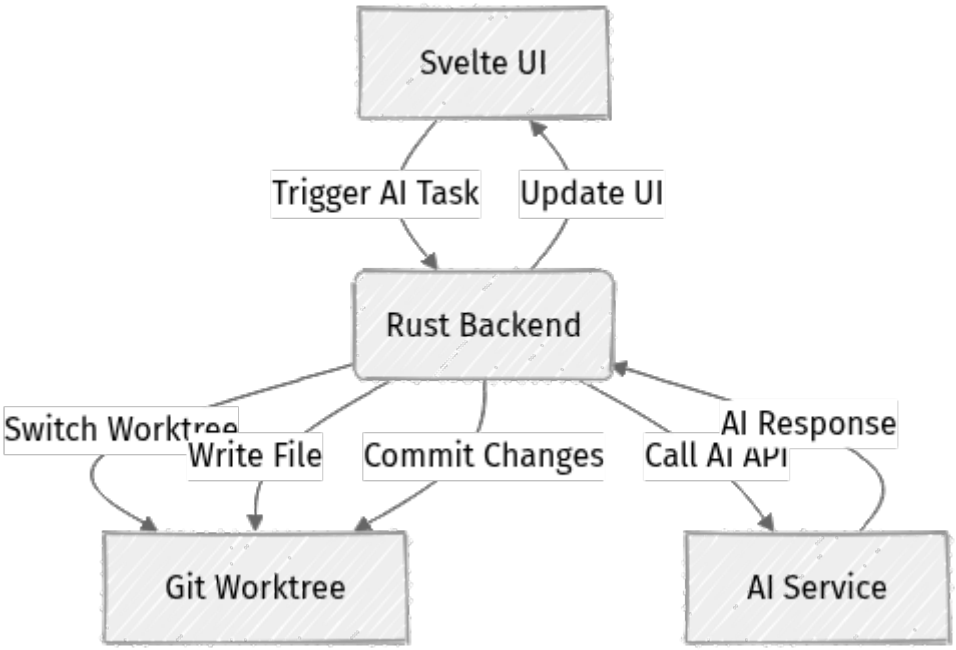
Integrating an AI agent requires careful orchestration between the frontend, the Rust backend, and the external AI API. The core idea is that a user action on a Kanban card triggers a backend process. This process then prepares a specific environment (a git worktree), invokes the AI, and writes the AI's output back into that environment.

Agent Interaction Data Flow

Here's the high-level flow we'll implement:

1. **Svelte UI (Frontend):** A user initiates an AI task from a Kanban card (e.g., clicks "Generate Code"). This sends a request to the Rust backend via Tauri's IPC mechanism, passing the card's ID, the worktree path, the task prompt, and the chosen AI model.
2. **Rust Backend (Orchestrator):**
    - Receives the request from the frontend.
    - Initializes or switches to the specific git worktree associated with the card.
    - Communicates with the external AI API (Claude or OpenAI) using the provided prompt and API key.
    - Receives the AI's generated text response.
    - Writes the AI-generated content into a new file within the active git worktree.
    - Optionally, performs `git add` and `git commit` to stage and save the AI's changes, providing a clear record.
    - Sends a success or failure message back to the Svelte UI.
3. **AI API (External Service):** Processes the prompt, applies its intelligence, and returns a generated text response.

4. **Git Worktree (Local Environment):** Serves as the isolated workspace where the AI agent performs its modifications. This ensures that agent changes are contained and don't conflict with other development efforts.



API Key Management Strategy

Accessing AI services like Claude or OpenAI requires an API key. For security, these keys must never be hardcoded directly into your application's source code. We will leverage environment variables for development and local testing. This is a standard and relatively secure practice for local environments. For production desktop applications, more robust OS-level secret management (e.g., system keychains, credential managers) would be the next step, but it's beyond the scope of this initial integration.

AI Client Abstraction

To maintain a clean and extensible Rust backend, we'll encapsulate the logic for interacting with AI APIs within a dedicated `ai_client` module. This module will handle HTTP requests, API key authentication, and response parsing, offering a simplified interface to the main application logic. This abstraction makes it easier to swap out AI providers or add new models in the future.

Step-by-Step Implementation

Let's build out the components required for our first AI agent integration.

1. Securely Manage AI API Keys with `dotenv`

First, ensure your AI API keys are handled securely. We'll use the `dotenv` crate to load environment variables from a `.env` file, preventing them from being committed to source control.

Create a `.env` file in the root of your Tauri project (where `Cargo.toml` is located) if you don't have one.

```
# .env
CLAUDE_API_KEY=sk-YOUR_CLAUDE_API_KEY_HERE
OPENAI_API_KEY=sk-YOUR_OPENAI_API_KEY_HERE
```

Important: Replace `sk-YOUR_CLAUDE_API_KEY_HERE` and `sk-YOUR_OPENAI_API_KEY_HERE` with your actual API keys. You only need to set the key for the model you intend to use.

Next, add `.env` to your `.gitignore` file to prevent accidental commitment:

```
# .gitignore
# ... other entries ...
.env
```

Now, open `Cargo.toml` and add the necessary dependencies for HTTP requests and environment variable loading:

```
# Cargo.toml

[dependencies]
tauri = { version = "2.0.0-beta.16", features = ["shell", "process"] } # Checked 2026-05-24
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.38.0", features = ["full"] } # Checked 2026-05-24
reqwest = { version = "0.12.5", features = ["json"] } # Checked 2026-05-24
dotenv = "0.15.0" # Checked 2026-05-24
```

Finally, modify your `src/main.rs` to load these environment variables at application startup:

```
// src/main.rs

// ... existing imports ...
use dotenv::dotenv; // Add this line
use std::env;      // Add this line

// ... existing tauri::command macros and mod declarations ...

// We'll add the actual implementation for this command later.
// For now, ensure it's declared for the main function to pick up.
#[tauri::command]
async fn run_ai_agent(card_id: String, worktree_path: String, prompt: String,
model_name: String) -> Result<String, String> {
    // Placeholder for now, full implementation below
    println!("AI agent invoked for card: {} in worktree: {} with prompt: {}
using model: {}", card_id, worktree_path, prompt, model_name);
    Ok(format!("AI agent task for card {} started (placeholder).", card_id))
}

fn main() {
    dotenv().ok(); // Load environment variables from .env file. `.ok()`
prevents crashing if file not found.
    // ... rest of your main function ...
    tauri::Builder::new()
        .invoke_handler(tauri::generate_handler![
            // ... existing commands ...
            run_ai_agent // Add our new command here
        ])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

This setup ensures that your API keys are loaded securely during development and runtime.

2. Rust Backend

- AI Client Module (`src/ai_client.rs`)

Create a new file `src/ai_client.rs`. This module will encapsulate the logic for interacting with different AI providers.

```
// src/ai_client.rs

use reqwest::Client;
use serde::{Deserialize, Serialize};
use std::env;
use std::collections::HashMap;

/// Enum to represent different AI models Kanbots can use.
/// We provide Claude and OpenAI (as a modern equivalent to Codex).
#[derive(Debug)]
pub enum AiModel {
    Claude,
    OpenAI,
}
```

```
impl AiModel {
    pub fn name(&self) -> &str {
        match self {
            AiModel::Claude => "Claude",
            AiModel::OpenAI => "OpenAI (GPT-4o)",
        }
    }
}

// --- Claude API Structures (Anthropic Messages API) ---
// See: https://docs.anthropic.com/claude/reference/messages_post

#[derive(Serialize, Debug)]
struct ClaudeMessage {
    role: String,
    content: String,
}

#[derive(Serialize, Debug)]
struct ClaudeRequest {
    model: String,
    max_tokens: u32,
    messages: Vec<ClaudeMessage>,
    #[serde(skip_serializing_if = "Option::is_none")] // Optional, for
example, system prompt
    system: Option<String>,
}

#[derive(Deserialize, Debug)]
struct ClaudeResponse {
    content: Vec<ClaudeContent>,
    #[serde(flatten)] // Capture other fields like 'id', 'model', 'usage',
etc.
    _other: HashMap<String, serde_json::Value>,
}

#[derive(Deserialize, Debug)]
struct ClaudeContent {
    #[serde(rename = "type")]
    content_type: String,
    text: String,
}

// --- OpenAI API Structures (Chat Completions API) ---
// See: https://platform.openai.com/docs/api-reference/chat/create

#[derive(Serialize, Debug)]
struct OpenAIRequest {
    model: String,
    messages: Vec<OpenAIMessage>,
    max_tokens: u32,
    #[serde(skip_serializing_if = "Option::is_none")] // Optional, for
example, temperature
    temperature: Option<f32>,
}

#[derive(Serialize, Debug, Deserialize)] // Deserialize for response parsing
too
struct OpenAIMessage {
    role: String,
    content: String,
```

```
}

#[derive(Deserialize, Debug)]
struct OpenAIResponse {
    choices: Vec<OpenAIChoice>,
    #[serde(flatten)]
    _other: HashMap<String, serde_json::Value>,
}

#[derive(Deserialize, Debug)]
struct OpenAIChoice {
    message: OpenAIMessage,
    #[serde(flatten)]
    _other: HashMap<String, serde_json::Value>,
}

/// Client for interacting with various AI models.
pub struct AiClient {
    client: Client,
    claude_api_key: Option<String>,
    openai_api_key: Option<String>,
}

impl AiClient {
    /// Creates a new `AiClient` by loading API keys from environment
variables.
    pub fn new() -> Self {
        let claude_api_key = env::var("CLAUDE_API_KEY").ok();
        let openai_api_key = env::var("OPENAI_API_KEY").ok();

        if claude_api_key.is_none() && openai_api_key.is_none() {
            eprintln!("⚠️ Warning: No AI API keys found. Please set
CLAUDE_API_KEY or OPENAI_API_KEY environment variable.");
        }

        AiClient {
            client: Client::new(),
            claude_api_key,
            openai_api_key,
        }
    }

    /// Generates code using the specified AI model.
    /// Returns the generated text or a descriptive error.
    pub async fn generate_code(&self, model_type: AiModel, prompt: &str) -> Re
sult<String, String> {
        match model_type {
            AiModel::Claude => self.generate_with_claude(prompt).await,
            AiModel::OpenAI => self.generate_with_openai(prompt).await,
        }
    }

    /// Internal method to interact with the Claude API.
    async fn generate_with_claude(&self, prompt: &str) -> Result<String, Strin
g> {
        let api_key = self.claude_api_key.as_ref()
            .ok_or_else(|| "Claude API key (CLAUDE_API_KEY) not set in
environment.".to_string())?;

        let url = "https://api.anthropic.com/v1/messages";
        // Using Claude 3 Opus as of 2026-05-24, known for strong reasoning
and code capabilities.
```

```
        let model = "claude-3-opus-20240229";

        let request_body = ClaudeRequest {
            model: model.to_string(),
            max_tokens: 1024, // Limit output length to prevent excessive
costs/response times
            messages: vec![
                ClaudeMessage {
                    role: "user".to_string(),
                    content: prompt.to_string(),
                },
            ],
            system: Some("You are a helpful and precise code generation
assistant. Only output code or explanations relevant to the user's request.".t
o_string()),
        };

        let response = self.client.post(url)
            .header("x-api-key", api_key)
            .header("anthropic-version", "2023-06-01") // Required Anthropic-
Version header
            .header("content-type", "application/json")
            .json(&request_body)
            .send()
            .await
            .map_err(|e| format!("Failed to send request to Claude API: {}",
e))?;

        let status = response.status();
        let response_text = response.text().await.map_err(|e| format!("Failed
to read Claude API response text: {}", e))?;

        if status.is_success() {
            let claude_response: ClaudeResponse = serde_json::from_str(&respon
se_text)
                .map_err(|e| format!
("Failed to parse Claude API response: {}. Raw response: {}", e,
response_text))?;

            if let Some(content_block) = claude_response.content.into_iter().f
ind(|c| c.content_type == "text") {
                Ok(content_block.text)
            } else {
                Err(format!("Claude API response missing expected text
content. Raw response: {}", response_text))
            }
        } else {
            Err(format!("Claude API error: Status {}. Response: {}", status, r
esponse_text))
        }
    }

    /// Internal method to interact with the OpenAI API.
    async fn generate_with_openai(&self, prompt: &str) -> Result<String, Strin
g> {
        let api_key = self.openai_api_key.as_ref()
            .ok_or_else(|| "OpenAI API key (OPENAI_API_KEY) not set in
environment.".to_string())?;

        let url = "https://api.openai.com/v1/chat/completions";
        // Using gpt-4o as a modern, highly capable model for code generation,
        // effectively superseding older Codex models. Checked 2026-05-24.
```

```
        let model = "gpt-4o";

        let request_body = OpenAIRequest {
            model: model.to_string(),
            messages: vec![
                OpenAIMessage {
                    role: "user".to_string(),
                    content: prompt.to_string(),
                },
            ],
            max_tokens: 1024, // Limit output length
            temperature: Some(0.7), // Controls randomness: lower for more
deterministic, higher for more creative
        };

        let response = self.client.post(url)
            .header("Authorization", format!("Bearer {}", api_key))
            .header("Content-Type", "application/json")
            .json(&request_body)
            .send()
            .await
            .map_err(|e| format!("Failed to send request to OpenAI API: {}",
e))?;

        let status = response.status();
        let response_text = response.text().await.map_err(|e| format!("Failed
to read OpenAI API response text: {}", e))?;

        if status.is_success() {
            let openai_response: OpenAIResponse = serde_json::from_str(&respon
se_text)
                .map_err(|e| format!
("Failed to parse OpenAI API response: {}. Raw response: {}", e,
response_text))?;

            if let Some(choice) = openai_response.choices.into_iter().next() {
                Ok(choice.message.content)
            } else {
                Err(format!("OpenAI API response missing choice content. Raw
response: {}", response_text))
            }
        } else {
            Err(format!("OpenAI API error: Status {}. Response: {}", status, r
esponse_text))
        }
    }
}
```

Explanation of `src/ai_client.rs`:

- **`AiModel` Enum:** This enum allows us to easily select which AI model to use, abstracting the underlying API calls.

- **Request/Response Structs:** These `struct` definitions, marked with `#[derive(Serialize, Deserialize)]`, mirror the JSON payloads expected by the Claude and OpenAI APIs. This ensures type-safe communication.
    - **Claude:** Interacts with Anthropic's `<https://api.anthropic.com/v1/messages>` endpoint. It requires specific headers like `x-api-key` and `anthropic-version`. We're using `claude-3-opus-20240229`, the latest powerful model as of 2026-05-24.
    - **OpenAI (Codex equivalent):** Interacts with OpenAI's `<https://api.openai.com/v1/chat/completions>` endpoint. It uses an `Authorization: Bearer <API_KEY>` header. We've chosen `gpt-4o` as a modern, highly capable model for code generation, as the original Codex models are largely superseded by newer OpenAI models like GPT-3.5 and GPT-4 series.
- **`AiClient` Struct:** This holds the `reqwest::Client` instance (for making HTTP requests) and the loaded API keys.
- **`new()`:** The constructor attempts to load API keys from environment variables. If none are found, it prints a warning, but allows the client to be created, enabling graceful degradation if a specific model isn't configured.
- **`generate_code()`:** This public method serves as the entry point for AI code generation. It dispatches the request to the appropriate internal method (`generate_with_claude` or `generate_with_openai`) based on the `AiModel` enum.
- **`generate_with_claude()` / `generate_with_openai()`:** These private methods handle the specifics of constructing HTTP requests, setting headers, sending JSON payloads, and parsing the responses for each API. They include robust error handling for network issues, API-specific error responses, and JSON deserialization failures. A `max_tokens` limit is set to manage response length and potential costs.

3. Rust Backend

- Agent Orchestration Logic (`src/main.rs`)

Now, let's connect the `AiClient` with our git worktree management. We'll modify the `run_ai_agent` command in `src/main.rs`. We'll also need to import our new `ai_client` module and the `git_worktree` module (which you should have from the previous chapter).

First, ensure `src/main.rs` includes the `mod` declarations for both modules:

```
// src/main.rs

mod git_worktree; // From previous chapter (Chapter 2)
mod ai_client;    // Our new module

// ... other uses ...
use git_worktree::{GitWorktree, WorktreeError}; // Ensure WorktreeError is
imported if used
use ai_client::{AiClient, AiModel};

// ... existing tauri::command macros ...

#[tauri::command]
async fn run_ai_agent(card_id: String, worktree_path: String, prompt: String,
model_name: String) -> Result<String, String> {
    println!("Invoking AI agent for card: {} in worktree: {} with prompt: {}
using model: {}", card_id, worktree_path, prompt, model_name);

    let ai_client = AiClient::new();
    let model_type = match model_name.as_str() {
        "Claude" => AiModel::Claude,
        "OpenAI" => AiModel::OpenAI,
        _ => return Err(format!("Unsupported AI model selected: {}. Please
choose 'Claude' or 'OpenAI'.", model_name)),
    };

    // 1. Ensure the worktree exists and switch to it
    // This provides the isolated environment for the AI agent.
    let worktree_manager = GitWorktree::new(&worktree_path)
        .map_err(|e| format!("Failed to initialize GitWorktree for {}: {}", wo
rktree_path, e))?;

    worktree_manager.switch_to_worktree()
        .map_err(|e| format!("Failed to switch to worktree {}: {}", worktree_p
ath, e))?;

    // 2. Call the AI client to generate code
    let generated_content = ai_client.generate_code(model_type,
&prompt).await?;

    // 3. Define a filename for the generated content.
    // In a more advanced system, the AI might suggest a filename or the user
could specify one.
    // For now, we'll use a descriptive name based on the card ID.
    let file_name = format!("{}_generated_task.md",
card_id); // Using .md for general content, could be .py, .js, etc.
    let file_path = std::path::PathBuf::from(&worktree_path).join(&file_name);

    // 4. Write the generated content to the file within the active worktree
    tokio::fs::write(&file_path, generated_content.as_bytes())
        .await
        .map_err(|e| format!("Failed to write generated content to file {}:
{}", file_path.display(), e))?;

    // 5. Stage and commit the changes
    // This creates a clear, auditable record of the AI agent's contribution.
```

```
    worktree_manager.add(&file_name)
        .map_err(|e| format!("Failed to stage file {}: {}", file_name, e))?;

    let commit_message = format!
("feat(ai): Kanbots AI agent generated content for card {}", card_id);
    worktree_manager.commit(&commit_message)
        .map_err(|e| format!("Failed to commit changes for card {}: {}", card_
id, e))?;


    Ok(format!("AI agent successfully generated content to {} and committed
changes.", file_path.display()))
}

// ... main function remains the same, ensure run_ai_agent is in
generate_handler! ...
```

Explanation of `run_ai_agent` command:

- **Arguments:** It now accepts `card_id`, `worktree_path`, `prompt`, and `model_name` from the frontend.
- **`AiClient` Initialization:** An `AiClient` instance is created, which automatically attempts to load API keys.
- **Model Selection:** The `model_name` string from the frontend is matched to our `AiModel` enum.
- **Worktree Preparation:** It uses `GitWorktree::new` to manage the worktree at `worktree_path` and `switch_to_worktree` to ensure the AI operates in the correct, isolated environment.
- **AI Invocation:** `ai_client.generate_code` is called with the selected model and the user's prompt. `await?` handles potential errors, propagating them up.
- **File Creation:** A descriptive filename (`{card_id}_generated_task.md`) is constructed, and the AI's response is written to this file within the active worktree using `tokio::fs::write`.
- **Git Commit:** The generated file is staged (`worktree_manager.add`) and committed (`worktree_manager.commit`) with a clear commit message. This makes the AI's contribution a traceable part of the version history.
- **Result:** A success message or an error is returned to the frontend.

4. Svelte Frontend

- UI Integration (`src/lib/components/KanbanCard.svelte`)

Now, let's add the UI elements to a Kanban card to trigger this AI agent. First, ensure your `Card` interface (or type) in Svelte includes a `worktreePath`, as we need this to tell the Rust backend where the agent should operate. This was likely added in Chapter 2 or 3.

```
// src/lib/types/kanban.ts (example, verify path in your project)
export interface Card {
    id: string;
    title: string;
    description: string;
    columnId: string;
    worktreePath?: string; // This property is crucial for agent operations
    // ... other properties you might have
}
```

Now, modify `src/lib/components/KanbanCard.svelte` to add a button, prompt input, model selection, and display agent status.

