ChatGPT

Plano Técnico de Implementação das Funcionalidades

Ordem de Implementação

Para otimizar o desenvolvimento, a seguinte ordem é recomendada, considerando dependências e complexidade:

1. **Timeline Visual do Projeto:** Implementar primeiro a timeline visual das fases do projeto. É um componente independente de UI e servirá de base para visualizar o cronograma gerado no passo seguinte. Além disso, não depende de outras novas funcionalidades, apenas de receber dados (fases e datas).
2. **Geração Automática de Cronograma (Briefing):** Em seguida, implementar a lógica que gera automaticamente as fases do cronograma a partir dos dados do briefing. Essa funcionalidade utilizará o componente de timeline (para exibir a prévia) e integrará com o estado global (armazenando o cronograma gerado no projeto criado).
3. **Sistema de Versões de Vídeo (Comparativo Visual):** Por último, implementar o versionamento de vídeos com comparativo visual. Essa parte é mais complexa e depende do estado global já ter o projeto criado. Embora seja independente da timeline, ela interage bastante com o Zustand (armazenando versões no projeto) e com a interface (lista de versões, players de vídeo, etc.), portanto fica por último após estabelecer a estrutura básica do projeto.

(Racionalização: Timeline é principalmente front-end visual, rápido para testar. Cronograma automático precisa do Timeline para mostrar resultado e de campos no store para salvar dados. Versões de vídeo altera bastante o store e interface, então implementamos por último para construir em cima da base pronta.)*

Dependências e Bibliotecas Necessárias

Antes de codificar, instale quaisquer bibliotecas utilitárias necessárias:

- **date-fns:** Biblioteca para manipulação de datas (cálculos de dias, formatação). Será utilizada na geração do cronograma e formatação de datas.
  Comando de instalação:

  ```
  npm install date-fns
  ```

  (Caso prefira Yarn: `yarn add date-fns`*.)*

Observação: Outras libs citadas como melhorias (por exemplo, visx/recharts para timeline avançada, ou integrações de upload real) não serão instaladas neste momento, pois usaremos soluções simples integradas. Certifique-se de que o projeto já tenha o **Zustand** instalado e configurado (conforme mencionado, ele já é usado como store global). As demais funcionalidades (React Hook Form, etc.) podem já existir no projeto, mas não são estritamente necessárias para executar os passos abaixo.

Implementação Passo-a-Passo

1. Timeline Visual do Projeto

Objetivo: Criar um componente visual para exibir a linha do tempo do projeto (fases, entregas e prazos) de forma horizontal e proporcional às datas. Cada fase será um segmento colorido indicando se está completa, em andamento, pendente ou atrasada. Vamos criar o componente e integrá-lo na página de detalhes do projeto.

- **Criar componente Timeline:** Crie o arquivo `components/widgets/Timeline.tsx` com o seguinte conteúdo:

```
"use client";

import React from "react";

// Definição da estrutura de uma fase do projeto
interface Phase {
  name: string;
  plannedStart: Date;
  plannedEnd: Date;
  completed: boolean;
}

interface TimelineProps {
  phases: Phase[];
  finalDueDate?: Date;  // prazo final do projeto (opcional)
}

const Timeline: React.FC<TimelineProps> = ({ phases, finalDueDate }) => {
  if (!phases || phases.length === 0) return null;

  // Ordena as fases por data de início para garantir ordem cronológica
  const sortedPhases = [...phases].sort(
    (a, b) => a.plannedStart.getTime() - b.plannedStart.getTime()
  );
  const timelineStart = sortedPhases[0].plannedStart;
  const lastPhaseEnd = sortedPhases[sortedPhases.length - 1].plannedEnd;
  // Determina o fim da timeline (considera prazo final se for após a última
fase)
  const timelineEnd =
    finalDueDate && finalDueDate > lastPhaseEnd ? finalDueDate :
lastPhaseEnd;
  const totalDurationMs = timelineEnd.getTime() - timelineStart.getTime();

  // Calcula a largura proporcional de cada fase em relação ao tempo total
  const columns: string[] = sortedPhases.map((phase) => {
    let durationMs = phase.plannedEnd.getTime() -
phase.plannedStart.getTime();
    if (durationMs <= 0) {
```

```
      // Fases instantâneas (mesmo dia) recebem um mínimo de duração visual
      durationMs = 12 * 60 * 60 * 1000; // 12h em ms (~0.5 dia)
    }
    const fraction = durationMs / totalDurationMs;
    const fractionStr = (fraction * 100).toFixed(2);
    return `${fractionStr}fr`;
  });
  const gridTemplate = columns.join(" ");

  // Define cor de fundo da fase conforme status e datas
  const getPhaseColorClass = (phase: Phase) => {
    if (phase.completed) return "bg-green-600";          // concluída: verde
    const now = new Date();
    if (phase.plannedEnd < now) return "bg-red-600";     // não concluída e
já passou do fim: vermelho (atrasada)
    if (phase.plannedStart <= now && phase.plannedEnd >= now) {
      return "bg-yellow-500";                            // em andamento:
amarelo
    }
    return "bg-blue-600";                                // futura/pendente:
azul
  };

  // Calcula posição do marcador de prazo final (linha vertical) se houver
prazo
  let finalDueMarkerStyle: React.CSSProperties | undefined;
  if (finalDueDate) {
    const startTime = timelineStart.getTime();
    const endTime = timelineEnd.getTime();
    const dueTime = finalDueDate.getTime();
    if (dueTime >= startTime && dueTime <= endTime) {
      const pct = ((dueTime - startTime) / (endTime - startTime)) * 100;
      finalDueMarkerStyle = { left: `${pct}%` };
    } else if (dueTime < startTime) {
      finalDueMarkerStyle = { left: "0%" };
    } else if (dueTime > endTime) {
      finalDueMarkerStyle = { left: "100%" };
    }
  }

  return (
    <div className="relative w-full p-4">
      {/* Barra de timeline usando CSS Grid */}
      <div
        className="grid items-center gap-1 w-full"
        style={{ gridTemplateColumns: gridTemplate }}
      >
        {sortedPhases.map((phase) => {
          const colorClass = getPhaseColorClass(phase);
          const phaseEndsAfterDue = finalDueDate && phase.plannedEnd >
finalDueDate;
```

```
          return (
            <div key={phase.name} className={`${colorClass} relative h-8`}>
              {/* Nome da fase centralizado + indicadores */}
              <span className="text-xs text-center whitespace-nowrap">
                {phase.name}
                {phase.completed && " ✔"}
                {/* Indicador de atraso */}
                {!phase.completed && phase.plannedEnd < new Date() && (
                  <span className="ml-1 text-red-200">(atrasado)</span>
                )}
                {/* Indicador de extrapolação do prazo */}
                {phaseEndsAfterDue && (
                  <span className="ml-1 text-yellow-200">(!)</span>
                )}
              </span>
            </div>
          );
        })}
      </div>
      {/* Marcador vertical do Prazo Final */}
      {finalDueMarkerStyle && (
        <div
          className="absolute top-0 bottom-0 border-l-2 border-yellow-400
opacity-70"
          style={finalDueMarkerStyle}
          title="Prazo final"
        />
      )}
      {/* (Opcional) Poderíamos adicionar um marcador "hoje" aqui para
referência */}
    </div>
  );
};

export default Timeline;
```

