Here’s the **expanded step-by-step workflow** that includes **initiations** and proper **code references** for each step:

**Step 1: Application Initialization**

This step involves initializing the React application, setting up routing, and connecting the Redux store.

**Files Involved:**

• index.tsx

• App.tsx

• store.tsx

**Workflow Details:**

1. **Rendering the Root Component (**index.tsx**):**

• The React application begins in index.tsx, where the root component is rendered.

• It sets up:

• **Redux Provider**: For global state management.

• **PersistGate**: To handle persistent Redux state.

• **RouterProvider**: For routing between application pages.

**Code Reference (**index.tsx**):**

import { createRoot } from 'react-dom/client';

import { Provider } from 'react-redux';

import { PersistGate } from 'redux-persist/integration/react';

import { createBrowserRouter, RouterProvider } from "react-router-dom";

import store from "./store.tsx";

import App from "./App";

const router = createBrowserRouter([

  {

    path: "/",

    element: <App />,

    children: [

      { path: "/record", element: <VideoAndSidebar mode="record" /> },

      { path: "/upload", element: <VideoAndSidebar mode="upload" /> },

    ],

  },

]);

const root = createRoot(document.getElementById('root')!);

root.render(

  <Provider store={store}>

    <PersistGate persistor={persistStore(store)}>

      <RouterProvider router={router} />

    </PersistGate>

  </Provider>

);

2. **Routing to** App.tsx**:**

• The App component is the root component where global app initialization begins.

• **Initiations in** App.tsx**:**

• Initializes piecesModelRef and xcornersModelRef for TensorFlow models.

• Checks the user authentication status via lichessTrySetUser.

**Code Reference (**App.tsx**):**

const App = () => {

  const piecesModelRef = useRef<GraphModel>();

  const xcornersModelRef = useRef<GraphModel>();

  const modelRefs = { piecesModelRef, xcornersModelRef };

  useEffect(() => {

    if (!userSelect().token) {

      lichessTrySetUser(navigate, dispatch);

    }

  }, []);

  return !loading && <Outlet context={modelRefs} />;

};

**Step 2: Loading Models and Resetting State**

This step sets up the TensorFlow models and clears previous game and corner states.

**Files Involved:**

• videoAndSidebar.tsx

• loadModels.ts

• gameSlice.ts

• cornersSlice.ts

**Workflow Details:**

1. **Initiation of Model Loading:**

• LoadModels is called in videoAndSidebar.tsx during the useEffect hook.

• **Location:** This is initiated when the VideoAndSidebar component mounts.

**Code Reference (**videoAndSidebar.tsx**):**

useEffect(() => {

  LoadModels(context.piecesModelRef, context.xcornersModelRef);

}, []);

2. **Implementation of Model Loading (**loadModels.ts**):**

• Loads the TensorFlow.js models for chess piece and corner detection.

**Code Reference (**loadModels.ts**):**

export const LoadModels = async (piecesModelRef: any, xcornersModelRef: any) => {

  piecesModelRef.current = await tf.loadGraphModel("/models/pieces-model.json");

  xcornersModelRef.current = await tf.loadGraphModel("/models/xcorners-model.json");

};

3. **Resetting State:**

• Resets the Redux states for game and corners.

• **Initiation:** Triggered in videoAndSidebar.tsx during useEffect.

**Code Reference (**videoAndSidebar.tsx**):**

useEffect(() => {

  dispatch(cornersReset());

  dispatch(gameResetStart());

  dispatch(gameResetMoves());

  dispatch(gameResetFen());

}, []);

Let’s address the **connection between Step 3 and Step 4**, as well as add **missing details** about how each step transitions to the next, ensuring a cohesive explanation of the workflow. I will expand the **steps between 3 and 4**, including additional details, triggers, and files. I’ll also elaborate on Step 5 by identifying all files involved in Lichess integration and how these integrations are initiated and processed.

**Step 3: Video Rendering and Frame Processing**

**Workflow Details:**

**1. Initiation of Video Rendering and Frame Processing**

• The video playback and frame processing begin when the Video component in video.tsx is rendered. This component orchestrates the video display and piece detection by initiating the findPieces function during the component mount phase.

• The findPieces function processes the video stream in real-time, analyzing each frame to detect chessboard corners and pieces.