```
<!-- src/lib/components/KanbanCard.svelte -->
<script lang="ts">
    import type { Card } from '$lib/types/kanban'; // Adjust path if needed
based on your project structure
    import { invoke } from '@tauri-apps/api/core';
    import { createEventDispatcher } from 'svelte';
    import { fade } from 'svelte/transition';

    export let card: Card;

    const dispatch = createEventDispatcher();
    let showEdit = false;
    let newTitle = card.title;
    let newDescription = card.description;
    let agentStatus: string = '';
    let isLoadingAgent = false;
    let aiPrompt = 'Generate a simple "hello world" Python script, including a
main function and a docstring.';
    let selectedModel = 'Claude'; // Default to Claude, allow user to pick

    function startEditing() {
        showEdit = true;
    }

    function saveCard() {
        dispatch('updateCard', {
            id: card.id,
            title: newTitle,
            description: newDescription,
        });
        showEdit = false;
    }

    function deleteCard() {
        dispatch('deleteCard', card.id);
    }
```

```
    async function runAgent() {
        if (!card.worktreePath) {
            agentStatus = 'Error: No worktree path defined for this card.
Please assign a worktree first.';
            return;
        }
        if (!aiPrompt.trim()) {
            agentStatus = 'Error: Please enter a prompt for the AI agent.';
            return;
        }

        isLoadingAgent = true;
        agentStatus = `Running ${selectedModel} agent... This may take a
moment.`;
        try {
            const result: string = await invoke('run_ai_agent', {
                cardId: card.id,
                worktreePath: card.worktreePath,
                prompt: aiPrompt,
                modelName: selectedModel,
            });
            agentStatus = result; // Display the success message from Rust
            console.log('AI Agent Result:', result);
        } catch (e: any) {
            agentStatus = `Agent Error: ${e}`;
            console.error('AI Agent Error:', e);
        } finally {
            isLoadingAgent = false;
        }
    }
</script>

<div
    class="bg-white p-4 rounded-lg shadow-md mb-3 border border-gray-200
hover:border-blue-400 cursor-grab"
    draggable="true"
    on:dragstart={(e) => {
        e.dataTransfer?.setData('text/plain', card.id);
        e.dataTransfer?.effectAllowed = 'move';
    }}
>
    {#if showEdit}
        <input
            type="text"
            bind:value={newTitle}
            class="w-full p-2 mb-2 border rounded-md"
            on:blur={saveCard}
            on:keydown={(e) => e.key === 'Enter' && saveCard()}
        />
        <textarea
            bind:value={newDescription}
            class="w-full p-2 mb-2 border rounded-md"
            on:blur={saveCard}
        ></textarea>
        <div class="flex justify-end space-x-2">
            <button on:click={saveCard} class="px-3 py-1 bg-green-500 text-
white rounded-md text-sm">Save</button>
            <button on:click={deleteCard} class="px-3 py-1 bg-red-500 text-
white rounded-md text-sm">Delete</button>
        </div>
    {:else}
        <h3 class="font-semibold text-lg mb-1">{card.title}</h3>
```

```
        <p class="text-gray-600 text-sm mb-2">{card.description}</p>
        <p class="text-gray-500 text-xs mb-2">Worktree: {card.worktreePath ||
'N/A'}</p>

        <div class="mt-4 border-t pt-3">
            <h4 class="font-medium text-md mb-2">AI Agent Task</h4>
            <textarea
                bind:value={aiPrompt}
                rows="3"
                class="w-full p-2 mb-2 text-sm border rounded-md bg-gray-50
focus:outline-none focus:ring-2 focus:ring-blue-500"
                placeholder="Enter AI prompt here..."
            ></textarea>
            <select bind:value={selectedModel} class="w-full p-2 mb-2 text-sm
border rounded-md bg-gray-50 focus:outline-none focus:ring-2 focus:ring-
blue-500">
                <option value="Claude">Claude (Anthropic)</option>
                <option value="OpenAI">OpenAI (GPT-4o)</option>
            </select>
            <button
                on:click={runAgent}
                disabled={isLoadingAgent || !card.worktreePath || !
aiPrompt.trim()}
                class="w-full px-4 py-2 bg-blue-600 text-white rounded-md
hover:bg-blue-700 disabled:opacity-50 disabled:cursor-not-allowed text-sm
font-semibold"
            >
                {#if isLoadingAgent}
                    Running Agent...
                {:else}
                    Run AI Agent
                {/if}
            </button>
            {#if agentStatus}
                <p class="mt-2 text-xs {agentStatus.startsWith('Error:') ?
'text-red-600' : 'text-gray-700'}" transition:fade>{agentStatus}</p>
            {/if}
        </div>

        <div class="flex justify-end mt-3 space-x-2">
            <button on:click={startEditing} class="px-3 py-1 bg-gray-200 text-
gray-800 rounded-md text-sm hover:bg-gray-300">Edit</button>
            <button on:click={deleteCard} class="px-3 py-1 bg-red-500 text-
white rounded-md text-sm hover:bg-red-600">Delete</button>
        </div>
    {/if}
</div>

<style>
    /* Add any specific styles for KanbanCard here */
</style>
```

Explanation of `KanbanCard.svelte` changes:

- **`runAgent()` function:** This asynchronous function is triggered by the "Run AI Agent" button.
  - It includes checks for `card.worktreePath` and a non-empty `aiPrompt` to provide immediate user feedback.
  - `isLoadingAgent` is set to `true` to disable the button and show a loading state, preventing multiple concurrent requests from the same card.
  - It calls the Tauri backend command `run_ai_agent` using `invoke`, passing the card's ID, its associated `worktreePath`, the `aiPrompt` from the textarea, and the `selectedModel`.
  - It updates `agentStatus` with success messages from the Rust backend or any errors encountered.
- **UI Elements:**
  - A `textarea` is provided for the user to input the AI prompt.
  - A `select` dropdown allows choosing between "Claude (Anthropic)" and "OpenAI (GPT-4o)".
  - A "Run AI Agent" button is dynamically enabled/disabled based on `isLoadingAgent`, the presence of a `worktreePath`, and a valid `aiPrompt`.
  - A paragraph displays the `agentStatus`, providing real-time feedback to the user. The text color changes to red for errors.
- **`transition:fade`:** A simple Svelte transition is used on the `agentStatus` paragraph for a smoother user experience.

Testing & Verification

Now that we've integrated the AI agent, let's verify it works as expected.

1. **Start the Kanbots application:**

```
cargo tauri dev
```

```
Observe the terminal output for any Rust compilation errors or warnings from
`AiClient::new()` regarding missing API keys.
```

1. **Ensure API Keys are Set:**
    - Verify that your `.env` file exists in the project root.
    - Confirm that `CLAUDE_API_KEY` or `OPENAI_API_KEY` (depending on which model you want to test) is correctly set with your actual API key.
    - **Crucial:** If you modified `.env`, you must restart `cargo tauri dev` for the changes to take effect.
2. **Create a New Kanban Card:** Add a new card to any column in your Kanbots application.
3. **Initialize a Worktree for the Card:**
    - From a previous chapter (Chapter 2 or 3), you should have functionality to associate a git worktree with a card. Use that to create a new worktree for your new card.
    - For example, if your base repository is `~/kanbots-repo`, you might create a worktree at `~/kanbots-repo/worktrees/card-123-feature`. Note the exact path.
4. **Enter a Prompt:** In the "AI Agent Task" section of the card, enter a clear, concise prompt. For example:
    - "Write a Python function `is_prime(n)` that checks if a number is prime. Include a docstring and example usage."
    - "Generate an HTML boilerplate with a title 'Kanbots Generated Page' and a simple h1 tag."
5. **Select AI Model:** Choose either "Claude (Anthropic)" or "OpenAI (GPT-4o)" from the dropdown, depending on which API key you've set up.
6. **Run AI Agent:** Click the "Run AI Agent" button.
    - Observe the `agentStatus` message updating in the Svelte UI. It should first show "Running Agent..." then a success or error message.
    - Check the terminal where `cargo tauri dev` is running for Rust backend `println!` messages and any potential errors.

7. **Verify Output:**
    - Once the agent reports success in the UI, open your terminal or file explorer.
    - Navigate to the worktree directory associated with your card (e.g., `~/kanbots-repo/worktrees/card-123-feature`).
    - You should find a new file there, named something like `your-card-id_generated_task.md` (or `.py`, `.html` if you adjust the extension in Rust).
    - Open this file. It should contain the code or text generated by the AI based on your prompt.
    - To verify the `git commit`, navigate to your base repository (`~/kanbots-repo`) and run `git log --oneline --graph --all`. You should see a new commit from the AI agent in its worktree branch.

Quick Debugging Checks:

- **Tauri Terminal Output:** The terminal running `cargo tauri dev` is your primary Rust backend log. Look for `println!` messages from `run_ai_agent` and any `eprintln!` warnings from `AiClient::new()` or errors from `reqwest` or `serde_json`.
- **Browser Developer Console (Svelte):** Press F12 in the Kanbots app window and check the "Console" tab for any frontend errors related to `invoke` calls or Svelte component rendering.
- **Environment Variables:** If you encounter "API key not set" errors, carefully re-check your `.env` file's spelling, location, and ensure you restarted `cargo tauri dev`.
- **Network Requests (Rust):** If AI API calls fail, the `ai_client.rs` module's error messages will be crucial. These typically indicate network issues, invalid API keys, or API-specific errors (e.g., rate limits).
- **File System Permissions:** Ensure the Kanbots application has write permissions to the worktree directories.

Production Considerations

While our current implementation is functional, deploying a desktop application with AI integration requires attention to several production-grade concerns:

- **API Key Security:** For a truly production-ready desktop app, relying solely on `.env` files is insufficient. Consider using OS-level secret management (e.g., `keytar` for Node.js/Electron, or native Rust crates that interface with system keychains like `keyring-rs`) to store API keys encrypted. This protects keys even if the `.env` file is accidentally exposed.
- **Rate Limiting and Cost Management:** AI APIs are metered and can be expensive. Implement:
    - **User Warnings:** Clearly inform users about potential costs associated with AI agent usage.
    - **Client-side Guards:** Basic rate limiting in the frontend to prevent accidental rapid-fire requests.
    - **Backend Throttling:** More robust rate limiting in the Rust backend to manage calls to the external APIs and provide clear feedback (`429 Too Many Requests`).
    - **Usage Tracking:** Log AI API call counts and token usage to help users monitor their spending.
- **Robust Error Handling & User Feedback:** The current `Result<String, String>` is simple. For production, define a custom error `enum` in Rust for more structured error types (e.g., `AiApiError::InvalidKey`, `AiApiError::RateLimitExceeded`, `WorktreeError::NotFound`). This allows the frontend to display highly specific, actionable error messages to the user.
- **Asynchronous Operations & Responsiveness:** AI API calls can introduce noticeable latency (~2-10 seconds, depending on the model and prompt complexity). Ensure the UI remains responsive using clear loading indicators, and that long-running operations in Rust are truly non-blocking, which `async`/`await` and `tokio` help facilitate.
- **Comprehensive Logging:** Implement detailed logging in the Rust backend for all AI interactions: prompts sent, full responses received, token counts, and all errors. This is invaluable for debugging, auditing agent behavior, and understanding usage patterns.

- **Agent Persona and Context Management:** As agents become more complex, managing their "persona" (e.g., "Developer," "Reviewer") and providing precise context will be critical to getting reliable outputs. This will be explored in later chapters.

Common Issues & Solutions

1. **"API key not set" or authentication errors:**
    - **Issue:** The `CLAUDE_API_KEY` or `OPENAI_API_KEY` environment variable is not correctly loaded or is missing.
    - **Solution:**
        - Verify the `.env` file exists in the project root of your Tauri project.
        - Ensure the variable name is exactly `CLAUDE_API_KEY` or `OPENAI_API_KEY`.
        - Confirm `dotenv().ok()` is called at the very beginning of your `main` function in `src/main.rs`.
        - **Crucial:** Always restart `cargo tauri dev` after making changes to `.env` or `Cargo.toml`.
        - Double-check the API key itself for typos, leading/trailing spaces, or expiration.
        - For OpenAI, ensure your key starts with `sk-` and is not an organization ID.
        - For Claude, ensure your key starts with `sk-ant-` and is not an organization ID.
2. **`invoke('run_ai_agent', ...)` fails with "command not found":**
    - **Issue:** The Tauri backend isn't correctly registering the `run_ai_agent` command.
    - **Solution:** Ensure `run_ai_agent` is explicitly included in the `tauri::generate_handler![]` macro within your `main.rs`. Recompile and restart the app. Check for any Rust compiler errors related to missing `#[tauri::command]` attributes.

3. **AI agent generates unexpected, incomplete, or empty output:**
    - **Issue:** The prompt might be too vague, the AI model is hallucinating, the `max_tokens` limit was hit, or the API call failed silently (though our `ai_client` tries to catch this).
    - **Solution:**
        - Check the Rust backend's console output for any errors from `ai_client` (e.g., JSON parsing errors, API specific errors).
        - Try a simpler, more direct prompt. Experiment with prompt engineering.
        - Temporarily increase `max_tokens` in `ai_client.rs` to see if the output was truncated.
        - If possible, inspect the raw `response_text` in the `ai_client.rs` code during debugging to see exactly what the API returned. This can reveal subtle API-side issues.
4. **File not found in worktree after agent runs:**
    - **Issue:** The agent successfully ran, but the file was written to the wrong location, or a file system error prevented writing.
    - **Solution:**
        - Double-check the `file_path` construction in `run_ai_agent` to ensure it's pointing to the correct worktree and filename. Use `file_path.display()` to print the full path in Rust logs.
        - Verify that the `worktree_path` passed from the frontend is correct and corresponds to an actual, initialized git worktree.
        - Check for file system permission errors in the Rust logs. The application needs write access to the worktree directory.

Summary & Next Step

You've just enabled your Kanbots application to leverage the power of external AI agents! You've successfully:

- Implemented secure API key management using environment variables and the `dotenv` crate.
- Created a robust Rust `AiClient` module to abstract and handle communication with Claude and OpenAI APIs.

- Extended the Rust backend to orchestrate AI agent tasks, including switching to specific git worktrees, invoking the AI, writing generated content to files, and committing changes to version control.
- Updated the Svelte frontend to allow users to trigger AI tasks from Kanban cards, provide prompts, select models, and view real-time status.

This milestone transforms Kanbots from a simple task manager into a nascent intelligent development assistant, capable of generating code and committing it directly into isolated work environments.

Next, in **Chapter 5: Multi-Agent Orchestration**, we'll build upon this foundation to enable multiple AI agents to work together on a single card, orchestrating more complex, persona-based development workflows.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- Tauri Official Documentation
- Svelte Official Documentation
- Anthropic Claude API Reference (Messages API)
- OpenAI Chat Completions API Reference
- Rust `reqwest` Crate Documentation
- Rust `dotenv` Crate Documentation
- Rust `tokio` Crate Documentation

CHAPTER 05

Orchestrating Multi-Agent Workflows with Personas

In the previous chapters, you've built a foundational Kanban board, integrated Git worktrees for isolated task contexts, and even enabled a single AI agent to perform basic tasks. This chapter marks a significant step forward: **orchestrating multiple AI agents to collaborate on a single task, each with a distinct persona.**

This milestone is critical because real-world development often involves multiple roles and handoffs. By simulating this with AI agents, we move beyond simple task automation towards a more intelligent, autonomous development assistant. By the end of this chapter, your Kanbots application will be able to initiate and manage sequential workflows, demonstrating how different AI "personalities" can contribute to a larger goal. You'll verify the workflow by observing agents making distinct, persona-aligned changes in a Git worktree, ultimately completing a small feature or refactoring task.

Project Overview

The Kanbots project aims to build a local-first, cross-platform desktop Kanban application. Its core innovation is the ability to attach AI agents to individual Kanban cards, using Git worktrees to provide isolated, version-controlled environments for agent execution. This allows for automated development workflows, where AI agents can generate, review, and refactor code directly within your local repository.

In this chapter, we specifically implement the multi-agent orchestration layer, enabling a "Developer" agent to write code and a "Reviewer" agent to critique it sequentially. This forms the basis for more complex, persona-driven automation within your development process.

Tech Stack Deep Dive

To achieve multi-agent orchestration, we'll leverage the following components:

- **Rust (Tauri Backend):** The primary orchestrator. Rust's strong type system, performance, and robust concurrency features make it ideal for managing Git worktrees, communicating with AI APIs, and coordinating agent steps. We'll extend our existing Tauri backend with new command handlers.
- **Svelte (Tauri Frontend):** Provides the interactive user interface. Svelte's reactivity and simplicity allow us to quickly build UI components for defining workflows and displaying real-time agent progress.
- **Git:** Essential for providing isolated contexts via worktrees and for tracking each agent's distinct contributions through commits.
- **AI Agent APIs (e.g., Claude Code, Codex):** While we'll use a simulated output for this chapter to focus on orchestration, the design accounts for integrating with real large language model (LLM) APIs. These APIs are the "brains" of our agents, interpreting instructions and generating code or reviews.

Milestone: Persona-Based Multi-Agent Workflow

This chapter focuses on delivering the capability for a Kanban card to execute a predefined, sequential workflow involving two distinct AI personas.

The key steps we'll implement are:

1. **Enhance Agent Structure:** Modify the Rust `Agent` struct to include an optional `persona` field.
2. **Define Workflow Steps:** Introduce a new Rust struct, `AgentWorkflowStep`, to describe individual actions within a multi-agent workflow.
3. **Implement Workflow Command:** Create a Tauri command in Rust that iterates through workflow steps, invokes agents with specific instructions and personas, and manages Git commits between steps.

4. **Integrate Frontend UI:** Update the Svelte frontend to allow users to define a simple two-step workflow, assign agents, and trigger the Rust command.
5. **Real-time Feedback:** Implement event emission from Rust to Svelte to provide live updates on agent progress.

By the end of this milestone, you will have a functional system where clicking a button in the UI triggers a sequence of automated actions by distinct AI personas, all tracked within a Git worktree.

Architecture: Orchestrating Collaborative AI

Moving from a single agent to multiple agents introduces new challenges and opportunities. We need a way to define agents, assign them roles (personas), and control the flow of execution between them.

The Concept of Personas

What are personas? In the context of AI agents, a persona is a defined role or identity that guides the agent's behavior, tone, and focus. Instead of a generic AI, an agent with a "Developer" persona will prioritize code generation, best practices, and implementation details. A "Reviewer" persona, conversely, will focus on code quality, security, performance, and adherence to standards.

Why do they exist? Personas help constrain the AI's output, making it more predictable and relevant to a specific part of a task. It's about providing clear context and expectations, much like assigning roles to human team members.

What problem do they solve? Without personas, a single agent might try to do everything, leading to unfocused or contradictory outputs. By splitting responsibilities, we leverage the AI's capabilities more effectively and enable complex workflows.

Workflow Design: Sequential Collaboration

For our first multi-agent workflow, we'll implement a sequential pattern:

1. **Developer Agent:** Generates initial code based on a task description.

2. **Reviewer Agent:** Reviews the generated code, suggests improvements, and refactors it.

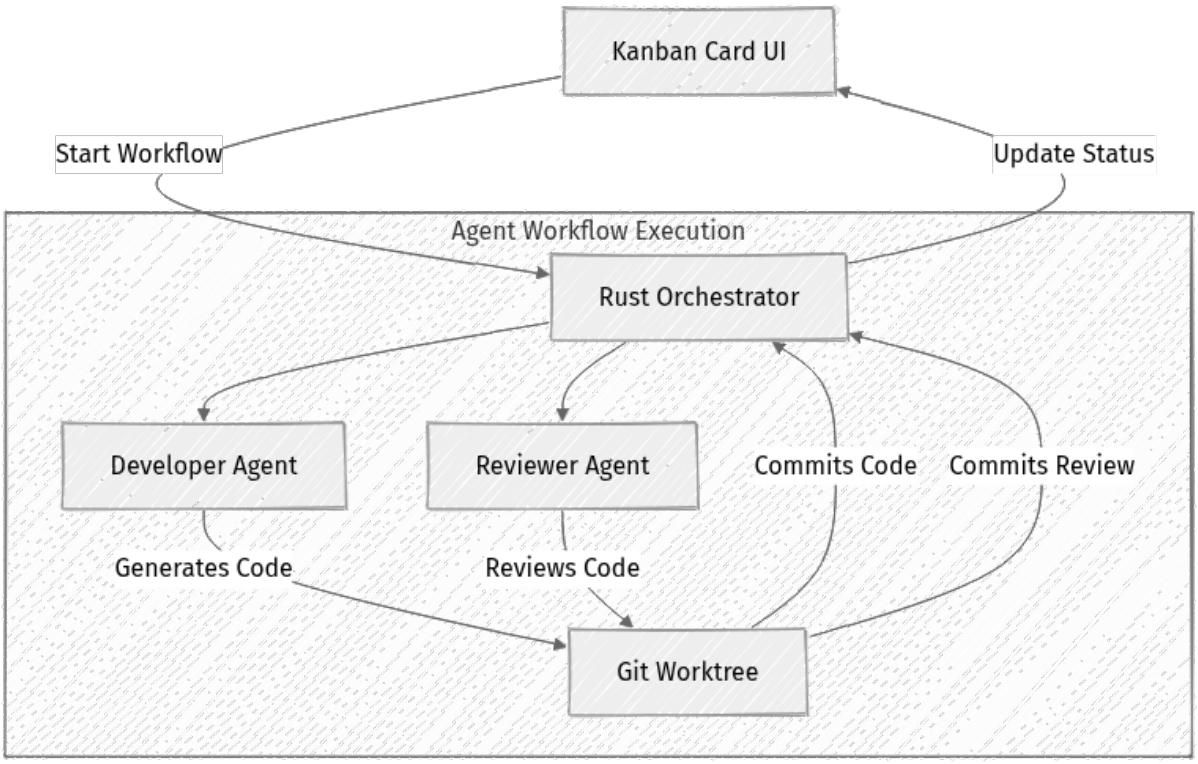
This sequential flow is simpler to manage than parallel execution initially and effectively demonstrates the core concept of agents building upon each other's work.

System Architecture Extension

To support this, we'll extend our existing architecture:

- **Agent Configuration:** Each agent instance will now include a `persona` field.
- **Workflow Definition:** A Kanban card will store a definition of a workflow, which is a sequence of agent tasks, each potentially assigned a specific persona. For this chapter, we'll hardcode a simple workflow in the frontend for demonstration.
- **Rust Orchestrator:** The Rust backend will gain new logic to:
    - Parse the workflow definition.
    - Execute agents sequentially, passing relevant context (e.g., file paths, previous agent's output) from one agent to the next.
    - Manage Git operations (staging, committing) between agent steps to track distinct contributions.
    - Update the Svelte frontend with real-time progress and outputs.
- **Svelte UI:** We'll add UI elements to define a simple workflow for a card and trigger its execution.

Here's a high-level view of the multi-agent workflow:



Data Flow

1. **UI Action:** The user clicks "Start Workflow" on a card, passing the card's ID, its associated worktree path, a list of `workflowSteps`, and the available `aiAgents` to the Rust backend.

2. **Rust Orchestration (`execute_agent_workflow`):**
    - Switches to the card's dedicated Git worktree.
    - **Loop through `workflowSteps`:**
        - For each step, it retrieves the specified `Agent` configuration.
        - It constructs a prompt for the AI, prepending the agent's persona (or an override from the workflow step) and including context (like previous agent's output).
        - Makes a simulated (or actual) call to the AI API.
        - **File I/O:** If an `output_path` is specified, the agent's output is written to that file within the worktree.
        - **Git Operations:** `git add .` and `git commit -m "..."` are executed to record the agent's contribution.
        - **Context Update:** Relevant output from the current agent is stored in a `HashMap` to be passed to the next agent in the sequence.
        - **UI Feedback:** Emits `agent_progress` events to the Svelte frontend.
3. **UI Update:** The Svelte frontend listens for `agent_progress` and `agent_workflow_completed` events, updating the `workflowStatus` display in real-time.

Step-by-Step Implementation

We'll start by modifying our Rust backend to support personas and then build the workflow execution logic.

1. Update Agent Structure (Rust Backend)

First, let's enhance our `Agent` struct in Rust to include a `persona` field. This allows us to define the role for each AI interaction.

Open `src-tauri/src/main.rs` and locate your `Agent` struct (or create one if you haven't already, based on previous chapters' single-agent integration).

```
// src-tauri/src/main.rs

// ... (other imports and structs)
```

```
#[derive(Debug, serde::Serialize, serde::Deserialize, Clone)]
pub struct Agent {
    pub id: String,
    pub name: String,
    pub api_key: String, // Production note: This should be handled securely
    pub model: String,
    pub persona: Option<String>, // New field for agent persona
    // Add other agent-specific configuration fields as needed
}

// ... (existing Tauri commands)
```

Explanation:

- We've added `persona: Option<String>`. Using `Option` allows us to define agents without a specific persona if needed, but for our workflow, it will be essential. This `persona` string will be injected directly into the AI prompt to guide its behavior.

2. Define Workflow Step and Command (Rust Backend)

Next, we need a way to define a sequence of agent actions and a new Tauri command to trigger this workflow.

Still in `src-tauri/src/main.rs`:

```
// src-tauri/src/main.rs

use std::collections::HashMap;
use std::path::PathBuf;
use tokio::process::Command; // For async process execution
use tauri::{AppHandle, Manager}; // For emitting events to the frontend
use tokio::fs; // For async file operations
use std::time::Duration; // For potential delays or timeouts (optional)

// ... (existing structs and Agent definition)

#[derive(Debug, serde::Serialize, serde::Deserialize, Clone)]
pub struct AgentWorkflowStep {
    pub agent_id: String, // ID of the agent to use for this step
    pub persona_override: Option<String>, // Optional: override agent's
default persona for this step
    pub instruction: String, // Specific instruction for this step
    pub output_path: Option<String>, // Where the agent should write its
output (e.g., "src/feature.rs")
    pub commit_message: String, // Git commit message for this step
}

/// Orchestrates a multi-agent workflow for a specific card.
#[tauri::command]
async fn execute_agent_workflow(
    app_handle: AppHandle,
    card_id: String,
    worktree_path: String,
    workflow_steps: Vec<AgentWorkflowStep>,
    ai_agents: Vec<Agent>, // Pass available agents to the command
```

```
) -> Result<String, String> {
    let card_repo_path = PathBuf::from(&worktree_path);

    // Ensure the worktree exists and switch to it
    if !card_repo_path.exists() {
        return Err(format!("Worktree path does not exist: {}", worktree_path))
;
    }
    // We assume the worktree is already initialized from previous steps
(Chapter 2).
    // For production, you'd add more robust checks/initialization here to
handle
    // cases where a worktree might be deleted or corrupted.

