CodeAnt AI

AI SCAN ANALYSIS REPORT

Repository: saii-1903/13-arrythmia-project

Scan ID: rtzoefzsah

Branch: main

Analysis Date: 05/02/2026, 12:28:57

Total Issues: 109

Scan Prompt

No prompt specified

Analysis Results

Total Issues Found: 109

Function Name	File	Type	Lines	Issue
convert_record	wfdb_to_json.py	Function	21-74	1 - When resampling the signal (fs != 250), the code does not adjust the annotation sample indices (ann.sample) to the new sampling rate; annotations remain in the original sample-space, so beat-to-segment mapping becomes incorrect after resampling. - Resampling shortens/lengthens ecg (ecg = resample(...); fs = 250) but ann.sample are used unchanged to build beat_labels and then compared to segment indices of the resampled signal, causing a sample-space mismatch.2 - The try/except lumps all FileNotFoundError cases together and prints "Record ... not found", which is misleading: a FileNotFoundError raised while writing output (e.g., due to a missing OUTPUT directory) will produce the same message even though the record was read successfully. - The entire read/process/write sequence (including fname.write_text(...)) is inside the try, so any FileNotFoundError raised during writing is caught and results in the same "Record ... not found" message.
convert_record	afdb_to_json.py	Function	15-66	1 - The function never ensures OUTPUT (converted/AFDB) exists before writing files; calling Path.write_text will fail if the directory is missing. - Proof: fname = OUTPUT / ... followed by fname.write_text(...) without any mkdir call will raise FileNotFoundError if OUTPUT's parent directory doesn't exist.2 - FileNotFoundError is caught and reported as "Record {record_name} not found", which is misleading because the same exception can arise from missing output directories or other I/O issues (so the error message is incorrect and hides the real cause). - Proof: the except FileNotFoundError handler prints "Record ... not found" but the try block includes fname.write_text(...) which can raise FileNotFoundError for a missing output dir, so the message can be incorrect.3 - The code assumes ann.aux_note is an iterable of strings and directly zips/uses "in" on its elements; if ann.aux_note is None or contains non-string entries this will raise TypeError - there is no handling for absent/non-string aux_note entries. - Proof: zip(ann.aux_note, ann.sample) will raise TypeError if ann.aux_note is None, and "AFIB" in aos will raise TypeError if aos is a non-string non-iterable (e.g. float), with no guard in the code.
_load_data_from_json	app.py	Function	61-99	1 - The function fails to handle ECG_CH_B when channels are provided inside SensorData[0]; it only checks SensorData[0]["ECG_CH_A"], so JSON files with SensorData containing only "ECG_CH_B" will raise a ValueError despite the docstring implying support for ECG_CH_B at least at top-level. - Proof: In the SensorData branch the code only tests "ECG_CH_A" in row and never looks for "ECG_CH_B", and because the top-level channel checks are in elif chained to that if, a file with SensorData present but only ECG_CH_B leaves signal=None and triggers the ValueError.2 - The function does not attempt to read any sampling-rate metadata from the JSON (e.g., fields like "fs", "sampling_rate", or per-record metadata in SensorData) and instead relies solely on filename heuristics; this can yield incorrect original_fs for many valid JSON files that include their own sampling frequency. - Proof: The code never accesses any sampling-rate keys in data or row and only sets original_fs based on filename substring checks (PTBXL/MITDB), so any fs fields in the JSON are ignored.

Function Name	File	Type	Lines	Issue
_compute_qrs_durations	app.py	Function	218-260	1 - The function returns a non-empty fallback np.array([80.0]) on any exception, while normal "no data" cases return an empty array; this inconsistent behavior can bias downstream feature computation and model training/retraining logic. - proof: The function returns np.array([]) in multiple early "no data" branches but in the except block returns np.array([80.0]) as a fallback.2 - It uses a broad try/except that catches all Exceptions, prints the error, and swallows it (returning the fallback). This masks real errors and prevents proper error propagation or logging, harming diagnosability and synergistic operation with other components. - proof: The code has "except Exception as e: print(...); return np.array([80.0])", which catches all exceptions and only prints the message.3 - Comment and code disagree on the physiological QRS filter range (comment mentions "e.g. 40ms to 250ms" but the implemented filter uses 30ms to 300ms), which is an explicit inconsistency in intended vs implemented behavior. - proof: The inline comment states 40-250ms while the filter uses (durations_ms >= 30) & (durations_ms <= 300).
_sanitize_features	app.py	Function	294-305	1 - The function only checks scalar floats (float and np.floating) and does not recursively sanitize nested containers (lists, tuples, dicts) or numpy arrays; NaN/Inf values inside those structures will remain and violate "Ensure no NaN/Inf for JSONB." - Proof: the code only tests isinstance(v, (float, np.floating)) and otherwise assigns the value unchanged, with no recursion or handling for lists/tuples/dicts/ndarrays so NaN/Inf inside them are left intact.2 - The function does not normalize other non-native numeric types (e.g., np.integer, np.bool_, np.int64, pandas numeric/scalar types) to native Python types, which can leave values that are not JSON/JSONB-serializable and therefore fail the intended sanitation. - Proof: only float and np.floating are converted to native float via float(v); other numpy/pandas scalar types are not checked or converted and are returned as-is.
process_and_save_record	app.py	Function	405-494	1 - The function never performs model inference or records model/training metadata: the DB schema includes model_pred_label, model_pred_probs, and training_round, but the function does not call any model/prediction routines or populate these fields - proof: the CREATE TABLE defines those columns but the INSERT only supplies filename, segment_index, segment_start_s, segment_duration_s, r_peaks_in_segment, features_json and there are no model/prediction calls in the function body. 2 - filename_key = str(file_path.relative_to(DATA_ROOT_DIR)) can raise a ValueError if file_path is not located under DATA_ROOT_DIR; the function does not handle this possibility - proof: pathlib.Path.relative_to raises ValueError for out-of-tree paths and that call is executed unguarded at the start of the function.
upload_and_process	app.py	Function	560-584	1 - The exception handling returns the raw exception message to the client (f"ERROR: Processing or Database Insertion Failed: {e}..."), exposing internal error details and potentially sensitive server information instead of logging the error and returning a generic error response. - Proof: the except block returns a formatted string containing {e} (the caught exception) in the HTTP response body.2 - If db_service.get_first_segment_id_by_filename(filename_key) returns a falsy value, the code unconditionally sets new_segment_id = 1 and redirects to that segment, which can lead to redirecting to a non-existent/incorrect segment ID instead of handling "no segments found" explicitly. - Proof: the code does "new_segment_id = db_service.get_first_segment_id_by_filename(...)" followed by "if not new_segment_id: new_segment_id = 1" then redirects to that ID.3 - The uploaded file's type/content is not validated (no check that the file is valid JSON or has an expected extension/MIME type) before saving and processing, allowing invalid or unexpected files to be passed into process_and_save_record. - Proof: the handler directly calls file.save(filepath) and then process_and_save_record(filepath) with no content-type, extension, or JSON validation.
api_xai	app.py	Function	591-749	1 - The function's exception handling for XAI failures states "The model needs to be retrained with the current 9-class list" but implements no retraining, logging, scheduling, or flagging mechanism - the code only returns a placeholder response. - Proof: the except block merely prints the traceback and sets pred_label/explanation/saliency placeholders with no calls to any retrain/reset/log/schedule function (reset_model is imported elsewhere but not invoked here).2 - The API returns inconsistent response schemas across branches: the short-segment rejection returns {"pred_label","confidence","explanation"} while artifact-rejection and success paths return "probs","saliency","classes" (and artifact path adds "quality_issues"), which can break clients expecting a stable response shape. - Proof: the short-segment branch returns keys 'pred_label','confidence','explanation', the artifact branch returns ' pred_label','probs','explanation','saliency','classes','quality_issues' , and the normal success branch returns