Detalhes: O componente está marcado com `"use client"` porque depende de valores do estado (ou props com dados do cliente). Ele recebe um array de fases (`phases`) e um prazo final opcional. Usamos **CSS Grid** para criar colunas proporcionais à duração de cada fase (definidas em fracao do total). A cor de cada segmento indica status (verde = completo, amarelo = em andamento, azul = futuro, vermelho = atrasado). Incluímos pequenos indicadores de texto: "✔" para fases concluídas, "(atrasado)" se a fase já ultrapassou a data de término atual sem estar completa, e "(!)" se a fase termina após o prazo final estipulado. Também desenhamos uma linha vertical semitransparente no ponto do prazo final para referência visual.

- **Integrar Timeline na página de projeto:** Abra (ou crie, caso não exista) a página de detalhes do projeto, presumivelmente em `app/events/[eventId]/page.tsx` (ou similar, conforme a estrutura do seu App Router). Insira a importação e o uso do componente Timeline. Por exemplo, ajuste o código assim:

```tsx
"use client";

import { useProjectsStore } from "@/store/useProjectsStore";
import Timeline from "@/components/widgets/Timeline";
// ... (outras imports, como componentes de versão que adicionaremos depois)

export default function EventDetailPage({ params }: { params: { eventId:
string } }) {
  // Obtém o projeto atual (por simplicidade, usamos um estado global para
currentProject)
  const event = useProjectsStore((state) => state.currentProject);
  if (!event) return <div>Carregando projeto...</div>;

  return (
    <div className="px-6 py-4">
      <h1 className="text-2xl font-bold mb-4">{event.name}</h1>
      {/* Timeline do projeto */}
      <Timeline phases={event.timeline} finalDueDate={event.finalDueDate} />

      {/* ... (mais conteúdo da página, ex: versões de vídeo, comentários,
etc.) */}
    </div>
  );
}
```

Observação: Aqui estamos assumindo que o projeto selecionado fica armazenado em `state.currentProject` via Zustand. Caso a aplicação use outra abordagem (por exemplo, buscar pelo `eventId` dos params ou utilizar server components), adapte a fonte de dados para obter as fases (`event.timeline`) e prazo (`event.finalDueDate`). O importante é que o componente `Timeline` receba a lista de fases do projeto atual. Inicialmente, enquanto ainda não implementamos a geração de cronograma, `event.timeline` pode estar vazio ou indefinido – o componente lida com isso retornando `null`.

- **Verificação (Timeline):** Abra o projeto no VSCode e **confirme** que o arquivo `components/widgets/Timeline.tsx` não apresenta erros de compilação (verifique se todas as variáveis e tipos estão definidos corretamente). Certifique-se de que a página de detalhes do evento importa e utiliza `<Timeline ...>` sem erros (o TypeScript deve reconhecer as props `phases` e `finalDueDate`). Neste momento, ao executar a aplicação, a timeline poderá não aparecer (pois ainda não há dados); isso é esperado. O importante é que nenhuma exceção ocorra e o layout da página de detalhe continue renderizando (mesmo que `Timeline` retorne null por não haver fases ainda).

2. Geração Automática de Cronograma a partir do Briefing

Objetivo: Quando o usuário preencher os dados iniciais do projeto (briefing) e solicitar, gerar automaticamente um cronograma (lista de fases com datas) baseado nessas informações. Incluiremos uma função utilitária para calcular as fases conforme regras simples (quantidade de vídeos, data do

evento, prazo final, etc.) e integraremos isso na página de criação de novo projeto. O resultado gerado será mostrado ao usuário para confirmação e então salvo no estado global ao criar o projeto.