    // Store agent map for easy lookup
    let agent_map: HashMap<String, Agent> = ai_agents.into_iter().map(|a| (a.i
d.clone(), a)).collect();

    // Context to pass between agents. This can store file contents, previous
outputs, etc.
    let mut current_context: HashMap<String, String> = HashMap::new();

    for (i, step) in workflow_steps.iter().enumerate() {
        let agent = agent_map.get(&step.agent_id)
            .ok_or_else(|| format!("Agent with ID {} not found for step {}", s
tep.agent_id, i))?;

        let actual_persona = step.persona_override.clone().or_else(|| agent.pe
rsona.clone());
        let persona_prefix = actual_persona.as_ref().map(|p| format!("You are
a {}. ", p)).unwrap_or_default();

        // Construct the prompt for the AI, including context from previous
steps
        let mut prompt = format!("{}{}", persona_prefix, step.instruction);

        // Append relevant context from previous agent's work
        if let Some(prev_output) = current_context.get("last_output") {
            prompt = format!("{} Previous output/summary: {}", prompt, prev_ou
tput);
        }
        if let Some(prev_output_file_path) = current_context.get("last_output_
file") {
            // In a real scenario, you'd read the file content here
            // and append it to the prompt if relevant for the current agent.
            // For now, we'll just mention the file.
            prompt = format!("{} Also consider the file generated at: {}", pro
mpt, prev_output_file_path);
        }

        // ⚡ Quick Note: For real code review, you'd read the actual file
content from `prev_output_file_path`
        // and include it in the prompt for the Reviewer agent to analyze.
This simulation simplifies that.

        app_handle.emit_all("agent_progress", format!("Card {}: Agent {} ({})
started step {}: {}",
            card_id, agent.name, actual_persona.clone().unwrap_or_default(),
i + 1, step.instruction))
            .map_err(|e| e.to_string())?;

        // Simulate AI API call (replace with actual API integration for
```

```
Claude Code or Codex)
        // For production, you would use a dedicated HTTP client (e.g.,
`reqwest`) and
        // serialize/deserialize JSON for the specific AI API.
        // Remember to securely load API keys (e.g., from environment
variables or OS secret store).

        // Example placeholder for actual API call (requires `reqwest` and
`serde_json` crates):
        /*
        let client = reqwest::Client::builder()
            .timeout(Duration::from_secs(60)) // Set a reasonable timeout
            .build()
            .map_err(|e| format!("Failed to build HTTP client: {}", e))?;

        let ai_api_url = match agent.model.as_str() {
            "claude-3-opus-20240229" => "https://api.anthropic.com/v1/
messages", // Example Claude API
            "gpt-4-turbo" => "https://api.openai.com/v1/chat/completions", //
Example OpenAI API
            _ => return Err(format!("Unsupported AI model: {}", agent.model)),
        };

        let request_body = serde_json::json!({
            "model": agent.model,
            "max_tokens": 2048, // Adjust based on expected output length
            "messages": [{"role": "user", "content": prompt}]
            // Add other model-specific parameters (e.g., temperature)
        });

        let response = client.post(ai_api_url)
            .header("x-api-key", &agent.api_key) // Securely retrieved API key
            .header("anthropic-version", "2023-06-01") // For Claude
            .header("content-type", "application/json")
            .json(&request_body)
            .send()
            .await
            .map_err(|e| format!("AI API request failed: {}", e))?;

        if !response.status().is_success() {
            let error_text = response.text().await.unwrap_or_else(|_| "Unknown
API error".to_string());
            return Err(format!("AI API returned error status {}: {}",
response.status(), error_text));
        }

        let json_response: serde_json::Value = response.json().await
            .map_err(|e| format!("Failed to parse AI API response JSON: {}",
e))?;

        let ai_output = json_response["content"][0]
["text"].as_str().unwrap_or_default().to_string();
        */

        // For now, let's simulate AI output:
        let ai_output = format!("Simulated AI output for {} persona: \"{}\"",
actual_persona.clone().unwrap_or_else(|| "generic agent".to_string()), prompt)
;
        println!("AI Agent Output for step {}: {}", i, ai_output); // Log to
console for debugging

        // Write agent output to file if specified
```

```
        if let Some(output_path) = &step.output_path {
            let full_path = card_repo_path.join(output_path);
            if let Some(parent) = full_path.parent() {
                fs::create_dir_all(parent).await.map_err(|e| format!("Failed
to create directories for {}: {}", full_path.display(), e))?;
            }
            fs::write(&full_path, &ai_output).await.map_err(|e| format!("Faile
d to write agent output to {}: {}", full_path.display(), e))?;
            println!("Wrote agent output to: {}", full_path.display());
            // Add the file path to context for subsequent agents if they need
to read it
            current_context.insert("last_output_file".to_string(), full_path.t
o_string_lossy().to_string());
        }

        // Git operations: Stage and commit changes
        // 🧠 Important: In a real scenario, you'd parse AI output for code
changes
        // and apply them precisely, not just dump raw output.
        // For this example, we assume the AI writes directly to output_path.
        let mut git_command = Command::new("git");
        git_command
            .current_dir(&card_repo_path)
            .arg("add")
            .arg("."); // Add all changes in the worktree

        let output = git_command.output().await.map_err(|e| format!
("Failed to stage changes in {}: {}", card_repo_path.display(), e))?;
        if !output.status.success() {
            return Err(format!("Git add failed: {}",
String::from_utf8_lossy(&output.stderr)));
        }

        let mut git_command = Command::new("git");
        git_command
            .current_dir(&card_repo_path)
            .arg("commit")
            .arg("-m")
            .arg(&step.commit_message)
            .env("GIT_AUTHOR_NAME",
agent.name.clone()) // Set author name for the commit
            .env("GIT_AUTHOR_EMAIL", format!("{}@kanbots.ai", agent.id)); //
Set author email

        let output = git_command.output().await.map_err(|e| format!
("Failed to commit changes in {}: {}", card_repo_path.display(), e))?;
        if !output.status.success() {
            // If no changes to commit, Git will return a non-zero status but
with a specific message.
            // We should not treat this as an error.
            if !String::from_utf8_lossy(&output.stderr).contains("nothing to
commit") {
                return Err(format!("Git commit failed: {}", String::from_utf8_
lossy(&output.stderr)));
            }
        } else {
            println!("Committed changes for step {}: {}", i, step.commit_messa
ge);
        }

        // Store relevant output in context for the next agent
        current_context.insert("last_output".to_string(), ai_output.clone());
```

```
    }

    app_handle.emit_all("agent_workflow_completed", format!
("Card {}: Workflow completed successfully!", card_id))
        .map_err(|e| e.to_string())?;

    Ok("Workflow executed successfully".to_string())
}

// Ensure your main function is asynchronous and registers the new command.
// Add `tokio = { version = "1", features = ["full"] }` to your Cargo.toml
// if you haven't already, as `tokio::process::Command` and `tokio::fs`
require it.
#[tokio::main]
async fn main() {
    tauri::Builder::default()
        .invoke_handler(tauri::generate_handler![
            // ... your existing commands here (e.g., create_worktree,
delete_worktree, execute_agent_task from previous chapters)
            execute_agent_workflow, // Add the new command
        ])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Required `Cargo.toml` updates: Ensure your `src-tauri/Cargo.toml` includes `tokio` with the `full` feature:

```
# src-tauri/Cargo.toml
[dependencies]
tauri = { version = "2.0.0-beta", features = ["fs-extra", "shell-open"] } #
Version as of 2026-05-24
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1", features = ["full"] } # Required for async process
and file operations
# reqwest = { version = "0.12", features = ["json"], optional = true } #
Uncomment for actual AI API calls
# dotenv = { version = "0.15", optional = true } # For loading API keys
from .env
```

If you decide to integrate actual AI APIs, uncomment `reqwest` and `dotenv` (or use another secure secret management approach).

Explanation of Rust Code:

- **`AgentWorkflowStep`:** This struct defines a single step in our workflow. It specifies which agent to use (`agent_id`), what instruction to give it, an optional `persona_override`, where to write its output, and the `commit_message` for Git.

- **`execute_agent_workflow` command:**
  - Takes `card_id`, `worktree_path`, a `Vec<AgentWorkflowStep>`, and a list of `ai_agents` (to look up by ID).
  - It iterates through each `workflow_step`.
  - For each step, it retrieves the corresponding `Agent` configuration.
  - It constructs a prompt, prepending the agent's persona (or an override) and including context from previous steps.
  - **AI API Call (Simulated):** The commented-out block shows where you would integrate with actual AI APIs like Claude Code or Codex. For this chapter, we're using a placeholder string to focus on the orchestration logic.
  - **UI Feedback:** `app_handle.emit_all("agent_progress", ...)` sends real-time updates to the Svelte frontend.
  - **File Output:** If `output_path` is specified, the agent's simulated output is written to a file within the worktree. `tokio::fs::create_dir_all` ensures the directory structure exists before writing.
  - **Git Operations:** After each agent step, `git add .` and `git commit -m "..."` are executed. This is crucial for tracking each agent's contribution and ensuring the next agent can see the changes. We also set `GIT_AUTHOR_NAME` and `GIT_AUTHOR_EMAIL` to attribute commits to the specific AI agent.
  - **Context Passing:** `current_context` is a `HashMap` used to pass information (like `last_output`) between sequential agents. This is a simple mechanism; for complex workflows, you might need a more structured context object or a shared "scratchpad" file within the worktree.
- **`main` function setup:** The `main` function is marked with `#[tokio::main]` to enable asynchronous operations, which are necessary for `tokio::process::Command` and `tokio::fs` functions. The `execute_agent_workflow` command is registered with Tauri's `invoke_handler`.

3. Frontend Integration (Svelte)

Now, let's update our Svelte frontend to define agents, create a simple workflow, and trigger the new Rust command.

Update Svelte Types

First, update your `src/lib/types.ts` to reflect the new Rust structures.

```
// src/lib/types.ts

export interface Agent {
  id: string;
  name: string;
  apiKey: string; // Stored securely, not directly in UI state usually
  model: string;
  persona?: string; // Optional persona field
}

export interface AgentWorkflowStep {
  agent_id: string;
  persona_override?: string;
  instruction: string;
  output_path?: string;
  commit_message: string;
}

export interface Card {
  id: string;
  title: string;
  description: string;
  status: 'todo' | 'in-progress' | 'done';
  worktreePath?: string; // Path to the associated git worktree
  // Add other card properties
  workflowStatus?: string; // To display current workflow status
}
```

Update Card Detail UI (Svelte)

We'll add a section to our `CardDetail.svelte` component (or a similar component) that allows us to define a simple workflow and trigger it.

```
<!-- src/routes/CardDetail.svelte -->
<script lang="ts">
  import { getContext, onMount, onDestroy } from 'svelte'; // Added onDestroy
  import { cards, agents } from '$lib/stores'; // Assuming you have agents in
a store
  import { invoke } from '@tauri-apps/api/tauri';
  import { listen } from '@tauri-apps/api/event';
  import type { Agent, AgentWorkflowStep, Card } from '$lib/types';

  export let cardId: string; // Passed as a prop to the component

  let currentCard: Card | undefined;
  let availableAgents: Agent[] = [];
  let workflowStatus: string = '';

  // Simple workflow definition for demonstration
  let workflowSteps: AgentWorkflowStep[] = [
    {
      agent_id: '', // Will be set by user selection
      persona_override: 'Senior Software Engineer',
      instruction: 'Generate a simple Rust function `add(a: i32, b: i32) ->
```

```
i32` in `src/lib.rs` that adds two integers. Ensure it includes a basic doc
comment.',
      output_path: 'src/lib.rs',
      commit_message: 'feat: add basic add function',
    },
    {
      agent_id: '', // Will be set by user selection
      persona_override: 'Code Reviewer',
      instruction: 'Review the `add` function in `src/lib.rs`. Ensure it
follows Rust best practices, is efficient, and improve its doc comment.',
      output_path: 'src/lib.rs', // Reviewer modifies the same file
      commit_message: 'refactor: review and improve add function',
    },
  ];

  let unlistenProgress: (() => void) | undefined;
  let unlistenCompleted: (() => void) | undefined;

  onMount(async () => {
    currentCard = $cards.find(c => c.id === cardId);
    availableAgents = $agents; // Load agents from store (assuming you
populate this from local storage or API)

    // Update agent_id placeholders with first available agent for now
    if (availableAgents.length > 0) {
      workflowSteps[0].agent_id = availableAgents[0].id;
      if (availableAgents.length > 1) {
        workflowSteps[1].agent_id = availableAgents[1].id;
      } else {
        // Fallback if only one agent, use same agent for both steps
        workflowSteps[1].agent_id = availableAgents[0].id;
      }
    }

    // Listen for agent progress updates from the backend
    unlistenProgress = await listen('agent_progress', (event) => {
      workflowStatus = (event.payload as string);
      console.log('Agent Progress:', workflowStatus);
    });

    unlistenCompleted = await listen('agent_workflow_completed', (event) => {
      workflowStatus = (event.payload as string);
      console.log('Workflow Completed:', workflowStatus);
      if (currentCard) {
        currentCard.workflowStatus = workflowStatus;
        $cards = $cards; // Trigger Svelte reactivity
      }
    });
  });

  onDestroy(() => {
    if (unlistenProgress) unlistenProgress();
    if (unlistenCompleted) unlistenCompleted();
  });

  async function startWorkflow() {
    if (!currentCard || !currentCard.worktreePath) {
      alert('Card or worktree path not found. Please initialize a worktree
first.');
      return;
    }
    if (availableAgents.length === 0) {
```

```
        alert('No AI agents configured. Please add agents first.');
        return;
    }
    if (!workflowSteps[0].agent_id || !workflowSteps[1].agent_id) {
        alert('Please select agents for both workflow steps.');
        return;
    }

    workflowStatus = 'Starting workflow...';
    try {
      const result = await invoke('execute_agent_workflow', {
        cardId: currentCard.id,
        worktreePath: currentCard.worktreePath,
        workflowSteps: workflowSteps,
        aiAgents: availableAgents, // Pass all agents for lookup by the Rust
backend
      });
      console.log('Workflow result:', result);
      workflowStatus = 'Workflow initiated successfully. Check console/
terminal for details.';
      if (currentCard) {
        currentCard.workflowStatus = 'Workflow in progress...';
        $cards = $cards;
      }
    } catch (error) {
      console.error('Failed to execute workflow:', error);
      workflowStatus = `Workflow failed: ${error}`;
      if (currentCard) {
        currentCard.workflowStatus = `Workflow failed: ${error}`;
        $cards = $cards;
      }
      alert(`Workflow execution failed: ${error}`);
    }
  }
</script>

{#if currentCard}
  <h2>{currentCard.title}</h2>
  <p>{currentCard.description}</p>
  <p>Worktree: <code>{currentCard.worktreePath || 'N/A'}</code></p>

  <h3>Agent Workflow</h3>
  <p>Current Status: <strong>{workflowStatus || currentCard.workflowStatus ||
'Idle'}</strong></p>

  {#each workflowSteps as step, i}
    <div style="border: 1px solid #eee; padding: 10px; margin-bottom: 10px;
border-radius: 5px;">
      <h4>Step {i + 1}: {step.persona_override || 'Generic Agent'}</h4>
      <p>Instruction: {step.instruction}</p>
      <p>Output Path: <code>{step.output_path || 'N/A'}</code></p>
      <p>Commit Message: <code>{step.commit_message}</code></p>
      <div>
        <label for="agentSelect-{i}">Select Agent:</label>
        <select id="agentSelect-{i}" bind:value={step.agent_id}>
          {#each availableAgents as agent (agent.id)}
            <option value={agent.id}>{agent.name} ({agent.model})</option>
          {/each}
        </select>
      </div>
    </div>
  {/each}
```

```
  <button on:click={startWorkflow}>Start Multi-Agent Workflow</button>

{:else}
  <p>Loading card details...</p>
{/if}

<style>
  /* Basic styling */
  h2, h3, h4 { color: #333; }
  button {
    background-color: #007bff;
    color: white;
    padding: 10px 15px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 1em;
    margin-top: 20px;
  }
  button:hover {
    background-color: #0056b3;
  }
  select {
    padding: 8px;
    border-radius: 4px;
    border: 1px solid #ddd;
    margin-left: 10px;
    min-width: 200px;
  }
  code {
      background-color: #f4f4f4;
      padding: 2px 4px;
      border-radius: 3px;
      font-family: monospace;
  }
</style>
```

Explanation of Svelte Code:

- **`workflowSteps`:** This array defines our fixed two-step workflow (Developer then Reviewer) with their respective instructions and persona overrides. In a more advanced application, this would be dynamic and configurable by the user via a form.

- **`onMount` and `onDestroy`:**
  - `onMount` loads `currentCard` and `availableAgents` from Svelte stores.
  - It initializes `workflowSteps` with the IDs of the first two available agents as a default.
  - Crucially, it sets up event listeners for `agent_progress` and `agent_workflow_completed` emitted by the Rust backend. This allows the UI to update in real-time.
  - `onDestroy` ensures that these event listeners are cleaned up when the component is unmounted, preventing memory leaks.
- **`startWorkflow` function:**
  - Performs basic validation (card and worktree exist, agents are selected).
  - Invokes the new `execute_agent_workflow` Tauri command.
  - Passes the `cardId`, `worktreePath`, the `workflowSteps` definition, and `availableAgents` (for the backend to look up agent details by ID).
  - Handles success and error states, updating `workflowStatus` on the UI.
- **UI Elements:** Displays the defined workflow steps, allows selecting agents for each step via a dropdown, and shows the real-time `workflowStatus`.

Testing & Verification

With the code in place, let's verify our multi-agent workflow. This is a crucial step to ensure the orchestration logic is sound and agents are making the expected changes.

1. **Start Kanbots:**
   - Launch your Tauri application using `cargo tauri dev` in the `src-tauri` directory.
   - Launch your Svelte frontend using `npm run dev` (or `yarn dev`) in your Svelte project root.

2. **Ensure a Card and Worktree Exist:**
    - In the Kanbots UI, create a new Kanban card (e.g., "Implement User Authentication").
    - Navigate to its detail view.
    - Use the functionality from a previous chapter (Milestone 2) to "Initialize Worktree" for this card. This will create the dedicated Git repository in your specified base path.
3. **Configure AI Agents:**
    - Ensure you have at least two `Agent` configurations stored in your application (or mock them in your Svelte `agents` store for testing). These agents should ideally have distinct names (e.g., "CodeDev", "CodeReview").
    - In a real setup, you'd provide valid API keys for models like `claude-3-opus-20240229` or `gpt-4-turbo`. For this chapter, our Rust backend simulates the output, so the `apiKey` can be a placeholder.
4. **Trigger the Workflow:**
    - Navigate back to the detail view of the card with the initialized worktree.
    - Observe the "Agent Workflow" section. You should see the two predefined steps ("Senior Software Engineer" and "Code Reviewer").
    - Use the dropdowns to select your configured agents for each step. If you only have one agent, select the same agent for both steps.
    - Click the "Start Multi-Agent Workflow" button.
5. **Observe UI Updates and Terminal Output:**
    - The `workflowStatus` on the UI should update, showing messages about agent progress as emitted by the Rust backend.
    - Check your terminal where you launched Tauri (`cargo tauri dev`) for the `println!` output from the Rust backend, indicating agent activity, simulated AI outputs, and file writes. You should see messages like "Agent ... started step ...", "Wrote agent output to: ...", and "Committed changes for step ...".

6. **Inspect the Git Worktree:**
    - After the workflow completes, open a terminal or file explorer and navigate to the `worktreePath` for your card (as displayed in the UI).
    - Inside this directory, run `git log --oneline --graph`. You should see at least two new commits:
        - One from the "CodeDev" or "Senior Software Engineer" persona (e.g., "feat: add basic add function").
        - One from the "CodeReview" or "Code Reviewer" persona (e.g., "refactor: review and improve add function").
    - Inspect the `src/lib.rs` file within the worktree. You should see the initial `add` function generated by the Developer, potentially with doc comments or improvements added by the Reviewer (even if simulated, the content should reflect the distinct steps).

Expected Outcome: You should see the UI updating with progress, and a `git log` in the worktree directory will clearly show distinct commits made by the "Developer" and "Reviewer" agents, demonstrating their sequential collaboration on the `src/lib.rs` file.

Production Considerations

Implementing multi-agent systems introduces several production-level concerns beyond basic functionality:

- **State Management & Persistence:** What happens if the application crashes mid-workflow? For critical tasks, you need to persist the workflow's current step, agent outputs, and context to disk. This allows for resuming interrupted workflows. This could involve storing workflow state in a local SQLite database or a JSON file.
- **Concurrency & Locking:** If multiple cards (and thus multiple worktrees) try to run workflows concurrently, ensure that Git operations on different worktrees don't conflict, and that AI API rate limits are respected. This might require a global queue or semaphore in the Rust backend for AI calls and Git operations to prevent resource exhaustion or rate-limit errors.
- **Cost Management:** AI API calls incur costs. Multi-agent workflows can quickly multiply these costs. Implement proper logging of API usage, and consider adding budget limits, user confirmations for complex workflows, or even a dry-run mode that simulates API calls without actual charges.

- **Robust Error Handling & Recovery:** Each step in a multi-agent workflow is a potential point of failure (AI hallucination, API error, Git command failure, file system error). Your orchestration logic needs to:
    - Catch errors gracefully, providing specific error messages.
    - Provide clear feedback to the user on which agent and which step failed.
    - Offer options like retry the current step, skip the current step, or rollback the worktree to a previous state.
- **Agent Communication & Context:** For more complex workflows, passing context as a simple `HashMap<String, String>` might become brittle. Consider a structured context object or a shared "scratchpad" within the worktree (e.g., a `workflow_context.json` file) that agents can both read and write to, allowing for richer, persistent context sharing.
- **Security:** AI API keys are paramount. Ensure they are never hardcoded and are securely managed (e.g., via OS-level secret management, environment variables, or encrypted local storage). Each agent should ideally only have access to the minimum necessary resources.

