ChatGPT

StoryArc – Product Requirements Document (PRD)

Introduction and Vision

StoryArc is a web application that empowers storytellers (screenwriters, novelists, attorneys, game designers, etc.) to craft modular, visual narratives. The goal is to provide a **drag-and-drop timeline** and **kanban-style board** for organizing story elements. Users can map out a storyline with arcs, scenes, and beats, visualizing the narrative flow and structure. This tool is developed in the emergent.sh AI-driven IDE, enabling a solo developer (with AI agent assistance) to rapidly build a production-ready app. The vision is to combine intuitive **timeline visualization**, hierarchical story organization, and real-time collaboration support to help creatives plan complex stories with ease.

Objectives and Goals

- **Visual Narrative Design:** Enable users to create and view story timelines in both horizontal and vertical orientations, showing how narrative events unfold over time.
- **Modular Structure:** Support a clear hierarchy of **Storyline → Arc → Scene → Beat**, so large stories can be broken into manageable parts. Users can expand/collapse or navigate this hierarchy easily.
- **Intuitive Drag-and-Drop:** Provide drag-and-drop editing for rearranging arcs, scenes, and beats on timelines or boards, making it simple to restructure narratives and see changes instantly.
- **Kanban Organization:** Offer an alternate kanban-style view to organize story components (e.g. arcs as columns and scenes as cards). This helps visualize story arcs in parallel and move scenes between arcs by dragging cards, or manage writing progress status in a familiar board layout.
- **Rich Metadata:** Allow users to annotate story events with details – each scene or beat can have a **metadata popup** showing its description, characters involved, location, time, notes, etc. This ensures important context is captured for each narrative beat.
- **Linked Story Elements:** Maintain databases of **characters**, **locations**, and **objects**. Users can link these to scenes or beats (like tagging which characters appear in a scene) for consistency. Clicking a character should highlight or list all scenes they're involved in, etc.
- **Templates for Structure:** Help users start with proven narrative structures. For example, include templates like **Dan Harmon's Story Circle** (8-step story embryo structure [1]), Hero's Journey, three-act structure, etc. Choosing a template will auto-generate arcs/scenes as placeholders that guide the user through that narrative formula.
- **Real-time Collaboration Ready:** Use Supabase as the backend with real-time sync so that updates to the story (adding/editing scenes, etc.) can reflect across clients instantly. This prepares the app for multi-user collaboration, even if initial version is single-user.
- **Fast Solo Development:** Leverage an AI-assisted development workflow (BMad method) and tools like Supabase and Shadcn UI to enable a lone developer to build quickly. Prioritize out-of-the-box solutions (auth, data storage, UI components) to reduce reinventing the wheel.
- **Future Extensibility:** Design the foundation such that future features (multi-user story sharing, AI story assistant suggestions, export to various formats) can be added without drastic changes.

User Persona and Use Cases

Target Users: StoryArc is aimed at creative professionals and hobbyists who plan narratives:
- *Screenwriters & Novelists* – for outlining plots, tracking subplots (arcs), and ensuring narrative beats align with structural frameworks (like Story Circle).
- *Game Designers* – for planning storylines in games, especially branching narratives or quests, by visually organizing plot arcs and events.
- *Attorneys & Legal Professionals* – for plotting case timelines or trial narratives, organizing evidence and testimonies as events on a timeline (with characters as witnesses, etc.), to build compelling courtroom stories.
- *Educators & Storytelling Coaches* – for teaching story structure using visual templates and allowing students to fill in their story beats.

Use Cases & Scenarios:
1. **Outline a New Story:** A novelist starts a new project, selects the "Hero's Journey" template, and StoryArc generates arcs (e.g. Ordinary World, Call to Adventure, etc.). The writer fills in scenes under each arc, drags them on the timeline to adjust pacing, and uses the kanban board to move scenes between acts.
2. **Manage Subplots:** A screenwriter creates multiple arcs within a storyline (main plot, subplot A, subplot B). On the timeline view, each arc is a separate row/lane; the writer drags scene cards along the horizontal timeline, aligning events from different arcs in chronological order. This helps visualize parallel story threads.
3. **Character Tracking:** A user adds a list of characters with bios. For each scene, they tag which characters and locations are involved. While reviewing, they click on a character (say, "Detective John") to see all scenes John appears in, ensuring the character's presence is consistent and their arc is coherent.
4. **Timeline of Events (Attorney):** A lawyer inputs key events of a case as beats with dates on a timeline (vertical view). Each beat has metadata (e.g. evidence, location, people involved). They drag beats to reorder if needed and use the metadata popups during trial prep to quickly recall details.
5. **Collaboration (Future):** Two game writers co-author a story. They share a project; as one adds a beat or edits a scene description, the other sees the update in real-time on their screen. They discuss structure in an integrated chat (future idea) or each take an arc to flesh out simultaneously.

Product Features

Hierarchical Story Structure (Storyline/Arc/Scene/Beat)

- A **Storyline** is the top-level container (equivalent to a project or script). It holds multiple **Arcs**.
- Each **Arc** represents a narrative thread or structural section (e.g. an act in a film, a subplot, or an attorney's line of argument). Arcs have titles and can be ordered.
- Each **Scene** belongs to an Arc. Scenes are major events or chapters. They contain one or more **Beats** (optional granular events or plot points). Scenes have titles, summary text, and potentially attributes like time or duration.
- **Beats** are the smallest unit – individual moments or events within a scene (e.g. a specific dialogue beat, a clue reveal, etc.). Beats have a short description.
- The UI will present this hierarchy in an outline form (like a collapsible tree or nested list) for quick navigation. Users can **create, rename, delete, or rearrange** arcs/scenes/beats. Drag-and-drop is enabled to move scenes between arcs or reorder beats within a scene, etc. The data integrity ensures beats move with their scene if the scene changes arc, etc.