Function Name	File	Type	Lines	Issue
				'pred_label','probs','explanation','saliency','classes'.
annotate_segment	app.py	Function	844-906	1 - The function never triggers model update/retraining or reset after saving a cardiologist annotation (no call to reset_model() or similar). - Proof: reset_model is imported at module scope but annotate_segment contains no call to reset_model and only calls db_service.update_annotation.2 - Validation is insufficient: it only checks `segment_id is None` and `doctor_rhythm_label is None`, so empty or blank strings (e.g., "") will pass validation and allow invalid annotations to be saved. - Proof: code uses `if segment_id is None or doctor_rhythm_label is None` which does not reject empty strings.3 - In the fallback for old frontends the code sets `doctor_ectopy_label = "None"` (the literal string) instead of using a consistent None/null value or an explicit enum value, creating inconsistent typing/semantics for the stored label. - Proof: fallback block contains the assignment `doctor_ectopy_label = "None"`.
export_corrected	app.py	Function	913-925	1 - The function re-imports export_corrected_segments inside the handler even though it is already imported at module level (redundant import). - proof: the module top contains "from export_corrected_segments import export_corrected_segments", so the in-handler "from export_corrected_segments import export_corrected_segments" is a duplicate.2 - The handler uses a broad except Exception and returns str(e) directly, which is poor error-handling practice and can leak internal error information. - proof: the code uses "except Exception as e: return jsonify({'error': str(e)})", which catches all exceptions and returns the raw exception string.3 - The handler calls export_corrected_segments synchronously in the request thread; if that export is long-running this will block the Flask worker and can cause timeouts or poor responsiveness instead of running the task asynchronously or in a background job. - proof: the handler invokes export_corrected_segments() directly, so its execution occurs in the request handler and blocks the Flask worker until it returns.
api_retrain_model	app.py	Function	998-1028	1 - The docstring lists xai.reset_model() but the function never calls xai.reset_model() (so the app does not trigger the model reset/reload step after starting retraining). - proof: the function body contains no call to reset_model() or xai.reset_model(), so the documented reset step is not executed.2 - The code opens "training_log.txt" in the parent process's current working directory while launching the retrain subprocess with cwd set to BASE_DIR / "models_training", which will place the log file in the wrong directory (likely unintended and confusing). - proof: open("training_log.txt", "w") is executed in the parent before Popen and creates the file in the parent's cwd; Popen(cwd=...) does not change where that already-open file was created.3 - The function starts retraining in the background without any check/lock to prevent concurrent retraining processes (multiple calls will spawn multiple retrain.py instances), which can lead to conflicting training runs and resource contention. - proof: the code unconditionally calls subprocess.Popen on each request and contains no mechanism (lock, PID check, or flag) to prevent multiple concurrent launches.
initializeZoom	zoom.js	ClassMethod	11-16	1 - initializeZoom only attaches mouse event listeners to the plot element and does not attach a global (document/window) mouseup listener; if the user presses inside the plot, drags outside, and releases the mouse outside the element, endZoom will not be called and zoom state may be left inconsistent (mouseleave calls cancelZoom rather than completing the zoom), breaking the intended synergy between startZoom and endZoom. - Proof: initializeZoom registers 'mouseup' only on this.plot and uses 'mouseleave' to call cancelZoom, so a mouseup outside the element cannot invoke endZoom.2 - initializeZoom does not register touch or pointer event handlers (touchstart/touchmove/touchend or pointerdown/pointermove/pointerup), so zoom interactions will not work on touch devices and the functionality is incomplete for non-mouse input. - Proof: initializeZoom only adds listeners for 'mousedown', 'mousemove', 'mouseup', and 'mouseleave' and contains no touch or pointer event registrations.
startZoom	zoom.js	ClassMethod	18-21	1 - startZoom does not check which mouse button was pressed (e.g., event.button === 0), so it will start a zoom on right-click/middle-click or other mouse buttons, which can conflict with expected UX (context menu, pan) and break synergy with other handlers. - proof: startZoom unconditionally sets this.isZooming = true and uses event.offsetX with no check of event.button, so any mousedown (left/right/middle) will trigger zoom start.2 - startZoom does not call event.preventDefault()/stopPropagation(), so starting a zoom can allow default browser behaviors (text selection, drag, context menu) that interfere with the intended zoom interaction. - proof: the function contains no calls to event.preventDefault() or event.stopPropagation(), so the browser's default/mouse

Function Name	File	Type	Lines	Issue
				handlers remain active when mousedown occurs.
endZoom	zoom.js	ClassMethod	31-47	1 - Leaves this.zoomStart set (stale) after both the early return for too-small selections and after completing the zoom; only isZooming is cleared, which can leave inconsistent internal state and cause incorrect behavior in subsequent interactions. - proof: endZoom sets this.isZooming = false but never resets this.zoomStart (cancelZoom is the only method that nulls zoomStart), so zoomStart remains the original value on both the early-return and success paths.
getTimeFromPosition	zoom.js	ClassMethod	54-58	1 - Does not guard against plot.clientWidth being zero or falsy, causing a division by zero and returning NaN/Infinity when computing the mapped time. - The code divides by this.plot.clientWidth with no check, so if clientWidth === 0 then (x/plotWidth) is Infinity (or leads to NaN when multiplied by 0), producing Infinity/NaN and there is no guard to prevent it.2 - Does not clamp or validate input x to the plot bounds; values <0 or >plotWidth will produce times outside the currentTimeRange and can lead to incoherent zoom ranges. - The function uses x directly in the linear mapping without clamping, so x < 0 yields times < currentTimeRange[0] and x > plotWidth yields times > currentTimeRange[1].3 - Does not validate that this.currentTimeRange is a valid [start,end] numeric pair or that timeRange is non-zero, which can produce incorrect results (or NaN) if the range is malformed or zero. - The code computes timeRange = this.currentTimeRange[1] - this.currentTimeRange[0] with no checks, so non-numeric/malformed entries produce NaN (and thus the function returns NaN), and if start===end timeRange === 0 the function will always return start regardless of x.
resetZoom	zoom.js	ClassMethod	60-63	1 - resetZoom() does not cancel or clear an in-progress zoom (it leaves isZooming and zoomStart unchanged) - proof: resetZoom only sets this.currentTimeRange and calls this.refreshDisplay(), it does not modify this.isZooming or this.zoomStart so an active drag remains active. 2 - resetZoom() calls the async this.refreshDisplay() without awaiting or returning the resulting Promise, leading to possible race conditions or unhandled rejections and preventing callers from knowing when the refresh completes - proof: resetZoom invokes this.refreshDisplay() but neither awaits nor returns its Promise, and refreshDisplay contains awaits (fetch) with no internal try/catch, so errors would propagate as an unobserved rejection and callers can't await completion.
refreshDisplay	zoom.js	ClassMethod	65-71	1 - Missing error handling: the function performs await fetch(...) and await response.json() with no try/catch, so network or parsing errors will produce unhandled promise rejections and break functionality - proof: the code directly awaits fetch and response.json with no surrounding try/catch or .catch handlers, so any thrown/rejected promise will propagate out of refreshDisplay.2 - No HTTP status check: the code does not verify response.ok or the HTTP status before calling response.json(), so error responses (4xx/5xx) may be treated as valid data - proof: the snippet calls response.json() unconditionally and never inspects response.ok or response.status.3 - No concurrency/cancellation handling: rapid successive calls to refreshDisplay can cause race conditions or stale data showing (no AbortController, no request sequencing or version check to ignore out-of-order responses) - proof: there is no AbortController, no token/version check, and updatePlot is called for every resolved response, allowing out-of-order responses to overwrite newer data.
file	index.html	N/A	1-884	1 - redrawECG computes visibleSamples = end - start but uses seg = ecgSignal.slice(start, end + 1) (which has length visibleSamples + 1) and then iterates for (let i = 0; i < visibleSamples; i++) with invVisible = 1/visibleSamples, so the final sample (index end) is never drawn and R-peak x calculations use a different denominator leading to misaligned coordinates - proven by the visibleSamples/end+1 slice and the for-loop/ invVisible usage in redrawECG. 2 - saveAnnotation hardcodes model_ectopy_label: "None" and sets model_rhythm_label from document.getElementById("arrLabel").innerText, so the payload always sends a fixed ectopy value and relies on UI text for the model rhythm rather than a parsed/stored model field - proven by those exact assignments in the saveAnnotation payload. 3 - populateRhythmDropdown returns early if (sel.options.length > 5) { return; }, which will prevent the dropdown from being repopulated with new serverClasses on subsequent loads and leave stale options - proven by the early-return guard at the top of populateRhythmDropdown.
analyze	analyze_dataset.py	Function	12-84	1 - The code drops composite labels returned by normalize_label (e.g. "Atrial Fibrillation + PVC Bigeminy") because it tests membership with "if lbl in CLASS_INDEX" against the joined

