Project Design

**Project Architecture & Technology Stack: LabGen AI Video Generator**

Document Version: 1.0

Date: August 16, 2025

Author: K. Brian, Principal Software Architect

### 1. Project Vision & Goals

The primary objective of the **LabGen AI Video Generator** is to create a web-based platform that autonomously transforms written scientific laboratory protocols into clear, concise, and visually engaging instructional videos. The system will parse raw text, generate a narrative and visual storyboard, synthesize corresponding media assets, and compile them into a final video output, thereby democratizing and simplifying scientific training.

### 2. System Architecture Overview

The system is designed as a distributed, service-oriented architecture to handle computationally intensive, long-running tasks gracefully. The core principle is to decouple the user-facing web application from the asynchronous, heavy-lifting AI pipeline.

#### 2.1. High-Level Diagram

+---------------------+ +---------------------+ +------------------+

| | | | | |

| Frontend Client |----->| Backend API |----->| Task Queue |

| (Next.js) | | (Python/FastAPI) | | (Celery/Redis) |

| |<-----| |<-----| |

+---------------------+ +----------+----------+ +--------+---------+

| |

| | (Dispatch Jobs)

| |

| v

+---------------------------------------------------+

| |

| AI Processing Workers |

| (Python: NLP, LLM, Image/Audio Gen, FFMPEG) |

| +--------------------------+------------------------+

|

| (Read/Write Assets)

|

v

+-------------------+

| |

| Data Storage |

| (Cloud Storage/FS)|

| |

+-------------------+

#### 2.2. Core Components

* Frontend Client: A Single Page Application (SPA) built with Next.js that provides the user interface for submitting protocols and viewing generated videos. It communicates with the backend via a RESTful API.
* Backend API Server: A lightweight, high-performance Python server using FastAPI. Its primary roles are to handle user requests, validate input, enqueue jobs into the task queue, and serve status updates and final video results. It *does not* perform the AI generation itself.
* AI Processing Pipeline: A set of background workers managed by Celery. These workers pull tasks from the queue and execute the multi-step AI workflow (parsing, storyboarding, asset generation, compilation). This architecture allows for horizontal scaling of workers based on load.
* Data Storage: A durable storage solution (e.g., AWS S3, Google Cloud Storage, or a local network file system) to store generated image assets, audio clips, and the final compiled videos.

#### 2.3. Overall Data Flow

1. Submission: The user submits the lab protocol text via the Next.js frontend.
2. Job Initiation: The frontend sends a POST request to the FastAPI backend.
3. Task Queuing: The backend validates the input, creates a unique task\_id, and pushes a generation job onto the Celery/Redis queue. It immediately returns the task\_id to the frontend.
4. Asynchronous Processing: A Celery worker picks up the job and begins the AI pipeline.
5. Asset Generation: The worker generates a storyboard and then dispatches parallel sub-tasks to generate all required visual and audio assets.
6. Compilation: Once all assets for a video are ready, a final task assembles them using FFMPEG into a single .mp4 file.
7. Storage & Status Update: The final video is saved to the designated Data Storage, and the task status in the database is updated to completed.
8. Result Retrieval: The frontend, which has been periodically polling the status endpoint using the task\_id, sees the 'completed' status and retrieves the video URL for the user.

### 3. Technology Stack Selection

The technology choices are optimized for development velocity, performance, scalability, and leveraging the existing AI/ML ecosystem.

| Component | Technology | Justification |
| --- | --- | --- |
| Frontend Framework | Next.js (React) | Provides an excellent developer experience with features like Server-Side Rendering (SSR) and Static Site Generation (SSG) for fast initial loads. The large React ecosystem ensures rich component availability. |
| Frontend Language | TypeScript | Enforces type safety, which is critical for building robust, maintainable, and scalable applications. Reduces runtime errors and improves code clarity and developer tooling. |
| Frontend Styling | Tailwind CSS | A utility-first CSS framework that enables rapid UI development directly within the markup, ensuring consistency and maintainability without writing custom CSS. |
| Backend Framework | Python & FastAPI | Python is the lingua franca of AI/ML. FastAPI provides exceptional performance on par with NodeJS, built-in async support (crucial for I/O with AI APIs), automatic data validation via Pydantic, and interactive API documentation. |
| Text Processing (NLP) | SpaCy | A production-grade, highly optimized NLP library. It excels at Named Entity Recognition (NER) for identifying equipment, chemicals, and actions, and is generally faster and more efficient than NLTK for these tasks. |
| Task Queuing | Celery & Redis | Celery is the industry standard for distributed task queues in Python. Redis is chosen as the message broker for its simplicity, low latency, and performance, which is ideal for our use case. |
| Video Compilation | FFMPEG | The undisputed open-source standard for video and audio manipulation. It's powerful, fast, and can be easily controlled via a Python wrapper like ffmpeg-python. |