- **Templating:** When starting a new Storyline, the user can choose a structure template. The app will then automatically create a set of arcs (and possibly placeholder scenes/beats) according to that template's formula (e.g. **Story Circle's eight stages** [1]). The placeholders will be labeled (e.g. "Need – The protagonist wants something") to guide the user, who can then fill in actual events.

Timeline Views (Horizontal & Vertical)

- **Horizontal Timeline:** A classic left-to-right timeline view. Time (or narrative progression) is on the X-axis. Each Arc is shown as a horizontal swimlane (row). Scene cards are plotted along the timeline in their respective arc's row. This allows the user to see at a glance how arcs interweave chronologically. Users can drag scene or beat cards along the timeline to adjust their sequence or timing. The timeline may use an arbitrary scale or sequence order (since fictional events may not have exact timestamps, an ordinal timeline is used).
- **Vertical Timeline:** An alternative top-to-bottom timeline view. In this orientation, time flows downward. Arcs could be represented as separate columns or lanes. This view might be useful for printing or for certain users (like legal timelines) who prefer vertical scrolling.
- **Timeline Controls:** Users can zoom in/out or switch units (e.g. switch between an "Act" timeline vs. a detailed scene timeline). They can toggle between horizontal/vertical layouts easily.
- **Event Display:** Each scene (or beat, if shown) appears as a **card/bar** on the timeline, labeled with its title and possibly an icon or color indicating its arc. Overlapping scenes (parallel events) will stack or appear on parallel lanes. Beats could be nested or shown as sub-items within scene cards (perhaps as smaller markers).
- **Drag-and-Drop on Timeline:** Users can drag a scene horizontally to change its chronological order. Dragging a scene vertically into a different arc lane will effectively reassign that scene to another arc (with a confirmation prompt). This interaction updates the underlying Arc/Scene relationship accordingly.

Kanban Board View

- The application provides a **Kanban-style board** as another way to organize story elements. In this view, each column represents either an Arc or another categorization (configurable, but by default we use Arc as the column). For example, each Arc is a column containing that arc's scenes as cards.
- Users can drag scene cards from one column to another, thereby moving the scene to a different arc. This gives a high-level overview of how scenes are distributed across arcs and allows rebalancing or reorganizing the narrative structure with a simple gesture.
- Alternatively, the user might switch the Kanban to a "status board" mode (future enhancement) where columns represent workflow states (To Do, Drafting, Editing, Done), and scene cards can be moved as they progress through writing stages. This would help writers track their progress. (In the initial version, focus is on arc-based columns, with potential to add status tracking later).
- Each scene card on the Kanban shows the scene title (and possibly a brief description or key icon like a character icon if main character present). Cards are color-coded or labeled by arc/story significance for quick visual distinction.
- The Kanban view complements the timeline: timeline is for chronological arrangement, while Kanban is for structural or status arrangement. Changes in one view reflect in the other (since they manipulate the same underlying data).

Metadata and Pop-ups

- Every Scene and Beat can store rich **metadata**. This includes: description (detailed text of what happens), a list of characters present, location, time (in story or actual date/time for real events), important objects, and notes (freeform text for ideas, foreshadowing, etc.).
- The UI will allow a quick view of this metadata via a popup or side panel. For example, clicking or double-clicking a scene card opens a **Scene Details** dialog. This dialog uses a Shadcn UI modal or drawer component to display all metadata fields in a user-friendly form. The user can edit these details here.
- **Character/Location linking:** Characters, locations, and objects are managed in their own lists (possibly accessible in a sidebar or separate section of the app). They can be created with attributes (name, description, image, etc.). In a scene's metadata, the user can add links to any number of characters, locations, or objects. This might be implemented via multi-select dropdowns (e.g. "Characters in this scene" dropdown shows all characters to pick from).
- When viewing a scene's metadata popup, the linked characters or locations are clickable – clicking one could bring up a mini-profile of that character or filter the timeline to highlight scenes involving that character. This **cross-linking** helps ensure consistency (e.g. you can check if an object introduced in early scenes is resolved later, by filtering to where that object appears).
- **Beat Details:** Beats, being smaller story moments, might have fewer metadata fields (perhaps just a short description, and optionally link characters/locations if needed). Beats could be edited inline on the timeline/kanban card (like a quick edit pencil icon) or via a similar popup.
- The metadata popups should follow a consistent design (using component library for inputs, tabs if needed for different sections of info) and should not obstruct the timeline too much – possibly appearing on the side or as overlays that can be easily closed.

Supabase Backend and User Accounts

- **User Authentication:** The app will use Supabase Auth for user signup/login. Each user has a secure account so their story projects are private. Initially it's single-user (each story belongs to one owner), but the design will allow adding collaborators in future.
- **Cloud Storage:** All data (stories, arcs, scenes, beats, characters, etc.) is stored in a Supabase PostgreSQL database. This means users can access their stories from anywhere by logging in, and data is persisted securely.
- **Real-time Updates:** Supabase's real-time capability will be leveraged so that any changes a user makes are pushed to their other devices or future collaborators instantly. For example, if a user opens the app in two browser tabs, adding a scene in one tab will immediately show up in the other. Supabase's **Realtime Postgres Changes** feature lets the client subscribe to insert/update events on tables [2], and it respects row-level security so users only receive updates for their own data [3]. This provides instantaneous feedback and lays groundwork for multi-user collaboration.
- **Performance and Scaling:** The app will fetch only necessary data (e.g. lazy-load beats when a scene is expanded) to keep performance snappy. Supabase can handle moderate loads and the data model is straightforward, so supporting dozens of arcs and hundreds of scenes per storyline should be feasible. We will ensure queries use indexes (e.g. foreign keys on IDs) for efficiency. As a solo-dev product, using Supabase avoids having to manage a custom server and provides high scalability out-of-the-box.