**Code Reference (**video.tsx**):**

useEffect(() => {

  findPieces(

    piecesModelRef,

    videoRef,

    canvasRef,

    playingRef,

    setText,

    dispatch,

    cornersRef,

    boardRef,

    movesPairsRef,

    lastMoveRef,

    moveTextRef,

    mode

  );

}, []);

**2. Core Functionality of findPieces**

• **Frame-by-Frame Analysis:**

• The findPieces function runs a continuous loop (requestAnimationFrame) to process each video frame.

• TensorFlow.js models (piecesModelRef and xcornersModelRef) are used for predictions on the video frame to detect chess pieces and board corners.

• **Key Functions and Operations:**

• **Chessboard Corner Detection:**

• The function calculates keypoints using chessboard corner references (cornersRef) and the current dimensions of the canvas.

• These keypoints are transformed into a usable format via getInvTransform and other utilities.

• **Chess Piece Detection:**

• The video frame is processed using the detect function, which generates bounding boxes and confidence scores for detected pieces.

• The detect function leverages the TensorFlow model (piecesModelRef) for frame analysis.

**Code Reference (**findPieces.ts**):**

const loop = async () => {

  const keypoints = getKeypoints(cornersRef, canvasRef);

  const { boxes, scores } = await detect(piecesModelRef, videoRef, keypoints);

  const update = getUpdate(scores, squares);

  state = updateState(state, update); *// Updates the internal game state*

  requestAnimationFrame(loop); *// Continues the loop for real-time processing*

};

requestAnimationFrame(loop);

**3. Detection of Chessboard Corners and Pieces**

• **Chessboard Corners:**

• The getKeypoints function determines the exact locations of the corners on the chessboard relative to the canvas size.

• Detected corners are updated dynamically in the Redux state using dispatch(cornersSet(payload)).

**Code Reference (**detect.ts **and** findPieces.ts**):**

export const detect = async (modelRef, videoRef, keypoints) => {

  const preds = modelRef.current.predict(image4D); *// Model prediction on the current frame*

  const { boxes, scores } = getBoxesAndScores(preds, width, height, videoWidth, videoHeight, padding, roi);

  return { boxes, scores };

};

**4. State Updates and Scoring System**

• After detecting the pieces and corners, the state is updated to reflect the current game status.

• **Score Calculation:**

• The calculateScore function evaluates the confidence scores of detected pieces and their respective board squares.

• The updated game state is processed to generate potential moves, identifying the best moves and scoring them.

**Code Reference (**findPieces.ts**):**

const { bestMove, bestMoves } = processState(state, movesPairsRef.current, possibleMoves);

if (bestMove !== null) {

  boardRef.current.move(bestMove.sans[0]);

}

const payload = makeUpdatePayload(boardRef.current, false);

dispatch(gameUpdate(payload)); *// Dispatch updated state to Redux*

**5. Rendering Detected Information**

• The renderState function is called to visually overlay the detected pieces and corners onto the video feed.

• This function uses the updated state to draw bounding boxes, labels, and other visual aids onto the canvas.

**Code Reference (**renderState.ts**):**

renderState(canvasRef.current, centers, boundary, state);

**6. Loop Continuation and Memory Management**

• The loop continues indefinitely using requestAnimationFrame to ensure real-time detection and updates.

• TensorFlow.js handles memory management by disposing of tensors that are no longer needed after each iteration.

**Code Reference (**findPieces.ts**):**

tf.dispose([boxes, scores]); *// Prevents memory leaks*

requestAnimationFrame(loop); *// Continues processing the next frame*

**7. Connection to Redux**

• Once the pieces and corners are detected, the updated state is dispatched to Redux via dispatch(gameUpdate(payload)) for global availability.

• This updated state is then accessible to other components, such as the sidebar and UI elements, ensuring synchronization across the application.

**8. Chessboard Visualization and Overlays**

• **How the Chessboard is Rendered**:

• The renderState function dynamically overlays detected pieces and chessboard corners onto the video feed.

• The canvasRef is used to draw bounding boxes around detected pieces and mark corners using transformed keypoints.

• These visual elements ensure the user can see the detected chessboard grid and pieces in real-time.

• **Integration with the Chessboard Grid**:

• The detected corners (cornersRef) are used to calculate transformations for accurate placement of the grid and pieces.

• This is essential for mapping detected pieces to valid board squares.