#### 3.1. AI Generation Implementations (Strategic Options)

We will architect the system to support two distinct AI generation backends. This allows us to start with a fast-to-market, high-quality solution (Option A) while building a more cost-effective, customizable alternative in parallel (Option B).

| Feature | Option A: Cloud API-Based (Initial Implementation) | Option B: Open Source / Self-Hosted |
| --- | --- | --- |
| Storyboard/Script | OpenAI GPT-4 | Fine-tuned LLM (e.g., Llama 3, Mistral) hosted via Hugging Face TGI or vLLM. |
| Image Generation | OpenAI DALL-E 3 | Stable Diffusion (e.g., SDXL) or a specialized model hosted on a dedicated GPU instance. |
| Video Generation | *Not applicable directly; we sequence DALL-E 3 images.* | Stable Video Diffusion (SVD) or similar models for generating short video clips instead of static images. |
| Narration (TTS) | OpenAI TTS-1 / TTS-1-HD | Open-source TTS model like Coqui XTTS for voice cloning and high-quality speech synthesis. |
| Pros | - Fastest Time-to-Market<br>- State-of-the-Art Quality<br>- Low Maintenance | - Lower Long-Term Cost<br>- Full Control & Customization<br>- No Vendor Lock-in |
| Cons | - High Operational Cost<br>- Vendor Lock-in<br>- Rate Limiting/Dependency | - High Initial Complexity<br>- Requires GPU Infrastructure<br>- Maintenance Overhead |

### 4. AI Processing Pipeline - Detailed Workflow

The AI pipeline is the core of the application, transforming raw text into a finished video. It is executed by a Celery worker.

1. Step 1: Protocol Ingestion & Parsing
   * The worker receives the raw protocol text as a string.
   * The text is sanitized and split into individual procedural steps using rule-based logic and sentence boundary detection.
2. Step 2: Structured Data Extraction (NLP)
   * For each step, a pre-trained SpaCy model performs Named Entity Recognition (NER).
   * The model identifies and labels key entities:
     + EQUIPMENT (e.g., "beaker," "pipette," "centrifuge")
     + SUBSTANCE (e.g., "distilled water," "hydrochloric acid")
     + ACTION (e.g., "pour," "mix," "heat," "measure")
     + QUANTITY (e.g., "10ml," "5 grams")
   * The output is a structured list of objects, each representing a step with its extracted entities.
3. Step 3: Storyboard Generation (LLM)
   * The structured data from Step 2 is formatted into a detailed prompt for a Large Language Model (LLM, e.g., GPT-4).
   * The LLM is instructed to act as a science educator and film director, converting the steps into a scene-by-scene storyboard.
   * The LLM's output is a structured JSON array. Each object in the array represents a scene and must conform to the following schema:

{

"scene\_number": 1,

"visual\_prompt": "A photorealistic image of a scientist's gloved hand carefully pouring 50ml of a clear blue liquid from a glass beaker into an Erlenmeyer flask on a clean lab bench.",

"narration\_script": "First, carefully pour fifty milliliters of the copper sulfate solution into the Erlenmeyer flask."