Function Name	File	Type	Lines	Issue
				string; composite labels are not split/expanded into valid classes and are therefore excluded from training-use counts - normalize_label returns " + ".join(...) for "+" inputs, and analyze only keeps normalized labels that are exact keys in CLASS_INDEX, so any joined composite string will not match and is filtered out.2 - The imbalance ratio calculation uses counts = Counter(valid) (which only contains observed classes) and computes min/max over counts.values(), so classes with zero occurrences (present in CLASS_NAMES/CLASS_INDEX but missing from the dataset) are ignored - this underestimates imbalance and yields incorrect ratio - Counter(valid) has no entries for absent classes, therefore min(counts.values()) cannot reflect zero-occurrence classes.3 - The DB connection is not managed in an exception-safe context (connect() used without a with-block); conn.close() is called unguarded, so an exception between connect() and close() can leak the connection - conn.close() is executed only after the cursor block with no try/finally, so any exception before that prevents conn.close() from running.
_load_model	xai.py	Function	89-159	1 - The function sets global _is_model_untrained = True on a load_state_dict RuntimeError but never resets _is_model_untrained to False on a subsequent successful load, causing the system to incorrectly think the model is untrained after a later successful reload. - Proof: the except branch unconditionally sets _is_model_untrained = True, and there is no assignment to False anywhere in _load_model (or before returning on a successful load), so once set it will persist across future reloads.
_compute_saliency	xai.py	Function	866-881	1 - The function never calls model.eval() or restores model.training, so stochastic layers (dropout/BatchNorm) may run in train mode and produce nondeterministic/incorrect saliency maps - the code contains no model.eval() or training-state handling. 2 - The function does not move or check the device of x (it only clones/detaches/requires_grad_), so passing a CPU tensor to a GPU model (or vice versa) will raise a device-mismatch at model(x). 3 - The function calls score.backward() (after only model.zero_grad()) and does not clear or restore model parameter gradients, so model.parameters() will have .grad populated after this call and those grads persist.
_analyze_attention	xai.py	Function	884-948	1 - The function hardcodes segment duration as 10s via secs_per_step = 10.0 / T, so index->time mapping is wrong for inputs not exactly 10s long - proof: the code unconditionally computes secs_per_step = 10.0 / T with no input/metadata used. 2 - The function silently returns an empty string on missing data or any exception, masking upstream errors - proof: it returns "" immediately if _last_attention is None and also returns "" in the except block after printing the error. 3 - Threshold logic is inconsistent and hardcoded (comment says >0.8 but code uses >0.7 then >0.5), leading to arbitrary peak detection behavior - proof: the comment "# Threshold: Find peaks > 0.8" is followed by peaks = np.where(attn_1d > 0.7)[0] and then a fallback to >0.5.
explain_segment	xai.py	Function	955-1066	1 - Saliency and attention are computed even when a clinical rule OVERRIDES the model, so saliency/attention can reflect the original model class (pred_idx) while pred_label/final_probs reflect the rule - proof: the code calls saliency = _compute_saliency(model, x, pred_idx) and attention_context = _analyze_attention(model) (which use the model/pred_idx) after computing pred_idx but after/independent of the override decision that sets final_prediction and final_probs. 2 - When _is_model_untrained is True and no rule overrides, the function still computes and returns model-based saliency and attention even though it replaces predictions and clears probabilities, producing misleading XAI outputs - proof: saliency and attention are computed before the if _is_model_untrained check that then replaces pred_label and final_probs with retraining notices.
predict_and_explain	xai.py	Function	1073-1130	1 - The exception return dict is missing the "cnn_featuremap" and "transformer_attention" keys that the normal return (and the docstring) promise. - Proof: the except block returns {"error":..., "pred_label":"Unknown", "probabilities":[], "saliency":[]} and does not include "cnn_featuremap" or "transformer_attention".2 - The function converts "probabilities" to a list but does not ensure "saliency" is JSON-serializable (no .tolist() or explicit conversion), leading to inconsistent/possibly non-serializable return types. - Proof: saliency is assigned via saliency = _compute_saliency(...) and returned directly, whereas probs is explicitly converted with probs.tolist(), so saliency's type is not normalized.
remove_baseline_wander	cleaning.py	Function	21-36	1 - The function does not validate that cutoff/normal_cutoff is within (0, 1) (i.e., cutoff < nyquist); if config or fs produce normal_cutoff <=0 or >=1, scipy.signal.butter will raise an error. - proof: normal_cutoff = cutoff / (0.5 * fs) is computed with no