- **Criar função geradora de cronograma:** Crie o arquivo utilitário `lib/scheduleGenerator.ts` contendo a função que monta o array de fases (`Phase[]`) com base nos inputs do briefing:

```
// lib/scheduleGenerator.ts
import { addDays, subDays } from "date-fns";

interface Phase {
  name: string;
  plannedStart: Date;
  plannedEnd: Date;
  completed: boolean;
}

/**
 * Gera um cronograma de fases do projeto com base nos dados fornecidos.
 * @param projectName   Nome do projeto (pode influenciar planejamento ou
identificação, opcionalmente)
 * @param numVideos     Quantidade de vídeos a serem produzidos no projeto
 * @param eventDate     Data do evento (caso haja gravação ao vivo; pode ser
undefined se não se aplica)
 * @param finalDueDate  Data limite final para entrega do projeto (pode ser
undefined se não definida)
 * @returns Array de fases planejadas (Phase[])
 */
export function generateScheduleFromBriefing(
  projectName: string,
  numVideos: number,
  eventDate?: Date,
  finalDueDate?: Date
): Phase[] {
  const now = new Date();
  const phases: Phase[] = [];

  // 1. Planejamento
  let planningEnd: Date;
  if (eventDate) {
    // Se há data de evento, planejamento vai até 1 dia antes do evento ou no
máximo 7 dias a partir de hoje (o que vier primeiro)
    const oneDayBeforeEvent = subDays(eventDate, 1);
    const maxPlanning = addDays(now, 7);
    planningEnd = oneDayBeforeEvent < maxPlanning ? oneDayBeforeEvent :
maxPlanning;
  } else {
    // Sem evento fixo: define ~3 dias de planejamento a partir de hoje
    planningEnd = addDays(now, 3);
  }
  phases.push({
```

```
    name: "Planejamento",
    plannedStart: now,
    plannedEnd: planningEnd,
    completed: false,
  });

  // 2. Gravação (só se houver evento ao vivo)
  let lastEnd = planningEnd;
  if (eventDate) {
    const eventStart = eventDate;
    const eventEnd = eventDate; // assumindo evento de um dia para
simplicidade
    phases.push({
      name: "Gravação",
      plannedStart: eventStart,
      plannedEnd: eventEnd,
      completed: false,
    });
    lastEnd = eventEnd;
  }

  // 3. Edição
  const editStart = addDays(lastEnd, 1);
  const daysPerVideo = 3; // suposição: 3 dias de edição por vídeo
  let editDurationDays = numVideos * daysPerVideo;
  let editEnd = addDays(editStart, editDurationDays);

  // 4. Revisão
  const reviewStart = addDays(editEnd, 1);
  const reviewDurationDays = 3; // 3 dias para revisão pelo cliente
  let reviewEnd = addDays(reviewStart, reviewDurationDays);

  // 5. Aprovação
  const approvalStart = addDays(reviewEnd, 1);
  const approvalDurationDays = 1; // 1 dia para aprovação final
  let approvalEnd = addDays(approvalStart, approvalDurationDays);

  // Ajuste baseado no prazo final, se fornecido
  if (finalDueDate) {
    const finalDueTime = finalDueDate.getTime();
    const plannedFinalTime = approvalEnd.getTime();
    if (plannedFinalTime > finalDueTime) {
      console.warn("Cronograma inicial ultrapassa o prazo final. Ajustando
fases...");
      // Calcula quantos dias além do prazo estamos
      const overrunDays = Math.ceil((plannedFinalTime - finalDueTime) /
(1000 * 60 * 60 * 24));
      let remainingOverrun = overrunDays;
      // Tenta reduzir dias da Revisão primeiro, depois Edição, para caber no
prazo
      if (remainingOverrun > 0) {
```

```
        const newReviewDuration = Math.max(reviewDurationDays -
remainingOverrun, 1);
        remainingOverrun -= (reviewDurationDays - newReviewDuration);
        reviewEnd = addDays(reviewStart, newReviewDuration);
      }
      if (remainingOverrun > 0) {
        const minEditDays = Math.ceil(numVideos * 1.5); // pelo menos ~1.5
dias por vídeo
        const newEditDuration = Math.max(editDurationDays -
remainingOverrun, minEditDays);
        editDurationDays = newEditDuration;
        editEnd = addDays(editStart, newEditDuration);
      }
      // Recalcula inícios/fins subsequentes após compressão
      if (reviewEnd < editEnd) {
        // garante que revisão não comece antes do fim da edição
        reviewEnd = addDays(editEnd, 1);
      }
      // Recalcula aprovação após eventual ajuste
      approvalEnd = addDays(reviewEnd, approvalDurationDays);
    }
  }

  // Adiciona as fases de Edição, Revisão e Aprovação com os valores
(ajustados ou originais)
  phases.push({
    name: "Edição",
    plannedStart: editStart,
    plannedEnd: editEnd,
    completed: false,
  });
  phases.push({
    name: "Revisão",
    plannedStart: reviewStart,
    plannedEnd: reviewEnd,
    completed: false,
  });
  phases.push({
    name: "Aprovação",
    plannedStart: addDays(reviewEnd, 1), // começa logo após fim da revisão
    plannedEnd: approvalEnd,
    completed: false,
  });

  // (Opcional) Poderíamos adicionar um marco final de "Entrega Final" igual
à approvalEnd
  return phases;
}
```

Este algoritmo produz um array de 5 fases básicas: **Planejamento**, **Gravação** (se aplicável), **Edição**, **Revisão** e **Aprovação**, distribuindo as datas de início/fim de acordo com o número de vídeos e prazos

fornecidos. Regras implementadas: - **Planejamento:** começa agora (`Date.now()`), vai até um dia antes do evento ou até ~7 dias a partir de hoje, o que ocorrer primeiro (caso haja evento). Sem evento, dura ~3 dias. - **Gravação:** ocorre na data do evento (mesmo início e fim no mesmo dia, simplificado). Ausente se não há data de evento. - **Edição:** começa no dia seguinte ao fim do planejamento ou gravação, dura ~3 dias por vídeo. - **Revisão:** começa no dia seguinte ao fim da edição, dura 3 dias. - **Aprovação:** começa no dia seguinte ao fim da revisão, dura 1 dia. - Se um **prazo final** (`finalDueDate`) foi definido e o cronograma calculado ultrapassa esse prazo, o código tenta **comprimir** as fases de revisão e edição para caber no prazo (reduzindo dias, mas garantindo um mínimo de 1 dia de revisão e ~1.5 dias por vídeo na edição). Ainda assim, se mesmo com ajuste alguma fase terminar depois do prazo, o indicador visual "(!)" aparecerá na timeline (como já tratado no componente Timeline). - Todas as fases iniciam com `completed: false` (nenhuma concluída no momento da geração inicial).

Após criar essa função, verifique se não há erros de import do date-fns. Certifique-se também de exportar a função como acima para poder usá-la em outras partes.

