This is a decisive move, Burnsie. You're shifting from waiting for direction to **defining the product** and providing the team with tangible assets they can use immediately.

Here is the Technical Design Document (TDD) for the V1 Teach-by-Doing (TbD) MVP. It is structured to be handed directly to a developer (or used by you) to build the system. It includes the data schema, the architecture, and the implementation plan.

Technical Design Document: Teach by Doing (TbD) MVP - V1.0

Status: Ready for Build

Author: Greg Burns

Date: November 21, 2025

Objective: To build a local, standalone prototype that ingests a screen recording of a user workflow and outputs a structured Pathway.json file, serving as the foundational "Data Ingestion Engine" for the FreeFuse PAD initiative.

---

1. System Overview

The TbD MVP is a **Screen-to-Schema** pipeline. It does not require real-time streaming or complex cloud infrastructure for V1. It focuses on the core problem: **Observation $\rightarrow$ Structure.**

1.1 The "Bridge" Strategy

To bypass the delay in the "TimeSuite" VCU project, this V1 uses a "Bridge Stack" of readily available, open-source CV tools to achieve immediate functionality.

- **Input:** A raw .mp4 video file of a user performing a task on their screen (e.g., navigating a software interface).
- **Core Logic:**
    1. **Segmentation:** Break video into "action events" based on visual changes.
    2. **Extraction:** Use OCR (Tesseract) to read UI text and Template Matching (OpenCV) to identify clicks/icons.
    3. **Structuring:** Map these events to the Pathway data schema.
- **Output:** A JSON file that machines (agents) and humans can read.

---

2. Data Schema Definition (The "Contract")

This schema is the most critical artifact. It defines the "common language" between the human expert and the AI agent.

2.1 The Root Object

JSON

```
{
  "pathway_id": "UUID-v4",
  "metadata": {
    "title": "Standard Operating Procedure: [Task Name]",
    "author_id": "user_123",
    "source_video": "s3://bucket/uploads/raw_video.mp4",
    "created_at": "ISO-8601 Timestamp",
    "total_duration_sec": 145.5
  },
  "nodes": []
}
```

2.2 The Node Object (The Atomic Unit)

Every detected event is a "Node." In V1, we focus on **ActionNodes**.

JSON

```
{
  "id": "node_1",
  "type": "action",
  "timestamp_start": 12.5,
  "timestamp_end": 14.2,
  "description": "User clicked 'File' menu",
  "data": {
    "action_type": "click", // enum: click, type, drag, scroll
    "ui_element_text": "File", // OCR result
    "ui_region": [10, 20, 100, 50], // [x, y, w, h] bounding box
    "confidence": 0.92
  },
  "next_node_id": "node_2"
}
```

---

3. Architecture & Tech Stack

This architecture is designed for local development now, with a clear path to GCP deployment (Cloud Run/Vertex AI) later.

3.1 The Stack

- **Language:** Python 3.9+

- **Video Processing:** OpenCV (cv2) for frame extraction and difference hashing.
- **OCR Engine:** Tesseract 5 (via pytesseract) for reading screen text.
- **Scene Detection:** PySceneDetect for splitting the video into logical "steps."
- **API Framework:** FastAPI (to serve the processing logic).
- **Container:** Docker (for portability).

3.2 The Pipeline Flow (The Algorithm)

Step 1: Ingest & Pre-process

- Upload .mp4.
- Convert to 1fps (frames per second) to reduce compute load for V1.

Step 2: Event Segmentation (The "When")

- Use PySceneDetect with ContentDetector algorithm.
- *Logic:* If pixel change > Threshold (30%), mark as a "New Scene" (new step).
- *Output:* A list of (start_time, end_time) tuples.

Step 3: Visual Extraction (The "What")

- For each scene, extract the **middle frame** (the stable state).
- Run **OCR** on the frame to get all text.
- *Heuristic:* If a specific small region changed significantly compared to the previous scene, assume that is the "Active Region" (e.g., a button press).

Step 4: Node Generation

- Create a Node object for each scene.
- Populate ui_element_text with the OCR text found in the "Active Region."
- Link nodes sequentially (node_1.next -> node_2).

---

4. Implementation Plan (Sprint 0)

This is the build order for your weekend sprint.

Phase 1: The Skeleton (Friday Night)

- [ ] Set up Python environment (requirements.txt: fastapi, opencv-python, pytesseract, scenedetect).
- [ ] Write schema.py: Pydantic models defining the JSON structure above.
- [ ] Write ingest.py: A script that accepts a file path and verifies it's a valid video.

Phase 2: The "Eyes" (Saturday)

- [ ] Implement segmentation.py: Use PySceneDetect to return a list of timestamps.

- [ ] Implement ocr.py: A function that takes a frame (image array) and returns text + bounding boxes.
- [ ] **Test:** Feed it a video of you clicking a "Save" button. Verify it outputs the text "Save".

Phase 3: The "Brain" & Output (Sunday)

- [ ] Implement pipeline.py: The orchestrator that loops through segments, calls OCR, and builds the JSON.
- [ ] Write main.py: The FastAPI entry point (POST /process-video).
- [ ] **Demo Artifact:** Run the script on your Surgical PoC usage video. Generate the output.json.

---

5. Deployment Strategy (GCP Ready)

Since you mentioned deploying to GCP:

1. **Dockerize:** Create a Dockerfile that installs system dependencies (libgl1, tesseract-ocr) and your Python requirements.
2. **Cloud Run:** Deploy the container to Google Cloud Run (Generation 2 for longer timeouts).
3. **Storage:** Use GCS buckets for video upload/storage.