Function Name	File	Type	Lines	Issue
				checks and scipy.signal.butter requires 0 < Wn < 1 and will raise ValueError if outside that range.2 - The function does not check signal length relative to filtfilt padlen (3*max(len(a), len(b))); filtfilt will raise an error for short signals, so short inputs can fail unexpectedly. - proof: the code calls filtfilt(b, a, signal) unconditionally and scipy.signal.filtfilt raises ValueError when signal length <= padlen (3*max(len(a), len(b))).3 - Unknown/unsupported cfg['method'] is silently ignored and the original signal is returned without warning or error, which can hide misconfiguration and lead callers to assume baseline removal occurred when it did not. - proof: if cfg['method'] != 'butterworth_highpass' the function simply returns the input signal with no logging or exception.
load_config	artifact_detection.py	Function	9-20	1 - The function returns the raw result of yaml.safe_load without validating or merging with fallback defaults; if the YAML exists but lacks the expected 'artifacts' keys (or keys within them), callers that access _CONFIG['artifacts'][...] will fail. - If CONFIG_PATH.exists() is True and the file's YAML omits 'artifacts' or nested keys, yaml.safe_load(...) is returned unchanged, so accesses like _CONFIG['artifacts']['flatline_threshold'] will raise KeyError/TypeError.2 - The function does not handle YAML parsing or file I/O errors (e.g., malformed YAML), so opening/parsing the config can raise exceptions and crash instead of falling back to safe defaults or surfacing a controlled error. - There is no try/except around open() or yaml.safe_load(), so OSError from file I/O or yaml.YAMLError from malformed YAML will propagate as exceptions.
check_signal_quality	artifact_detection.py	Function	24-105	1 - Docstring claims "sqi_score": float # 0.0 to 1.0, but the function returns the raw calculate_sqi_score value (0-100) and does not normalize it to 0-1 - Proof: the docstring annotates sqi_score as 0.0-1.0 while the code sets numeric_sqi = calculate_sqi_score(signal, fs) (and calculate_sqi_score's docstring and implementation use a 0.0-100.0 scale) with no normalization applied. 2 - The function's logic and thresholds (e.g., comparing numeric_sqi to 40.0 and 25.0 and capping to 40.0 when artifacts exist) operate on a 0-100 scale while the docstring and return type suggest a 0-1 scale, causing inconsistent semantics - Proof: the code uses comparisons numeric_sqi > 40.0, numeric_sqi < 25.0 and sets numeric_sqi = 40.0, which are incompatible with a 0.0-1.0 interpretation stated in the docstring.
compute_metrics	utils.py	Function	14-18	1 - The function does not validate that y_true and y_pred have the same length; passing mismatched-length inputs will raise exceptions (accuracy_score/f1_score) rather than being handled gracefully. - Proof: the code only checks len(y_true)>0 and then calls accuracy_score(y_true, y_pred) / f1_score without verifying len(y_pred)==len(y_true), so sklearn will raise a ValueError on mismatched lengths.2 - confusion_matrix is forced to use labels=list(range(len(CLASS_NAMES))). If y_true/y_pred contain labels outside that range or a different label set, those labels will be ignored or misrepresented, producing an incorrect confusion matrix. - Proof: the code calls confusion_matrix(y_true, y_pred, labels=list(range(len(CLASS_NAMES)))) which restricts the matrix to those integer labels and will not account correctly for any other label values/types in the inputs.3 - Inconsistent handling of empty inputs: accuracy and macro_f1 are set to 0.0 when y_true is empty, but the confusion matrix is still computed (and returned) unconditionally, which is inconsistent and can be misleading. - Proof: the code conditionally sets acc/macro_f1 when len(y_true)>0 else 0.0, but always executes confusion_matrix(...) afterwards, so an empty y_true yields 0.0 metrics but still returns a (zero) confusion matrix.
save_plot_prediction	utils.py	Function	20-29	1 - The function assumes indices from probs map directly into the global CLASS_NAMES list without validating lengths; if len(probs) > len(CLASS_NAMES) this will raise an IndexError when accessing CLASS_NAMES[i]. - proof: topk uses indices from probs and the list comprehension does CLASS_NAMES[i] with no bounds check, so any i >= len(CLASS_NAMES) will raise IndexError (e.g., probs length 20 vs CLASS_NAMES length 13).2 - The function formats probability values with f"{p:.2f}" but does not ensure p is a plain Python float/NumPy scalar; passing PyTorch tensors or other non-scalar types will raise a TypeError (no conversion like .item() is performed). - proof: the code uses f"{p:.2f}" without converting p, and objects like torch.Tensor do not support the 'f' format specifier and will raise a TypeError when formatted.
_load_data_from_json	ecgprocessor.py	Function	40-80	1 - The function does not check for an empty signal array: if the JSON contains an empty list for the channel, np.max/np.min will raise an exception (and downstream processing will fail). - proof: The code only tests signal is None, then calls np.max(np.abs(signal)) with no empty-array check, and np.max on an empty numpy array raises a ValueError ("zero-size array to

Function Name	File	Type	Lines	Issue
				reduction operation maximum").2 - The normalization step can divide by zero when the signal is constant (np.max(signal) == np.min(signal)), producing NaN/Inf values that will propagate to downstream functions; there is no guard against this. - proof: The code performs (signal - np.min(signal)) / (np.max(signal) - np.min(signal)) when the threshold passes but does not check if the denominator is zero.
_preprocess	ecgprocessor.py	Function	83-109	1 - The function does not validate that self.target_fs is sufficiently high for the hard-coded filter cutoffs (0.5, 40, 50±1, 60±1 Hz). - Proof: nyquist = 0.5*self.target_fs and the 60 Hz bandstop uses (60+1)/nyquist = 122/self.target_fs so for self.target_fs <= 122 the normalized cutoff >= 1 and scipy.signal.butter will raise a ValueError, yet no check/adaptation is performed in the code.2 - The function does not guard against very short input signals relative to filtfilt's padding requirements. - Proof: the code applies filtfilt with order-3 filters (len(b)=4) so padlen = 3*(4-1)=9 and filtfilt requires len(signal) > padlen, but no length check or fallback is implemented before calling filtfilt.
_r_peak_detection	ecgprocessor.py	Function	111-133	1 - The refinement picks np.argmax(signal[start:end]) on the signed waveform so inverted (negative) R-peaks will not be found correctly (it should consider magnitude or polarity) - proof: the code literally calls np.argmax on the raw segment (no abs or argmin), so negative R-peaks would yield the wrong index. 2 - Peak indices are computed on integrated_signal derived from np.diff(signal) (length N-1) but then used directly to index the original signal (length N), creating an index/alignment mismatch (off-by-one risk) - proof: squared_signal = np.diff(signal) -> len(squared_signal)=len(signal)-1 and integrated_signal is convolved from that, yet find_peaks indices are applied directly into signal. 3 - No deduplication or refractory enforcement is applied to refined_r_peaks before returning, so duplicates or very-close indices can occur and produce zero RR intervals downstream - proof: refined_r_peaks is appended to and returned as-is with no unique/sorting/min-distance checks.
_calculate_nonlinear_hrv	ecgprocessor.py	Function	183-212	1 - The >=20-beat guard makes the function return defaults for 5s segments: with SEGMENT_DURATION_S = 5.0 you would need 21 R-peaks in a segment to produce 20 RR intervals (21 beats / 5s = 252 bpm) which is physiologically implausible, so typical segments will always hit the early return. - Proof: SEGMENT_DURATION_S is 5.0 (global) and RR_intervals = beats - 1, so reaching 20 RR intervals requires 21 beats in 5s 252 bpm, outside normal ranges. 2 - The inner try/except ImportError around "from biosppy.signals.hrv import sample_entropy" is redundant/unreachable because sample_entropy is already imported at module top. - Proof: the file-level globals include "from biosppy.signals.hrv import sample_entropy", so the inner import cannot raise ImportError in a successfully loaded module.
_sanitize_features	ecgprocessor.py	Function	240-260	1 - Does not recursively sanitize nested structures (lists, dicts, nested dicts): NaN/Inf values inside lists or sub-dictionaries are left unchanged, so the function does not guarantee that the entire feature payload is JSON-serializable. - proof: the code iterates only top-level items and only handles float/np.floating, assigning all other values unchanged and never recursing into nested lists/dicts.2 - Does not convert numpy integer or boolean scalar types (np.integer, np.bool_) to native Python types: only np.floating is handled, so numpy integer/bool scalars can cause json.dumps to raise TypeError. - proof: the function checks isinstance(value, float) and isinstance(value, np.floating) but has no branch for np.integer or np.bool_, leaving those numpy scalars unconverted.3 - Leaves numpy arrays (np.ndarray) unchanged (and does not replace NaN/Inf inside them): JSON cannot serialize numpy arrays directly and arrays may contain NaN/Inf, so the function can fail to ensure JSONB compatibility. - proof: np.ndarray values fall into the else branch and are copied as-is without conversion or NaN/Inf checks.
_extract_segment_features	ecgprocessor.py	Function	263-311	1 - The function computes RMSSD as np.sqrt(np.mean(np.diff(rr_intervals_ms)**2)) while only checking len(rr_intervals_ms) > 0, so when rr_intervals_ms has length 1 np.diff(...) is empty, np.mean(empty) yields NaN and RMSSD becomes NaN (with a runtime warning).
process_and_save_record	ecgprocessor.py	Function	336-407	1 - segments_processed is incremented for every successful cur.execute call regardless of whether the INSERT actually inserted/updated a row (ON CONFLICT ... DO NOTHING can result in no row change). The printed "Successfully inserted/updated {segments_processed}" is therefore misleading and does not reflect actual rows inserted/updated. - Proof: the code does segments_processed += 1 immediately after cur.execute(...) with ON CONFLICT DO NOTHING, and there is no check of cur.rowcount, so conflict-suppressed non-inserts are still counted.2 - The database connection (conn) is not guaranteed to be closed on unexpected exceptions raised inside