- **Atualizar a página de Novo Projeto (Briefing):** Agora, integre a geração de cronograma na interface onde o usuário insere os dados iniciais do projeto. Abra `app/events/new/page.tsx` (página de criação de um novo evento/projeto). Adicione os estados para os campos do briefing, um botão para gerar cronograma e exibir a prévia, e ajuste a ação final de criar projeto. Por exemplo:

```
"use client";

import React, { useState } from "react";
import { useRouter } from "next/navigation";
import { generateScheduleFromBriefing } from "@/lib/scheduleGenerator";
import { useProjectsStore } from "@/store/useProjectsStore";

export default function NewEventPage() {
  const router = useRouter();
  const createProject = useProjectsStore((state) => state.createProject);

  // Campos do formulário de briefing
  const [projectName, setProjectName] = useState("");
  const [numVideos, setNumVideos] = useState(1);
  const [eventDate, setEventDate] = useState<string>("");       //
armazenamos datas como string para compatibilidade com <input type="date">
  const [finalDueDate, setFinalDueDate] = useState<string>("");
  const [generatedTimeline, setGeneratedTimeline] =
useState<ReturnType<typeof generateScheduleFromBriefing>>([]);

  // Gera cronograma ao clicar no botão
  const handleGenerateTimeline = () => {
    if (!projectName) {
      alert("Por favor, informe um nome para o projeto antes de gerar o
cronograma.");
      return;
    }
    const eventDateObj = eventDate ? new Date(eventDate) : undefined;
```

```
    const finalDueDateObj = finalDueDate ? new Date(finalDueDate) :
undefined;
    const phases = generateScheduleFromBriefing(projectName, numVideos,
eventDateObj, finalDueDateObj);
    setGeneratedTimeline(phases);
  };

  // Cria o projeto usando os dados preenchidos e o cronograma gerado
  const handleCreateProject = () => {
    if (!projectName) {
      alert("Nome do projeto é obrigatório.");
      return;
    }
    const eventDateObj = eventDate ? new Date(eventDate) : undefined;
    const finalDueDateObj = finalDueDate ? new Date(finalDueDate) :
undefined;
    // Gera fases se ainda não foi gerado (segurança caso o usuário pule a
etapa de clicar em "Gerar Cronograma")
    const phases = generatedTimeline.length
      ? generatedTimeline
      : generateScheduleFromBriefing(projectName, numVideos, eventDateObj,
finalDueDateObj);

    // Chama a ação do Zustand para criar o projeto no estado global
    createProject({
      name: projectName,
      eventDate: eventDateObj,
      finalDueDate: finalDueDateObj,
      timeline: phases,
      videos: []  // inicia sem vídeos (adicionado depois via upload de
versões)
    });

    // Navega para a página de detalhes do projeto recém-criado.
    // Aqui assumimos que a página de destino lê o currentProject do store
(como implementado antes).
    router.push("/events/current");
  };

  return (
    <div className="max-w-xl mx-auto p-6 bg-neutral-900 text-white rounded">
      <h1 className="text-2xl font-bold mb-4">Novo Projeto – Briefing</h1>
      {/* Formulário de Briefing */}
      <div className="space-y-4">
        <div>
          <label className="block font-medium mb-1">Nome do Projeto:</label>
          <input
            type="text"
            value={projectName}
            onChange={(e) => setProjectName(e.target.value)}
            className="w-full px-3 py-2 rounded text-black"
```

```
            placeholder="Ex: Vídeo Institucional do Evento X"
          />
        </div>
        <div>
          <label className="block font-medium mb-1">Número de Vídeos:</label>
          <input
            type="number"
            value={numVideos}
            onChange={(e) => setNumVideos(Number(e.target.value))}
            className="w-full px-3 py-2 rounded text-black"
            min={1}
          />
        </div>
        <div>
          <label className="block font-medium mb-1">Data do Evento
(Gravação):</label>
          <input
            type="date"
            value={eventDate}
            onChange={(e) => setEventDate(e.target.value)}
            className="px-3 py-2 rounded text-black"
          />
          <small className="block text-gray-400">Deixe em branco se não
houver evento ao vivo.</small>
        </div>
        <div>
          <label className="block font-medium mb-1">Prazo Final para
Entrega:</label>
          <input
            type="date"
            value={finalDueDate}
            onChange={(e) => setFinalDueDate(e.target.value)}
            className="px-3 py-2 rounded text-black"
          />
          <small className="block text-gray-400">Se houver uma data limite
de entrega.</small>
        </div>
        {/* Botão para gerar cronograma automaticamente */}
        <button
          type="button"
          onClick={handleGenerateTimeline}
          className="mt-4 px-4 py-2 bg-purple-600 hover:bg-purple-700
rounded"
        >
          Gerar Cronograma
        </button>
      </div>

      {/* Prévia do cronograma gerado (se existir) */}
      {generatedTimeline.length > 0 && (
        <div className="mt-6">
```

```
          <h2 className="text-xl font-semibold mb-2">Cronograma Sugerido:</
h2>
          <ul className="mb-4">
            {generatedTimeline.map((phase, idx) => (
              <li key={idx} className="text-sm mb-1">
                <strong>{phase.name}:</strong>{" "}
                {phase.plannedStart.toLocaleDateString()} &rarr;{" "}
                {phase.plannedEnd.toLocaleDateString()}
                {phase.plannedEnd < new Date() && " (⚠ atrasado)"}
                {finalDueDate && phase.plannedEnd > new Date(finalDueDate)
&& " (além do prazo)"}
              </li>
            ))}
          </ul>
          <p className="text-gray-300 text-sm">
            *Você poderá ajustar as datas acima antes de salvar o projeto
(funcionalidade futura).
          </p>
        </div>
      )}

      {/* Botão final para salvar/criar o projeto */}
      <div className="mt-6 text-center">
        <button
          type="button"
          onClick={handleCreateProject}
          className="px-6 py-3 bg-green-600 hover:bg-green-700 font-medium
rounded"
        >
          Salvar Projeto
        </button>
      </div>
    </div>
  );
}
```

Vamos destacar os principais pontos dessa implementação: - Utilizamos `useState` para controlar os campos do formulário: nome do projeto, número de vídeos, data do evento e prazo final. (Se o projeto já usar **React Hook Form** e **Zod** para validação, você poderia integrar esses campos nesse esquema; aqui usamos state simples para clareza.) - O botão **"Gerar Cronograma"** (`handleGenerateTimeline`) pega os valores atuais preenchidos, faz as devidas conversões para Date, e usa `generateScheduleFromBriefing` (a função criada) para obter a lista de fases. O resultado é armazenado em `generatedTimeline` no state local. - Se `generatedTimeline` tiver conteúdo, renderizamos uma prévia do cronograma: uma lista `<ul>` mostrando cada fase com datas formatadas (aqui usamos `toLocaleDateString()` para simplicidade; poderíamos usar `format` do date-fns para formatação consistente, como "dd/MM/yyyy"). Também indicamos visualmente se alguma fase está atrasada em relação à data atual ou além do prazo final definido (usando textos ⚠). - O botão **"Salvar Projeto"** (`handleCreateProject`) verifica campos obrigatórios, garante que temos as fases (gerando-as na hora se o usuário não clicou em gerar antes), e então chama `createProject` do store global. Passamos os dados do projeto, incluindo `timeline: phases` (array de fases calculado) e

inicializamos `videos: []` vazio, pois ainda não há versões enviadas. - Após criar o projeto no estado global, fazemos um `router.push` para redirecionar. No exemplo, usamos `"/events/current"` assumindo que essa rota exibe o projeto atual usando `currentProject`. **Ajuste este comportamento de navegação conforme sua lógica:** por exemplo, se preferir, pode navegar para uma rota com o ID do projeto (`/events/${newId}`), dependendo de como o Zustand e rotas estão configurados. O importante é chegar na página de detalhes do projeto, onde já integramos a timeline e (em breve) as versões de vídeo.