Code Reference (renderState.ts):

export const renderState = (canvas, centers, boundary, state) => {

  const ctx = canvas.getContext('2d');

  ctx.clearRect(0, 0, canvas.width, canvas.height);

*// Draw bounding boxes for pieces*

  centers.forEach(center => {

    ctx.beginPath();

    ctx.arc(center[0], center[1], 5, 0, 2 \* Math.PI);

    ctx.fillStyle = "red";

    ctx.fill();

  });

*// Draw grid based on detected corners*

  boundary.forEach(line => {

    ctx.beginPath();

    ctx.moveTo(line[0][0], line[0][1]);

    ctx.lineTo(line[1][0], line[1][1]);

    ctx.strokeStyle = "blue";

    ctx.stroke();

  });

};

**Step 4: Dispatching Redux Actions for State Updates**

**Workflow Details:**

1. **Triggering Redux Updates:**

• After updating the local state in findPieces, it dispatches updates to the Redux store using dispatch(gameUpdate(...)).

**Code Reference (**findPieces.ts**):**

const payload = makeUpdatePayload(boardRef.current, false);

dispatch(gameUpdate(payload)); *// Dispatching the updated game state*

2. **Handling the Update in Redux (**gameSlice.ts**):**

• The dispatched action is handled in the Redux slice gameSlice.ts, which updates the global state.

**Code Reference (**gameSlice.ts**):**

const gameSlice = createSlice({

  name: 'game',

  initialState,

  reducers: {

    gameUpdate: (state, action) => {

*// Merges the new game state into Redux store*

      state.moves = action.payload.moves;

    },

  },

});

3. **Connecting to Sidebar Components:**

• Once the Redux state is updated, components like PlaySidebar dynamically react to these changes using Redux selectors (gameSelect).

**Code Reference (**playSidebar.tsx**):**

const moves = useSelector(gameSelect).moves;

**Step 5: Updating State and Interacting with Sidebar**

This step involves real-time synchronization of game state (Redux) with UI components (sidebars) and triggers necessary updates.

**Files Involved:**

• playSidebar.tsx

• recordSidebar.tsx

• gameSlice.ts

• cornersSlice.ts

**Workflow Details:**

1. **State Updates Drive Sidebar Interaction:**

• **Redux State Updates (**gameSlice.ts**)** trigger changes in the **sidebar components** such as PlaySidebar and RecordSidebar.

**Code Reference (**playSidebar.tsx**):**

const moves = useSelector(gameSelect).moves;

2. **User Interactions Affect State:**

• Sidebar actions, like recording or playing moves, dispatch new actions to Redux. These actions further modify the global state.

**Code Reference (**recordSidebar.tsx**):**

const handleRecordStart = () => {

  dispatch(recordGameStart());

};

3. **Connection to Step 5:**

• The sidebar components prepare **game data** (e.g., PGN) required for Lichess integration.

**Step 6: Lichess Integration**

**Files Involved:**

• videoAndSidebar.tsx

• lichess.ts

**Workflow Details:**

1. **Triggering Lichess Integration:**

• When moves are updated in Redux (gameSlice.ts), videoAndSidebar.tsx monitors these updates. If in “broadcast” mode, it calls lichessPushRound.

**Code Reference (**videoAndSidebar.tsx**):**

useEffect(() => {

  if (mode === "broadcast") {

    lichessPushRound(token, broadcastPgn, study.id);

  }

}, [moves]);

2. **Pushing Data to Lichess (**lichess.ts**):**

• lichessPushRound sends the PGN (Portable Game Notation) to Lichess for broadcasting.

**Code Reference (**lichess.ts**):**

export const lichessPushRound = (token, pgn, roundId) => {

  fetchResponse(token, `/api/broadcast/round/${roundId}/push`, { body: pgn, method: "POST" });

};

3. **Other Lichess Operations:**

• Data such as account details (lichessGetAccount), PGN imports (lichessImportPgn), and active games (lichessGetPlaying) are also integrated.

4. **Translating Chessboard State to Lichess:**

• **PGN Preparation**:

• The detected chessboard state (from step 4) is used to generate Portable Game Notation (PGN) representing the current game state.

• PGN includes all moves played so far, starting from the initial position (START\_FEN).

• **Synchronization with Lichess**:

• When a move is detected and added to the state, videoAndSidebar.tsx monitors these updates.

• In broadcast mode, it prepares the PGN and sends it to Lichess via the lichessPushRound function.

Code Reference (videoAndSidebar.tsx):

useEffect(() => {

  if (mode === "broadcast" && moves.length > 0) {

    const broadcastPgn = [

      `[FEN "${START\_FEN}"]`,

      `[Moves "${moves.join(' ')}"]`,

    ].join("\r\n");

    lichessPushRound(token, broadcastPgn, study.id);

  }

}, [moves]);