Function Name	File	Type	Lines	Issue
				the with conn.cursor() block (there is no finally/try/finally around conn.close). An unhandled exception will bypass the conn.commit() and conn.close(), potentially leaking DB connections. - Proof: conn.close() is only called after the with-block and commit; any exception escaping the with-block will skip those lines and there is no finally to ensure close().3 - _sanitize_features does not convert non-float numpy scalar types (e.g., np.int64) or other non-serializable numpy objects into native Python types; such values would be left as-is and can cause json.dumps(sanitized_features) to raise a TypeError during insertion. - Proof: _sanitize_features only handles float and np.floating specifically and otherwise leaves values unchanged, so a np.int64 remains a numpy scalar which json.dumps cannot serialize.
main	export_sql_segments_to_json.py	Function	54-94	1 - The function writes JSON files to EXPORT_DIR without ensuring the directory exists (EXPORT_DIR is defined but not created), which will raise FileNotFoundError if the directory is missing. - EXPORT_DIR is set to Path("retraining_data") and the code never calls mkdir; Path.write_text does not create parent directories and will raise FileNotFoundError if that directory is absent.
get_segment_list	db_service.py	Function	23-52	1 - The function uses a bare except: which swallows all exceptions and returns an empty list, masking database errors and making callers unable to distinguish "no data" from an error condition. - Proof: the code contains "except:" with no exception type and immediately "return []", so any exception (including DB errors) is caught and converted to an empty list.2 - The function does not log or re-raise exceptions on failure (combined with the bare except), so failures are silently ignored and debugging/monitoring of DB issues is impossible. - Proof: the except block only returns [] and contains no logging calls or "raise", so no error information is recorded or propagated.
update_annotation	db_service.py	Function	123-251	1 - used_for_training is unconditionally set to FALSE in the UPDATE, so no records will ever be marked for retraining despite the function claiming to "classify the mistake for retraining" - Proof: the SQL literal contains "used_for_training = FALSE" with no conditional logic, so every update always resets that column to False.2 - The function contains duplicated conn.commit() and return cur.rowcount > 0 lines (second block unreachable), indicating a logic/flow bug - Proof: immediately after cur.execute the code does "conn.commit(); return cur.rowcount > 0" and then has another "conn.commit(); return cur.rowcount > 0" which is unreachable due to the prior return.3 - model_rhythm_label and model_ectopy_label default to None in the signature but the function raises ValueError if they are None - a signature/behavior mismatch that can mislead callers - Proof: the signature sets model_rhythm_label=None and model_ectopy_label=None but the code immediately checks "if model_rhythm_label is None or model_ectopy_label is None: raise ValueError(...)", causing omitted args to raise.
save_model_prediction	db_service.py	Function	256-276	1 - The function unconditionally returns True even when an exception occurs, so callers (including model training/retraining logic) cannot determine whether the database update actually succeeded. - proof: the function ends with "return True" after the try/except/finally and the except block neither changes the return value nor re-raises, so an exception path still returns True.2 - Exceptions are swallowed and only printed (no error propagation or clear error signaling and no explicit rollback), causing silent failures and preventing upstream logic from handling update errors properly. - proof: the except block only does print("DB ERROR save_model_prediction:", e) and does not re-raise the exception or call conn.rollback(), so errors are neither propagated nor rolled back.
get_min_segment_id_with_signal	db_service.py	Function	280-300	1 - The function catches all exceptions, prints an error, and returns 0, thereby conflating "no matching segment" with "database error"; this can silently break downstream/model training logic that expects a distinct failure vs. empty-result signal. - Proof: The code returns 0 both when no row/value is found (return ... else 0) and in the except block (except ... return 0), so an error and an empty result are indistinguishable. 2 - Error reporting uses print(...) instead of proper structured logging or raising, which harms observability and makes it hard for calling code to react appropriately. - Proof: The except block calls print("DB ERROR get_min_segment_id_with_signal:", e) and does not log or re-raise the exception, so callers cannot catch the original error and the message only goes to stdout.
get_first_segment_id_by_filename	db_service.py	Function	322-345	1 - The function swallows all exceptions and returns 0 on error, which hides database failures and prevents callers from distinguishing between a genuine segment_id of 0 and an operational error. - Proof: the except block does "print(...)" then "return 0", so any Exception yields 0.2 - Error reporting uses print(...) instead of a proper logging mechanism, which is

Function Name	File	Type	Lines	Issue
				inappropriate for reliable production diagnostics and monitoring. - Proof: the error path calls print("DB ERROR get_first_segment_id_by_filename:", e) rather than using the logging module.3 - The function conflates "no matching row" and "error occurred" by returning 0 in both cases; it should return a distinct sentinel (e.g., None) or raise an exception so callers can differentiate the cases. - Proof: on no row it does "return row[0] if row else 0", and on exception it also "return 0", so both cases produce 0.
clean_json	import_to_sql.py	Function	44-57	1 - The function only checks isinstance(obj, float) and thus does not handle NumPy scalar types (e.g., np.float64, np.int64). NumPy NaN/inf scalars will not be detected/converted to None and numeric NumPy scalars won't be converted to native Python types, causing potential JSON serialization or downstream errors. - Proof: isinstance(np.float64(np.nan), float) is False so a np.float64 NaN bypasses the float branch and remains unconverted.2 - The function does not handle numpy.ndarray (or other sequence types like tuples). NumPy arrays containing NaN/inf will be returned unmodified (and not converted to lists), so recursive cleaning and JSON-compatibility for array data are not guaranteed. - Proof: isinstance(np.array([np.nan]), list) is False so a numpy.ndarray hits the final return and is not recursively cleaned or converted.
compute_pr_interval	import_to_sql.py	Function	121-138	1 - The function broadly catches all exceptions (bare "except Exception") and silently returns None, which hides errors and failures in ECG delineation or data issues; this prevents proper error propagation, logging, and makes debugging or any downstream model training/retraining logic fragile. - Proof: the function ends with "except Exception: return None", so any exception is swallowed and no logging or re-raising occurs.
fetch_annotated_segments	db_loader.py	Function	10-67	1 - r_peaks normalization is inconsistent: when r_peaks is a string the code converts elements to int, but when it's a list/tuple it simply does list(r_peaks) without ensuring elements are ints. This produces inconsistent element types in "r_peaks_in_segment" and can break downstream/model training code that expects integers. - Proof: the code uses "rp = [int(x) for x in r_peaks.split(',') ...]" for string input but "elif isinstance(r_peaks, (list, tuple)): rp = list(r_peaks)" for list/tuple, so no int cast is applied in the latter branch.2 - The function uses bare except: clauses (in JSON loading and r_peaks parsing) which silently swallow all exceptions and make debugging and error propagation impossible. - Proof: both "except:" blocks after json.loads and the r_peaks int conversion catch all exceptions with no logging or re-raise, so errors are silently ignored.3 - The SELECT query has no ORDER BY, so results (and any limit applied) are nondeterministic across calls. That can harm reproducibility and any model training/retraining logic that relies on a stable data ordering. - Proof: the query string only selects from ecg_features_annotatable and conditionally appends " LIMIT ...", but contains no "ORDER BY" clause, so SQL does not guarantee row order.
save_to_db	synthesize_arrhythmias.py	Function	307-342	1 - Uses hardcoded sampling rate (360) and duration (10.0) in the INSERT instead of using the global FS and DURATION constants, breaking consistency/synergy with module-level settings. - Proof: the VALUES tuple passed to cur.execute contains the literal 10.0 and 360 while the module defines FS=360 and DURATION=10 which are not used.2 - Does not ensure cursor/connection are closed on exceptions (no finally or context manager), so DB resources may leak if an error occurs. - Proof: conn and cur are created inside the try but the except only prints the error and never calls cur.close() or conn.close(), so an early exception can leave them open.
get_output_gain	ECG_Workstation.py	Function	86-94	1 - Inconsistent interpretation of target_mv between modes: get_output_gain() returns np.clip(self.target_mv / 5.0, ...) when self.is_calibrated is False, but audio_callback uses gain = self.target_mv directly in Calibration mode, so the same slider value is interpreted differently depending on code path. - Proof: get_output_gain uncalibrated branch divides by 5.0, while audio_callback's Calibration branch uses target_mv raw.2 - Misleading docstring / unit handling: the docstring claims the function computes DAC amplitude "based on target mV and calibration", yet the uncalibrated branch treats target_mv as a UI percentage (dividing by 5) rather than mV, so the documentation does not match the implemented behavior. - Proof: uncalibrated branch comment says "target_mv acts as raw 0-1 percentage (scaled 0-5 for UI feel)" and the code divides target_mv by 5.0.
load_mit_record	ECG_Workstation.py	Function	96-127	1 - Resampling uses np.linspace(0, original_len, target_len) which includes original_len as an endpoint while xp=np.arange(original_len) ranges 0..original_len-1, so the interpolation will request x==original_len (outside xp) and be extrapolated instead of purely interpolated - np.linspace(..., original_len) yields a value beyond xp[-1]==original_len-1 as