- **Verificação (Cronograma Automático):** Após colar o código acima, **confirme** no VSCode que:
- O import de `generateScheduleFromBriefing` está resolvendo corretamente (o arquivo `lib/scheduleGenerator.ts` foi criado e exporta a função).
- Não há erros de tipo nos usos de `createProject` e nos parâmetros passados. (Iremos implementar/atualizar `createProject` no store no próximo passo, portanto temporariamente pode haver um erro se ainda não fez a alteração no store – realize a próxima etapa e volte para verificar.)
- Inicie a aplicação local e navegue até a página de criação de novo projeto. Teste preencher alguns dados (por exemplo, 2 vídeos, com/sem data de evento, com um prazo final) e clique em **Gerar Cronograma**. Você deve ver a lista de fases calculadas aparecer. Verifique se as datas e indicações de atraso/prazo fazem sentido. *(Ex.: Se colocou prazo muito curto, deve aparecer "(além do prazo)" em alguma fase.)* Essa prévia confirma que a geração está funcionando.
- Ainda na página de novo projeto, após gerar o cronograma, clique em **Salvar Projeto**. Isso deve chamar `createProject` (ainda vamos definir melhor no store) e redirecionar para a página de detalhes do projeto. Nesse momento, como não implementamos totalmente o store e navegação por ID, pode ser que a rota `/events/current` exiba o projeto criado via `currentProject`. Mais adiante, confirmaremos se a timeline aparece com os dados salvos quando integrarmos o store.

3. Sistema de Versões de Vídeo com Comparativo Visual

Objetivo: Permitir o upload e gerenciamento de múltiplas versões de um mesmo vídeo do projeto, bem como fornecer uma ferramenta para comparar lado a lado a versão atual e a anterior. Esta funcionalidade envolve ajustes no estado global (store) para armazenar as versões, criação de componentes de UI para listar versões e para o comparativo, e integração desses componentes na página do projeto.

- **Atualizar o store global (Zustand) para suportar versões:** Abra o arquivo `store/useProjectsStore.ts` (ou equivalente onde está definido o estado de projetos). Precisamos:
- Definir estruturas de dados para versão de vídeo e entregável de vídeo.
- Incluir esses campos no tipo Project (incluindo a timeline do projeto, se ainda não adicionamos).
- Adicionar ações na store: `createProject` (se não implementado ou para substituir `addProject` anterior) e `addVideoVersion`.

Edite/complete o arquivo `store/useProjectsStore.ts` conforme abaixo (incluindo as novas interfaces e funções dentro da criação do Zustand store):

```
import { create } from "zustand";

// Definição de tipos para Timeline e Versões de Vídeo
interface Phase {
  name: string;
```

```
  plannedStart: Date;
  plannedEnd: Date;
  completed: boolean;
}
interface VideoVersion {
  id: string;
  name: string;       // ex: "v1", "v2", "Final"
  url: string;        // URL do arquivo de vídeo (local blob URL ou URL
remota)
  uploadedAt: Date;
}
interface VideoDeliverable {
  id: string;
  title: string;
  versions: VideoVersion[];
}
interface Project {
  id: string;
  name: string;
  eventDate?: Date;
  finalDueDate?: Date;
  timeline: Phase[];           // fases do projeto (cronograma)
  videos: VideoDeliverable[];  // lista de vídeos do projeto, cada qual com
suas versões
}

interface ProjectsState {
  projects: Project[];
  currentProject: Project | null;
  createProject: (data: Omit<Project, "id">) => void;
  addVideoVersion: (file: File, deliverableId?: string) => void;
  // (Outras ações como selecionar projeto, marcar fase concluída, etc.,
podem ser adicionadas futuramente)
}

export const useProjectsStore = create<ProjectsState>((set) => ({
  projects: [],
  currentProject: null,

  createProject: (data) => set((state) => {
    const newId = Date.now().toString();  // gera um ID simples (timestamp)
    // Se nenhum vídeo foi fornecido, inicializa com um deliverable padrão
    const initialVideos: VideoDeliverable[] = data.videos &&
data.videos.length > 0
      ? data.videos
      : [{ id: `${newId}-vid1`, title: "Vídeo 1", versions: [] }];
    const newProject: Project = {
      id: newId,
      name: data.name,
      eventDate: data.eventDate,
      finalDueDate: data.finalDueDate,
```

```
      timeline: data.timeline,
      videos: initialVideos
    };
    return {
      projects: [...state.projects, newProject],
      currentProject: newProject
    };
  }),

  addVideoVersion: (file, deliverableId) => set((state) => {
    if (!state.currentProject) return {};
    // Identifica o deliverable alvo (usa o primeiro vídeo caso não seja
especificado)
    const deliverables = state.currentProject.videos;
    const deliverableIndex = deliverableId
      ? deliverables.findIndex(d => d.id === deliverableId)
      : 0;
    if (deliverableIndex < 0) return {};  // se índice inválido, não faz nada
    const deliverable = deliverables[deliverableIndex];

    // Cria objeto da nova versão
    const newVersionNumber = deliverable.versions.length + 1;
    const versionName = `v${newVersionNumber}`;
    const newVersion: VideoVersion = {
      id: `${deliverable.id}-v${newVersionNumber}`,
      name: versionName,
      url: URL.createObjectURL(file),  // cria URL local (blob) para o vídeo
      uploadedAt: new Date()
    };
    // Atualiza a lista de versões do deliverable
    const updatedDeliverable: VideoDeliverable = {
      ...deliverable,
      versions: [...deliverable.versions, newVersion]
    };
    // Atualiza o projeto atual com o novo deliverable
    const updatedProject: Project = {
      ...state.currentProject,
      videos: state.currentProject.videos.map((d, idx) =>
        idx === deliverableIndex ? updatedDeliverable : d
      )
    };

    return {
      projects: state.projects.map(p => p.id === updatedProject.id ?
updatedProject : p),
      currentProject: updatedProject
    };
  })
}));
```