Common Issues & Solutions

1. **Agents Losing Context / Unfocused Output:**
    - **Issue:** An agent in a later step doesn't seem to understand what the previous agent did or the overall goal, leading to irrelevant or contradictory output.
    - **Solution:**
        - **Prompt Engineering:** Refine agent personas and instructions to be very specific and constrain output. Clearly state the overall task and the agent's current sub-task.
        - **Context Passing:** Ensure `current_context` accurately reflects all necessary information. For code-related tasks, it's often better to pass the actual content of relevant files (read from the worktree) rather than just summaries or file paths.
        - **Iterative Prompting:** If a single prompt is too complex, break it down. For example, a reviewer might first be asked to list issues, then asked to fix them based on that list.
2. **Conflicting Changes in Worktrees:**
    - **Issue:** Git operations fail because the worktree is in an unexpected state (e.g., uncommitted changes from a user, merge conflicts from a previous failed run).
    - **Solution:**
        - **Pre-workflow Check:** Before running a workflow, check the worktree for uncommitted changes using `git status`. Prompt the user to commit or stash them.
        - **Atomic Commits:** Ensure each agent step performs a clear `git add .` and `git commit -m "..."` to isolate its changes. This makes it easier to track and potentially revert individual agent contributions.
        - **Error Handling for Git:** Catch specific Git error messages (e.g., "nothing to commit") and handle them gracefully, differentiating them from actual failures.

3. **AI Hallucinations or Unexpected Outputs:**
    - **Issue:** An agent generates nonsensical code, irrelevant text, or gets stuck in a repetitive loop.
    - **Solution:**
        - **Strong Prompting:** The `persona` field and `instruction` are your primary tools. Be explicit about expected output format (e.g., "only output Rust code, no prose").
        - **Output Validation:** Implement backend logic to parse and validate AI output. For code, this could mean basic syntax checking (e.g., using a Rust linter or `rustfmt`) or checking for specific keywords. If output is invalid, retry the agent with a corrective prompt or flag it for user intervention.
        - **Timeouts:** Implement timeouts for AI API calls to prevent agents from hanging indefinitely.
        - **User Intervention:** Provide UI controls to pause a workflow, inspect agent output, and manually correct or guide the agent. This is crucial for debugging and recovering from AI failures.

Summary & Next Step

You've successfully built the foundation for multi-agent collaboration within Kanbots! You can now:

- Define AI agents with specific personas that guide their behavior.
- Orchestrate sequential workflows where agents build upon each other's work.
- Observe real-time progress and verify distinct contributions through Git commits, clearly attributing changes to specific AI personas.

This chapter was a significant leap, demonstrating how to leverage AI agents not just for individual tasks, but as a coordinated team. The ability to define personas and orchestrate their interactions opens up vast possibilities for automating complex development processes directly on your desktop.

In the next chapter, we will focus on enhancing the **UI Feedback & Control**. We'll refine the user interface to provide more detailed, real-time insights into agent activity, allow users to intervene in workflows (e.g., pause, resume, cancel), and improve the overall user experience of interacting with your AI development team.

References

- Tauri Documentation (v2): https://tauri.app/v2/guides/
- Svelte Documentation (v5): https://svelte.dev/docs
- Rust Standard Library `std::process::Command`: https://doc.rust-lang.org/std/process/struct.Command.html
- Tokio `tokio::process::Command` & `tokio::fs`: https://docs.rs/tokio/latest/tokio/process/struct.Command.html
- Git Documentation: https://git-scm.com/doc
- Anthropic Claude API Documentation (for reference): https://docs.anthropic.com/claude/reference/messages_post

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Real-time Agent Progress and User Control UI

Interacting with AI agents can often feel like giving a command to a black box. You trigger a task, wait, and eventually, an output appears. For a multi-agent system like Kanbots, this lack of transparency can lead to frustration and inefficiency. This chapter addresses that challenge by equipping our Kanbots application with real-time feedback and user controls.

By the end of this milestone, your Kanbots application will provide a dynamic interface that displays agent progress, streams logs, and allows users to pause, resume, or cancel agent tasks directly from the Kanban board. This dramatically improves the user experience, giving operators crucial insights and control over complex AI workflows.

Project Overview: Bringing Agents to Life

In previous chapters, we established the core Kanban board and the ability to associate AI agents with cards, executing tasks within isolated git worktrees. This chapter focuses on closing the feedback loop: making agent actions visible and controllable. We're moving from a fire-and-forget model to an interactive, observable system, which is critical for debugging, managing costs, and iterating on AI-driven development.

Tech Stack for Real-time Interaction

The core technologies enabling this real-time interaction are:

- **Tauri (v2):** Provides the robust Inter-Process Communication (IPC) layer between our Rust backend and Svelte frontend. Its event system is ideal for streaming updates, and its command system handles user actions.
- **Rust:** The backend language, leveraging `tokio` for asynchronous task management and `std::sync` primitives for safe concurrent state management (e.g., `Arc<Mutex>`, `AtomicBool`, `Notify`).

- **Svelte (5):** The frontend framework, designed for reactivity, which will efficiently update the UI as new agent logs and status changes arrive.

Milestone: Interactive Agent Management

This chapter aims to deliver the following functional improvements:

1. **Real-time Log Streaming:** Agent output and progress messages from the Rust backend will stream directly to the Svelte frontend, displayed within the respective Kanban card.
2. **Dynamic Agent Status:** The UI will reflect the current state of an agent (e.g., "idle", "running", "paused", "cancelled", "completed").
3. **User Control Buttons:** Implement "Run," "Pause," "Resume," and "Cancel" buttons for each agent, allowing direct user intervention.
4. **Backend Agent State Management:** The Rust backend will maintain an internal state for each running agent, enabling it to respond to control signals.

Architecture: Two-Way IPC for Control and Observation

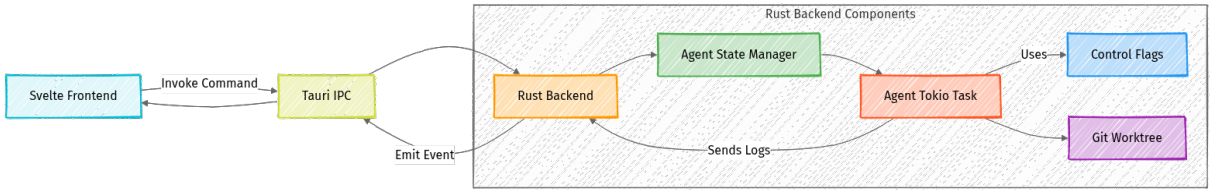
Achieving real-time interaction requires a well-defined two-way communication channel between our Rust backend and Svelte frontend. Tauri's IPC mechanisms are perfectly suited for this, allowing us to emit events from Rust to Svelte and invoke commands from Svelte to Rust.

Communication Flow

1. **Backend to Frontend (Observation):** As an AI agent executes its task in the Rust backend, it will periodically emit custom events (e.g., `agent_log`) containing progress messages, status updates, and any generated output. The Svelte frontend listens for these events and updates the UI dynamically.
2. **Frontend to Backend (Control):** The Svelte frontend exposes interactive buttons (e.g., "Pause", "Resume", "Cancel"). When a user clicks one, it invokes a corresponding Tauri command in the Rust backend. The backend then processes this command to alter the agent's execution state.

Agent State Management in Rust

The Rust backend will maintain a central `AgentStateManager` to track each active agent. This manager will hold references to the agent's background task (`tokio::task::JoinHandle`) and control primitives (`Arc<tokio::sync::Notify>` for pause/resume, `Arc<std::sync::atomic::AtomicBool>` for cancellation). When a command arrives from the frontend, the manager signals the appropriate agent task.



Step-by-Step Implementation

We'll begin by enhancing the Rust backend to emit events and manage agent states, then update the Svelte frontend to consume these events and provide the control UI.

1. Backend: Emitting Real-time Agent Logs and Progress

First, we'll modify our `run_agent_task` command to send progress updates to the frontend using Tauri's `AppHandle`.

File: `src-tauri/src/main.rs`

We need to define a struct for our event payload and then use `app_handle.emit_all` to send it.

```
// src-tauri/src/main.rs

use std::collections::HashMap;
use std::sync::{Arc, Mutex};
use tokio::task::JoinHandle;
use chrono::Utc; // For timestamps

// ... (existing imports like tauri, tauri_plugin_shell, serde)

// Define the AgentLog struct for event payload
// 🧠 Important: This struct must derive `Clone` and `serde::Serialize`
// for Tauri's event system to work correctly.
#[derive(Clone, serde::Serialize)]
struct AgentLog {
    card_id: String,
    agent_id: String,
```

```
    message: String,
    timestamp: String,
    log_type: String, // e.g., "info", "progress", "error", "status"
}

// ... (existing greet, create_worktree, switch_worktree commands)

// The `run_agent_task` command will be modified later to integrate state
management.
// For now, let's just make it emit events.
// We are temporarily commenting out the `state` parameter to focus on event
emission.
// It will be re-added in the next section.
#[tauri::command]
async fn run_agent_task_placeholder(
    app_handle: tauri::AppHandle, // Add this parameter to access Tauri's
event system
    card_id: String,
    agent_id: String,
    task_description: String,
) -> Result<String, String> {
    // 📌 Key Idea: Use AppHandle to emit events from the backend to the
frontend.
    // `emit_all` sends the event to all listening webview windows.

    // Emit a "starting" event
    app_handle.emit_all(
        "agent_log",
        &AgentLog {
            card_id: card_id.clone(),
            agent_id: agent_id.clone(),
            message: format!("[{}] Starting task: {}", agent_id, task_descript
ion),
            timestamp: Utc::now().to_rfc3339(),
            log_type: "info".to_string(),
        },
    ).map_err(|e| e.to_string())?;

    // Simulate agent work and progress
    for i in 1..=3 {
        let progress_message = format!("[{}] Working on step {}/3...", agent_i
d, i);
        app_handle.emit_all(
            "agent_log",
            &AgentLog {
                card_id: card_id.clone(),
                agent_id: agent_id.clone(),
                message: progress_message.clone(),
                timestamp: Utc::now().to_rfc3339(),
                log_type: "progress".to_string(),
            },
        ).map_err(|e| e.to_string())?;
        tokio::time::sleep(tokio::time::Duration::from_secs(2)).await; //
Simulate work
    }

    // Simulate modifying a worktree (from previous chapter)
    let worktree_path = format!("/tmp/kanbots_worktrees/{}/{}", card_id, agent
_id);
    std::fs::create_dir_all(&worktree_path)
        .map_err(|e| format!("Failed to create worktree dir: {}", e))?;
    let file_content = format!("// Generated by {} for card {}\nconsole.log('T
```

```
ask completed!');", agent_id, card_id);
    let file_path = format!("{}/output.js", worktree_path);
    std::fs::write(&file_path, file_content)
        .map_err(|e| format!("Failed to write file to worktree: {}", e))?;

    app_handle.emit_all(
        "agent_log",
        &AgentLog {
            card_id: card_id.clone(),
            agent_id: agent_id.clone(),
            message: format!("[{}] Task completed successfully. File created:
{}", agent_id, file_path),
            timestamp: Utc::now().to_rfc3339(),
            log_type: "info".to_string(),
        },
    ).map_err(|e| e.to_string())?;

    Ok(format!("Agent {} completed task for card {}", agent_id, card_id))
}

fn main() {
    tauri::Builder::default()
        .plugin(tauri_plugin_shell::init())
        .invoke_handler(tauri::generate_handler![
            greet,
            create_worktree,
            switch_worktree,
            run_agent_task_placeholder // Register the temporary command
        ])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Cargo.toml additions: Ensure `chrono` is added with the `serde` feature:```toml

src-tauri/Cargo.toml

[dependencies] chrono = { version = "0.4", features = ["serde"] }

... other dependencies

```
- `app_handle: tauri::AppHandle`: This parameter grants access to the Tauri
application instance, allowing us to emit events.
- `app_handle.emit_all("agent_log", &payload)`: This sends an event named
`"agent_log"` to *all* listening webview windows. The `AgentLog` struct is
serialized into JSON and sent as the event payload.
- `tokio::time::sleep`: Used to simulate work, making the progress updates
visible over time.
- `chrono::Utc::now().to_rfc3339()`: Provides a standardized timestamp for
each log entry.

### 2. Backend: Implementing Agent Control (Pause/Resume/Cancel)

Implementing robust pause/resume/cancel for asynchronous tasks in Rust
```

```
requires careful use of concurrency primitives. We'll introduce a global
`AgentStateManager` to track and control agents.

**File:** `src-tauri/src/main.rs`

First, add the necessary `tokio` dependency if not already present, ensuring
`full` features for `task` and `sync`.

**Cargo.toml additions:**```toml
# src-tauri/Cargo.toml
[dependencies]
tokio = { version = "1", features = ["full"] } # Use "full" for convenience in
tutorials
# ... other dependencies
```

Now, define the `AgentStateManager` and integrate it into our Tauri application.

```rust
// src-tauri/src/main.rs

// ... (existing imports)
use std::sync::atomic::{AtomicBool, Ordering}; // For atomic cancellation flag
use tokio::sync::Notify; // For pause/resume signaling

// Global state to track agent tasks and their control flags
// 🧠 Important: Mutexes are crucial for safely sharing mutable state across
threads.
// Arc allows multiple ownership of the same data.
struct AgentStateManager {
    // Stores handles to the running agent tasks, allowing us to abort them.
    tasks: Mutex<HashMap<String, JoinHandle<()>>>, // Key: card_id-agent_id
    // Notify is used for pause/resume. When `notify.notified().await` is
called,
    // it blocks until `notify.notify_one()` or `notify.notify_waiters()` is
called.
    pause_notifies: Mutex<HashMap<String, Arc<Notify>>>,
    // AtomicBool is a simple, thread-safe flag for cancellation.
    cancel_flags: Mutex<HashMap<String, Arc<AtomicBool>>>,
}

impl AgentStateManager {
    fn new() -> Self {
        AgentStateManager {
            tasks: Mutex::new(HashMap::new()),
            pause_notifies: Mutex::new(HashMap::new()),
            cancel_flags: Mutex::new(HashMap::new()),
        }
    }

    // Add a new agent task's control primitives to the manager
    fn add_task(
        &self,
        key: String,
        handle: JoinHandle<()>,
        pause_notify: Arc<Notify>,
        cancel_flag: Arc<AtomicBool>,
    ) {
        self.tasks.lock().unwrap().insert(key.clone(), handle);
```

```
        self.pause_notifies.lock().unwrap().insert(key.clone(), pause_notify);
        self.cancel_flags.lock().unwrap().insert(key, cancel_flag);
    }

    // Signal an agent task to cancel
    fn signal_cancel(&self, key: &str) -> bool {
        if let Some(flag) = self.cancel_flags.lock().unwrap().get(key) {
            flag.store(true, Ordering::SeqCst); // Set the cancellation flag
            // ⚡ Quick Note: Aborting the JoinHandle is a best-effort, non-
graceful stop.
            // The AtomicBool provides a graceful exit path for the agent task
itself.
            if let Some(handle) = self.tasks.lock().unwrap().remove(key) {
                handle.abort(); // Attempt to abort the Tokio task immediately
            }
            self.remove_task_entries(key);
            true
        } else {
            false
        }
    }

    // Signal an agent task to pause
    fn signal_pause(&self, key: &str) -> bool {
        if let Some(notify) = self.pause_notifies.lock().unwrap().get(key) {
            // ⚠️ What can go wrong: `Notify` does not inherently "pause" a
task.
            // It unblocks tasks that are `await`ing on it. To pause, the
agent task
            // itself must check a state and then `await notify.notified()`
when paused.
            // For now, we'll just log this as a signal. The agent task logic
needs to implement the actual waiting.
            println!("Signaling PAUSE for agent: {}", key);
            // We don't call notify_one() here, as that would *unpause* a
waiting task.
            // The agent task will detect the pause state and then wait on the
notify.
            true
        } else {
            false
        }
    }

    // Signal an agent task to resume
    fn signal_resume(&self, key: &str) -> bool {
        if let Some(notify) = self.pause_notifies.lock().unwrap().get(key) {
            notify.notify_one(); // Unblocks one task waiting on this notify
            println!("Signaling RESUME for agent: {}", key);
            true
        } else {
            false
        }
    }

    // Helper to remove all entries for a task after completion or
cancellation
    fn remove_task_entries(&self, key: &str) {
        self.tasks.lock().unwrap().remove(key);
        self.pause_notifies.lock().unwrap().remove(key);
        self.cancel_flags.lock().unwrap().remove(key);
    }
```

```
}

// Make AgentStateManager available globally via Tauri's managed state
// ⚡ Real-world insight: Tauri's `manage` allows you to inject shared,
immutable
// references to state into your command functions.
#[derive(Default)]
struct AppState {
    agent_manager: AgentStateManager,
}

// Update main function to register our managed state
fn main() {
    tauri::Builder::default()
        .plugin(tauri_plugin_shell::init())
        .manage(AppState::default()) // Register our state manager here
        .invoke_handler(tauri::generate_handler![
            greet,
            create_worktree,
            switch_worktree,
            run_agent_task, // This will be our new, updated command
            pause_agent,    // New command for pausing
            resume_agent,   // New command for resuming
            cancel_agent    // New command for canceling
        ])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}

// New Tauri commands for control
#[tauri::command]
async fn pause_agent(
    card_id: String,
    agent_id: String,
    state: tauri::State<'_, AppState>, // Access managed state
) -> Result<(), String> {
    let key = format!("{}-{}", card_id, agent_id);
    if state.agent_manager.signal_pause(&key) {
        Ok(())
    } else {
        Err(format!("Agent {} not found or not running.", key))
    }
}

#[tauri::command]
async fn resume_agent(
    card_id: String,
    agent_id: String,
    state: tauri::State<'_, AppState>,
) -> Result<(), String> {
    let key = format!("{}-{}", card_id, agent_id);
    if state.agent_manager.signal_resume(&key) {
        Ok(())
    } else {
        Err(format!("Agent {} not found or not running.", key))
    }
}

#[tauri::command]
async fn cancel_agent(
    card_id: String,
    agent_id: String,
```

```
    state: tauri::State<'_, AppState>,
) -> Result<(), String> {
    let key = format!("{}-{}", card_id, agent_id);
    if state.agent_manager.signal_cancel(&key) {
        Ok(())
    } else {
        Err(format!("Agent {} not found or not running.", key))
    }
}

// Modify run_agent_task to register its task and check for cancellation/pause
// This replaces the `run_agent_task_placeholder` from the previous step.
#[tauri::command]
async fn run_agent_task(
    app_handle: tauri::AppHandle,
    state: tauri::State<'_, AppState>, // Access managed state
    card_id: String,
    agent_id: String,
    task_description: String,
) -> Result<String, String> {
    let task_key = format!("{}-{}", card_id, agent_id);
    let cancel_flag = Arc::new(AtomicBool::new(false));
    let pause_notify = Arc::new(Notify::new());

    // Clone Arc for the spawned task and for the manager
    let cancel_flag_for_task = cancel_flag.clone();
    let pause_notify_for_task = pause_notify.clone();
    let app_handle_for_task = app_handle.clone();
    let card_id_for_task = card_id.clone();
    let agent_id_for_task = agent_id.clone();
    let state_for_task = state.clone(); // Clone the state for task cleanup

    // Spawn the agent logic into a separate Tokio task
    let handle = tokio::spawn(async move {
        // Initial status update
        app_handle_for_task.emit_all(
            "agent_log",
            &AgentLog {
                card_id: card_id_for_task.clone(),
                agent_id: agent_id_for_task.clone(),
                message: format!("[{}] Starting task: {}", agent_id_for_task,
task_description),
                timestamp: Utc::now().to_rfc3339(),
                log_type: "status_running".to_string(), // Specific status
type
            },
        ).unwrap_or_else(|e| eprintln!("Error emitting log: {}", e));


        for i in 1..=5 { // More steps to better demonstrate pause/cancel
            // Check for cancellation signal
            if cancel_flag_for_task.load(Ordering::SeqCst) {
                app_handle_for_task.emit_all(
                    "agent_log",
                    &AgentLog {
                        card_id: card_id_for_task.clone(),
                        agent_id: agent_id_for_task.clone(),
                        message: format!("[{}] Task cancelled by user.", agent
_id_for_task),
                        timestamp: Utc::now().to_rfc3339(),
                        log_type: "status_cancelled".to_string(),
                    },
```

```
                ).unwrap_or_else(|e| eprintln!("Error emitting log: {}", e));
                state_for_task.agent_manager.remove_task_entries(&task_key); /
/ Clean up
                return; // Exit the task gracefully
            }

            // ⚠️ What can go wrong: Implementing true pause.
            // To truly pause, the agent task needs to `await
pause_notify_for_task.notified().await;`
            // when a pause signal is received. For this demonstration, we'll
simulate a pause point
            // and emit a log, but not actually block the task.
            // A full implementation would involve a shared `AtomicBool` for
`is_paused` and the `Notify`
            // to release the task from its waiting state.
            if i == 3 {
                 app_handle_for_task.emit_all(
                    "agent_log",
                    &AgentLog {
                        card_id: card_id_for_task.clone(),
                        agent_id: agent_id_for_task.clone(),
                        message: format!("[{}] Simulating potential pause
point.", agent_id_for_task),
                        timestamp: Utc::now().to_rfc3339(),
                        log_type: "info".to_string(),
                    },
                ).unwrap_or_else(|e| eprintln!("Error emitting log: {}", e));
                // In a production system, a pause would look like:
                // if is_paused_flag.load(Ordering::SeqCst) {
                //     app_handle_for_task.emit_all("agent_log", ...
"status_paused")...;
                //     pause_notify_for_task.notified().await; // Blocks until
resumed
                //     app_handle_for_task.emit_all("agent_log", ...
"status_running")...;
                // }
            }

            let progress_message = format!("[{}] Working on step {}/5...", age
nt_id_for_task, i);
            app_handle_for_task.emit_all(
                "agent_log",
                &AgentLog {
                    card_id: card_id_for_task.clone(),
                    agent_id: agent_id_for_task.clone(),
                    message: progress_message.clone(),
                    timestamp: Utc::now().to_rfc3339(),
                    log_type: "progress".to_string(),
                },
            ).unwrap_or_else(|e| eprintln!("Error emitting log: {}", e));
            tokio::time::sleep(tokio::time::Duration::from_secs(1)).await; //
Simulate work
        }

        // Simulate modifying a worktree
        let worktree_path = format!("/tmp/kanbots_worktrees/{}/{}", card_id_fo
r_task, agent_id_for_task);
        std::fs::create_dir_all(&worktree_path)
            .unwrap_or_else(|e| eprintln!("Failed to create worktree dir:
{}", e));
        let file_content = format!("// Generated by {} for card {}\nconsole.lo
g('Task completed!');", agent_id_for_task, card_id_for_task);
```

```
        let file_path = format!("{}/output.js", worktree_path);
        std::fs::write(&file_path, file_content)
            .unwrap_or_else(|e| eprintln!("Failed to write file to worktree:
{}", e));

        app_handle_for_task.emit_all(
            "agent_log",
            &AgentLog {
                card_id: card_id_for_task.clone(),
                agent_id: agent_id_for_task.clone(),
                message: format!("[{}] Task completed successfully. File
created: {}", agent_id_for_task, file_path),
                timestamp: Utc::now().to_rfc3339(),
                log_type: "status_completed".to_string(),
            },
        ).unwrap_or_else(|e| eprintln!("Error emitting log: {}", e));

        state_for_task.agent_manager.remove_task_entries(&task_key); // Clean
up on completion
    });

    state.agent_manager.add_task(task_key, handle, pause_notify, cancel_flag);