Function Name	File	Type	Lines	Issue
				shown in the code. 2 - The function computes duration_sec = original_len / MIT_RESAMPLE_RATE instead of using the record's sampling rate (e.g. record.fs), so duration and target_len will be incorrect for records whose sampling frequency differs from MIT_RESAMPLE_RATE - record.fs is read by wfdb.rdrecord but never used.
audio_callback	ECG_Workstation.py	Function	129-232	1 - The code unpacks self.complex_gen.generate_segment(self.waveform) as "sig, _" but generate_segment returns three values (signal, r_peaks, metadata), so this will raise a ValueError for too many values to unpack. - Proof: generate_segment ends with "return signal, r_peaks, metadata" yet the call is "sig, _ = self.complex_gen.generate_segment(...)" which expects 2 values.2 - period_samples = int(self.fs / freq) can divide by zero if freq == 0 (e.g., self.bpm == 0 for NSR/PVC), causing a ZeroDivisionError before the later check for period_samples == 0. - Proof: for NSR/PVC freq = self.bpm/60.0, so bpm==0 => freq==0 and int(self.fs / freq) triggers division by zero.
__init__	ECG_Workstation.py	Function	235-247	1 - The constructor never binds the window close protocol to the on_close cleanup method, so closing the window via the window manager won't run the cleanup logic - proof: __init__ calls _setup_ui/_start_audio_stream/_start_scan_plot but there is no self.protocol("WM_DELETE_WINDOW", self.on_close) (nor any protocol binding) anywhere in the shown class while an on_close method exists. 2 - _start_audio_stream() is invoked synchronously in __init__ with no error handling, so failures opening the audio device will raise during construction and crash the app before the mainloop - proof: _start_audio_stream constructs and starts sd.OutputStream without try/except and is called directly in __init__, so any exception will propagate out of the constructor.
update_bpm_slider	ECG_Workstation.py	Function	367-377	1 - The local variable "freq" is computed (freq = value / 60.0) but never used - dead/unnecessary code inside the function. - Proof: freq is assigned and never referenced again in the function, so the value is unused.2 - The unit-labeling decision is overly narrow: it treats only engine.waveform values "NSR" and "PVC" as BPM and everything else as Hz, so many cardiac waveforms from the UI (e.g., "Sinus Bradycardia", "Sinus Tachycardia", "Atrial Fibrillation") will be labeled "Hz" instead of "BPM". - Proof: wave_options contains strings like "Sinus Bradycardia" and "Atrial Fibrillation" which are not in ["NSR","PVC"], therefore the else branch runs and sets the label to "Hz".
update_amp_slider	ECG_Workstation.py	Function	379-386	1 - update_amp_slider does not handle the "Calibration" mode: toggle_calibration_mode sets the amp_val_label to "DAC Level (Raw)" and reconfigures the slider to 0-1 for calibration, but update_amp_slider only checks engine.is_calibrated and will overwrite the label (and display percent) when the user moves the slider during calibration. - proof: toggle_calibration_mode sets amp_slider.from_=0,to=1 and amp_val_label="DAC Level (Raw)" while leaving engine.is_calibrated False, and update_amp_slider only branches on engine.is_calibrated so any slider movement hits the else branch and replaces the label. 2 - update_amp_slider unconditionally computes pct = (value / 5.0) * 100 assuming the slider range is 0-5; that assumption is invalid when the slider is reconfigured to 0-1 for calibration, so the displayed percentage (and semantics) will be incorrect during calibration or any other mode that changes the slider range. - proof: the else branch always divides by 5.0 to compute pct regardless of the slider's actual from_/to range (which toggle_calibration_mode sets to 0-1), making the percent value wrong.
change_waveform	ECG_Workstation.py	Function	397-401	1 - change_waveform relies on update_bpm_slider to pick BPM vs Hz but update_bpm_slider only treats self.engine.waveform in ["NSR", "PVC"] as BPM while the UI waveform list contains many cardiac names like "Sinus Bradycardia" and "Atrial Fibrillation", so selecting those will make the label show "Hz" incorrectly - proof: update_bpm_slider's conditional is literally "if self.engine.waveform in ['NSR', 'PVC']" yet the option menu contains many other cardiac strings, so they will fall into the "Hz" branch. 2 - change_waveform sets self.engine.complex_buffer = None without any synchronization while the audio stream is started with callback=self.engine.audio_callback (started in _start_audio_stream), so this unsynchronized write can race with the audio callback's reads - proof: _start_audio_stream creates sd.OutputStream(callback=self.engine.audio_callback) which invokes the callback on the audio thread but no locks are used around complex_buffer.
toggle_calibration_mode	ECG_Workstation.py	Function	403-436	1 - When entering Calibration the function sets self.engine.mode = "Calibration" but does not update the operation-mode segmented button (self.mode_seg). - proof: mode_seg was created with values ["Synthetic","MIT-BIH"] and toggle_calibration_mode sets self.engine.mode but never calls