Non-Functional Requirements

- **Usability:** The interface must be clean and intuitive, as non-technical users (writers, etc.) will be using it. Minimal setup to start a project, in-app tips or onboarding for how to use timeline and kanban features. Drag-and-drop and editing interactions should be smooth and with visual feedback (e.g. highlight drop targets).
- **Responsiveness:** The application should work on various screen sizes. Ideally, it should be usable on tablets (touch support for dragging on a touchscreen). Mobile usage might be limited for complex drag-drop UI, but read-only or minor editing on mobile should be considered.
- **Reliability:** Use supabase's managed backend to ensure data reliability (ACID transactions, etc.). Implement auto-save on edits (each change is saved to DB immediately so no data loss if user closes the app suddenly).
- **Security & Privacy:** Enable Row-Level Security on all tables so each user's data is isolated and protected. Ensure that one user cannot access another's stories in the database. All communications should be over HTTPS. For now, the app is single-tenant per user; when collaboration is introduced, we will carefully extend permissions (e.g. an invited co-author will be added to a story's access list). Sensitive data (like a lawyer's case notes) must remain private to authorized users only. Supabase's RLS policies will enforce this (each row will have an `owner_id` and queries will be limited to those where `owner_id == auth.uid()` [4]).
- **Compatibility & Deployment:** The project should run in cloud IDEs like Replit without special hardware. That means using Node and browser-based tech. We avoid any heavy proprietary frameworks that don't run on Replit. The stack is essentially JavaScript/TypeScript for both front and back (with Supabase). We also ensure that the app can be containerized or deployed easily (for example, it could be hosted on Vercel or Netlify for the frontend, with Supabase as backend service).

Out of Scope (for Initial Version)

- **Text Editing for Script**: The app is for outlining/organization, not writing the full script content. We won't build rich text editors or screenplay formatting (though scene descriptions can be written in plain text).
- **Advanced Branching Narratives:** Branching story logic (like conditional paths in games) is not explicitly handled in the first version. The linear timeline assumes a mostly linear narrative (though multiple arcs can simulate parallel threads). In future, we might add features for branching paths or interactive stories.
- **In-app AI Writing Assistant:** While we plan for AI support later (e.g. an "AI Muse" that suggests plot points or checks story consistency), the initial product will not include generative AI features for content.
- **Third-Party Integrations:** Export to Final Draft or import from other tools, integration with writing software, etc., will be considered later. Initially, a basic export (like JSON or plain text outline) might be provided, but not a polished export to industry-standard formats.

Development Approach and AI Workflow (BMad)

This project is being developed by a solo developer with the assistance of specialized AI agents following the **BMad (Breakthrough Method for Agile AI-Driven Development) workflow**. In this methodology, each AI agent takes on a specific role to support both **ideation** (planning) and **implementation** of the project. The roles and their responsibilities are defined clearly as follows:

- **Analyst AI:** In the early stage, the Analyst agent helped research and clarify requirements. It analyzed the needs of storytellers across different domains and studied existing narrative tools. The Analyst ensured that the PRD covers all core user needs and that the solution will be competitive and useful. (For example, researching narrative structures like the Story Circle and identifying must-have features for timeline tools were guided by the Analyst.)
- **Product Manager (PM) AI:** The PM agent took the insights from the Analyst and shaped the Product Requirements Document. Acting as a product manager, it defined the scope, prioritized features, and made sure the product vision is clear. The PM agent ensured the PRD's objectives and features align with user goals and business value. It structured the PRD (this document) in a logical, comprehensive way, and set the acceptance criteria for each feature.
- **Architect AI:** Once the PRD was finalized, the Architect agent designed the technical solution outlined in the Architecture Document (below). The Architect translated the requirements into a concrete plan: defining the data schema, choosing the tech stack (Supabase, React with Shadcn UI, etc.), outlining the component structure, and planning how real-time sync and security will be handled. It ensured the design allows **fast development** (e.g. using reusable components, leveraging BaaS features) and future extensibility (for multi-user and AI features). The architect agent's output is a blueprint that the developer will follow.
- **Developer (Dev) AI:** With a solid PRD and architecture in place, the Dev agent will handle the coding tasks. Acting as a software engineer, it will implement the React front-end (using the Shadcn UI component system for consistency and speed) and set up the Supabase backend (creating tables, policies, and wiring up real-time listeners). The Dev agent will work within the emergent.sh IDE or Replit, writing code for each feature in short sprints (guided by the detailed plan). It will refer back to the PRD and Architecture docs to ensure the functionality matches the specifications. Where needed, the Dev agent may also write scripts (or Supabase Edge Functions) for things like applying templates or complex queries.
- **Quality Assurance (QA) AI:** The QA agent will assist in testing the application. It will generate and run test cases for critical user flows (creating a story, dragging scenes on the timeline, linking characters, etc.). QA will verify that acceptance criteria from the PRD are met and that the app is free of major bugs. It will test security measures (ensuring one user cannot fetch another's data, etc.), and performance (e.g. the app remains responsive with many timeline items). Any issues will be reported for fixes. The QA agent essentially acts as an automated tester and reviewer, ensuring the final product is robust and user-ready.

By clearly delineating these roles, the BMad AI-assisted workflow ensures that the project is thoroughly planned and executed, even with a solo human developer. The **Analyst/PM/Architect** phases produce a strong blueprint (which we have in this PRD and the following Architecture Document), and the **Dev/QA** phases ensure high-quality implementation. This approach accelerates development while maintaining clarity and quality at each step of the process.

