# App Architectures & User Flows: YouTube Transcript Tool

Following the feasibility assessment, this document outlines potential architectures and user flows for the proposed YouTube transcription tool, focusing on server-side processing for the Whisper model.

**Recommended Architecture: Web Application**

This is considered the most robust and accessible approach.

* **Frontend:**
  + **Technology:** React or Next.js (using create\_react\_app or create\_nextjs\_app templates for rapid development with UI components).
  + **User Interface (UI):**
    - Clean landing page with a clear value proposition.
    - Prominent input field for the YouTube URL.
    - Optional: Dropdown for desired transcription language (if using multilingual models).
    - Optional: Simplified quality/speed selector (e.g., “Fastest”, “Balanced”, “Most Accurate”) mapping to different Whisper models (tiny/base, small/medium, large) on the backend.
    - “Transcribe” button.
    - Status display area (e.g., “Idle”, “Downloading Audio”, “Transcribing (this may take several minutes)…”, “Complete”).
    - Results area: Displays the transcribed text directly (e.g., in a scrollable textarea).
    - Download buttons for .txt and .srt file formats.
  + **Functionality:** Handles user input, sends requests to the backend API, displays status updates (potentially via polling or WebSockets), renders the final transcript, and handles file downloads.
* **Backend:**
  + **Technology:** Flask (using create\_flask\_app template).
  + **API:** Defines endpoints to:
    - Receive transcription requests (YouTube URL, options).
    - Provide status updates for ongoing jobs.
    - Deliver the final transcript text and/or file download links.
  + **Core Logic:**
    - Validates the incoming YouTube URL.
    - Uses pytube to download the appropriate audio stream from the URL.
    - Calls the openai-whisper library to perform transcription using the selected model size. **This step MUST run on a server with adequate resources (GPU recommended for speed/accuracy).**
    - Formats the output into .txt and .srt files.
    - Stores results temporarily for retrieval by the frontend.
  + **Job Management (Optional but Recommended):** Implement a task queue (e.g., Celery with Redis/RabbitMQ) to handle transcription jobs asynchronously. This prevents API requests from timing out during long transcriptions and allows for scaling.
* **Infrastructure:**
  + Web server to host the frontend and backend (Flask) application.
  + One or more compute instances with GPUs dedicated to running the Whisper model (or usage of a cloud GPU service/Whisper API).
  + Potentially a database for user accounts, job tracking, or storing results.
  + Task queue broker (if using asynchronous processing).

**User Flow (Web Application):**

1. **Visit:** User navigates to the web application URL.
2. **Input:** User pastes a valid YouTube video URL into the input field.
3. **Configure (Optional):** User selects desired language or quality/speed setting.
4. **Initiate:** User clicks the “Transcribe” button.
5. **Feedback (Frontend):** UI updates to show “Processing…” or more detailed status (e.g., “Downloading audio…”).
6. **Request (Frontend -> Backend):** Frontend sends the URL and options to the backend API.
7. **Processing (Backend):**
   * Backend validates the request.
   * Backend (or a queued worker) downloads the audio using pytube.
   * Backend/worker runs the audio through the selected Whisper model on a GPU-enabled machine.
   * Backend saves the generated .txt and .srt files or stores the text content.
8. **Completion Notification (Backend -> Frontend):** Backend signals completion to the frontend (e.g., via API polling response or WebSocket message).
9. **Display Results (Frontend):**
   * UI updates to “Complete”.
   * The transcribed text is displayed in the results area.
   * Download buttons for .txt and .srt become active.
10. **Download (Optional):** User clicks download buttons to save the files.

**Alternative Architecture: Browser Extension + Backend**

* **Frontend (Extension):** UI integrated into the browser, potentially appearing as a button on YouTube pages. Extracts URL, sends to backend, displays status/results.
* **Backend:** Identical to the Web Application backend.
* **User Flow:** Similar, but initiated from the YouTube page via the extension button.
* **Pros:** Convenience for users already on YouTube.
* **Cons:** Requires separate development for different browsers, extension store approval process, potentially less discoverable.

This detailed outline provides a solid foundation for planning the development of the YouTube transcription tool.