* + }

1. Step 4: Asset Generation (Parallel Tasks)
   * The worker iterates through the JSON storyboard. For each scene, it dispatches two parallel sub-tasks:
     + Image Generation: The visual\_prompt is sent to the image generation model (e.g., DALL-E 3 or Stable Diffusion) to create a .png image file.
     + Audio Generation: The narration\_script is sent to the Text-to-Speech (TTS) model to create a .mp3 audio file.
   * This parallelism significantly reduces the total generation time.
2. Step 5: Video Assembly (FFMPEG)
   * Once all image and audio assets for the task are successfully generated, a final worker task is initiated.
   * Using an FFMPEG command (via a Python wrapper), the worker performs the following:
     + Sequences the images in order of scene\_number.
     + Sets a standard duration for each image (e.g., 5 seconds).
     + Overlays the corresponding narration audio for each image segment.
     + Adds silent crossfades between scenes for a smooth transition.
     + Encodes the final output as a web-optimized .mp4 file.

### 5. API Endpoint Specification (RESTful Design)

The API is designed to support the asynchronous nature of the video generation process.

| Endpoint | Method | Description |
| --- | --- | --- |
| POST /api/v1/generate | POST | Initiates a new video generation job.<br/>Body: { "protocol\_text": "..." }<br/>Success Response (202 Accepted): { "task\_id": "uuid-v4-string" }<br/>Failure Response (400 Bad Request): { "error": "..." } |
| GET /api/v1/status/{task\_id} | GET | Checks the status of a generation job.<br/>Success Response (200 OK): `{ "task\_id": "...", "status": "pending |
| GET /api/v1/video/{task\_id} | GET | Retrieves the final video URL. Only works if status is 'completed'.<br/>Success Response (200 OK): { "task\_id": "...", "video\_url": "https://storage.googleapis.com/..." }<br/>Failure Response (404 Not Found): { "error": "Video not ready or task failed" } |

### 6. Project Directory Structure (Monorepo)

A monorepo structure using

pnpm workspaces (or similar) is recommended to manage shared code and streamline development between the frontend and backend.

/labgen-ai-video/

├── apps/

│ ├── frontend/ # Next.js Application

│ │ ├── app/

│ │ ├── components/

│ │ ├── public/

│ │ ├── next.config.js

│ │ └── package.json

│ └── backend/ # FastAPI Application

│ ├── app/

│ │ ├── api/ # API Routers/Endpoints

│ │ ├── core/ # Config, settings

│ │ ├── models/ # Pydantic models

│ │ └── services/ # Business logic

│ ├── workers/ # Celery task definitions

│ │ ├── pipeline.py

│ │ └── \_\_init\_\_.py

│ ├── main.py # FastAPI app entrypoint

│ └── requirements.txt

├── packages/

│ └── shared-types/ # Shared TypeScript types/interfaces

│ └── index.ts

├── infra/

│ ├── docker-compose.yml # For local development

│ └── Dockerfile.backend

├── .gitignore

├── package.json # Root package.json for monorepo scripts

└── README.md

### 7. Development Roadmap

The project will be developed in distinct, iterative phases to manage complexity and deliver value incrementally.

1. Phase 1: Core Backend & API Scaffolding (2 Sprints)
   * Set up the monorepo structure.
   * Develop the FastAPI application with the /generate, /status, and /video endpoints.
   * Integrate Celery and Redis. Implement a "dummy" worker that simulates a long-running task and updates status.
   * Set up Docker for local development.
2. Phase 2: AI Pipeline Implementation - Option A (3 Sprints)
   * Implement the full AI pipeline using the Cloud API-based approach (OpenAI).
   * Step 2: Integrate SpaCy for NER.
   * Step 3: Integrate GPT-4 for storyboard generation.
   * Step 4: Integrate DALL-E 3 and OpenAI TTS for asset generation.
   * Step 5: Integrate FFMPEG for video compilation.
   * *Goal: A fully functional end-to-end video generation system.*
3. Phase 3: Frontend Development (3 Sprints)
   * Build the Next.js application UI/UX.
   * Develop the protocol submission form.
   * Implement the status polling logic and the results page with the video player.
   * Ensure responsive design for desktop and mobile.
4. Phase 4: Full Integration, Testing & Deployment (2 Sprints)
   * Connect the frontend to the live backend API.
   * Conduct end-to-end testing and bug fixing.
   * Develop CI/CD pipelines.
   * Deploy the initial version (MVP) to a cloud provider (e.g., AWS, GCP).
5. Phase 5: AI Pipeline Implementation - Option B (4+ Sprints)
   * Research and select the best open-source models (LLM, Image, TTS).
   * Set up GPU-enabled infrastructure for hosting these models.
   * Develop an abstraction layer in the backend to allow switching between "Option A" and "Option B" generation pipelines.
   * Implement the full pipeline using the self-hosted models.
6. Phase 6: Advanced Features & Optimization (Ongoing)
   * Performance monitoring and cost optimization.
   * User accounts and history.
   * Implementation of Stable Video Diffusion (SVD) for dynamic scenes.
   * Advanced video editing features (e.g., text overlays, background music).