StoryArc – Architecture Document

Architectural Overview

The StoryArc application is designed with a **modern full-stack JavaScript architecture** optimized for rapid development and future scalability. The architecture consists of:

- **Frontend:** A React-based single-page application (SPA) for the user interface, using the **Shadcn UI** component system (which builds on Radix UI and Tailwind CSS) for a consistent, accessible design language [5]. This provides a library of pre-built, customizable components (buttons, modals, drag-drop lists, etc.), allowing the solo developer to assemble the UI rather than design from scratch. The frontend is primarily responsible for rendering the timeline and kanban views, handling user interactions (drag-and-drop, clicks), and displaying modals/popups for metadata. State management can be handled with React Context or Zustand for things like the current story data loaded.

- **Backend:** Supabase (a Backend-as-a-Service built on PostgreSQL) serves as the primary backend. It provides the database, authentication, file storage (if needed), and real-time subscription services. We do not maintain a separate server; the React app communicates directly with Supabase's endpoints (via the Supabase JS client library). Supabase was chosen to offload typical backend tasks: it gives instant RESTful APIs and a reliable Postgres database with minimal setup, perfect for a solo dev scenario. Business logic that can't be done purely on the client (if any) could be implemented as Supabase Edge Functions (small serverless functions) or database stored procedures, but the aim is to keep it simple with mostly direct CRUD operations from the client.

- **Realtime Sync:** A key architectural feature is leveraging Supabase's real-time capabilities. The client will subscribe to changes on relevant tables (stories, arcs, scenes, etc.) so that any updates are pushed to the UI instantly. Under the hood, Supabase's realtime uses Postgres logical replication to listen for inserts/updates and broadcasts them via WebSockets to connected clients [2]. This eliminates the need for a custom WebSocket server. We will set up listeners in the React app that update state when, say, a new scene is inserted by the user (or by a collaborator in the future). Because Supabase integrates with Postgres RLS, these realtime messages are filtered by security policies – users will only receive updates for data they are allowed to read [3]. This architecture ensures the app is ready for multi-user collaboration without extra complexity.

- **Development/Hosting Environment:** During development, the emergent.sh IDE and Replit provide an environment to code and test the app. The app can be run locally (in the Replit container) with a development server (e.g. using Vite or Create React App for React). For production, the frontend can be built as a static bundle and deployed on a platform like Vercel or served via any static host, since it's just an SPA. The Supabase backend is a cloud service, so no backend server deployment is needed apart from configuring the Supabase project. Replit compatibility has been considered: all code is in JavaScript/TypeScript, and no build step requires proprietary OS-specific tools. The real-time and database connection use standard web protocols which Replit supports. Environment variables (Supabase keys, etc.) will be used in Replit to keep secrets out of code.

Diagrammatically, the architecture can be seen as:

User ⇆ React App (Shadcn UI components) ⇆ Supabase (DB + Auth + Realtime).

The React App contains all UI/UX logic, and it communicates with Supabase via JS SDK calls (HTTPS for REST or WebSocket for realtime). Supabase handles storing data and enforcing security rules. There is no separate application server in between, which accelerates development and reduces maintenance.

Data Schema (Database Design)

The database schema is centered around the story hierarchy and linked entities. All data is stored in **PostgreSQL** tables on Supabase. Below is the schema with key tables and relationships:

- **users** – Supabase provides an `auth.users` table for authenticated users. We may have a public profile table if needed, but for our purposes, we mainly use the user's UUID from Supabase Auth as the `user_id` foreign key in other tables to denote ownership. Each data record will reference the user who owns it (for RLS enforcement).

- **stories** – Represents a Storyline (top-level narrative project).

- **Fields:** `id` (UUID primary key), `user_id` (UUID, references auth.users.id), `title` (text), `description` (text, optional longer synopsis), `created_at` (timestamp).
- **Relationships:** A story has many arcs. `user_id` indicates the owner. In the future, a separate join table (e.g. `story_collaborators`) could list additional user_id with access.

- **arcs** – Represents a Story Arc or major section, tied to a story.

- **Fields:** `id` (UUID PK), `story_id` (FK to stories.id), `title` (text, e.g. "Main Plot" or "Act I"), `order_index` (integer for ordering arcs within a story), `color` or `label` (optional styling for UI).
- **Relationships:** arc belongs to a story; story deletion cascades to arcs. An arc has many scenes.

- **scenes** – Represents a Scene (or chapter/event) within an arc.

- **Fields:** `id` (UUID PK), `arc_id` (FK to arcs.id), `title` (text), `description` (text, the detailed narrative or notes for the scene), `order_index` (integer ordering within the arc), `timestamp` (optional – could store a narrative chronological marker or date for real events), `metadata_json` (JSONB, to store any additional metadata like custom fields).
- **Relationships:** scene belongs to an arc (and via arc to a story). Scenes have many beats. If a scene is moved to a different arc, we update its `arc_id`. Could have a composite index on (arc_id, order_index) for quickly sorting scenes.

- **beats** – Represents a Beat, the smallest narrative unit under a scene.

- **Fields:** `id` (UUID PK), `scene_id` (FK to scenes.id), `summary` (text, a brief description of the beat), `order_index` (integer order within the scene), possibly `timestamp` or `offset` (if we want to timestamp beats relative to scene).
- **Relationships:** beat belongs to a scene. We might cascade delete beats if a scene is deleted. Beats can be re-ordered by changing their order_index.

- **characters** – Represents a character entity that can be linked to scenes.

- **Fields:** `id` (UUID PK), `story_id` (FK to stories.id, indicating this character belongs to a specific story project), `name` (text), `description` (text, bio or notes), maybe `avatar_url` (text if linking to an image in Supabase storage).
- **Relationships:** characters are scoped to a story. One story has many characters. (We might also allow global characters across stories for the same user, but likely each story has its own cast.)

- **locations** – Similar structure to characters.