    Ok(format!("Agent {} task started for card {}", agent_id, card_id))
}
```

- **`AgentStateManager`:** This struct centralizes the management of running agent tasks. It uses `Arc<Mutex<HashMap<...>>>` to safely store `JoinHandle`s, `Notify` objects, and `AtomicBool` flags.
    - `Arc<T>`: Enables multiple owners of the same data, crucial for sharing state between the `AgentStateManager` and the spawned `tokio::task`.
    - `Mutex<T>`: Provides exclusive access to the `HashMap`s, preventing data races when multiple threads try to modify them.
    - `JoinHandle<()>`: A handle to a spawned Tokio task. Calling `.abort()` on it attempts to cancel the task.
    - `AtomicBool`: A thread-safe boolean. We use it as a flag for graceful cancellation. The agent task periodically checks this flag.
    - `tokio::sync::Notify`: A synchronization primitive for signaling. `notify.notified().await` will block until `notify.notify_one()` is called. This is the foundation for pause/resume.
- **`tauri::State<'_, AppState>`:** This allows our Tauri commands to access the shared `AppState` instance, which holds our `AgentStateManager`.

- **`tokio::spawn`:** Each `run_agent_task` now spawns the actual agent logic into a separate Tokio task. This ensures the Tauri command returns immediately, keeping the UI responsive, while the agent runs in the background.
- **Cancellation Logic:** The agent task periodically checks `cancel_flag_for_task.load(Ordering::SeqCst)`. If true, it performs cleanup and exits. The `signal_cancel` function also calls `handle.abort()` as a forceful backup.
- **Pause/Resume Caveat:** The current `signal_pause` only logs the intent. For a true pause, the agent task itself would need to enter a waiting state (`await pause_notify_for_task.notified().await;`) when a `pause_agent` command is issued. The `signal_resume` command would then call `notify.notify_one()` to unblock it. This requires a more complex state machine within the agent task, which is beyond the scope of this UI-focused chapter but important for production.

3. Frontend: Displaying Real-time Logs and Control Buttons

Now, let's update our Svelte frontend component to listen for these events and provide the UI elements.

File: `src/lib/components/KanbanCard.svelte`

```
<!-- src/lib/components/KanbanCard.svelte -->
<script lang="ts">
  import { createEventDispatcher, onMount, onDestroy } from 'svelte';
  import { invoke } from '@tauri-apps/api/core';
  import { listen } from '@tauri-apps/api/event';
  import type { Agent, KanbanCard as CardType } from '../types'; // Alias
KanbanCard to avoid conflict
  import { cardStore } from '../stores'; // Assuming you have a store for
cards

  export let card: CardType; // Use the aliased type
  export let columnId: string;

  const dispatch = createEventDispatcher();

  // Stores logs per agent
  let agentLogs: { [agentId: string]: { message: string; timestamp: string;
log_type: string }[] } = {};
  // Stores current status per agent
  let agentStatuses: { [agentId: string]: 'idle' | 'running' | 'paused' |
'cancelled' | 'completed' | 'error' } = {};

  // Function to scroll logs to the bottom
  function scrollToBottom(agentId: string) {
    const logContainer = document.querySelector(`.log-output-${agentId}`);
    if (logContainer) {
      logContainer.scrollTop = logContainer.scrollHeight;
```

```
    }
  }

  onMount(async () => {
    // Listen for agent log events
    const unlisten = await listen<{ card_id: string; agent_id: string;
message: string; timestamp: string; log_type: string }>(
      'agent_log',
      (event) => {
        const { card_id, agent_id, message, timestamp, log_type } =
event.payload;
        if (card_id === card.id) {
          if (!agentLogs[agent_id]) {
            agentLogs[agent_id] = [];
          }
          agentLogs[agent_id] = [...agentLogs[agent_id], { message, timestamp,
log_type }];

          // Update status based on specific log types
          if (log_type === 'status_running') {
            agentStatuses[agent_id] = 'running';
          } else if (log_type === 'status_cancelled') {
            agentStatuses[agent_id] = 'cancelled';
          } else if (log_type === 'status_completed') {
            agentStatuses[agent_id] = 'completed';
          } else if (log_type === 'error') {
             // If an error log comes in, and not already cancelled/completed,
set to error
            if (!['cancelled', 'completed'].includes(agentStatuses[agent_id]))
{
                agentStatuses[agent_id] = 'error';
            }
          }
          // The pause/resume status will be updated on command invocation,
not log type

          agentLogs = agentLogs; // Trigger Svelte reactivity
          agentStatuses = agentStatuses; // Trigger Svelte reactivity

          // Scroll to bottom after updates
          setTimeout(() => scrollToBottom(agent_id), 0);
        }
      }
    );

    onDestroy(() => {
      unlisten();
    });
  });

  async function runAgent(agentId: string, taskDescription: string) {
    try {
      agentStatuses[agentId] = 'running';
      agentLogs[agentId] = []; // Clear previous logs for a new run
      agentLogs = agentLogs;
      agentStatuses = agentStatuses;
      await invoke('run_agent_task', { cardId: card.id, agentId,
taskDescription });
    } catch (error) {
      console.error('Error running agent:', error);
      agentLogs[agentId] = [...(agentLogs[agentId] || []), { message: `Error:
${error}`, timestamp: new Date().toISOString(), log_type: 'error' }];
```

```
      agentStatuses[agentId] = 'error';
      agentLogs = agentLogs;
      agentStatuses = agentStatuses;
    }
  }

  async function pauseAgent(agentId: string) {
    try {
      await invoke('pause_agent', { cardId: card.id, agentId });
      agentStatuses[agentId] = 'paused'; // Optimistic UI update
      agentLogs[agentId] = [...(agentLogs[agentId] || []), { message: `[$
{agentId}] Pause signal sent.`, timestamp: new Date().toISOString(), log_type:
'info' }];
      agentStatuses = agentStatuses;
      agentLogs = agentLogs;
    } catch (error) {
      console.error('Error pausing agent:', error);
      agentLogs[agentId] = [...(agentLogs[agentId] || []), { message: `Error
pausing: ${error}`, timestamp: new Date().toISOString(), log_type: 'error' }];
      agentLogs = agentLogs;
    }
  }

  async function resumeAgent(agentId: string) {
    try {
      await invoke('resume_agent', { cardId: card.id, agentId });
      agentStatuses[agentId] = 'running'; // Optimistic UI update
      agentLogs[agentId] = [...(agentLogs[agentId] || []), { message: `[$
{agentId}] Resume signal sent.`, timestamp: new Date().toISOString(),
log_type: 'info' }];
      agentStatuses = agentStatuses;
      agentLogs = agentLogs;
    } catch (error) {
      console.error('Error resuming agent:', error);
      agentLogs[agentId] = [...(agentLogs[agentId] || []), { message: `Error
resuming: ${error}`, timestamp: new Date().toISOString(), log_type:
'error' }];
      agentLogs = agentLogs;
    }
  }

  async function cancelAgent(agentId: string) {
    try {
      await invoke('cancel_agent', { cardId: card.id, agentId });
      agentStatuses[agentId] = 'cancelled'; // Optimistic UI update
      agentLogs[agentId] = [...(agentLogs[agentId] || []), { message: `[$
{agentId}] Cancel signal sent.`, timestamp: new Date().toISOString(),
log_type: 'error' }];
      agentStatuses = agentStatuses;
      agentLogs = agentLogs;
    } catch (error) {
      console.error('Error canceling agent:', error);
      agentLogs[agentId] = [...(agentLogs[agentId] || []), { message: `Error
canceling: ${error}`, timestamp: new Date().toISOString(), log_type:
'error' }];
      agentLogs = agentLogs;
    }
  }

  // ... (existing drag/drop handlers, if any)
</script>
```

```
<div class="kanban-card" draggable="true" on:dragstart on:dragend>
  <h3>{card.title}</h3>
  <p>{card.description}</p>

  {#each card.agents as agent (agent.id)}
    <div class="agent-section">
      <h4>Agent: {agent.name} ({agent.persona})</h4>
      <p>Status: <span class="agent-status status-{agentStatuses[agent.id] ||
'idle'}">{agentStatuses[agent.id] || 'idle'}</span></p>
      <div class="agent-controls">
        {#if agentStatuses[agent.id] === 'running'}
          <button on:click={() => pauseAgent(agent.id)}>Pause</button>
          <button on:click={() => cancelAgent(agent.id)}>Cancel</button>
        {:else if agentStatuses[agent.id] === 'paused'}
          <button on:click={() => resumeAgent(agent.id)}>Resume</button>
          <button on:click={() => cancelAgent(agent.id)}>Cancel</button>
        {:else if ['idle', 'completed', 'cancelled',
'error'].includes(agentStatuses[agent.id]) || !agentStatuses[agent.id]}
          <button on:click={() => runAgent(agent.id, `Develop feature for card
${card.id}`)}>Run Agent</button>
        {/if}
      </div>
      <div class="agent-logs">
        <h5>Logs:</h5>
        <div class="log-output log-output-{agent.id}">
          {#each agentLogs[agent.id] || [] as log}
            <p class="log-entry log-{log.log_type}">
              [{new Date(log.timestamp).toLocaleTimeString()}] {log.message}
            </p>
          {/each}
        </div>
      </div>
    </div>
  {/each}

  <!-- ... (existing card content) -->
</div>

<style>
  /* Add basic styling for the new elements */
  .kanban-card {
    background-color: white;
    padding: 15px;
    border-radius: 8px;
    box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
    margin-bottom: 10px;
    cursor: grab;
  }
  .agent-section {
    margin-top: 15px;
    padding-top: 10px;
    border-top: 1px solid #eee;
  }
  .agent-controls button {
    margin-right: 5px;
    padding: 5px 10px;
    border-radius: 4px;
    border: 1px solid #ccc;
    background-color: #f0f0f0;
    cursor: pointer;
  }
  .agent-controls button:hover {
```

```
    background-color: #e0e0e0;
  }
  .agent-logs {
    margin-top: 10px;
    max-height: 150px; /* Limit height for scrollability */
    overflow-y: auto;
    background-color: #f9f9f9;
    border: 1px solid #ddd;
    padding: 8px;
    border-radius: 4px;
    font-family: monospace;
    font-size: 0.85em;
  }
  .log-output p {
    margin: 0;
    line-height: 1.4;
  }
  /* Specific log type styling */
  .log-error {
    color: red;
    font-weight: bold;
  }
  .log-progress {
    color: #007bff; /* Blue for progress */
  }
  .log-status_running {
    color: green;
  }
  .log-status_completed {
    color: darkgreen;
  }
  .log-status_cancelled {
    color: orange;
    font-weight: bold;
  }

  /* Agent status badge styling */
  .agent-status {
      padding: 2px 6px;
      border-radius: 4px;
      font-weight: bold;
      font-size: 0.8em;
      text-transform: uppercase;
  }
  .status-idle { background-color: #e0e0e0; color: #555; }
  .status-running { background-color: #d4edda; color: #155724; }
  .status-paused { background-color: #fff3cd; color: #856404; }
  .status-cancelled { background-color: #f8d7da; color: #721c24; }
  .status-completed { background-color: #cce5ff; color: #004085; }
  .status-error { background-color: #f8d7da; color: #721c24; }
</style>
```

- **`onMount` and `listen`:** The `KanbanCard` component sets up an event listener for `agent_log` events from the Tauri backend when it mounts.
- **`agentLogs` and `agentStatuses`:** These Svelte reactive variables store the streaming logs and the current status for each agent on the card, keyed by `agent.id`.

- **`invoke` for Control:** Functions like `runAgent`, `pauseAgent`, `resumeAgent`, and `cancelAgent` use `invoke` to call the corresponding Rust Tauri commands. The UI updates optimistically, assuming the command will succeed.
- **Conditional Rendering:** The control buttons (`Pause`, `Resume`, `Cancel`, `Run Agent`) are conditionally rendered based on the agent's current status, providing an intuitive user experience.
- **Log Display and Scrolling:** A `div` with `overflow-y: auto` is used to display agent logs. The `scrollToBottom` function ensures new logs are always visible.
- **Styling:** Basic CSS is added to differentiate log types (e.g., errors in red) and status badges for better visual feedback.
- **`agentLogs = agentLogs;` and `agentStatuses = agentStatuses;`:** These are essential Svelte patterns to trigger reactivity when modifying an object or array directly.

4. Frontend: Type Definitions

Ensure your Svelte project has the necessary type definitions.

File: `src/lib/types.ts` (create if it doesn't exist, or update)

```
// src/lib/types.ts

export interface Agent {
  id: string;
  name: string;
  persona: string;
  // Add other agent properties as needed
}

export interface KanbanCard {
  id: string;
  title: string;
  description: string;
  columnId: string;
  agents: Agent[]; // Ensure agents array is part of the card
  // Add other card properties as needed
}

export interface KanbanColumn {
  id: string;
  title: string;
  cardIds: string[];
}
```

5. Frontend: Update Card Store with Agents

Make sure your `cardStore` includes an `agents` array for each card, with some dummy data for testing.

File: `src/lib/stores.ts`

```
// src/lib/stores.ts

import { writable } from 'svelte/store';
import type { KanbanCard, KanbanColumn } from './types';

// Initial dummy data for demonstration
const initialCards: KanbanCard[] = [
  { id: 'card-1', title: 'Implement User Auth', description: 'Setup JWT
authentication for API endpoints.', columnId: 'todo',
    agents: [
      { id: 'agent-1-dev', name: 'Claude Code Dev', persona: 'Developer' },
      { id: 'agent-1-rev', name: 'Codex Reviewer', persona: 'Code Reviewer' },
    ]
  },
  { id: 'card-2', title: 'Design Database Schema', description: 'Outline
tables and relationships for user data.', columnId: 'in-progress',
    agents: [
      { id: 'agent-2-arch', name: 'Claude Code Architect', persona: 'Architect
' },
    ]
  },
  { id: 'card-3', title: 'Refactor Legacy Module', description: 'Improve
readability and performance of old code.', columnId: 'todo',
    agents: [
      { id: 'agent-3-opt', name: 'Codex Optimizer', persona: 'Refactorer' },
    ]
  },
];

const initialColumns: KanbanColumn[] = [
  { id: 'todo', title: 'To Do', cardIds: ['card-1', 'card-3'] },
  { id: 'in-progress', title: 'In Progress', cardIds: ['card-2'] },
  { id: 'done', title: 'Done', cardIds: [] },
];

export const cardStore = writable<KanbanCard[]>(initialCards);
export const columnStore = writable<KanbanColumn[]>(initialColumns);
```

Testing & Verification

It's time to see our real-time feedback and control in action.

1. **Start the Tauri App:**

```
npm run tauri dev
```

1. **Inspect the UI:**
    - The Kanbots application should launch. Navigate to a card that has agents assigned (e.g., "Implement User Auth").
    - You should see a new "Agent" section for each agent, displaying its name, persona, and an initial "Status: idle". A "Run Agent" button should be visible, along with an empty "Logs" area.
2. **Run an Agent:**
    - Click the "Run Agent" button for one of the agents (e.g., "Claude Code Dev").
    - **Expected behavior:**
        - The agent's "Status" should immediately change to "running".
        - The "Run Agent" button should be replaced by "Pause" and "Cancel" buttons.
        - The "Logs" area should start displaying real-time messages, indicating the agent's progress (e.g., "Starting task...", "Working on step X/5...", "Simulating potential pause point...", "Task completed successfully..."). Logs should scroll automatically.
        - Finally, the status should change to "completed", and the control buttons should revert to "Run Agent".
3. **Test Pause/Resume (Conceptual):**
    - Start an agent. While it's running, click "Pause".
    - **Expected behavior:** The status should change to "paused", and a log entry like `[agent-X] Pause signal sent.` should appear.
    - Click "Resume".
    - **Expected behavior:** The status should revert to "running", and a log entry like `[agent-X] Resume signal sent.` should appear.
    - Note: As discussed, the agent task in Rust doesn't actually block on pause in this simplified example. The UI changes reflect the signal being sent. A full implementation would require the agent's internal logic to actively wait.

4. **Test Cancel:**
    - Start another agent. While it's running, click "Cancel".
    - **Expected behavior:**
        - The status should change to "cancelled", and a log entry like `[agent-X] Cancel signal sent.` should appear.
        - The logs should also show `[agent-X] Task cancelled by user.`.
        - The control buttons should revert to "Run Agent".
5. **Verify Worktree Output:**
    - After an agent completes successfully, check your `/tmp/kanbots_worktrees/` directory (or wherever your worktrees are configured). You should find the worktree directory for the card and agent (e.g., `/tmp/kanbots_worktrees/card-1/agent-1-dev/`), containing the `output.js` file generated by the agent.

If the logs appear in real-time, statuses update, and buttons respond as described, you've successfully implemented real-time feedback and control.

Production Considerations

Implementing real-time interaction and control for background processes involves several critical concerns for a production-grade application:

- **IPC Performance and Throttling:**
    - **Why it matters:** Emitting events too frequently (e.g., hundreds per second) can overwhelm the IPC channel, leading to UI lag or dropped messages.
    - **Solution:** Batch log messages, debounce frequent updates, or only emit critical status changes. Consider different event channels for high-frequency (e.g., raw agent output) vs. low-frequency (e.g., major status changes) data.

- **Agent State Consistency and Reconciliation:**
    - **Why it matters:** The UI's displayed status must accurately reflect the backend's actual state. Discrepancies lead to user confusion and incorrect actions.
    - **Solution:** Implement a dedicated "status" event from the backend for definitive state changes (e.g., `status_running`, `status_paused`, `status_completed`). The frontend should primarily rely on these explicit status events, rather than inferring state from general log messages. Implement a mechanism for the frontend to query the backend for an agent's true state on initialization or after network interruptions.
- **Graceful Pause/Resume Implementation:**
    - **Why it matters:** A truly paused agent should release computational resources and avoid unnecessary API calls. A simple signal might not stop an agent mid-computation.
    - **Solution:** The agent's task logic must be designed to be interruptible. This often involves periodically checking a `should_pause` flag or `await`ing on a `Notify` at natural break points (e.g., before making an API call, after processing a chunk of data). This is a complex engineering challenge.
- **Security of Control Signals:**
    - **Why it matters:** Malicious code injected into the frontend (though less likely in a desktop app, still a concern) shouldn't be able to arbitrarily control agents or other system processes.
    - **Solution:** Tauri's IPC is designed with security in mind, providing a strict allowlist for commands. Ensure input validation on `card_id` and `agent_id` parameters in Rust commands to prevent path traversal or other injection attacks.
- **Resource Management for Aborted Tasks:**
    - **Why it matters:** When an agent is cancelled, its associated resources (e.g., temporary files, open network connections, spawned sub-processes) should be cleaned up to prevent resource leaks.
    - **Solution:** The `cancel_task` logic in Rust should ensure proper cleanup. For child processes, ensure they are terminated. For file system resources, delete temporary worktree data.

- **Error Reporting and User Feedback:**
    - **Why it matters:** Users need clear, actionable feedback when an agent encounters an error or a control command fails.
    - **Solution:** Enhance error logs with more detail. Display user-friendly error messages in the UI, potentially with suggestions for resolution.

Common Issues & Solutions

1. **Frontend UI not updating (logs or status):**
    - **Issue:** Logs or status changes don't appear or update in the Svelte UI after an agent starts.
    - **Solution:**
        - **Rust `emit_all` Check:** In `src-tauri/src/main.rs`, verify that `app_handle.emit_all("agent_log", &payload)` is being called correctly and that the event name `"agent_log"` matches the frontend's `listen` call exactly. Check for `unwrap_or_else` blocks in Rust that might be silently swallowing errors during event emission.
        - **Svelte Reactivity:** Ensure you are reassigning the entire object/array to trigger Svelte's reactivity, e.g., `agentLogs = agentLogs;` and `agentStatuses = agentStatuses;`.
        - **Console Logs:** Add `console.log(event.payload)` inside your Svelte `listen` callback to confirm events are actually being received by the frontend.
        - **Browser Dev Tools:** Check the Network tab for any WebSocket traffic or errors if Tauri uses it for IPC (though for `listen`/`invoke`, it's typically direct IPC).

2. **Control commands (Pause/Resume/Cancel) have no effect:**
    - **Issue:** Clicking "Pause" or "Cancel" doesn't seem to affect the agent's execution.
    - **Solution:**
        - **Tauri Command Registration:** Double-check that `pause_agent`, `resume_agent`, and `cancel_agent` are correctly listed in `tauri::generate_handler!` in `src-tauri/src/main.rs`.
        - **`invoke` Call Parameters:** Verify the command name and parameters (`cardId`, `agentId`) in your Svelte `invoke` calls match the Rust command signatures exactly.
        - **Rust `AgentStateManager` Logic:** Debug the `signal_pause`, `signal_resume`, and `signal_cancel` methods within the `AgentStateManager`. Use `println!` statements to confirm they are being called and that the correct `JoinHandle`, `Notify`, or `AtomicBool` is being accessed.
        - **Agent Task Responsiveness:** For graceful control (especially pause/resume), the `tokio::spawn` task must be designed to periodically check for signals (e.g., `AtomicBool` for cancellation) or `await` on `Notify` for pausing. If the agent's internal loop is long-running and synchronous without these checks, it won't respond to external signals.

3. **Agent state inconsistencies in UI:**
    - **Issue:** The UI shows an agent as "running" but it has completed, or it shows "paused" but the backend is still running (due to the simplified pause implementation).
    - **Solution:**
        - **Explicit Status Events:** Rely less on inferring status from generic log messages. Have the backend emit explicit `agent_log` events with `log_type: "status_running"`, `"status_completed"`, `"status_cancelled"`, etc., to provide definitive state changes.
        - **Cleanup in `AgentStateManager`:** Ensure that the `AgentStateManager` correctly removes task entries (`tasks`, `pause_notifies`, `cancel_flags`) when an agent task completes or is cancelled. If not, stale entries can lead to incorrect state reporting.
        - **Race Conditions:** For shared mutable state in Rust, always use `Arc<Mutex<T>>` to prevent data races. Ensure all access to the `AgentStateManager` is properly locked.

Summary & Next Step

You've successfully enhanced Kanbots with crucial real-time feedback and control mechanisms. Users can now see live agent progress, review logs, and intervene in agent execution by pausing, resuming, or canceling tasks. This significantly improves the usability and debugging experience for orchestrating complex AI workflows.

What's ready now:

- The Rust backend can emit real-time log and progress events to the frontend.
- The Rust backend can receive and process user control commands (pause, resume, cancel) via Tauri IPC.
- The Svelte frontend dynamically displays agent status and streams logs.
- Users can control agent execution via interactive UI buttons.

In the next chapter, we will consolidate these features by implementing robust error handling and comprehensive logging. We'll focus on how to manage and report issues that arise from AI API calls, git operations, or agent execution, making our Kanbots application more resilient and easier to debug in a production environment.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- Tauri v2 Inter-Process Communication (IPC) Guide
- Tauri v2 State Management Guide
- Svelte 5 Tutorial
- Tokio: An asynchronous runtime for Rust
- Rust `std::sync::atomic` Documentation
- Rust `std::sync::Mutex` Documentation
- Rust `Arc` Documentation

CHAPTER 07

Securing API Keys and Robust Error Handling

In this chapter, we elevate Kanbots from a functional prototype to a more robust, production-minded application. We'll tackle two critical aspects: the secure management of sensitive AI API keys and the implementation of comprehensive error handling and logging. These elements are non-negotiable for any application that interacts with external services or handles user data, ensuring both security and a smooth user experience.

By the end of this milestone, your Kanbots application will no longer store API keys in plain text or crash silently. Instead, it will securely load credentials, gracefully handle expected and unexpected failures from AI agents or Git operations, and provide clear feedback to the user and logs for debugging. This significantly improves the application's reliability, maintainability, and trustworthiness.

Project Overview

Kanbots is a desktop Kanban application designed to orchestrate multi-agent AI workflows on individual task cards. Each card can host one or more AI agents (e.g., Claude Code, Codex) that execute tasks within isolated Git worktrees. This architecture facilitates persona-based development, allowing agents to generate, review, and refactor code, mimicking a collaborative development team. This chapter, as part of Milestone 7, focuses on hardening the application's core by addressing critical security and reliability concerns that are essential for any real-world deployment.

Tech Stack

For this chapter, we primarily work with the following technologies:

- **Rust (Backend):** Handles API key retrieval, AI agent orchestration, Git worktree management, and robust error handling.

- **Tauri:** Provides the cross-platform desktop shell and the Inter-Process Communication (IPC) layer between Rust and Svelte.
- **Svelte (Frontend):** Manages the user interface, displays agent progress, and presents error feedback.
- **AI Agent APIs (e.g., Claude Code/Codex):** The external services for which we are securing API keys.
- **Rust Crates:**
  - `std::env`: For reading environment variables.
  - `thiserror`: For defining custom, idiomatic Rust error types.
  - `serde`: For serializing/deserializing data, including errors, across the IPC boundary.
  - `log` & `env_logger`: For structured logging within the Rust backend.

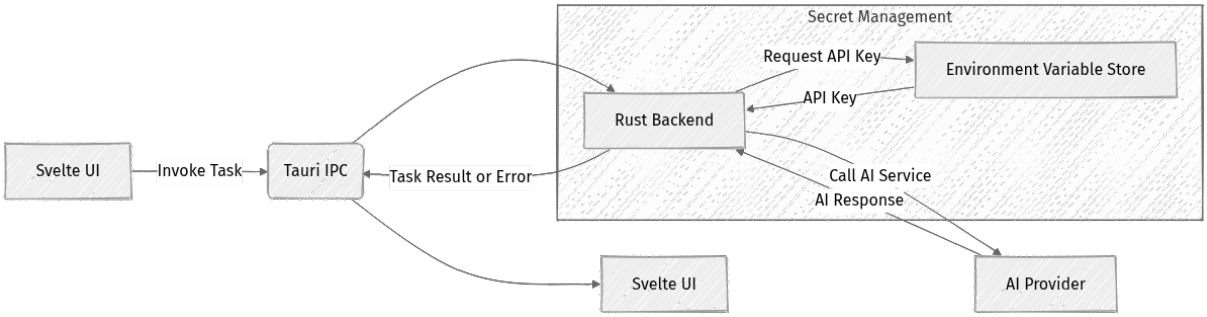
Architecture and Design

Working with external AI APIs means dealing with API keys—sensitive credentials that grant access to powerful services and often incur costs. Exposing these keys, even accidentally, can lead to security breaches, unauthorized usage, and unexpected bills. For a desktop application like Kanbots, which runs locally, we need a robust strategy for key management. Similarly, a well-defined error handling architecture ensures stability and provides clear user feedback.

API Key Management Strategy

We will use **environment variables** for securely loading API keys. This approach prevents keys from being hardcoded in the source code or committed to version control. While OS-level secret stores (like macOS Keychain or Windows Credential Manager) offer even greater security, environment variables provide a good balance of security and simplicity for a project guide and are a common first step in production systems.

The Rust backend will be solely responsible for retrieving API keys from the environment. The Svelte frontend will initiate AI tasks via Tauri IPC, and the Rust backend will then load the necessary API key before making any calls to the external AI provider.

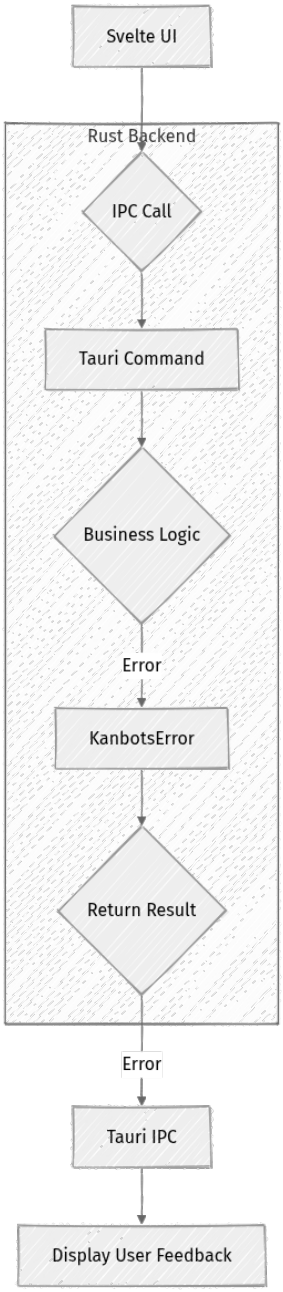


Robust Error Handling Design

An application that crashes or silently fails frustrates users and makes debugging a nightmare. Robust error handling means:

- **Anticipating failures:** AI API rate limits, network issues, invalid Git commands, agent hallucinations, or missing API keys.
- **Structured errors:** Defining custom error types that encapsulate specific failure scenarios, allowing for precise handling.
- **Graceful degradation:** Informing the user when something goes wrong without crashing, and offering actionable advice.
- **Logging:** Recording errors and application events for post-mortem analysis and operational insights.

Rust's `Result` enum (`Ok(T)` or `Err(E)`) is central to this. We'll define a custom error enum, `KanbotsError`, to represent various failure points in the application. This error type will be serializable, allowing it to be safely transmitted across the Tauri IPC boundary to the Svelte frontend.



Step-by-Step Implementation

We'll start by modifying the Rust backend to load API keys securely, then introduce a custom error type and integrate logging. Finally, we'll update the Svelte frontend to display these structured errors.

1. Update `src-tauri/Cargo.toml`

First, add the necessary dependencies for error handling and logging to your Rust project.

File: `src-tauri/Cargo.toml`