Tab 2

**LabGen: AI-Powered Experiment Visualizer**

Document Version: 1.1

Date: August 16, 2025

Author: K. Brian, Principal Software Architect

LabGen is a web application designed to transform written scientific laboratory protocols into clear, narrated video demonstrations. By leveraging AI for text-to-speech and image generation, it provides a dynamic way to visualize complex experimental procedures, making science more accessible and easier to understand.

This project features a decoupled architecture with a static frontend and a powerful FastAPI backend that supports multiple AI generation pipelines.

### Key Features

* Text-to-Video Generation: Converts step-by-step lab protocols into a complete video sequence.
* Realistic Lab Environments: Generates images depicting scientists performing actions in a modern laboratory setting.
* Spoken Narration: Each step of the protocol is narrated using AI-powered Text-to-Speech (TTS) for clarity.
* Dual AI Backends: Easily switch between two generation pipelines:
  + An OpenAI-based pipeline (simulating DALL-E and TTS APIs).
  + A Hugging Face-based pipeline (simulating local or self-hosted models).
* Asynchronous Processing: Utilizes background tasks to handle long-running video generation jobs without blocking the user interface.
* Simple & Intuitive UI: A clean, single-page interface for protocol submission and video playback.

### System Architecture

The application is composed of two primary components: a static frontend and a FastAPI backend. The backend manages the core logic and can be configured to use one of two distinct AI pipelines.

1. Frontend (  
   labgen\_frontend):
   * A static single-page application built with HTML, Tailwind CSS, and vanilla JavaScript.
   * Responsible for capturing the user's protocol input.
   * Communicates with the backend via REST API calls to initiate generation, poll for status, and retrieve the final video.
2. Backend (  
   labgen\_backend):
   * A robust API built with Python and FastAPI.
   * Exposes endpoints for managing the video generation lifecycle.
   * Uses BackgroundTasks for non-blocking, asynchronous job processing.
   * Maintains task status in-memory (for this version).
3. AI Generation Pipelines:
   * The core video generation process is modular and selected via an environment variable (GENERATION\_BACKEND).
   * Pipeline Flow:
     1. Protocol Parser: Extracts structured steps, actions, and entities from the raw text.
     2. Storyboard Generator: Creates a scene-by-scene storyboard with visual prompts and narration scripts.
     3. Asset Generation: For each scene, it generates an image and a corresponding audio narration file using the selected backend (OpenAI or Hugging Face).
     4. Video Assembler: Combines the generated images and audio files into a final MP4 video (simulated using FFmpeg placeholders).

### Technology Stack

| Component | Technology |
| --- | --- |
| Frontend | HTML5, CSS3, Vanilla JavaScript, Tailwind CSS, Lucide Icons |
| Backend | Python 3.x, FastAPI, Uvicorn |
| AI (Simulated) | OpenAI API (DALL-E, TTS), Hugging Face Models (Stable Diffusion, TTS) |

### Project Structure

The repository is organized into two main directories,

labgen\_frontend and labgen\_backend.