Explicação e considerações: - Definimos as interfaces `VideoVersion` (uma versão de vídeo, com id único, nome da versão, URL e data de upload) e `VideoDeliverable` (um item de vídeo do projeto, que pode ter várias versões). Incluímos `timeline: Phase[]` e `videos: VideoDeliverable[]` dentro de `Project`. - A ação `createProject` agora inicializa o projeto com um ID gerado (aqui usando timestamp por simplicidade; futuramente poderíamos usar UUID). Também garante que exista ao menos um `VideoDeliverable` no novo projeto: se `data.videos` não vier preenchido, criamos um default com `id "{projectId}-vid1"`, título "Vídeo 1" e sem versões ainda. - A ação `addVideoVersion` faz o seguinte: - Verifica se há um `currentProject` selecionado no estado; se não, retorna sem fazer nada (por segurança). - Determina em qual `VideoDeliverable` adicionar a nova versão. Se um `deliverableId` foi passado, procura esse; senão, usa o primeiro da lista (no nosso caso, cada projeto começa com um único deliverable "Vídeo 1", então ele será alvo). - Cria um objeto `newVersion` com `id` único (combina o id do deliverable com `-vN`), um nome sequencial "vN", e gera um URL blob local com `URL.createObjectURL(file)` para poder reproduzir o vídeo imediatamente no front-end. Define `uploadedAt` com o timestamp atual. - Atualiza a lista de versões do deliverable alvo, depois atualiza o projeto atual e a lista global de projetos com essa nova versão incluída. - Retorna o novo estado, atualizando tanto `projects` (a lista) quanto `currentProject`. Isso garante que qualquer componente que esteja exibindo detalhes do projeto ou lista de versões seja re-renderizado via Zustand.

Observação: Esta implementação **simula o upload** apenas armazenando o vídeo localmente (blob URL). Em um cenário real, você faria o upload do arquivo para um servidor ou storage (S3, etc.) e, somente após sucesso, adicionaria a versão com a URL fornecida pelo backend. Aqui, mantivemos tudo no front-end para simplificar os testes locais. Lembre-se de que o URL blob gerado existe apenas enquanto a página estiver aberta; ao recarregar, as URLs podem invalidar. Para persistência real, seria necessário backend.

Se o seu projeto já possuía um método `addProject` ou similar, você pode optar por substituir pela lógica de `createProject` acima ou ajustar nomes conforme preferência. O importante é que agora o projeto seja criado com as propriedades novas (`timeline`, `videos`). Certifique-se de atualizar qualquer parte do código que chamava o antigo `addProject` para usar `createProject` (no nosso código de briefing acima, já usamos `createProject`).

- **Criar componente de Lista de Versões:** Agora, crie um componente para exibir as versões de vídeo de um determinado deliverable e permitir adicionar novas versões (upload de arquivo). Vamos criar `components/video/VersionList.tsx`:

```
"use client";

import React from "react";
import { useProjectsStore } from "@/store/useProjectsStore";

const VersionList: React.FC = () => {
  // Obtemos as versões do primeiro deliverable do projeto atual (para
simplificar)
  const versions = useProjectsStore(
    (state) => state.currentProject?.videos[0]?.versions
  ) || [];
  const addVideoVersion = useProjectsStore((state) => state.addVideoVersion);

  const handleFileChange = (e: React.ChangeEvent<HTMLInputElement>) => {
```

```
    const file = e.target.files?.[0];
    if (file) {
      addVideoVersion(file);  // adiciona ao primeiro vídeo do projeto atual
      e.target.value =
"";    // reseta o input para permitir upload do mesmo arquivo nome
futuramente, se preciso
    }
  };

  return (
    <div className="mb-6">
      <h2 className="text-lg font-semibold mb-2">Versões do Vídeo</h2>
      {versions.length === 0 && (
        <p className="text-sm text-gray-300">Nenhuma versão enviada ainda.</
p>
      )}
      {versions.length > 0 && (
        <ul className="text-sm mb-3">
          {versions.map((v) => (
            <li key={v.id}>
              <span>{v.name}</span>
              {" - "}
              <span className="text-gray-400">
                {v.uploadedAt.toLocaleDateString()}
              </span>
              {v.name.toLowerCase().includes("final") && (
                <strong className="text-green-500 ml-1">(Final)</strong>
              )}
            </li>
          ))}
        </ul>
      )}
      {/* Input para fazer upload de nova versão */}
      <label className="block text-sm font-medium mb-1">Upload de nova
versão:</label>
      <input
        type="file"
        accept="video/*"
        onChange={handleFileChange}
        className="text-sm file:mr-3 file:py-2 file:px-4 file:rounded
file:border-0
                   file:text-sm file:bg-purple-600 file:text-white
hover:file:bg-purple-700"
      />
      <p className="text-xs text-gray-400 mt-1">
        Selecione um arquivo de vídeo para enviar como nova versão.
      </p>
    </div>
  );
};
```

```
export default VersionList;
```

Esse componente: - Usa o Zustand para pegar a lista de versões (`versions`) do primeiro vídeo do `currentProject` e a ação `addVideoVersion`. - Lista as versões existentes mostrando o nome (v1, v2, ...) e a data de upload formatada simples (usando `toLocaleDateString` do JS). - Marca com **(Final)** em verde se o nome da versão contém "final" (por exemplo, se no futuro marcarmos alguma versão como "Final"). - Exibe um input do tipo file estilizado para permitir upload. Ao selecionar um arquivo, chama `addVideoVersion(file)` do store, que adiciona a versão. Em seguida, limpa o valor do input file (isso permite que se o usuário quiser enviar novamente um arquivo com o mesmo nome, o onChange detecte). - Utilizamos classes Tailwind com prefixo `file:` para estilizar o botão de upload de arquivo, seguindo o tema (roxo para combinar com tema Dracula, semelhante ao botão de gerar cronograma).

