Based on the detailed analysis in your documents, here are my suggestions for fixing the three confirmed bugs:

Bug #7: State Update Race in handleCellUpdate

Problem: `scheduleAutoSave` is called with mixed state - new `pendingUpdates` but old `cellsToDelete`.

Fix: Use `useEffect` to coordinate state updates:

```
// Remove scheduleAutoSave from inside setPendingUpdates
const handleCellUpdate = async (cell, newData, prevData) => {
  // ... validation logic ...
  setPendingUpdates(prev => updatePendingUpdates(prev, updatedCell));
  // Don't call scheduleAutoSave here
};

// Add useEffect to handle auto-save scheduling
useEffect(() => {
  if (pendingUpdates.length > 0 || cellsToDelete.length > 0) {
    scheduleAutoSave(pendingUpdates, cellsToDelete);
  }
}, [pendingUpdates, cellsToDelete]);
```

This ensures `scheduleAutoSave` is always called with the latest state values after React has processed all updates.

Bug #8: Duplicate Pending Updates

Problem: Rapid sequential operations can cause race conditions where multiple state updates use the same `prev` value.

Fix: Use functional state updates with proper queuing:

```
// Create a state update queue to prevent race conditions
const [updateQueue, setUpdateQueue] = useState([]);

const queuePendingUpdate = (updatedCell) => {
  setPendingUpdates(prev => {
    // Always check for existing cell by ID to prevent duplicates
    const existingIndex = prev.findIndex(c => c._id === updatedCell._id);
    if (existingIndex >= 0) {
      // Replace existing
      const newArray = [...prev];
      newArray[existingIndex] = updatedCell;
      return newArray;
    } else {
      // Add new
      return [...prev, updatedCell];
    }
  });
```

```
};
```

Alternatively, use a more robust approach with `useReducer`:

```
const pendingUpdatesReducer = (state, action) => {
  switch (action.type) {
    case 'ADD_OR_UPDATE':
      const existingIndex = state.findIndex(c => c._id === action.cell._id);
      if (existingIndex >= 0) {
        const newState = [...state];
        newState[existingIndex] = action.cell;
        return newState;
      }
      return [...state, action.cell];
    case 'REMOVE':
      return state.filter(c => c._id !== action.cellId);
    case 'CLEAR':
      return [];
    default:
      return state;
  }
};

const [pendingUpdates, dispatchPendingUpdates] =
useReducer(pendingUpdatesReducer, []);
```

Bug #15: Async State Updates Interference

Problem: Multiple async operations (like delete row + delete column) can interfere and cause data loss.

Fix: Implement operation queuing or locking:

Option 1 - Operation Queue:

```
const [operationQueue, setOperationQueue] = useState([]);
const [isProcessingOperation, setIsProcessingOperation] = useState(false);

const queueOperation = async (operation) => {
  return new Promise((resolve, reject) => {
    setOperationQueue(prev => [...prev, { operation, resolve, reject }]);
  });
};

useEffect(() => {
  const processQueue = async () => {
    if (operationQueue.length === 0 || isProcessingOperation) return;

    setIsProcessingOperation(true);
    const { operation, resolve, reject } = operationQueue[0];

    try {
      const result = await operation();
```

```
      resolve(result);
    } catch (error) {
      reject(error);
    } finally {
      setOperationQueue(prev => prev.slice(1));
      setIsProcessingOperation(false);
    }
  };

  processQueue();
}, [operationQueue, isProcessingOperation]);

// Usage:
const handleDeleteRowBtnClick = async (currentRowIndex) => {
  return queueOperation(async () => {
    // ... existing deletion logic
  });
};
```

Option 2 - State-based Locking (Simpler):

```
const [isOperationInProgress, setIsOperationInProgress] = useState(false);

const handleDeleteRowBtnClick = async (currentRowIndex) => {
  if (isOperationInProgress) return; // Prevent concurrent operations

  setIsOperationInProgress(true);
  try {
    // ... existing deletion logic
  } finally {
    setIsOperationInProgress(false);
  }
};

const handleDeleteColumnBtnClick = async (currentColumnIndex) => {
  if (isOperationInProgress) return; // Prevent concurrent operations

  setIsOperationInProgress(true);
  try {
    // ... existing deletion logic
  } finally {
    setIsOperationInProgress(false);
  }
};
```

The queuing approach (Option 1) is more robust as it ensures operations complete in order, while the locking approach (Option 2) is simpler but may frustrate users if they click rapidly. Choose based on your UX requirements.