```
# src-tauri/Cargo.toml
[dependencies]
# ... existing dependencies ...
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
thiserror = "1.0" # For easy error definition
log = "0.4" # For logging API
env_logger = "0.11" # For logging implementation
```

- **`serde`**: Already likely present, but ensure `features = ["derive"]` is enabled for serializing custom error types.
- **`thiserror`**: Simplifies creating custom error types by automatically implementing `std::error::Error` and `Display`.
- **`log`**: A facade for logging, allowing you to use `info!`, `error!`, etc., without coupling directly to a specific logger implementation.
- **`env_logger`**: A simple logger that prints log messages to stderr, configurable via the `RUST_LOG` environment variable.

Checked: `thiserror = "1.0"`, `log = "0.4"`, `env_logger = "0.11"` are current stable versions as of 2026-05-24.

2. Define Custom Error Types (Rust Backend)

Create a new module for our custom error types. This centralizes error definitions and makes our code cleaner.

File: `src-tauri/src/error.rs` (Create this new file)

```
use thiserror::Error;
use serde::{Serialize, Deserialize};

/// Custom error type for Kanbots application.
/// This enum categorizes various potential failures, making error handling
more precise.
#[derive(Debug, Error, Serialize, Deserialize)]
pub enum KanbotsError {
    #[error("API Key Error: {0}")]
    ApiKey(String),
    #[error("AI Agent Error: {0}")]
    AiAgent(String),
    #[error("Git Worktree Error: {0}")]
    GitWorktree(String),
    #[error("IPC Error: {0}")]
    Ipc(String),
    #[error("Unknown Error: {0}")]
    Unknown(String),
}

// Implement From traits for easier error conversion from standard library
errors.
```

```
// This allows us to use the '?' operator with standard errors, converting
them
// automatically into our custom KanbotsError.

impl From<std::env::VarError> for KanbotsError {
    fn from(err: std::env::VarError) -> Self {
        KanbotsError::ApiKey(format!("Environment variable access error: {}",
err))
    }
}

// ⚡ Quick Note: In a real project, you would add `From` implementations for
// other external library error types you interact with, e.g., HTTP client
errors,
// Git library errors (like `git2::Error`), or specific AI client SDK errors.
// This ensures all errors are unified under `KanbotsError`.
```

Explanation:

- `#[derive(Debug, Error, Serialize, Deserialize)]`:
  - `Debug`: Allows us to print the error for debugging.
  - `Error`: From `thiserror`, automatically implements `std::error::Error` and `Display`, making our error type compatible with Rust's error ecosystem.
  - `Serialize`, `Deserialize`: From `serde`, essential for converting our Rust error enum into JSON for transmission over Tauri's IPC to the Svelte frontend.
- **Error Variants**: `ApiKey`, `AiAgent`, `GitWorktree`, `Ipc`, `Unknown` provide distinct categories for different failure modes. The `#[error("...")]` attribute defines the display message for each variant.
- `From<std::env::VarError> for KanbotsError`: This `impl` block allows any `std::env::VarError` (e.g., when an environment variable is not found) to be automatically converted into a `KanbotsError::ApiKey` variant. This is incredibly useful with the `?` operator.

3. Implement Secure API Key Loading and Logging (Rust Backend)

Now, modify `src-tauri/src/main.rs` to use our custom `KanbotsError` and integrate logging.

File: `src-tauri/src/main.rs`

```
// ... existing imports ...
use std::env;
use log::{info, error, warn, debug}; // Import logging macros
mod error; // Import our custom error module
```

```
use error::KanbotsError; // Use our KanbotsError type

// Define constants for environment variable names
const CLAUDE_API_KEY_ENV: &str = "CLAUDE_API_KEY";
const CODEX_API_KEY_ENV: &str = "CODEX_API_KEY";

/// Helper function to get API key from environment, returning KanbotsError on
failure.
fn get_api_key(env_var_name: &str) -> Result<String, KanbotsError> {
    info!("Attempting to load API key from environment: {}", env_var_name);
    env::var(env_var_name)
        .map_err(|e| {
            // Log the detailed error internally, but provide a user-friendly
message for the UI
            error!("Failed to load API key {}: {}", env_var_name, e);
            KanbotsError::ApiKey(format!
("Environment variable '{}' not set or invalid.", env_var_name))
        })
}

/// Example command to use the API key and simulate AI agent tasks.
/// Now returns our custom KanbotsError for robust error propagation.
#[tauri::command]
async fn run_ai_agent_task(agent_name: String, task_description: String) -> Re
sult<String, KanbotsError> {
    info!("Received request to run agent '{}' with task: '{}'", agent_name, ta
sk_description);

    let api_key_env_var = match agent_name.as_str() {
        "Claude Code" => CLAUDE_API_KEY_ENV,
        // "Codex" => CODEX_API_KEY_ENV, // Uncomment/add if integrating Codex
        _ => {
            warn!("Attempted to run unsupported AI agent: {}", agent_name);
            return Err(KanbotsError::AiAgent(format!("Unsupported AI agent:
{}", agent_name)));
        }
    };

    // Attempt to get the API key. The '?' operator will automatically convert
    // std::env::VarError into KanbotsError::ApiKey due to our `From` impl.
    let api_key = get_api_key(api_key_env_var)?;

    // ⚡ Real-world insight: In a production system, `api_key` would be
passed
    // to an actual AI client library here to make an HTTP request.
    // For this guide, we'll simulate the call and potential failures.

    // Simulate an AI API call. Let's introduce a simulated failure condition.
    if task_description.contains("fail_me") {
        error!("Simulating AI agent failure for task: '{}'", task_description)
;
        return Err(KanbotsError::AiAgent("Simulated AI agent processing
error.".to_string()));
    }

    debug!("Agent '{}' is processing task '{}' with API key (value hidden)", a
gent_name, task_description);
    info!("Agent '{}' successfully processed task: '{}'", agent_name, task_des
cription);

    // Placeholder for actual AI agent logic output
    Ok(format!("Agent '{}' completed task: '{}' (API key loaded from {})", age
```

```
nt_name, task_description, api_key_env_var))
}

fn main() {
    // Initialize the logger. `env_logger` reads the RUST_LOG environment
variable
    // to determine which log levels to display (e.g., RUST_LOG=info,
RUST_LOG=debug).
    env_logger::init();
    info!("Kanbots Rust backend starting up.");

    tauri::Builder::default()
        .invoke_handler(tauri::generate_handler![
            // ... other commands ...
            run_ai_agent_task,
        ])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Explanation:

- **Imports**: We now import `log` macros and our `KanbotsError` type from the `error` module.
- **`get_api_key` Return Type**: It now returns `Result<String, KanbotsError>`, allowing us to propagate our custom error type.
- **Error Logging**: Inside `get_api_key`, `error!` is used to log the failure, providing internal debugging information. The `KanbotsError::ApiKey` variant is returned with a user-friendly message.
- **`run_ai_agent_task` Return Type**: This command also returns `Result<String, KanbotsError>`. This is crucial for Tauri's IPC to correctly handle errors and send them to the frontend as structured JSON.
- **`?` Operator**: The `api_key = get_api_key(api_key_env_var)?;` line concisely handles potential errors from `get_api_key`. If `get_api_key` returns an `Err`, `run_ai_agent_task` immediately returns that `Err`.
- **Simulated Failure**: We've added a conditional `if task_description.contains("fail_me")` to simulate an AI agent processing error, demonstrating how a `KanbotsError::AiAgent` would be returned.
- **Logging**: `info!`, `warn!`, `error!`, and `debug!` macros are used at various points to provide observability into the application's runtime.
- **`env_logger::init()`**: Called in `main` to set up basic logging. By default, it prints to stderr. Its verbosity can be controlled using the `RUST_LOG` environment variable (e.g., `RUST_LOG=info` for informational messages and above, `RUST_LOG=debug` for more verbose output).

4. Setting Environment Variables

Before running Kanbots, you need to set the environment variable for your AI API key. Remember to replace `"your_claude_api_key_here"` with your actual API key.

On Linux/macOS (Bash/Zsh):

```
export CLAUDE_API_KEY="your_claude_api_key_here"
# Or for Codex if integrated
export CODEX_API_KEY="your_codex_api_key_here"
```

On Windows (Command Prompt):

```
set CLAUDE_API_KEY="your_claude_api_key_here"
rem Or for Codex if integrated
set CODEX_API_KEY="your_codex_api_key_here"
```

On Windows (PowerShell):

```
$env:CLAUDE_API_KEY="your_claude_api_key_here"
# Or for Codex if integrated
$env:CODEX_API_KEY="your_codex_api_key_here"
```

Important: These commands set the variable only for the current terminal session. To make them persistent, you'd add them to your shell's profile file (e.g., `~/.bashrc`, `~/.zshrc`, `~/.profile` on Linux/macOS, or configure System Environment Variables on Windows).

5. UI Feedback for Errors (Svelte Frontend)

The Svelte frontend needs to be updated to gracefully handle errors returned from the Rust backend. Tauri's `invoke` mechanism handles `Result` types from Rust, converting `Err` variants (if they are `Serialize`) into stringified JSON that can be parsed in TypeScript.

Update `src/lib/tauri.ts` (or similar IPC wrapper)

Ensure your IPC call can catch errors and parse the structured `KanbotsError`.

File: `src/lib/tauri.ts` (or wherever you define Tauri invoke calls)

```
import { invoke } from "@tauri-apps/api/tauri";
```

```
// Define a TypeScript interface that mirrors our Rust KanbotsError enum
structure.
// This allows the frontend to work with typed errors.
interface KanbotsError {
  ApiKey?: string; // If the error is an ApiKey variant, this property will
exist
  AiAgent?: string; // If the error is an AiAgent variant, this property will
exist
  GitWorktree?: string;
  Ipc?: string;
  Unknown?: string;
  // A generic message property for easy display, populated after parsing
  message: string;
}

export async function invokeRunAiAgentTask(
  agentName: string,
  taskDescription: string,
): Promise<string> {
  try {
    const response = await invoke<string>("run_ai_agent_task", {
      agentName,
      taskDescription,
    });
    return response;
  } catch (error: any) {
    console.error("IPC call failed:", error);
    let kanbotsError: KanbotsError;
    try {
      // Tauri IPC errors from Rust `Result::Err` variants (if Serialize)
      // are often stringified JSON of the Rust error enum.
      const parsedError = JSON.parse(error);
      kanbotsError = parsedError as KanbotsError;

      // Extract the specific error message from the first (and usually only)
      // property of the parsed error object.
      // Example: { "ApiKey": "Environment variable 'CLAUDE_API_KEY' not
set..." }
      kanbotsError.message = Object.values(kanbotsError)[0] as string;
    } catch (parseError) {
      // If parsing fails, it's an unexpected error, so treat it as an Unknown
error.
      kanbotsError = { Unknown: String(error), message: String(error) };
    }
    throw kanbotsError; // Re-throw the structured error for the UI component
to handle
  }
}
```

Explanation:

- **`KanbotsError` Interface**: Defines the expected structure of errors coming from the Rust backend, ensuring type safety in TypeScript.
- **`try...catch` Block**: Wraps the `invoke` call to gracefully handle potential errors.

- **JSON Parsing**: When `invoke` returns an error from a Rust `Result::Err` (where the error type is `Serialize`), it typically comes as a stringified JSON. We attempt to `JSON.parse()` this string into our `KanbotsError` interface.
- **Message Extraction**: The `Object.values(kanbotsError)[0]` line is a clever way to extract the actual error message string, as our `KanbotsError` enum variants are structured like `{"VariantName": "message string"}`.
- **Re-throwing**: The structured `kanbotsError` is re-thrown, allowing the specific Svelte component that initiated the task to catch and display it.

Update a Svelte Component (e.g., `src/routes/KanbanCard.svelte`)

Modify a component that interacts with the `run_ai_agent_task` command to display errors dynamically.

File: `src/routes/KanbanCard.svelte` (simplified example)

```
<script lang="ts">
  import { invokeRunAiAgentTask } from '$lib/tauri';
  import { createEventDispatcher } from 'svelte';

  export let cardId: string;
  export let taskTitle: string;
  export let agentType: string = 'Claude Code'; // Default agent

  let agentOutput: string = '';
  let errorMessage: string | null = null;
  let isLoading: boolean = false;

  const dispatch = createEventDispatcher();

  async function startAgentTask() {
    errorMessage = null; // Clear previous errors before a new attempt
    isLoading = true;
    agentOutput = 'Agent is thinking...'; // Provide immediate feedback

    try {
      const response = await invokeRunAiAgentTask(agentType, taskTitle);
      agentOutput = response;
      dispatch('agentCompleted', { cardId, output: response });
    } catch (err: any) {
      console.error('Frontend caught agent error:', err);
      // Display the structured error message from the backend
      errorMessage = err.message || 'An unknown error occurred with the
agent.';
      agentOutput = `Error: ${errorMessage}`; // Update output area as well
      dispatch('agentFailed', { cardId, error: errorMessage });
    } finally {
      isLoading = false; // Always reset loading state
    }
  }
</script>

<div class="kanban-card">
```

```
  <h3>{taskTitle}</h3>
  <p>Agent: {agentType}</p>
  <button on:click={startAgentTask} disabled={isLoading}>
    {#if isLoading}
      Running...
    {:else}
      Run Agent
    {/if}
  </button>

  {#if agentOutput}
    <div class="agent-output">
      <h4>Agent Output:</h4>
      <pre>{agentOutput}</pre>
    </div>
  {/if}

  {#if errorMessage}
    <div class="error-message">
      <h4>Error:</h4>
      <p>{errorMessage}</p>
      <p>Check the application logs (terminal) for more details.</p>
    </div>
  {/if}
</div>

<style>
  .kanban-card {
    border: 1px solid #ccc;
    padding: 10px;
    margin-bottom: 10px;
    background-color: #f9f9f9;
    border-radius: 5px;
  }
  .agent-output {
    margin-top: 10px;
    padding: 8px;
    background-color: #e0f7fa;
    border-left: 3px solid #00bcd4;
  }
  .error-message {
    margin-top: 10px;
    padding: 8px;
    background-color: #ffebee;
    border-left: 3px solid #f44336;
    color: #f44336;
  }
</style>
```

Explanation:

- **`startAgentTask` Function**: Now includes a `try...catch` block.
- **Error Display**: If `invokeRunAiAgentTask` throws an error, it's caught, and `errorMessage` is set to the `message` property of our structured error.

- **Conditional Rendering**: The Svelte template conditionally displays the `errorMessage` div (`{#if errorMessage}`), providing clear, immediate feedback to the user.
- **Loading State**: `isLoading` state is used to disable the button during agent execution and show "Running..." feedback, improving UX.

Testing & Verification

Let's ensure our new security and error handling measures are working as expected.

1. **Verify API Key Loading (Success Case):**
    - Set your `CLAUDE_API_KEY` environment variable in your terminal before starting the app.
    - Run Kanbots: `RUST_LOG=info cargo tauri dev` (or `npm run tauri dev`).
    - In the Kanbots UI, run an agent task (e.g., "Write a simple Rust function").
    - **Expected:**
        - The Rust backend terminal output should show `info` level logs confirming the API key was loaded (e.g., `info: Attempting to load API key from environment: CLAUDE_API_KEY`).
        - The UI should show the success message from the agent.
    - **Check:** Ensure the actual key value is not printed in the logs, only the confirmation of its loading.

2. **Verify API Key Loading (Failure Case):**
    - **Unset** your `CLAUDE_API_KEY` environment variable (e.g., `unset CLAUDE_API_KEY` on Linux/macOS, or close and reopen the terminal to clear it).
    - Run Kanbots: `RUST_LOG=info cargo tauri dev`.
    - In the Kanbots UI, run an agent task.
    - **Expected:**
        - The Rust backend terminal should show `error` level logs indicating the environment variable is not set (e.g., `error: Failed to load API key CLAUDE_API_KEY: ...`).
        - The Kanbots UI should display an error message like "Error: Environment variable 'CLAUDE_API_KEY' not set or invalid."
    - **Check:** The UI feedback is clear and the application doesn't crash.
3. **Verify Agent Task Error Handling (Simulated Failure):**
    - Ensure your `CLAUDE_API_KEY` is set.
    - Run Kanbots: `RUST_LOG=info cargo tauri dev`.
    - In the Kanbots UI, create a card with a task title that includes `"fail_me"` (e.g., "Implement user auth - fail_me").
    - Run the agent task on this card.
    - **Expected:**
        - The Rust backend terminal should show `error` level logs indicating the simulated AI agent failure (e.g., `error: Simulating AI agent failure for task: 'Implement user auth - fail_me'`).
        - The Kanbots UI should display an error message like "Error: Simulated AI agent processing error."
    - **Check:** The specific error message from the backend is correctly propagated and displayed.