- **Criar componente de Comparativo de Versões:** Agora o componente que coloca duas versões lado a lado para comparar visualmente. Vamos criar `components/video/VersionCompare.tsx`:

```
"use client";

import React, { useRef } from "react";
import { useProjectsStore } from "@/store/useProjectsStore";

const VersionCompare: React.FC = () => {
  const versions = useProjectsStore(
    (state) => state.currentProject?.videos[0]?.versions
  ) || [];
  // Precisamos de pelo menos duas versões para comparar
  if (versions.length < 2) return null;

  const latestVersion = versions[versions.length - 1];
  const prevVersion = versions[versions.length - 2];

  // Referências para os elementos de vídeo (para controlar reprodução)
  const videoRef1 = useRef<HTMLVideoElement>(null);
  const videoRef2 = useRef<HTMLVideoElement>(null);

  // Função para dar play simultâneo nos dois vídeos do início
  const handlePlayBoth = () => {
    if (videoRef1.current && videoRef2.current) {
      videoRef1.current.currentTime = 0;
      videoRef2.current.currentTime = 0;
      videoRef1.current.play();
      videoRef2.current.play();
    }
  };

  return (
    <div className="mb-6">
      <h2 className="text-lg font-semibold mb-3">Comparativo de Versões</h2>
```

```
      <div className="flex gap-4">
        {/* Versão anterior */}
        <div className="flex-1">
          <h3 className="text-base font-medium mb-1">Versão:
{prevVersion.name}</h3>
          <video
            ref={videoRef1}
            src={prevVersion.url}
            controls
            className="w-full rounded border border-gray-600"
          />
          {/* Espaço para lista de comentários da versão anterior, se houver
*/}
          {/* <CommentsList versionId={prevVersion.id} /> */}
        </div>
        {/* Versão atual */}
        <div className="flex-1">
          <h3 className="text-base font-medium mb-1">
            Versão: {latestVersion.name} (Atual)
          </h3>
          <video
            ref={videoRef2}
            src={latestVersion.url}
            controls
            className="w-full rounded border border-gray-600"
          />
          {/* Espaço para lista de comentários da versão atual, se houver */}
          {/* <CommentsList versionId={latestVersion.id} /> */}
        </div>
      </div>
      {/* Botão para reproduzir os dois vídeos simultaneamente */}
      <button
        type="button"
        onClick={handlePlayBoth}
        className="mt-3 px-4 py-2 bg-blue-600 hover:bg-blue-700 rounded text-
sm"
      >
        Reproduzir ambos do início
      </button>
      <p className="text-xs text-gray-400 mt-1">
        *Clique em play nos dois vídeos ou use o botão acima para iniciá-los
juntos.*
      </p>
    </div>
  );
};

export default VersionCompare;
```

Esse componente: - Obtém as versões do primeiro vídeo do projeto atual, e se existirem pelo menos duas, define `latestVersion` (última da lista) e `prevVersion` (penúltima). - Renderiza dois players

de vídeo `<video>` lado a lado (usando flexbox `.flex.gap-4` para colunas). O vídeo da esquerda é a versão anterior, o da direita é a versão atual (marcada como "(Atual)" no título). - Usa `useRef` para manter referências aos dois elementos de vídeo, permitindo controlar via código. Implementamos um botão "Reproduzir ambos do início" que, ao ser clicado, reseta ambos os vídeos para o segundo 0 e dá play nos dois quase simultaneamente. Isso ajuda o usuário a ver as diferenças sincronizadamente desde o começo. (Uma melhoria futura seria sincronizar tempo de execução continuamente, mas mantivemos simples com um botão.) - Incluímos comentários indicando onde um componente de comentários poderia entrar (`<CommentsList versionId={...} />`). Ou seja, se houver um sistema de comentários implementado, poderíamos filtrá-los por `versionId` e mostrar os relacionados a cada versão abaixo do respectivo vídeo. Isso facilitaria comparar não apenas os vídeos em si, mas também verificar se feedbacks dados na versão anterior foram resolvidos na versão atual. - O botão e textos seguem o estilo geral (botão azul, texto pequeno explicativo em cinza).

- **Integrar os componentes de versão na página do projeto:** Agora que temos `VersionList` e `VersionCompare`, volte ao arquivo `app/events/[eventId]/page.tsx` (detalhes do evento) e importe esses componentes, renderizando-os abaixo da Timeline. Ficará algo assim (dando continuidade ao exemplo da integração anterior):

```
import VersionList from "@/components/video/VersionList";
import VersionCompare from "@/components/video/VersionCompare";

export default function EventDetailPage({ params }) {
  const event = useProjectsStore((state) => state.currentProject);
  if (!event) return <div>Carregando projeto...</div>;

  return (
    <div className="px-6 py-4">
      <h1 className="text-2xl font-bold mb-4">{event.name}</h1>
      <Timeline phases={event.timeline} finalDueDate={event.finalDueDate} />

      {/* Seção de versões de vídeo */}
      <div className="mt-6">
        <VersionList />
        <VersionCompare />
      </div>

      {/* ... demais seções, ex: comentários gerais do projeto ... */}
    </div>
  );
}
```

Agora, ao visualizar a página do projeto, você terá: - A **Timeline** do projeto exibindo as fases e status (implementada no passo 1). - A seção **Versões do Vídeo** listando as versões existentes e o input para adicionar nova versão. - O **Comparativo de Versões** que aparece automaticamente somente quando há 2 ou mais versões disponíveis, mostrando sempre a última e a penúltima.

Detalhe: Decidimos por simplicidade tratar sempre o primeiro vídeo do projeto (`videos[0]`). Se no futuro houver múltiplos vídeos por projeto, você poderia expandir a UI para selecionar qual deliverable

visualizar/comparar. Mas como nosso foco é a mecânica de versões, mantivemos um vídeo único por projeto para não complicar a interface agora.

