Here is a technical summary of the challenges we faced during the development of **Chronosearch v2/v3**. These are excellent talking points for an interview because they show you can debug complex "Full Stack + Cloud" issues.

1. The "Zombie Code" Deployment

- **The Symptom:** You made changes to the code (like fixing the password length), but the error (`72 bytes limit`) kept happening exactly as before.
- **The Root Cause:** We were using a script (`setup_cloud.py`) that defined the Modal app but didn't explicitly "mount" the local `backend` folder. Modal was running a **cached, stale version** of the code from weeks ago, ignoring your local updates.
- **The Fix:** We switched to **Module-Based Deployment**.
- Command: `modal deploy -m backend.main`
- This forces Python to treat the folder as a package and syncs the exact local files to the cloud container.

2. The "72-Byte" Bcrypt Crash

- **The Symptom:** The backend crashed with `password cannot be longer than 72 bytes`, even when you manually sent a short password like "12345678".
- **The Root Cause:** It wasn't actually the password length. The library we were using, **`passlib`**, is old and unmaintained. It tried to read the version of the modern `bcrypt` library to check for compatibility but failed (`AttributeError: has no attribute '__about__'`). This crash triggered a misleading error message about password length.
- **The Fix:** We removed `passlib` entirely and rewrote `auth.py` to use **pure `bcrypt`**. This eliminated the dependency conflict and the error.

3. The "404 Not Found" / Connection Failed

- **The Symptom:** The frontend said "Connection Failed," and the Modal logs showed **zero** activity (no function calls).
- **The Root Cause:** A classic **API Mismatch**.
- **Frontend (`api.js`)** was sending a POST request to **`/login`** with **JSON** data.
- **Backend (`api.py`)** was listening on **`/token`** and expecting **Form Data** (`OAuth2PasswordRequestForm`).
- Because the URL and Content-Type didn't match, FastAPI rejected the request before it even entered your function code.
- **The Fix:** We renamed the backend endpoint to `/login` and updated it to accept a Pydantic model (`UserAuth`) so it could read the JSON.

4. The `ModuleNotFoundError` in Cloud

- **The Symptom:** The app deployed successfully, but as soon as a request hit it, it crashed with `No module named 'backend'`.

- **The Root Cause:** Python's "Relative Imports" (e.g., `from .common import app`) depend on where the script is executed from. In the cloud, the file structure didn't match the local environment's "root" context.
- **The Fix:** We standardized the imports to be absolute (`from backend.common import app`) and ensured the `modal deploy` command was run from the root directory so the package structure was preserved in the cloud.

5. Google OAuth "Origin Not Allowed"

- **The Symptom:** The Google Login popup appeared but immediately showed an error: `The given origin is not allowed for the given client ID`.
- **The Root Cause:** Google's security policy requires you to explicitly whitelist every domain that can initiate a login. `localhost:5173` was not on the list.
- **The Fix:** We updated the **Google Cloud Console** credentials to include `http://localhost:5173` in the "Authorized JavaScript Origins."

6. The "Infinite Indexing" Loop (Design Flaw)

- **The Symptom:** Every time you opened a video to watch it, the "Indexing" process would start again.
- **The Root Cause:** The `VideoPlayer` component (or the `upload` flow) didn't have a clear separation between **"Process Video"** (Write) and **"Watch Video"** (Read). The player was triggering the heavy AI job instead of just checking if the vectors already existed.
- **The Fix (Planned):** We are implementing a **"Write Once, Read Forever"** architecture where the Upload triggers the AI, and the Player only queries the database status.