4. **Verify Logging Levels:**
    - Run Kanbots with different `RUST_LOG` settings:
        - `RUST_LOG=warn cargo tauri dev`: You should only see `warn` and `error` messages.
        - `RUST_LOG=debug cargo tauri dev`: You should see all `info`, `warn`, `error`, and `debug` messages, including the hidden API key message (`debug: Agent ... with API key (value hidden)`).
    - **Check:** The logs provide useful context for debugging at different verbosity levels.

Production Considerations

While environment variables are a good start for managing secrets, true production desktop applications often require more robust secret management and advanced observability.

OS-Level Secret Management

For maximum security, especially for user-specific API keys, consider using OS-level secret storage:

- **Rust `keyring` crate:** This crate (`keyring = "2.0"` as of 2026-05-24) provides a cross-platform interface to OS secret services (macOS Keychain, Windows Credential Manager, Linux Secret Service). It's the recommended approach for desktop apps storing sensitive user credentials.
    - **Usage:** Instead of `std::env::var`, you'd use `keyring::Entry::new("Kanbots", "claude_api_key")?.get_password()`.
    - **Tradeoffs:** Adds a dependency, requires user permission dialogs on first access, and is more complex to set up. However, it's significantly more secure than environment variables, especially if the user's system itself is compromised or the app is distributed.

Advanced Logging and Observability

- **Structured Logging:** For larger applications, consider structured logging (e.g., with `slog` or `tracing` crates). This outputs logs in a machine-readable format (like JSON), making them easier to parse, filter, and analyze with tools.
- **Remote Reporting:** For deployed applications, you might integrate with error tracking services (e.g., Sentry, Bugsnag) or log aggregation platforms (e.g., ELK stack, Grafana Loki). This requires user consent and careful privacy considerations for a local-first desktop app.
- **User-Configurable Log Levels:** Allow users to adjust log verbosity from the UI (e.g., for debugging purposes). This can be implemented by exposing a Tauri command to set the `RUST_LOG` level dynamically.

Rate Limiting and Cost Management

AI APIs are not free and often have strict rate limits. Implement client-side rate limiting and potentially user-configurable usage limits to prevent runaway costs or hitting API limits.

- **Client-side throttling:** Use a token bucket algorithm or simple delay between API calls within the Rust backend to manage the frequency of requests to the AI provider.
- **User settings:** Allow users to set a maximum daily or monthly spend/usage, and warn them when they approach the limit. This could be integrated into the Kanbots settings.

Common Issues & Solutions

1. **API Keys Exposed in Logs/Code:**
    - **Issue:** Accidentally printing the actual API key in logs, or hardcoding it in source code.
    - **Solution:** Never log the key itself. Always use environment variables or OS secret stores. Review your code and log outputs carefully. Use tools like `git-secrets` to prevent committing sensitive data to version control.
2. **Generic Error Messages:**
    - **Issue:** The UI simply says "An error occurred" without details, leaving the user confused.
    - **Solution:** Use structured error types (like `KanbotsError`) to categorize errors. Propagate specific error messages from the backend to the frontend. Ensure frontend logic extracts and displays these details (e.g., `err.message`).
3. **Silent Failures:**
    - **Issue:** An operation fails, but the application neither crashes nor informs the user, leading to incorrect state or data loss.
    - **Solution:** Make every potentially failing operation return a `Result`. Always handle the `Err` variant (at least by logging it). Use logging (`error!`, `warn!`) to catch and record unexpected states, even if the UI can't immediately show everything.
4. **Tauri IPC Error Handling Confusion:**
    - **Issue:** The `catch` block in TypeScript receives a vague `error: any` from `invoke`, making it hard to process.
    - **Solution:** Remember that Rust `Result` types where `Err` implements `Serialize` will be stringified JSON when sent over IPC. Always try to `JSON.parse()` the error string and cast it to your expected TypeScript error interface. This provides a structured error object for your frontend logic.

Summary & Next Step

You've made significant strides in hardening Kanbots. By implementing secure API key loading via environment variables, defining a robust custom error handling strategy in Rust, and integrating clear UI feedback in Svelte, your application is now more reliable and user-friendly. The logging infrastructure provides crucial observability, making future debugging and maintenance much simpler.

What's ready now:

- API keys for AI agents are loaded securely from environment variables.
- The Rust backend uses a custom `KanbotsError` enum for structured error reporting.
- Errors from the backend are gracefully propagated to the Svelte frontend.
- The Svelte UI provides clear, user-friendly feedback when errors occur.
- Basic logging is in place to track application activity and errors.

The next step would typically involve further refining the UI/UX, perhaps implementing more advanced multi-agent coordination features, or preparing for packaging and deployment. We'll focus on the latter in the next chapter, ensuring your Kanbots application is ready to be shared.

References

- Tauri Documentation: Commands
- Rust Standard Library: `std::env`
- Rust `thiserror` Crate Documentation
- Rust `log` Crate Documentation
- Rust `env_logger` Crate Documentation
- Svelte Documentation
- Rust `keyring` Crate Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

Logging Agent Activities and Deployment Considerations

Debugging and understanding the behavior of a multi-agent system like Kanbots can be incredibly challenging without proper visibility. In this final chapter, we'll equip our Kanbots application with robust logging capabilities to capture agent activities, inputs, outputs, and any errors. This provides the essential observability needed to diagnose issues, track performance, and even audit AI agent decisions.

Beyond observability, this chapter also guides you through the critical steps of preparing your Kanbots application for distribution. We'll explore Tauri's deployment features, focusing on how to package your application for various operating systems and important considerations like secure API key management and application signing.

By the end of this chapter, you will have a Kanbots application that not only orchestrates AI agents effectively but also provides clear insights into their operations through structured logs. You'll also understand the process of building and preparing your desktop application for real-world use, making it ready to share or deploy within your team.

Project Overview: Kanbots - AI-Powered Kanban

Kanbots is an open-source, local-first Kanban desktop application designed to streamline development workflows. It allows users to create Kanban cards, each capable of hosting and orchestrating multiple AI agents (such as Claude Code or Codex). These agents operate within isolated Git worktrees, enabling parallel task execution, code generation, review, and other development activities. The application leverages a multi-persona approach, assigning specific roles to agents to manage complex tasks effectively.

Tech Stack Deep Dive for Observability and Deployment

This chapter focuses on two critical aspects: observability and distribution. Our core technologies, Tauri and Rust, provide powerful tools for both.

- **Tauri (v2):** The framework for building our cross-platform desktop application. Its robust build system handles packaging and distribution.
- **Rust (Latest Stable):** The backend language powering our application logic, including agent orchestration and Git worktree management.
- **`tracing` crate (v0.1.40):** Rust's idiomatic, structured logging framework. It allows us to emit rich, machine-readable log events.
- **`tracing-subscriber` (v0.3.18):** Provides the configuration layer for `tracing`, allowing us to direct logs to files, consoles, and filter by level.
- **`tracing-appender` (v0.2.2):** Enables efficient, non-blocking file logging and log rotation, crucial for production applications.
- **Svelte (v5):** Our frontend framework, which interacts with the Rust backend via Tauri's IPC for UI updates and triggering agent actions.

Build Plan: Logging and Distribution Milestones

This chapter breaks down into two main milestones:

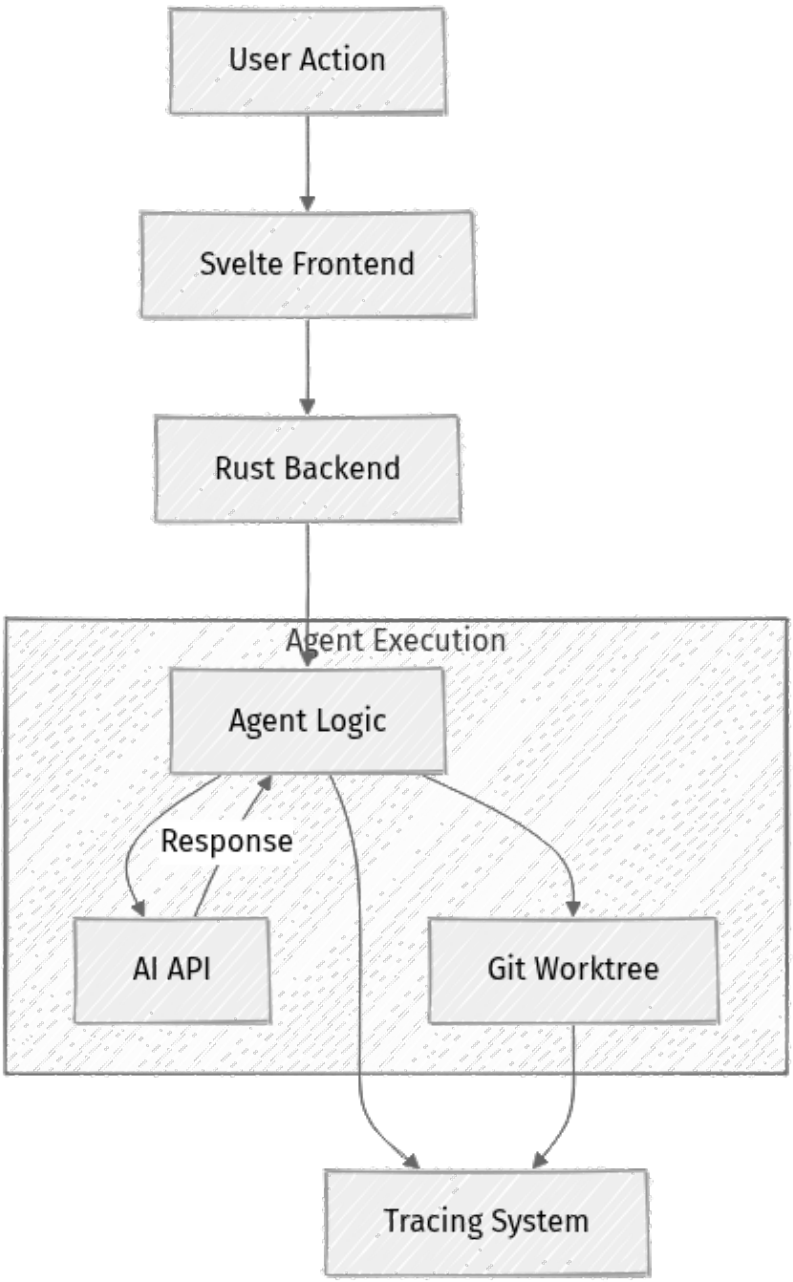
1. **Implement Structured Logging:** Integrate the `tracing` ecosystem into the Rust backend to capture detailed, machine-readable logs of agent activities. This will involve adding dependencies, configuring a file appender with rotation, and instrumenting key agent operations with tracing macros.
2. **Prepare for Deployment:** Configure Tauri's build settings to package the application for various operating systems. This includes setting application metadata, handling icons, and understanding the process for generating distributable installers.

Architecture: Integrating Observability

For Kanbots, logging serves several critical purposes: debugging, auditing agent actions, monitoring performance, and managing AI API costs. We will use `tracing` to emit structured logs that are both human-readable (in development) and machine-parseable (for analysis). These logs will be written to daily rotating files in the application's data directory.

Log File Flow

The following diagram illustrates how user actions flow through the system, trigger agent logic, and how the `tracing` system captures events from both agent operations and Git worktree interactions, outputting them for monitoring.



- **User Action:** Initiates a task in the Svelte UI.
- **Svelte Frontend:** Communicates with the Rust backend via IPC.
- **Rust Backend:** Orchestrates agent logic.
- **Agent Logic:** Contains the core intelligence, interacting with external AI APIs and local Git worktrees.
- **Tracing System:** Captures events from `Agent Logic` and `Git Worktree` operations.
- **Log File:** Stores structured log events, typically rotated daily.
- **Debugging/Monitoring:** Engineers use these logs to understand system behavior.

Step-by-Step Implementation: Agent Logging with `tracing`

We'll integrate the `tracing` crate into our Rust backend to provide structured, file-based logging.

1. Add `tracing` Dependencies

Open your `Cargo.toml` file located in the `src-tauri` directory. Add the following dependencies under the `[dependencies]` section. These versions were current as of 2026-05-24.

```
# src-tauri/Cargo.toml

[dependencies]
# ... other dependencies ...
tracing = "0.1.40"
tracing-subscriber = { version = "0.3.18", features = ["env-filter", "fmt", "a
nsi", "json"] }
tracing-appender = "0.2.2"
```

- **`tracing = "0.1.40"`:** This is the core `tracing` crate. It provides the macros (`info!`, `error!`, `debug!`, etc.) that you'll use to emit log events throughout your Rust code.
- **`tracing-subscriber = "0.3.18"`:** This crate provides the "subscriber" implementation, which is responsible for collecting, filtering, and formatting the events emitted by `tracing`.
    - `env-filter`: Enables filtering log events based on environment variables (e.g., `RUST_LOG=info`). This is very useful for controlling verbosity without recompiling.
    - `fmt`: Provides a formatter for human-readable log output, typically to the console or a file.
    - `ansi`: Adds ANSI color codes to console output for better readability during development.
    - `json`: Enables structured JSON output, which is ideal for machine parsing, analysis, and integration with log management systems.
- **`tracing-appender = "0.2.2"`:** This crate provides utilities for efficiently writing logs to files. We'll use it for non-blocking I/O and log file rotation.

2. Configure `tracing` in `main.rs`

Next, we'll set up the `tracing` subscriber in our `main.rs` to direct logs to a file and the console. We'll also make the log file path configurable and ensure logs are written to a platform-appropriate application data directory.

Open `src-tauri/src/main.rs` and add the following code.

```
// src-tauri/src/main.rs

use tracing::{info, error, warn, debug, Level};
use tracing_subscriber::{
    fmt::{self, MakeWriter},
    EnvFilter,
    prelude::*,
};
use tracing_appender::rolling::{RollingFileAppender, Rotation};
use std::path::PathBuf;
use std::fs;

// ... other imports for your Tauri commands ...

/// Helper function to get the application's data directory.
/// This function determines a platform-specific directory for persistent
application data.
/// For a running Tauri app, `app_handle.path_resolver().app_data_dir()` is
more precise.
/// Here, we use a default config to get a base path suitable for early
logging setup.
fn get_app_data_dir() -> PathBuf {
    // This will typically resolve to a platform-specific directory, e.g.:
    // Linux:   ~/.config/kanbots
    // macOS:   ~/Library/Application Support/com.kanbots.dev
    // Windows: C:\Users\Username\AppData\Roaming\Kanbots
    tauri::api::path::app_data_dir(&tauri::Config::default())
        .expect("Failed to get application data directory")
}

/// Sets up the global tracing subscriber for structured logging.
/// Logs are directed to both a daily rotating file (JSON format) and the
console (compact format).
/// Log levels are controlled by the RUST_LOG environment variable, defaulting
to INFO.
fn setup_logging() {
    let app_data_dir = get_app_data_dir();
    let log_dir = app_data_dir.join("logs");

    // Ensure the log directory exists, creating it if necessary.
    fs::create_dir_all(&log_dir).expect("Failed to create log directory");

    // Configure a rolling file appender: new log file daily, named
"kanbots.log".
    let file_appender = RollingFileAppender::new(Rotation::DAILY, &log_dir, "k
anbots.log");
    // Wrap the file appender in a non-blocking writer to prevent logging from
blocking the main thread.
    let (non_blocking_appender, _guard) = tracing_appender::non_blocking(file_
```

```
appender);

    // Build the tracing subscriber registry.
    tracing_subscriber::registry()
        // Filter logs based on the RUST_LOG environment variable, defaulting
to INFO.
        .with(EnvFilter::from_default_env().add_directive(Level::INFO.into()))
        // Layer for file logging: JSON format, no ANSI colors, compact.
        .with(
            fmt::layer()
                .with_writer(non_blocking_appender) // Direct output to the
non-blocking file writer
                .json() // Output logs in structured JSON format
                .with_ansi(false) // Disable ANSI colors for file logs
                .compact(), // Use a compact format for fields
        )
        // Layer for console logging: compact format, with ANSI colors.
        .with(
            fmt::layer()
                .with_writer(std::io::stdout) // Direct output to standard
console
                .with_ansi(true) // Enable ANSI colors for console logs
                .compact(), // Use a compact format for fields
        )
        // Initialize the global default subscriber.
        .init();

    info!("Kanbots logging initialized to: {:?}", log_dir);
}


fn main() {
    // Initialize logging as the very first step in main.
    setup_logging();

    tauri::Builder::default()
        .invoke_handler(tauri::generate_handler![
            // ... your existing command handlers here ...
            // e.g., agent_start_task, git_create_worktree, etc.
            agent_start_task // Ensure this example command is included
        ])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}

// Example of how to use tracing in your Tauri commands:
// This command simulates an AI agent starting a task.
#[tauri::command]
async fn agent_start_task(card_id: String, agent_id: String,
task_description: String) -> Result<(), String> {
    // Log the initiation of an agent task using info! macro with structured
fields.
    info!(
        card_id = %card_id,
        agent_id = %agent_id,
        task = %task_description,
        "AI agent task initiated."
    );

    // Simulate agent work and a potential error condition.
    if task_description.contains("fail") {
        // Log an error if a simulated failure occurs.
```

```
        error!(
            card_id = %card_id,
            agent_id = %agent_id,
            error = "Simulated failure condition",
            "Agent task failed unexpectedly."
        );
        return Err("Agent task simulation failed.".into());
    }

    // Log a debug message for intermediate processing steps.
    debug!(
        card_id = %card_id,
        agent_id = %agent_id,
        "Agent is processing task..."
    );

    // ... actual agent logic, git operations, API calls would go here ...
    // Use info!, debug!, warn!, error! macros as appropriate for different
stages.

    // Example: When making an API call to an AI service.
    info!(
        card_id = %card_id,
        agent_id = %agent_id,
        api_endpoint = "Claude Code API",
        prompt_length = 500,
        "Calling AI API for code generation."
    );

    // Example: When receiving a response from the AI API.
    debug!(
        card_id = %card_id,
        agent_id = %agent_id,
        response_length = 1200,
        "Received response from AI API."
    );

    // Example: When a file is created or modified in the agent's worktree.
    info!(
        card_id = %card_id,
        agent_id = %agent_id,
        file_path = "src/new_feature.rs",
        "Agent created file in worktree."
    );

    // Log successful completion of the agent task.
    info!(
        card_id = %card_id,
        agent_id = %agent_id,
        "AI agent task completed successfully."
    );
    Ok(())
}
```

Explanation of the Logging Setup:

1. **`get_app_data_dir()`:** This helper function leverages Tauri's API (`tauri::api::path::app_data_dir`) to determine a platform-appropriate directory for storing application data. This is crucial for cross-platform compatibility and ensures logs are stored persistently and in a user-specific location without cluttering the project directory. We use `tauri::Config::default()` to get a base path.
2. **`setup_logging()`:**
    - **Log Directory Creation:** It first constructs a path for a `logs` subdirectory within the application's data directory and ensures it exists.
    - **`RollingFileAppender`:** Creates a file appender that automatically rotates log files daily. This prevents a single log file from growing indefinitely, which is a common issue in long-running applications. Files will be named like `kanbots.log.YYYY-MM-DD`.
    - **`tracing_appender::non_blocking`:** Wraps the file appender with a non-blocking writer. This is a critical performance optimization: logging operations are offloaded to a background thread, preventing them from blocking your application's main execution flow and ensuring a smooth user experience.
    - **`tracing_subscriber::registry().with(...)`:** This is where the `tracing` ecosystem is configured.
        - **`EnvFilter`:** Allows you to control the verbosity of logs using the `RUST_LOG` environment variable (e.g., `RUST_LOG=debug` to see all debug messages). It defaults to `INFO` level if `RUST_LOG` is not set.
        - **File Layer:** Configures a `fmt::layer()` to write to the `non_blocking_appender`. Crucially, `.json()` ensures logs are emitted in a structured JSON format, making them easy to parse by log analysis tools. `.with_ansi(false)` prevents color codes from polluting the file.
        - **Console Layer:** Configures a separate `fmt::layer()` to write to `std::io::stdout` (your terminal). `.with_ansi(true)` enables colorized output, which is very helpful for debugging during development.
    - **`.init()`:** Activates the configured subscriber globally, making `tracing` macros available throughout your application.

3. **`main()`:** The `setup_logging()` function is called at the very beginning of your `main` function. This ensures that the logging system is fully initialized and ready to capture events from the moment your application starts.
4. **Tracing in Commands:**
    - You use `info!`, `debug!`, `warn!`, and `error!` macros (similar to `println!`) to emit log events.
    - **Structured Fields:** The syntax `card_id = %card_id` is key. It creates named key-value pairs in your log output. The `%` ensures the value is formatted using its `Display` trait. These structured fields are invaluable for filtering and querying logs later. For example, you could easily search for all log entries related to a specific `card_id`.

3. Update Frontend to Trigger Logging (Optional)

While the logging logic resides in Rust, ensure your Svelte frontend calls the relevant Tauri commands (e.g., `agent_start_task`) that you've instrumented with `tracing` macros. Your existing frontend code should already do this. The important part is that when these commands are invoked from the UI, the Rust backend will now automatically generate detailed log entries.

Step-by-Step Implementation: Deployment Preparation

Tauri makes building for deployment straightforward. We'll configure `tauri.conf.json` and discuss the build process.