- **Verificação (Versões de Vídeo):** Após inserir o código acima, verifique no VSCode:
- O arquivo do **store** não tem erros de tipo e inclui as novas ações (`createProject` e `addVideoVersion`). Confirme que `Project.timeline` e `Project.videos` existem no tipo e são usados na `createProject` (deve casar com o que passamos no `handleCreateProject` do briefing).
- Os componentes `VersionList` e `VersionCompare` devem compilar sem erros e estar devidamente importados na página do evento.
- Abra o app em modo desenvolvimento e teste o fluxo completo:
    1. Crie um novo projeto pela tela de **Briefing** (Nova Projeto). Depois de clicar em **Salvar Projeto**, você deve ser levado à página de detalhes. Verifique se a **Timeline** aparece preenchida com as fases e barras coloridas correspondentes (se o cronograma foi gerado e salvo corretamente, deve mostrar). Cada fase concluída inicialmente será nenhuma (então todas coloridas como futuras ou atrasadas dependendo das datas relativas a hoje).
    2. Na seção **Versões do Vídeo**, deve constar "Nenhuma versão enviada ainda." para um projeto recém-criado. Use o campo "Upload de nova versão" para selecionar um arquivo de vídeo do seu computador (qualquer vídeo de teste). Ao selecionar, a lista deve atualizar mostrando **v1** com a data atual. O comparativo não aparece ainda pois só há uma versão.
    3. Faça um segundo upload no mesmo projeto (se quiser, pode usar o mesmo arquivo novamente para simular v2). Agora a lista deve mostrar **v1** e **v2**, e o componente **Comparativo de Versões** deve surgir abaixo. Verifique que consegue dar play nos dois vídeos (eles tocarão independentemente, ou juntos se clicar no botão de reproduzir ambos). Veja se as referências de versão estão corretas (título mostrando "v1" e "v2 (Atual)").
    4. (Opcional) Teste também cenários de comentário, se existir um sistema de comentários integrado, para pensar em como filtrá-los por versão. No nosso caso, sem implementar isso, apenas garantimos que nenhum erro acontece nos placeholders de `<CommentsList>` (eles estão comentados, então não executam nada).
    5. Inspecione o estado global via DevTools do Zustand (se disponível) ou logs, para assegurar que `currentProject` contém a nova versão após cada upload e que a estrutura (`videos[0].versions`) está sendo atualizada corretamente.
- Por fim, verifique visualmente se todos os componentes estão estilizados de acordo com o tema. As cores usadas (verdes, azuis, roxos, etc.) foram escolhidas com base no tema Dracula e Tailwind padrão. Ajuste qualquer cor ou classe caso o seu `tailwind.config.js` tenha cores personalizadas do tema (por exemplo, se há uma classe `primary` customizada, você poderia usá-la em vez de `bg-purple-600`, etc.).

Prompt Final – Validações e Próximos Passos

Após implementar todas as etapas acima, utilize o VSCode (ou Copilot Chat) para revisar e testar os pontos a seguir:

1. **Estrutura do Store:** Abra `store/useProjectsStore.ts` e confirme que as interfaces **Project**, **Phase**, **VideoDeliverable** e **VideoVersion** estão declaradas corretamente, e que as funções **createProject** e **addVideoVersion** aparecem sem erros de tipo. Peça ao Copilot para

validar se a propriedade `timeline` do projeto está sendo populada e usada (por exemplo: *"A função createProject está incluindo corretamente timeline e videos no objeto Project?"*).

2. **Integridade do Timeline:** Verifique no **browser** se o componente **Timeline** renderiza as fases adequadamente. Por exemplo, pergunte ao Copilot no chat se as fases estão ordenadas e coloridas de acordo com as condições (*"O componente Timeline está distinguindo corretamente fases concluídas, em andamento e atrasadas?"*). Embora a validação visual final seja manual, o Copilot pode inspecionar o código para conferir a lógica.
3. **Geração de Cronograma:** Confirme, através de testes manuais ou consulta ao Copilot, se a função **generateScheduleFromBriefing** está retornando um array de 4-5 fases conforme esperado dado certos inputs. Você pode usar o console do navegador ou um teste unitário rápido para checar (ex.: inserir 2 vídeos, data de evento hoje +10 dias, prazo final hoje +15 dias, etc., e ver se as fases respeitam essas datas). Peça ao Copilot para analisar possíveis edge cases (*"O que acontece se eu não fornecer data de evento nem prazo final? Quantas fases a função retorna?"*).
4. **Fluxo de Criação e Navegação:** Use o Copilot Chat para revisar a parte do código que faz `router.push("/events/current")`. Se necessário, ajuste conforme a arquitetura de rotas do seu app. Por exemplo, pergunte: *"Devo manter '/events/current' ou usar o ID do projeto na navegação?"*. Certifique-se de que, ao criar um projeto, a tela de detalhes realmente exiba o projeto correto. Isso pode envolver lógica adicional (por exemplo, talvez configurar a página de detalhe para ler do store em vez de usar o param ID diretamente durante a ausência de backend).
5. **Upload e Versões:** Peça ao Copilot para verificar a implementação de **VersionList** e **VersionCompare**. Por exemplo: *"A função addVideoVersion está atualizando corretamente currentProject e projects?"*. Depois, teste você mesmo fazendo upload de arquivos conforme descrito e veja se nenhum erro ocorre (ver console do navegador para mensagens ou warnings). Copilot pode ajudar a confirmar se usar `URL.createObjectURL` repetidamente é seguro neste contexto (lembrando que em dev tudo bem, mas em produção deve-se liberar URLs blob com `URL.revokeObjectURL` eventualmente – não crítico para agora, mas bom notar).
6. **Interface e Usabilidade:** Revise com o Copilot se há melhorias rápidas de UX possíveis. Por exemplo, **Copilot** pode sugerir adicionar um indicador de carregamento ao fazer upload (embora nosso caso seja instantâneo pois não há upload real). Ou validar campos do formulário (nome não vazio já fazemos com alert, mas poderíamos desabilitar botões se inválido, etc.). Anote essas sugestões para futuras iterações.
7. **Teste Final Completo:** Finalmente, rode a aplicação no navegador e realize um **teste completo do fluxo**: criar um projeto com briefing -> gerar cronograma -> salvar -> visualizar timeline -> fazer dois uploads de vídeo -> comparar versões. Verifique se tudo funciona como esperado. Qualquer comportamento inesperado, use o Copilot Chat para investigar o código relacionado. Por exemplo, se a timeline não aparecer, peça: *"Quais possíveis motivos para event.timeline estar vazio na página de detalhe?"* e corrija de acordo (pode ser que `createProject` não esteja sendo chamado, ou a navegação não esteja certa, etc.).

Se todos os pontos acima estiverem corretos, as três funcionalidades estarão implementadas com sucesso.

Boa programação! Agora o Visual Studio Code (com Copilot) deve ser capaz de auxiliar no refinamento final. Certifique-se de que cada arquivo criado/modificado foi salvo e reconhecido pelo projeto (veja no Explorador de arquivos do VSCode se eles aparecem nos locais certos). Em caso de dúvidas ou erros, utilize o Copilot Chat solicitando correções pontuais nos trechos apresentados. Cada etapa foi pensada para seguir os padrões do projeto (Next.js App Router, Tailwind com tema escuro, Zustand para estado global, componentes client-side onde necessário), garantindo consistência com o restante da aplicação.