Function Name	File	Type	Lines	Issue
				self.mode_seg.set() or otherwise updates that widget, so the segmented control's visible selection remains unchanged.2 - The function saves a measured calibration value into self.engine.calibration_gain but then ignores it: it unconditionally sets the slider and self.engine.target_mv to 1.0 on finish. - proof: the code assigns self.engine.calibration_gain = self.amp_slider.get(), then immediately sets self.amp_slider.set(1.0) and self.engine.target_mv = 1.0, and App.update_amp_slider never reads calibration_gain (it sets engine.target_mv directly), so the saved calibration value is not used to map slidermV in this class.
load_mit_data	ECG_Workstation.py	Function	438-457	1 - The function updates Tkinter widgets from the background thread (calls to self.mit_progress.set via update_prog, and self.mode_seg.set / self.change_mode / self.load_btn.configure inside _thread). Tkinter is not thread-safe and all UI updates must be performed on the main thread. - Proof: update_prog calls self.mit_progress.set and is passed as progress_callback to self.engine.load_mit_record running inside the background _thread, and that same _thread directly calls self.mode_seg.set, self.change_mode and self.load_btn.configure after load, all of which mutate Tk widgets from the worker thread.2 - The background thread does not catch exceptions from self.engine.load_mit_record; if that call raises, the thread will terminate silently and the UI remains in the disabled/"Downloading..." state (progress/button not restored). - Proof: _thread directly calls success = self.engine.load_mit_record(...) with no try/except, so any exception will abort the thread before the UI is reset, leaving the button disabled and text as "Downloading...".
_animate_scan	ECG_Workstation.py	Function	472-509	1 - The function uses a bare "except Exception: pass", which silently swallows all exceptions and hides errors from queue processing and plotting (makes debugging and correct inter-component behavior impossible). - Proof: The try block wraps queue processing and plotting and ends with "except Exception: pass", so any Exception raised inside (including during get_nowait(), set_ydata, or draw_idle) will be ignored.2 - The loop checks "while not self.engine.plot_queue.empty()" and then calls "get_nowait()", which is a race condition (queue can become empty between the empty() check and get_nowait()), leading to potential Empty exceptions that are currently being silently suppressed. - Proof: The code uses empty() then immediately get_nowait(), which can raise queue.Empty if another thread emptied the queue in between, and that exception is caught and discarded by the surrounding "except Exception: pass".
on_close	ECG_Workstation.py	Function	511-528	1 - Calls sys.exit() at the end, causing abrupt process termination which prevents a graceful shutdown and proper cleanup of background threads/resources (instead of letting the mainloop exit cleanly) - proof: the last line of on_close() is sys.exit(), which raises SystemExit and will terminate the process immediately unless caught, preventing any further shutdown code from running.2 - Does not signal or shut down the SignalEngine or any of its background worker threads (no engine shutdown/join), so engine threads or callbacks may continue running or access destroyed GUI state - proof: on_close() never calls any method on self.engine to stop or join threads; self.engine is not signaled or shut down in this function.3 - Uses a broad try/except Exception: pass when stopping/closing the audio stream, which silently swallows errors and can hide failures during shutdown - proof: the stream stop/close calls are wrapped in try: ... except Exception: pass, which suppresses any exceptions raised.
file	README.md	N/A	1-72	1 - The README does not perform or document any checks/verifications that "the code is calling each function correctly" or that functions "work synergistically"; there is no section describing call graphs, inter-module interaction, or runtime integration tests as the instruction requires. - The README only lists files and quick-start commands (project structure and how to run components) and contains no sections describing call graphs, inter-module interactions, or runtime/integration tests.2 - The README lacks any detailed verification of the model training and retraining logic (no description of training loop, hyperparameters, checkpointing/reloading behavior, validation checks, or how the dashboard-triggered retraining integrates with saved models), so it does not satisfy the requirement to check whether model training/retraining logic is proper. - Apart from a brief "Retraining" note to run train_balanced.py or use the dashboard, the README provides no details on training loops, hyperparameters, checkpointing/reloading, validation, or dashboard-model integration.
apply_clinical_rules	rhythm_orchestrator.py	Function	9-33	1 - The function defaults missing numerical features to 0 (e.g., hr = features.get("mean_hr", 0)), which can produce false positives (e.g., missing mean_hr -> hr == 0 -> returns "Sinus Bradycardia")

Function Name	File	Type	Lines	Issue
				- proof: features.get uses 0 as default so an empty dict yields hr=0 and the hr<45 branch returns "Sinus Bradycardia", 0.8, "HR < 45 bpm". 2 - Extracted features pr_interval and qrs_width are never used, indicating incomplete/incorrect rule implementation (dead/unutilized inputs) - proof: pr and qrs_width are assigned from features.get but are not referenced elsewhere in the function. 3 - Confidence scores and thresholds are hardcoded with no mechanism for calibration or integration with model training/retraining, so the function cannot be updated or synchronized with a learning pipeline - proof: numeric thresholds (hr > 150, rr_cv > 0.15, hr < 45) and confidences (0.7, 0.8, 0.65) are literal constants in the code with no parameters, config, or hooks for external calibration.
decide	rhythm_orchestrator.py	Function	39-105	1 - The arbitration does not implement the documented priority "Strong Rule Match -> Rule Label": rule_conf is never used as a general gate/threshold and only two specific label-based overrides are applied (e.g., SVT override uses rule_lbl equality, not a rule_conf check) - proof: apply_clinical_rules returns ("Supraventricular Tachycardia", 0.7, ...) for HR>150 but the decide() code overrides ML whenever rule_lbl == "Supraventricular Tachycardia" with no check on rule_conf (no >=0.9 or similar). 2 - When a clinical rule override occurs, the function still returns the original ml_prediction probabilities rather than adjusted probabilities for the rule, causing inconsistent outputs - proof: the final returned dict always includes "probabilities": ml_probs which is set once from ml_prediction and never modified when rule-based overrides change final_label.
run_validation_suite	automated_test.py	Function	70-185	1 - The generator sometimes returns two values for Ventricular Fibrillation but run_validation_suite always unpacks three (sig, r_peaks, meta), causing a ValueError when processing "Ventricular Fibrillation". - Proof: generate_segment contains "if primary_label == 'Ventricular Fibrillation': ... return signal, []" while run_validation_suite does "sig, r_peaks, meta = gen.generate_segment(label)".2 - Several test_classes strings do not match the exact substrings the generator checks (e.g., test has "2nd Degree AV Block Type 1" but generator looks for "Type 1 Wenckebach", "2nd Degree AV Block Type 2" vs "Type 2 Mobitz II", "1st Degree AV Block" vs "1' AV Block", "3rd Degree AV Block" vs "3' AV Block"), so those specific conditional branches will not be triggered. - Proof: run_validation_suite's list contains the shown human-readable forms while generate_segment performs substring checks like "Type 1 Wenckebach", "Type 2 Mobitz II", "1' AV Block", and "3' AV Block" which are not present in the test strings.
file	IMPLEMENTATION_PLAN_REMOVE_COMPOSITES_FIXED.md	N/A	1-71	1 - RETRAIN_CLASS_NAMES is built by checking the raw CLASS_NAMES for the literal " + " (and not normalizing them first). This contradicts the "Normalization First" rule and can miss composite names that use a different spacing/format (e.g., "AF+PVC"). - Proof: The comprehension shown uses if c not in EXCLUDED_PURE_SINUS and " + " not in c with no call to normalize_label, so a CLASS_NAMES entry "AF+PVC" would not contain the literal " + " and therefore would not be filtered as intended.
count	check_counts.py	Function	8-10	1 - SQL injection / unsafe query building: the function interpolates the raw where string into an f-string SQL statement (cur.execute(f"... WHERE {where}")), which definitively allows injection or malformed SQL if where contains untrusted input. - Proof: the code uses cur.execute(f"SELECT COUNT(*) FROM ecg_features_annotatable WHERE {where}") placing the raw where value directly into the SQL string without parameterization or sanitization.2 - Does not return the count value: the function only prints the count (print(...)) and returns None, which prevents callers from programmatically using the count and is not appropriate for modular or testable code. - Proof: the function body calls print(f"{label}: {cur.fetchone()[0]}") and contains no return statement, so it always returns None.3 - No error handling or transaction management and reliance on a global cursor: the function directly uses the module-level cur without handling exceptions, commits, or ensuring the cursor/connection state, which can lead to uncaught DB errors or inconsistent transaction state. - Proof: cur is a module-level cursor (cur = conn.cursor()) and the function calls cur.execute(...) with no try/except, conn.commit(), or rollback, so exceptions will propagate and transactions aren't explicitly managed.
file	IMPLEMENTATION_PLAN_SPLIT_MODELS_GUARDED.md	N/A	1-101	1 - Summary and guards use the name get_rhyth_label_idx (missing "m"), but the code snippet defines get_rhythm_label_idx - inconsistent function naming that will break references. - Proof: The "Summary of Guards" lists `get_rhyth_label_idx`, while the "Rhythm Mapper" code block defines `def get_rhythm_label_idx(...)`, showing a clear name mismatch in the document.2 - The snippets and assertions reference RHYTHM_CLASS_INDEX and ECTOPY_CLASS_INDEX, but the document only defines RHYTHM_CLASS_NAMES and