1. Configure `tauri.conf.json` for Production

Open `src-tauri/tauri.conf.json`. This file is the central configuration for how your Tauri application is built, packaged, and behaves. The structure shown here is compatible with Tauri v2. Review and customize the fields as described.

```
// src-tauri/tauri.conf.json

{
  "build": {
    "devPath": "../dist",
    "distDir": "../dist",
    "beforeDevCommand": "npm run dev",
    "beforeBuildCommand": "npm run build"
```

```
  },
  "package": {
    "productName": "Kanbots",
    "version": "0.1.0",
    "description": "Kanban board with AI agents for development workflows.",
    "authors": ["Your Name or Team Name"],
    "homepage": "https://your-project-website.com"
  },
  "tauri": {
    "bundle": {
      "active": true,
      "targets": "all",
      "identifier": "com.yourcompany.kanbots",
      "icon": [
        "icons/32x32.png",
        "icons/128x128.png",
        "icons/128x128@2x.png",
        "icons/icon.icns",
        "icons/icon.ico"
      ],
      "resources": [],
      "externalBin": [],
      "copyright": "Copyright (c) 2026 Your Name",
      "deb": {
        "depends": []
      },
      "macOS": {
        "entitlements": null,
        "exceptionDomain": "",
        "frameworks": [],
        "providerShortCode": null,
        "signingIdentity": null,
        "hardenedRuntime": true,
        "resources": [],
        "infoPlist": {}
      },
      "windows": {
        "certificateThumbprint": null,
        "digestAlgorithm": "sha256",
        "timestampUrl": ""
      }
    },
    "security": {
      "csp": null
    },
    "windows": [
      {
        "fullscreen": false,
        "resizable": true,
        "title": "Kanbots",
        "width": 1200,
        "height": 800
      }
    ]
  }
}
```

Key fields to note and customize:

- **`build.devPath` / `build.distDir`:** These point to your Svelte frontend's build output directory. `../dist` is a common default.
- **`build.beforeDevCommand` / `build.beforeBuildCommand`:** These commands tell Tauri how to start your frontend in development mode and how to build it for production. Ensure `npm run dev` and `npm run build` (or `yarn` / `pnpm` equivalents) are correctly configured in your frontend's `package.json`.
- **`package.productName`:** The user-facing name of your application, displayed in the OS.
- **`package.version`:** The version number of your application. Follow semantic versioning (e.g., `0.1.0`, `1.0.0`).
- **`package.description`, `package.authors`, `package.homepage`:** Important metadata for your application's installers and about dialogs.
- **`tauri.bundle.identifier`: CRITICAL.** This is a unique identifier for your application, usually in reverse domain name notation (e.g., `com.yourcompany.kanbots`). It's used by the operating system for various purposes, including application sandboxing, storing user data, and identifying your app for updates. **Ensure you change `com.kanbots.dev` to something unique to your project.**
- **`tauri.bundle.icon`:** Paths to your application icons. Tauri uses these to generate platform-specific icons. Ensure these files exist in your `src-tauri/icons/` directory.
- **`tauri.bundle.targets`:** Defines which installer types Tauri should build. Set to `"all"` to build for all supported targets for your host OS, or specify specific targets like `["msi", "dmg", "appimage"]`.
- **`tauri.bundle.macOS.signingIdentity` / `tauri.bundle.windows.certificateThumbprint`:** These fields are for code signing. Code signing is crucial for production applications to establish trust with users and operating systems. Setting them up requires developer accounts (Apple Developer Program, Microsoft Partner Center) and specific certificates. For now, leaving them `null` is fine, but be aware of their purpose for future production readiness.
- **`tauri.bundle.macOS.hardenedRuntime`:** A security feature for macOS applications, generally recommended to be `true` for production builds.

2. Create Application Icons

Ensure you have a set of application icons in the `src-tauri/icons/` directory as specified in `tauri.conf.json`. These icons are used by the operating system for the application executable, taskbar, and various menus. If you don't have them, you can use online tools (search for "Tauri icon generator" or "app icon generator") to create a set from a single high-resolution image.

Example structure:``` src-tauri/ └── icons/ ├── 32x32.png ├── 128x128.png ├── 128x128@2x.png ├── icon.icns (macOS specific) └── icon.ico (Windows specific)

```
### 3. Build the Application for Distribution

Once `tauri.conf.json` is configured, building your application for distribution is a single command. Navigate to your `src-tauri` directory in the terminal.
```bash
# In your project root (where Cargo.toml is in src-tauri/)
cd src-tauri
cargo tauri build
```

This command will:

1. Execute `npm run build` (or `pnpm build`, `yarn build`) in your frontend directory (`../dist`) to compile your Svelte application into static assets.
2. Compile your Rust backend in `release` mode (with optimizations).
3. Bundle the compiled frontend and backend, along with your specified assets and icons, into platform-specific installers or application bundles.

The generated installers and application bundles will be located in `src-tauri/target/release/bundle/`. You'll typically find `.dmg` (macOS), `.msi` or `.exe` (Windows), and `.deb` or `.AppImage` (Linux) files there, depending on your `tauri.bundle.targets` setting and host OS.

Testing & Verification: Log Inspection and Build Check

After implementing logging and preparing for deployment, it's crucial to verify everything works as expected.
```

1. Verify Agent Logging

1. **Run Kanbots in Development Mode:** Open your terminal in the `src-tauri` directory and execute:

```
cargo tauri dev
```

```
You should see console output from your `tracing` setup, including the `info!`
message confirming logging initialization.
```

1. **Trigger an Agent Task:** Interact with the Kanbots UI to start an AI agent task on a card (e.g., initiate the `agent_start_task` command).
2. **Locate the Log File:**
    - In your terminal, look for the initial `info!` message: `Kanbots logging initialized to: ...`. This will tell you the exact path to your log directory.
    - Navigate to this directory (e.g., `~/.config/kanbots/logs` on Linux, `~/Library/Application Support/com.yourcompany.kanbots/logs` on macOS).
    - You should find a file named `kanbots.log.YYYY-MM-DD` (e.g., `kanbots.log.2026-05-24`).
3. **Inspect Log Contents:** Open the log file using a text editor. You should see JSON-formatted entries reflecting the agent's activity:

```
    {"timestamp":"2026-05-24T12:00:00.000000Z","level":"INFO","target":"kanbot
s_app::main","fields":{"message":"AI agent task initiated.","card_id":"card-12
3","agent_id":"agent-456","task":"generate hello world"},"span":{"name":"agent
_start_task"},"spans":[{"name":"agent_start_task"}]}
    {"timestamp":"2026-05-24T12:00:00.100000Z","level":"DEBUG","target":"kanbo
ts_app::main","fields":{"message":"Agent is processing task...","card_id":"car
d-123","agent_id":"agent-456"},"span":{"name":"agent_start_task"},"spans":[{"n
ame":"agent_start_task"}]}
    // ... more log entries for API calls, file creations, etc. ...
```

```
Look for `INFO` messages marking the start and completion of tasks, `DEBUG`
for intermediate steps, and `ERROR` if you simulated a failure condition. The
structured fields (`card_id`, `agent_id`, `task`) should be present,
confirming the JSON output.
```

2. Verify Application Build

1. **Run the Build Command:** Open your terminal in the `src-tauri` directory and execute:

```
cargo tauri build
```

```
This process can take several minutes.
```

1. **Check Output Directory:** Once the build completes, navigate to `src-tauri/target/release/bundle/`.
2. **Inspect Bundles/Installers:**
    - **macOS:** You should find a `.dmg` disk image and/or an `.app` bundle.
    - **Windows:** You should find an `.msi` installer and/or a standalone `.exe` executable.
    - **Linux:** You should find a `.deb` package (for Debian/Ubuntu-based systems) or an `.AppImage` file (for broader Linux compatibility).
3. **Test the Built Application:** Install or run the generated package on your system. Confirm that the application launches correctly, the UI is fully functional, and that AI agents can still perform tasks. Crucially, verify that this built application also generates log files in the correct location (as confirmed in step 1). This ensures your production build includes all logging components.

Production Considerations: Operations and Security

Deploying a desktop application involves more than just building a package; it requires careful consideration of security, maintenance, and user experience.

Secure API Key Handling

🧠 **Important:** Never hardcode sensitive API keys directly into your application's source code or bundle them into the executable. This exposes them to anyone who can inspect your application.

For desktop applications, robust methods for handling secrets include:

- **Environment Variables:** This is the simplest and recommended starting point. Users set environment variables (e.g., `CLAUDE_API_KEY`, `CODEX_API_KEY`) in their shell before launching the application. Your Rust backend then reads these using `std::env::var`.
    - **Tradeoff:** Requires manual user setup and isn't as secure as OS-level storage if the user's environment is compromised.
- **OS-level Secret Management:** For higher security, leverage platform-specific credential managers like macOS Keychain, Windows Credential Manager, or `libsecret` on Linux. These store secrets securely at the operating system level.
    - **Tradeoff:** More complex to implement, often requires additional Rust crates (e.g., `keyring`) and platform-specific configurations.
- **Encrypted Configuration File:** A configuration file stored locally that is encrypted at rest and decrypted at runtime using a user-provided password or an OS-level secret.
    - **Tradeoff:** Adds complexity for key management and user interaction.

Recommendation for Kanbots: Start with environment variables for ease of development and initial distribution. For a truly production-grade application, especially if dealing with sensitive user data or high-value API keys, investigate OS-level secret management.

Log Rotation and Retention

Our `tracing-appender` configuration already handles daily log rotation. For a production system, consider further enhancements:

- **Compression:** Implement automatic compression of old log files (e.g., using `gzip`) to save disk space, especially for applications generating high volumes of logs.
- **Retention Policy:** Define and enforce a policy for automatically deleting log files older than a certain period (e.g., 30 days). This prevents log files from consuming excessive disk space over time. This can be managed by a small background task within your Rust app or an external script.

Performance Impact of Logging

While `tracing` with non-blocking appenders is highly efficient, excessive `debug!` or `trace!` logging in a production build can still introduce a minor performance overhead and generate very large log files.

- **Sensible Log Levels:** Use `info!` for significant events, `debug!` for detailed debugging (which is often disabled or filtered out in release builds by default via `RUST_LOG`), `warn!` for non-critical issues, and `error!` for critical failures.
- **Conditional Logging:** Avoid expensive computations or data serialization within `debug!` or `trace!` macros unless those levels are explicitly enabled. For example, avoid complex string formatting if the log message won't be seen.

Application Signing

Code signing is a non-negotiable step for distributing production desktop applications:

- **Trust and Authenticity:** Code signing assures users that the application comes from a verified developer and has not been tampered with since it was signed.
- **OS Security Features:** macOS Gatekeeper, Windows SmartScreen, and other OS security mechanisms will block or heavily warn users about unsigned applications. A signed application appears legitimate and runs without undue friction.
- **Updates:** Code signing is often a prerequisite for secure over-the-air update mechanisms.

Setting up code signing requires:

- Obtaining a developer certificate from a trusted Certificate Authority (e.g., Apple Developer Program, Microsoft Partner Center).
- Configuring your build environment with these certificates.
- Updating `tauri.conf.json` with the appropriate `signingIdentity` (macOS) or `certificateThumbprint` (Windows).

This is a complex, platform-specific process that typically falls outside the scope of a single project chapter, but it's a critical consideration for any public distribution.

Distribution Channels

Once your application is built and signed, you need a strategy for how users will acquire it:

- **Direct Download:** Host the installers (DMG, MSI, AppImage) on your project's website. This offers the most control but requires you to manage updates.
- **App Stores:**
  - **macOS App Store:** Offers broad reach but requires significant changes to adhere to Apple's strict sandboxing rules and review process.
  - **Microsoft Store:** Generally easier to get into than Apple's store.
  - **Linux Package Managers:** Distribute via `.deb` for Debian/Ubuntu-based systems, `.rpm` for Fedora/RHEL, or `AppImage` for wider compatibility across various Linux distributions.

Common Issues & Solutions

1. **Logs Not Appearing or Empty:**
    - **Issue:** You run the app, trigger agents, but the log file is empty or doesn't exist.
    - **Solution:**
        - **Check Console Output:** Verify the initial `Kanbots logging initialized to: ...` message in your terminal to confirm the log directory path and that `setup_logging()` ran.
        - **`setup_logging()` Call:** Ensure `setup_logging()` is called at the very beginning of your `main` function in `src-tauri/src/main.rs`.
        - **`RUST_LOG` Environment Variable:** Check if the `RUST_LOG` environment variable is set to a level that filters out all your messages (e.g., `RUST_LOG=off` or `RUST_LOG=error` when you're only emitting `info!` messages). Try setting `RUST_LOG=debug` to get more verbose output.
        - **File Permissions:** Ensure your application has write permissions to the determined log directory.
2. **`cargo tauri build` Fails:**
    - **Issue:** The build process errors out, often related to frontend compilation or specific platform targets.
    - **Solution:**
        - **Frontend Build:** First, try running `npm run build` (or `pnpm build`, `yarn build`) independently in your frontend directory (`../`) to ensure it compiles successfully. Fix any frontend-specific errors there.
        - **Dependencies:** Confirm all Rust (`cargo update`) and Node.js (`npm install`) dependencies are correctly installed and up-to-date.
        - **Tauri Configuration:** Double-check `src-tauri/tauri.conf.json` for any syntax errors or incorrect paths, especially for `bundle.identifier` and `icon` paths.
        - **Cross-Compilation:** If you're attempting to build for a different OS (e.g., macOS on Linux, or Windows on macOS), you might need specific cross-compilation tools (e.g., `mingw` for Windows builds on Linux) or virtual machines. It's generally easiest to build for a target OS on that OS itself.

3. **Sensitive Information in Logs:**
    - **Issue:** API keys, personal user data, or other secrets are accidentally written to log files.
    - **Solution:**
        - **Code Review:** Rigorously review all `info!`, `debug!`, `error!`, etc., calls to ensure no sensitive data is passed directly into log messages or structured fields. This includes prompt contents sent to AI APIs or responses received.
        - **Redaction/Masking:** For production systems, implement a log redaction mechanism. This could be a custom `tracing-subscriber` layer that automatically masks or removes sensitive patterns (e.g., API key formats, email addresses, credit card numbers) before writing to the log file.
        - **Access Control:** Ensure log files themselves are protected with appropriate file system permissions, limiting access to authorized users.

Summary & Next Steps

Congratulations! You've reached the end of building Kanbots, a sophisticated desktop Kanban application capable of orchestrating multi-agent AI workflows using Git worktrees. In this final chapter, we've equipped Kanbots with essential observability through structured logging and explored the critical steps for preparing your application for real-world deployment.

You now have a functional desktop application that:

- Provides an interactive Kanban board for task management.
- Integrates AI agents (like Claude Code or Codex) for automated task execution.
- Manages isolated task environments using Git worktrees, preventing conflicts.
- Orchestrates complex multi-agent workflows with persona-based assignments for efficient development.
- Offers clear insights into agent activities and system behavior via robust, structured logging.

- Is configured and ready for packaging and distribution across Windows, macOS, and Linux.

This project serves as a strong foundation for building more advanced AI-powered desktop tools. From here, consider these next steps to evolve Kanbots:

- **Enhanced UI/UX:** Implement more sophisticated real-time feedback for agent progress, richer control over agent tasks (pause, stop, re-run), and user customization options for the board and agents.
- **More Agent Types and Tooling:** Integrate with additional specialized AI models or provide agents with access to more tools (e.g., web search, file system manipulation with careful sandboxing, external APIs).
- **Advanced Workflow Orchestration:** Develop more complex dependency management between agent tasks, allowing for sophisticated multi-stage pipelines.
- **Cloud Integration (Optional):** While local-first is a key principle, you might explore syncing board state or agent progress to a cloud service for collaboration among a team.
- **AI Cost Monitoring:** Integrate with billing APIs from AI providers (if available and permissible) to display real-time cost estimates within the application for each agent's activity.

Keep iterating, keep building, and continue to explore the exciting possibilities of AI-enhanced development!

References

- Tauri Documentation
- Tauri Configuration Reference
- Tracing Crate Documentation
- Tracing Subscriber Crate Documentation
- Tracing Appender Crate Documentation
- Rust `std::env` Module
- Svelte Official Website

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

Building Kanbots: AI Agents, Git Worktrees, and Desktop Automation

Orchestrating Development with AI Agents and Isolated Workspaces

Modern software development often involves managing numerous tasks, collaborating with team members, and increasingly, leveraging AI for assistance. Imagine a tool that brings all these elements together: a personal Kanban board where each task card can host its own AI agents, operating in isolated Git environments, and collaborating on code generation, review, or other development workflows.

This guide will walk you through building **Kanbots**, a desktop Kanban application designed to do exactly that. We'll combine the power of a local-first desktop application with the intelligence of AI agents and the robustness of Git worktrees to create a unique development automation platform.

Why Build Kanbots?

The motivation behind Kanbots is to address several practical challenges in modern development:

- **Local-First Control and Privacy:** Unlike many cloud-based AI tools, Kanbots operates as a desktop application. This gives you direct control over your environment, data, and AI interactions, enhancing privacy and allowing direct filesystem access for agents.
- **Isolated AI Task Execution:** AI agents often need to modify code or interact with a specific project context. Git worktrees provide a clean, isolated environment for each agent to operate without interfering with your main codebase or other agents. This prevents conflicts and simplifies context management.

- **Multi-Agent Collaboration:** Real-world development tasks are complex. Kanbots enables you to orchestrate multiple AI agents, each potentially with a different persona (e.g., "Developer," "Reviewer"), to collaborate on a single task card. This mimics human team dynamics for automated workflows.
- **Enhanced Development Workflow:** Automate repetitive coding tasks, generate boilerplate, perform code reviews, or even prototype features directly from your task board, freeing up developer time for more complex problem-solving.
- **Understanding Modern Stack Integration:** This project offers a hands-on opportunity to integrate a modern desktop framework (Tauri), a reactive frontend (Svelte), powerful version control (Git), and external AI APIs into a cohesive, functional system.

What We're Building

Kanbots will be a cross-platform desktop application featuring:

- An interactive Kanban board UI to manage tasks.
- The ability to attach one or more AI agents to individual task cards.
- Dynamic creation and management of Git worktrees, providing isolated sandboxes for each agent's execution.
- Orchestration logic for multi-agent workflows, allowing agents to collaborate sequentially or in parallel.
- Persona-based task assignment, enabling agents to adopt specific roles (e.g., code generation, review, testing).
- Real-time UI feedback on agent progress, outputs, and control mechanisms (pause, resume).
- Secure handling of AI API keys and robust error reporting.

By the end of this guide, you will have a functional Kanbots application and a deep understanding of how to architect desktop applications with Rust, manage complex Git operations programmatically, and orchestrate intelligent agents for practical development tasks.

Core Technologies and Versions

We'll be leveraging a powerful and modern stack for Kanbots:

- **Tauri v2:** The framework for building cross-platform desktop applications using web technologies. Tauri's Rust backend provides performance, security, and direct system access, while its web frontend offers flexibility.
- **Svelte 5:** A reactive JavaScript framework for building the user interface. Svelte compiles your components into highly optimized vanilla JavaScript, resulting in small bundle sizes and excellent performance.
- **Rust:** The primary language for Kanbots' backend logic, handling file system interactions, Git operations, and AI API orchestration.
- **TypeScript:** Used for type-safe frontend development with Svelte, improving code quality and maintainability.
- **AI Agent APIs:** We will integrate with external AI models like Claude Code (from Anthropic) or Codex (from OpenAI, or its successors).
    - **Claude Code:** Exact stable version checked unknown as of 2026-05-24. Please confirm the latest stable release from Anthropic's official documentation.
    - **Codex (or successors):** Exact stable version checked unknown as of 2026-05-24. Please confirm the latest stable release from OpenAI's official documentation.
- **Git:** The distributed version control system, fundamental for managing worktrees and code changes.
    - Exact stable version checked unknown as of 2026-05-24. Please confirm the latest stable release from Git's official documentation.

Prerequisites and Setup

Before we begin, ensure you have the following installed and configured on your desktop operating system (Windows, macOS, or Linux):

1. **Rust Toolchain:** Install `rustup` by following the instructions on the official Rust website. This will provide `rustc`, `cargo`, and other necessary tools.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

1. **Node.js & npm/yarn:** For the Svelte frontend. We recommend using a Node.js LTS version.

```
# For Node.js (e.g., via nvm)
nvm install --lts
nvm use --lts
# Or directly from nodejs.org
```

1. **Git:** Ensure Git is installed and configured with your user name and email.

```
# Check if installed
git --version
# Configure (if not already)
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

1. **Code Editor:** A robust code editor like VS Code with Rust Analyzer and Svelte extensions is highly recommended.
2. **AI API Keys:** Obtain API keys for your chosen AI agent (e.g., Claude Code or OpenAI's models). Keep these secure; we will discuss secure handling in a later chapter.

Architecture at a Glance

Kanbots follows a clear architectural pattern:

- **Tauri Core:** The application is built with Tauri, where a Rust backend handles system-level operations, and a Svelte web frontend provides the user interface.
- **IPC Communication:** The Svelte frontend communicates with the Rust backend via Tauri's Inter-Process Communication (IPC) layer. This allows UI actions to trigger backend logic and backend events to update the UI.
- **Rust Backend Orchestration:** The Rust backend is the brain of Kanbots. It manages the creation, deletion, and switching of Git worktrees, orchestrates calls to external AI agent APIs, and handles the intricate logic of multi-agent workflows.
- **Isolated Agent Environments:** Each Kanban card can be associated with one or more AI agents. When an agent is active, the Rust backend ensures it operates within its own dedicated Git worktree. This worktree is a separate, lightweight copy of a Git repository, allowing agents to make changes without affecting the main branch or other agents.

- **External AI Intelligence:** The Rust backend makes secure HTTP requests to AI agent APIs (like Claude Code or OpenAI's services) to send prompts and receive generated outputs.
- **Local-First Data:** While AI interactions are external, the core Kanban board data and agent states will be managed locally, prioritizing user control and responsiveness.

Learning Path

This guide is structured into incremental milestones, allowing you to build and verify Kanbots step-by-step.

Setting Up Your Kanbots Workshop: Tauri v2 and Svelte 5

Configure the development environment for Tauri (v2) and Svelte (5), initialize the project, and verify the basic desktop application setup.

Building the Core Kanban Board UI

Develop the interactive Kanban board user interface with functionality for adding, editing, and moving cards and columns.

Mastering Git Worktrees for Isolated Agent Tasks

Implement Rust backend logic to dynamically create, manage, and switch between git worktrees, providing isolated execution environments for each Kanban card's tasks.

Integrating Your First AI Agent: Claude Code or Codex

Connect a single AI agent (e.g., Claude Code or Codex) to a Kanban card, enabling it to perform a simple task within its dedicated git worktree.

Orchestrating Multi-Agent Workflows with Personas

Design and implement the system for orchestrating multiple AI agents to collaborate on a task, demonstrating a practical code generation and review cycle using distinct personas.

Real-time Agent Progress and User Control UI

Enhance the frontend UI to display real-time agent progress, outputs, and provide user controls like pausing or resuming agent tasks.

Securing API Keys and Robust Error Handling

Implement secure storage and retrieval for AI API keys and establish robust error handling mechanisms for agent failures and API communication issues.

Logging Agent Activities and Deployment Considerations

Integrate comprehensive logging for agent activities and outputs, and discuss essential considerations for packaging and deploying the Kanbots desktop application.

Important Considerations and Potential Pitfalls

Building a system like Kanbots involves managing several complex interactions. As we progress, we'll discuss strategies for:

- **Managing Git Worktree State:** Ensuring worktrees are correctly created, cleaned up, and switched without data loss or conflicts.
- **Agent Orchestration Complexity:** Designing workflows that prevent deadlocks, manage concurrent agent actions, and handle unexpected outputs.
- **AI Agent Reliability:** Addressing potential hallucinations, infinite loops, or API rate limits from AI models.
- **Security:** Securely handling sensitive API keys and managing agent access to the local filesystem.
- **Performance:** Optimizing frequent Git operations and AI API calls to maintain a responsive application.
- **Debugging:** Strategies for debugging distributed logic across the frontend, Rust backend, and multiple AI agents.

This guide will provide practical solutions and best practices to navigate these challenges, ensuring you build a robust and functional application. Let's get started on bringing your AI-powered development assistant to life.

References

- Tauri Official Documentation
- Svelte Official Documentation
- Rust Programming Language Official Website

- Git Official Documentation
- Anthropic Claude Documentation
- OpenAI API Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.