<details> <summary><strong><code>labgen\_frontend/</code></strong></summary>

labgen\_frontend/

├── index.html # Main application HTML file

├── script.js # Client-side logic for UI and API interaction

└── style.css # Custom CSS styles

</details> <details> <summary><strong><code>labgen\_backend/</code></strong></summary>

labgen\_backend/

├── app/

│ ├── api/

│ │ └── endpoints/

│ │ └── generation.py # API routes for video generation

│ ├── core/

│ │ └── config.py # Application configuration loader

│ ├── models/

│ │ └── schemas.py # Pydantic models for API requests/responses

│ ├── services/

│ │ ├── protocol\_parser.py # Parses raw protocol text

│ │ ├── storyboard\_generator.py # Creates a storyboard from parsed steps

│ │ ├── openai\_\*.py # Placeholder OpenAI asset generators

│ │ ├── huggingface\_\*.py # Placeholder Hugging Face asset generators

│ │ ├── video\_assembler.py # Simulates video creation

│ │ └── placeholder\_services.py # Main background task logic

│ └── main.py # FastAPI application entry point

├── requirements.txt # Python dependencies

└── .env.example # Example environment variables file

</details>

### Setup and Installation

#### Prerequisites

* Python 3.8+ and pip
* A web browser

#### 1. Clone the Repository

git clone <repository\_url>

cd LabGen-AI-App

#### 2. Backend Setup

Navigate to the backend directory and set up a virtual environment.

cd labgen\_backend

# Create and activate a virtual environment

python -m venv venv

source venv/bin/activate # On Windows, use: venv\Scripts\activate

# Install dependencies

pip install -r requirements.txt

*Note: The requirements.txt file should contain fastapi and uvicorn[standard]. If it's empty, please add them.*

#### 3. Frontend Setup

The frontend is a static site and requires no build process. It can be run using any simple HTTP server or by opening the

index.html file directly in your browser.

### Configuration

The backend is configured using environment variables. Create a .env file in the labgen\_backend directory by copying the .env.example.

# In labgen\_backend/

cp .env.example .env

Now, edit the.env file to configure the application:

# File: labgen\_backend/.env

# Set the desired AI generation backend.

# Options: "openai" or "huggingface"

GENERATION\_BACKEND="openai"

# --- OpenAI Configuration ---

# Add your OpenAI API key here. This is required if GENERATION\_BACKEND is "openai".

# OPENAI\_API\_KEY="sk-..."

# --- Hugging Face Configuration (Placeholder) ---

# In a real implementation, you would specify paths or URLs to your models.

# HUGGINGFACE\_IMG\_MODEL\_PATH="runwayml/stable-diffusion-v1-5"

# HUGGINGFACE\_TTS\_MODEL\_PATH="suno/bark-small"

* GENERATION\_BACKEND: This is the most important setting.
  + Set to openai to use the (simulated) OpenAI API pipeline.
  + Set to huggingface to use the (simulated) Hugging Face pipeline.
* OPENAI\_API\_KEY: If using the openai backend, you would place your actual API key here in a real implementation.
* Hugging Face Paths: The placeholder services for Hugging Face show how local model paths or identifiers would be used.

### Running the Application

#### 1. Start the Backend Server

Ensure you are in the

labgen\_backend directory with your virtual environment activated.

# In labgen\_backend/

uvicorn app.main:app --reload

The API server will start, typically on

http://127.0.0.1:8000. You can access the interactive API documentation at http://127.0.0.1:8000/docs.

#### 2. Start the Frontend

Open a new terminal and navigate to the

labgen\_frontend directory. A simple way to serve the file is using Python's built-in HTTP server.

# In labgen\_frontend/

python -m http.server 8080

You can now access the LabGen web app at

http://localhost:8080.

### How to Use

1. Open the Web App: Navigate to http://localhost:8080 in your browser.
2. Paste Protocol: Copy your scientific protocol and paste it into the large text area.
3. Generate Video: Click the "Generate Video" button.
4. Monitor Progress: The UI will update to show the status of the generation process (e.g., "Parsing protocol...", "Rendering video frames...").
5. View Result: Once complete, a video player will appear, ready for you to watch the generated (placeholder) demonstration.
6. Start Over: Click the "Start Over" button to reset the interface and generate a new video.

### API Endpoints

The backend provides the following REST API endpoints, prefixed with

/api/v1.

| Method | Endpoint | Description |
| --- | --- | --- |
| POST | /generate | Initiates a new video generation job. Accepts a JSON body with a protocol string. Returns a task\_id. |
| GET | /status/{task\_id} | Checks the status of a specific job (PENDING, PROCESSING, COMPLETED, FAILED). |
| GET | /video/{task\_id} | Retrieves the URL of the final video once the job is COMPLETED. |

Tab 3