- **Fields:** `id` (PK), `story_id`, `name`, `description`. (Could also have coordinates or image if relevant, but not initially.)
- **Relationships:** one story has many locations.

- **objects** – Tangible items of significance in the story (e.g. "Magic Sword", "Murder Weapon").

- **Fields:** `id` (PK), `story_id`, `name`, `description`.
- **Relationships:** one story has many objects.

- **scene_characters** – A join table to link characters to scenes (many-to-many relationship, since a scene can have multiple characters and a character appears in multiple scenes).

- **Fields:** `scene_id` (FK to scenes), `character_id` (FK to characters). Both as composite PK (or add an id).
- We will typically query this to find which characters are in a scene or which scenes a character is in. Similarly, we can have `scene_locations` and `scene_objects` join tables for linking locations/objects to scenes. *(Alternatively, we could use a single join table like* `scene_entities` *with a type field for char/loc/obj, but separate tables make queries simpler and enforce proper foreign keys.)*

Entity Relationships: In summary, `stories -> arcs -> scenes -> beats` form a hierarchy. Characters/locations/objects all link back to a story, and scenes link to them via join tables. The `user_id` ownership is at least stored at story level (each story knows its owner). For convenience and security, we might also include `user_id` on arcs, scenes, etc., redundantly (ensuring it's set to the story's owner) to simplify writing RLS policies (so that every table can have an RLS check on user_id). This duplication can be maintained via triggers or just set on creation from the client.

All tables have `created_at`/`updated_at` timestamps (Supabase can auto-generate those) to track changes.

Row-Level Security (RLS): We will enable RLS on every table containing user data. The basic policy for each will be along the lines of: *allow the authenticated user to select/insert/update/delete rows where* `user_id = auth.uid()`. For example, on the `stories` table:

```
ALTER TABLE stories ENABLE ROW LEVEL SECURITY;
CREATE POLICY "Allow story owner full access" ON stories
FOR ALL USING (user_id = auth.uid()) WITH CHECK (user_id = auth.uid());
```

Similarly, for child tables like arcs or scenes, if we carry `user_id`, we use the same pattern. If not carrying `user_id` on a child, we can join through the story (e.g. policy on arcs could check that the arcs.story_id has a story with user_id = auth.uid()). Supabase provides JWT functions like `auth.uid()` to get the current user's ID [4]. Initially, only owners can access their data. In the future for collaboration, we might extend policies to allow shared access (for example, having a `story_collaborators` table and writing policies to allow users listed there). The current approach, however, isolates each user's data entirely, which is simplest and most secure for a solo use scenario.

Application Components (Frontend Structure)

The React frontend will be organized into modular components corresponding to the major parts of the UI. We will utilize a component-based architecture, making use of the Shadcn UI library components where possible and creating custom components for app-specific functionality. The tentative component structure is as follows:

- **App (Root Component):** Initializes Supabase client (using the provided URL and anon key), sets up context providers (for auth state and perhaps a global store of the current story data). It contains the routing if we have multiple pages/views (e.g. a login page, a main app page). Once logged in, the main interface is shown. The App may use a layout with a sidebar (for story list, characters list, etc.) and a main workspace.

- **StoryDashboard Page:** The main page when a story is open. This could be a page component that includes the toolbar (buttons to switch views, toggle template, etc.) and either the Timeline view or Kanban view based on user selection. It manages high-level state like "currentView = timeline or kanban".

- **TimelineView Component:** Responsible for rendering the timeline visualization.

- Internally, it may generate subcomponents like **TimelineGrid** (the structure of lanes and time axis if needed) and **TimelineLane** for each Arc.
- Each **TimelineLane** takes an arc and renders that arc's scenes in the horizontal or vertical timeline. Scenes can be represented by **SceneCard** components placed at positions relative to their order/ timestamp. We might use a library or custom logic to make these draggable. When a SceneCard is dragged within the same lane, its order_index is updated; if dragged to another lane, its arc_id changes (and order_index re-computed).

- The timeline might have a component for a time scale or markers if needed (e.g. showing act boundaries or dates). This could be a simple ruler on top or side, but for now, order-based positioning might not require a detailed scale beyond sequential slots.

- **KanbanView Component:** Manages the kanban board interface.

- It contains multiple **KanbanColumn** components, one per Arc (or per status, if that mode is used). The column displays a header (arc title) and a list of **SceneCard** components. We might use something like the HTML5 Drag and Drop API or a React drag-drop library (like `@dnd-kit` or `react-beautiful-dnd`) to handle reordering and moving cards between columns.
- A **SceneCard** in Kanban shows minimal info (title, perhaps an icon if it has beats or a character count). It should be a reusable component possibly shared with the timeline (styled differently depending on context). Shadcn's card or draggable primitives can be utilized here.

- **SceneCard Component:** Represents a scene in a visual form (used in timeline and kanban).

- It will be a styled card (Tailwind/Shadcn styling) that can display the scene title and maybe a short summary on hover. If the scene has beats, we might indicate that (e.g. a small stack icon or a number of beats).
- SceneCard has events for onClick (to open metadata popup), onDragStart/End for drag-drop operations. It receives props for its current arc and position.

- **BeatItem Component:** If we decide to visually represent beats inside scenes (perhaps as sub-items or bullet points in a scene's card or in the metadata panel), BeatItem will handle that. However, on the timeline, we might not individually drag beats (they move with their scene). Beats might primarily be managed in the Scene detail view instead of directly on the timeline (to keep the timeline visually clear).

- **Metadata Modal Components:**

- **SceneModal** (or SceneDetailsPanel): A component that renders the detailed form of a scene's metadata. It includes fields for title (editable), description (textarea), list selectors for characters/locations/objects (multi-select checkboxes or tokens), and a list of Beats. Beats can be created, edited, or reordered here (maybe as a simple list of text items with add/remove buttons).
- This modal uses Shadcn UI's dialog or sheet component for a consistent look. It is triggered when a SceneCard is double-clicked or when a context menu "Edit Scene" is clicked.
- **CharacterModal, LocationModal, ObjectModal:** Forms to add/edit these entities. Likely simpler (just name, description, image URL).
- We may also have a **ConfirmDialog** for actions like "Are you sure you want to delete this scene (and all its beats)?" – using a Shadcn confirmation dialog pattern.

- **Sidebar Components:** The app could have a sidebar showing the list of all characters, locations, objects for quick access. For instance, a **CharacterList** that lists character names; clicking one filters timeline to highlight scenes with that character. This might be an advanced feature (if time permits).

The sidebar could also list all arcs/scenes in text form (outline view) as an alternate navigation, enabling quick jumping or reordering via drag in a tree structure.

- **Top Bar / Menu:** A top navigation or toolbar might include: a dropdown to switch stories (if multiple story projects exist), buttons to toggle Timeline/Kanban view, an "Add" menu to create new arc/scene, an "Import Template" action, and user account menu (profile, logout). We'll use Shadcn UI components like dropdown menus, icons, etc., here.

- If in the future we integrate an AI assistant, a button like "Ask AI for suggestion" could live here too (opening perhaps a chat panel). This is noted for extensibility.

- **State Management:** We will likely keep the data in React state using hooks. A global store (using Context or Zustand) can hold the current story's data: e.g. an object with {arcs: [...], scenes: [...], characters: [...]}. The components will subscribe to relevant parts. When the Supabase realtime listener fires (e.g. a new scene added), we update this state. For initial simplicity, we might fetch all data for a story on load and then listen for changes. Alternatively, we could fetch arcs, then fetch scenes for each arc, etc., but since the dataset is not huge, a single bulk fetch with joins (or multiple quick fetches) is fine.

- **Routing:** If using React Router or similar, we might have routes like `/stories/:storyId` for the main editor, and `/login` for auth. Supabase auth can handle session persistence; the App component will check if user is logged in (maybe via Supabase's auth on start) and redirect accordingly.

The component design emphasizes reusability and clarity. For example, SceneCard is used in two places (timeline, kanban) but defined once. The use of Shadcn UI ensures base components like modals, lists, buttons have a consistent style and behavior (since Shadcn builds on accessible Radix UI primitives). This saves development time and ensures a professional UI without a dedicated designer.

Backend API and Integration

With Supabase, we rely mostly on its auto-generated APIs and client library rather than writing a custom API server. The React app will interact with the backend as follows:

- **Supabase Client:** In the frontend, we instantiate a Supabase client with the project URL and anon key. We use this for all database operations and auth. For example, to fetch data: `supabase.from('scenes').select('*').eq('arc_id', someArcId)` will retrieve scenes for an arc. Similarly, `insert`, `update`, and `delete` calls are made through this client. These calls invoke Supabase's REST API under the hood. Because of RLS, the requests include the user's JWT from login, and only authorized data is returned.

- **Fetching Initial Data:** When a user opens a story, the app will fetch all relevant data. We have options:

- Use Supabase's ability to select and filter: e.g. fetch arcs with `select * where story_id = X` and for each arc fetch scenes, etc. We might perform multiple queries (one for arcs, one for scenes, one for beats, characters, etc.). This is straightforward and can run in parallel.
- Alternatively, use a single call with foreign table embedding if enabled (Supabase allows a syntax like `.select("*, scenes(*, beats(*)), arcs(*), characters(*)")` using PostgREST feature). However, that can get complex. A simpler approach is fine for now.

- The data will be stored in state once fetched.

- **CRUD Operations:** Each user action translates to a database operation:

- Creating a new arc -> `supabase.from('arcs').insert({story_id: X, title: "...", order_index: N})`. On success, the realtime subscription will also soon notify the app of the new arc (or we can optimistically update the state immediately).
- Reordering scenes -> perhaps multiple updates (if we need to swap order indexes). We can handle ordering on the client by computing new order values and send updates. (Alternatively, use an ordering scheme that avoids needing to update many records at once, e.g., use floating point or timestamps to allow insert between, but that might be overkill. Simpler to re-number on drop and batch update).
- Deleting a scene -> `supabase.from('scenes').delete().eq('id', sceneId)`. We have foreign key ON DELETE CASCADE for beats so they go too, and similarly join table entries.

- Linking a character to a scene -> `supabase.from('scene_characters').insert({scene_id: A, character_id: B})`. Or removing via delete. The UI will call these based on user toggling a link.

- **Supabase Functions (optional):** For template insertion, we might implement it client-side: e.g., upon selecting a template, the client inserts the predefined arcs and scenes in sequence. This is fine for initial version. However, to ensure atomicity (all or nothing) and maybe reuse, we could create a Postgres function or use Supabase's RPC mechanism: e.g., an RPC call like `apply_template(story_id, template_name)` which performs a series of inserts in one transaction. This would be more advanced and possibly done later. Initially, the client can perform a few insert calls in succession (given one developer, ease of implementation is key).

- **Edge Functions:** If we find any logic that is better done server-side (for security or complexity), we'll consider a Supabase Edge Function. For example, if generating an export file (PDF/markdown) or complex search, an Edge Function could be used. The architecture leaves this possibility open, but none are strictly required for MVP.

- **External Integrations:** Not planned for v1. But architecture-wise, if in future we integrate with an AI API (for story suggestions) or need to send emails (for invites), those calls would be made from the client or an Edge function. Using the client for AI (with API keys) is possible but not secure to expose keys; likely we'd use a secure function for that. We note this as a future extension point.

Security and Permissions Strategy

Security is critical since user content can be sensitive. We've covered Row-Level Security policies under data schema. Here's a recap and additional measures:

- **Authentication:** Only authenticated users can access the main app. Supabase's auth will manage this (we'll use a pre-built Login component or redirect to Supabase's auth UI). The session token is stored and used for all requests. We will not allow anonymous access to any data reads or writes on the database; all tables will require an authenticated role. For example, we configure Supabase so that anon (unauthenticated) cannot read any of the story tables. The only open access might be a public "templates" table if we have one with generic template definitions (that could be readable by anyone), but even that is optional.

- **Row-Level Security:** Enabled on story content tables as described. Example policy snippet for scenes:
  ```
  CREATE POLICY "Scene owner can do all" ON scenes
     FOR ALL USING (auth.uid() = user_id) WITH CHECK (auth.uid() = user_id);
  ```
  (Assuming we propagate user_id to scenes). If not, a slightly more complex policy checking via join on story is needed. But to keep it simple, we will duplicate user_id on arcs, scenes, etc. upon creation (the client knows current user's ID from the auth session). This duplication costs a bit of extra storage but simplifies RLS logic.
  With these policies, even if a malicious client tried to request another story's ID explicitly, the query would return zero rows. The realtime subscription likewise won't send events for others' data because of RLS filtering.

- **Storage Security:** If we use Supabase Storage (for uploading images for characters or so), we'll secure buckets with policies as well. For instance, a bucket "avatars" can enforce that file paths must contain the user's ID or specific story ID. For initial version, we might not have file uploads, but it's part of extensibility.

- **Front-end Authorization Checks:** We will also ensure on the front-end that we only request or manipulate data for the current story/user. This is secondary (the primary guard is the backend RLS). But for example, we won't even show the UI for switching stories that don't belong to the user. Also, any attempt to load data is scoped by user's session automatically via Supabase.

- **Preventing Data Loss:** Use database transactions for multi-step operations if possible. For instance, if the user deletes a story, that might cascade delete a lot of data. We might implement a "trash" or at least a confirmation to avoid accidental mass deletion. Supabase can handle cascades, but we ensure the user intent is clear.

- **Dependency security:** Shadcn UI components and other libraries are open-source; we'll keep them updated. Supabase libraries are official and maintained. We'll pin versions to avoid breaking changes.

Real-time Sync Implementation

Real-time synchronization is implemented through Supabase's real-time channels. Here's how we plan to set it up in the app:

- Upon loading a story, after initial data fetch, the app will subscribe to Postgres changes on the relevant tables for that story. Supabase allows filtering events by for example: `supabase.channel('schema-db-changes')...`. We can subscribe in a few ways:
- **Channel per story:** Join a realtime channel for each story ID, with filters so that we only get events for that story. For example, subscribe to `INSERT`/`UPDATE`/`DELETE` on tables arcs, scenes, beats where story_id matches the current story. This could be done by using Postgres function triggers that publish to specific topics, or simpler, use the Supabase client's built-in filtering: (Supabase's JS client can filter in the `.on()` call by specifying an `eq: {story_id: currentId}` condition for each table subscription).
- Alternatively, subscribe to user-wide changes and filter in client: e.g. subscribe to all `stories` events for user's own. But more efficient is to target by story.

- When an event is received (say a new scene is inserted by our own action or by another collaborator), the client will update the local state accordingly. Supabase real-time payloads include the new record and type of change. Our handler will, for example, on `INSERT` to scenes: add the new scene to the scenes state list (if it matches current story), on `UPDATE`: merge changes (like if scene title changed), on `DELETE`: remove it from state. This will automatically cause React to re-render the timeline/kanban with updated data. The UI might also show a subtle notification for changes coming in.

- **Conflict resolution:** Since initially only one user edits their data, conflicts (two people editing same field) are not an issue. In future multi-user, the "last write wins" approach of the database will be our default; we might later implement more complex conflict handling or locking if needed, but likely not necessary for story outlining (collaborators can coordinate who edits what, or we could even eventually add operational transform if doing simultaneous script editing, but that's beyond scope now).

- **Testing realtime:** We will test that if the same user opens two sessions, operations sync correctly. This also tests that the subscription is correctly limited by RLS (one user shouldn't get another's events). Given Supabase's feature set, we are leveraging proven infrastructure instead of writing our own socket server, which again accelerates development.

Tech Stack and External Libraries

To summarize the chosen tech stack and justify each part in terms of the solo-dev, fast development requirement:

- **React + Vite:** React is used for its component model which fits our needs (dynamic lists, interactive DnD, modals). Vite (or Next.js or CRA) could be used; Vite is likely for simplicity in Replit (fast bundler, zero-config). This gives hot-reload during development for quick iteration.

- **TypeScript:** We will likely write in TypeScript for type safety, especially interacting with Supabase (which can generate types from the DB schema). This prevents common mistakes and serves as documentation. The emergent.sh IDE supports TS.
- **Supabase (Postgres, Auth, Storage, Realtime):** We offload as much backend work as possible here. In addition to points discussed, Supabase also gives us a nice web UI to manage the database (for debugging or manually seeding template data).
- **Shadcn UI (Radix + Tailwind):** Using this modern component library means we avoid spending time on styling basic UI controls. Shadcn components are copied into our project (so we can tweak them if needed) and give us accessible, well-tested building blocks [6]. Tailwind CSS utility classes will be used to handle custom layout styling (e.g. positioning timeline elements). This is faster than writing lots of custom CSS.
- **Drag and Drop Library:** Possibly `@dnd-kit` or `react-beautiful-dnd` to implement the drag/drop. These abstract away the details of HTML5 drag events and make reordering lists or moving items between lists simpler. We will choose one that is compatible with our component structure. For timeline dragging, we might handle it manually if needed (calculating positions), but a library could still help with the draggability and drop detection.
- **State Management:** As mentioned, likely React Context or a lightweight global store library. Redux is probably overkill; we prefer simplicity.
- **Testing Tools:** For QA, if writing automated tests, we could use Jest (for any pure functions or util modules) and perhaps React Testing Library for component testing. However, given the focus on AI QA, we might rely on the QA agent to do scenario testing through the UI (possibly using headless browser or just logically stepping through state changes).

This stack ensures that as a solo developer, we can move quickly: no need to build auth (Supabase does it), no need to design custom UI kit (Shadcn provides), no need for complex server code (all in client and DB). Everything is JavaScript/TypeScript, which avoids context switching between languages.

Extensibility and Future Plans

We've consciously designed the system to accommodate future enhancements without large refactoring:

- **Multi-User Collaboration:** The initial design already isolates user data and uses real-time updates. To allow multiple users on one story, we would introduce a **collaboration model**: e.g. a story may have a list of allowed users. Technically, we'd add a join table `story_collaborators (story_id, user_id, role)` and adjust RLS policies to allow those users to read/write that story's data. The frontend would include a UI for the story owner to invite others (maybe via email). Because our data model ties everything to story_id and story has an owner, we might extend it so that either the owner's ID or collaborators' IDs are permitted. Supabase's RLS can handle this with an `OR` in the policy (check if auth.uid() is in a subquery of collaborators). The real-time subscriptions remain nearly the same – collaborators would subscribe to changes on the story they share. Our UI already can handle updates from others. Minor tweaks like locking perhaps if two users try to edit the exact same text could be considered, but often last-save-wins is acceptable at outline level.

- **AI Assistant Integration:** Given emergent.sh is an AI-centric IDE, integrating AI features in the app is likely. Some possible additions:

- An “AI Suggest” button on a scene could call an OpenAI API (or local model) to generate suggestions for what could happen next in that scene or to brainstorm a beat. This would likely be implemented via a Supabase Edge Function (to keep API keys safe) or directly from client if using an open API. The architecture can accommodate this by adding a module that calls the API and returns suggestions to the UI (displayed maybe in a sidebar or a modal). No fundamental change to data model needed – perhaps a new table for AI suggestion history if we want to store them.
- AI consistency checks: an agent could read through the entire story data and produce a report (this can also be done via a background function).

- None of these conflict with the current design; they are additive. The clean separation (frontend vs DB) and use of an open API like Supabase means we can plug such features in without major rewrites.

- **Exports and Integration:** We anticipate users might want to export their work. Since the data is structured, we can create exporters:

- For example, export to PDF or Word: likely generate a document with each scene and its details. We might use a library like jsPDF or an Edge Function with a PDF service. Or provide a simple Markdown export (which users can convert as needed). Because we have a clear story->arc->scene breakdown, writing an exporter is straightforward by traversing that structure.
- Integration with other apps: possibly allow exporting to timeline tools or importing from them. Our data model is basic enough to map from/to JSON, which is good.

- We would design export functions to operate either in the client (for small text exports) or on server for heavier tasks. The architecture being modular means we can add an “Export” component or a page for selecting format and generating output without touching the core.

- **Scalability Considerations:** If the user base grows or the app gets more complex:

- The tech choices scale up: Supabase can scale the DB and has generous limits; if needed we could self-host Postgres. The front-end can be served via CDNs for many users.
- If some logic becomes heavy on client (say computing a very large timeline), we can optimize by pagination or virtualization (rendering only what’s in view).

- For now, given solo dev, we assume moderate usage, but it’s good to know the stack can grow.

- **Testing and QA Automation:** In future, we might integrate a test suite that runs when deploying new versions. The QA agent could be leveraged to run through critical flows in a staging environment. The architecture doesn’t hinder adding e2e tests (we could use something like Cypress for UI testing down the road).

Deployment Plan

For completeness, a quick note on deployment: The app can be containerized as a Node app serving the built files (or simply deploy static files to a static host). Supabase requires no deployment aside from using their cloud service (or a self-host instance). Environment variables (Supabase project URL, anon key) will be set in the deployment environment. We’ll also include a toggle for dev vs prod (different Supabase keys perhaps). Using Replit for hosting is possible (Replit could run a web server for the React app), but likely we use Replit mainly for development and testing. For production, a service like Netlify or Vercel is ideal for hosting the frontend, and it can connect to the live Supabase backend.

Continuous integration isn't a big concern for a solo project, but version control (GitHub) will be used to manage the code. The emergent.sh environment likely integrates with git as well, or we ensure to export code out for versioning.

Conclusion

The architecture of StoryArc balances **immediacy** (using high-level services and libraries to get results fast) with **extensibility** (a clear data model and modular components ready for multi-user and AI enhancements). By assigning distinct AI agents to planning and development roles (Analyst, PM, Architect, Dev, QA), we have produced a detailed plan that a solo developer can confidently execute. The end result will be a robust web application where storytellers can visually craft and manage narratives. The system is secure, real-time collaborative, and built on a modern tech stack that will allow it to grow in functionality. With this blueprint in hand, the development can proceed with clarity on what to build and how each part should work, ensuring a smooth path from idea to a working product.

---

[1] Storytelling Guide: Dan Harmon Story Circle Explained
https://www.studiobinder.com/blog/dan-harmon-story-circle/

[2] [3] Realtime - Postgres changes | Supabase Features
https://supabase.com/features/realtime-postgres-changes

[4] Row Level Security | Supabase Docs
https://supabase.com/docs/guides/database/postgres/row-level-security

[5] [6] My Tiny Guide to Shadcn, Radix, and Tailwind | by Mairaj Pirzada | Medium
https://medium.com/@immairaj/my-tiny-guide-to-shadcn-radix-and-tailwind-da50fce3140a