Function Name	File	Type	Lines	Issue
				ECTOPY_CLASS_NAMES (no index/mapping is shown), causing an undefined/inconsistent identifier issue. - Proof: The code returns `RHYTHM_CLASS_INDEX.get(...)` and uses `ECTOPY_CLASS_INDEX[...]`, yet only `RHYTHM_CLASS_NAMES = [...]` and `ECTOPY_CLASS_NAMES = [...]` are declared in the document with no `_CLASS_INDEX` mappings provided.
__init__	retrain.py	Function	48-50	1 - The constructor opens the log file directly without ensuring the log file's parent directory exists, which can raise FileNotFoundError if the directory hasn't been created beforehand. - proof: __init__ calls open(log_file, 'w') directly and contains no code to create parent directories (e.g., os.makedirs), so open will raise FileNotFoundError if the path's directory is missing.2 - The constructor uses mode 'w' (truncate) so it always overwrites any existing log file instead of appending, which can unintentionally lose prior logs. - proof: the code uses open(log_file, 'w', ...) and Python's 'w' mode truncates existing files on open.3 - There is no error handling around opening the log file (no try/except or fallback), so failures to open the file will raise exceptions that are not handled by the logger. - proof: __init__ contains no try/except block, so any exception from open(...) will propagate out of the constructor.
__init__	retrain.py	Function	90-191	1 - The constructor checks for patient_id/admission_id columns but does not include them in the SELECT or extract them from each row; patient_id and admission_id are always left as None, which will break any downstream patient-level splitting or logic that depends on actual patient/admission IDs. - proof: The SELECT only requests segment_id, arrhythmia_label, raw_signal, annotation_type, used_for_training, and the loop sets patient_id = None and admission_id = None before appending samples, so no real IDs are stored.2 - The code unconditionally filters by mistake_target in the WHERE clause without verifying that the mistake_target column exists in the table; if that column is absent the query will raise a runtime SQL error. - proof: target_filter injects "AND mistake_target = '...'" into the query but available_cols is never checked for 'mistake_target' before executing cur.execute(query).
analyze_and_suggest	balance_dataset.py	Function	16-122	1 - The code calls min([c for c in counts.values() if c > 0]) without handling the case where no class has >0 samples; if all counts are zero this produces an empty sequence and raises a ValueError (function will crash) - Proof: counts is a non-empty dict of zeros when there are no rows, so the list comprehension yields [] and min([]) raises ValueError since there's no guard against an empty list. 2 - The database connection is not protected by try/finally or a context manager; conn.close() is only called after successful fetch, so exceptions during query/fetch will leave the connection open (resource leak) - Proof: conn.close() is executed only after the with cur: block with no surrounding try/finally, so any exception in execute/fetchall will skip conn.close() and leave the connection open.
run_check	verify_db_checklist.py	Function	15-22	1 - The function catches all exceptions and only prints an error message without re-raising or returning any failure indicator, so callers cannot programmatically detect check failures (this can break orchestration of downstream model training/retraining logic). - Proof: the code uses "except Exception as e:" and the except block only calls print(f" Check Failed: {e}") with no raise or return, so exceptions are swallowed and not propagated.2 - The function returns no structured result (no boolean/status or exception), only printing to stdout; therefore it cannot be used by calling code to make deterministic decisions about proceeding with model training/retraining. - Proof: there is no return statement in the function (implicit return None), so no structured status is provided to callers.
check_0_connections	verify_db_checklist.py	Function	27-36	1 - Hardcoded database name in the SQL ("WHERE datname = 'ecg_analysis'") rather than using the provided connection's actual database (e.g., current_database() or conn parameters), so the function can give incorrect results when called with a connection to a different DB. - proof: the query string is a literal "WHERE datname = 'ecg_analysis'" and the function never inspects conn, so it will always filter for ecg_analysis regardless of the provided connection.2 - The function only prints results and returns nothing, preventing downstream code from programmatically using the query results for further checks or orchestration (breaking synergy with other functions expecting structured outputs). - proof: the function only calls print() and contains no return statement, so callers cannot access the fetched rows.3 - No error handling / no finally block: if get_cursor or cur.execute/cur.fetchall raises an exception the cursor will not be closed, risking resource leaks. - proof: cur.close() is called only at the end with no try/finally or except, so any exception before that will skip cursor cleanup.
check_1_db_table	verify_db_checklist.py	Function	38-47	1 - The function only prints information and does not return a

Function Name	File	Type	Lines	Issue
				boolean/structured result indicating whether the table exists, so it cannot be used programmatically by downstream checks or orchestration logic. - proof: the function has no return statement and only uses print(), so callers always receive None and no structured result.2 - There is no exception handling/finally block around DB operations; if an exception occurs before cur.close(), the cursor can leak (cursor.close() may never run). - proof: cur = get_cursor(conn) and cur.close() are not guarded by try/finally/with, so any exception from execute()/fetchone() will bypass cur.close().3 - The function does not validate or raise an error when the table is missing (it merely prints the to_regclass result), so it does not enforce the "check" semantics expected from a verification routine. - proof: the code prints the value of res from to_regclass but contains no conditional check or raise if res is None (table missing).
check_2_columns	verify_db_checklist.py	Function	49-76	1 - The function queries data_type and is_nullable but never uses them - it therefore fails to validate expected data types and nullability for required columns (important for correct model training/retraining). - Proof: the SQL selects column_name, data_type, is_nullable but the code only does cols = [r[0] for r in rows] and never inspects r[1] or r[2].2 - The cursor is not managed in a try/finally (or context manager); if cur.execute or fetchall raises, cur.close() will not run, causing a potential resource leak. - Proof: cur.close() is only called after execute/fetchall, so any exception from those calls will skip the close.3 - The function only prints results and does not return any structured status (e.g., boolean or list of missing/invalid columns), preventing programmatic coordination with other checks or downstream training/retraining logic. - Proof: there is no return statement and all outputs are produced via print(), so callers receive no return value.
check_3_domain	verify_db_checklist.py	Function	78-89	1 - The function lacks exception handling for database errors (missing table/columns or query failures); any SQL error will raise and break the checklist flow instead of being handled gracefully. - Proof: there is no try/except in the function and cur.execute(...) calls will propagate any psycopg2 errors out of the function.2 - The cursor is not guaranteed to be closed on error (no try/finally or context manager), leading to potential resource leaks if an exception occurs. - Proof: cur.close() is called only at the end of the function and there is no try/finally or with-context to ensure close() runs on exceptions.
check_4_integrity	verify_db_checklist.py	Function	91-110	1 - The function does not guarantee cursor cleanup on errors (no try/finally or context manager), so if cur.execute or fetchone raises an exception the cursor will not be closed, causing a resource leak. - Proof: cur.close() is called only after the execute/fetchone calls and there is no try/finally or with-statement, so any exception before cur.close() will exit the function without closing the cursor.2 - The function only prints the check results and returns nothing, preventing other code from programmatically consuming these integrity counts for downstream logic (breaking synergistic function composition). - Proof: the function has no return statement and only uses print(...) for n1 and n2, so callers cannot obtain the counts programmatically.
check_5_used_flag	verify_db_checklist.py	Function	112-127	1 - The function does not protect cursor/resource cleanup on error: it calls cur.close() only at the end, so an exception raised by cur.execute or cur.fetchone will skip closing the cursor (should use try/finally or a context manager). - proof: cur.close() is executed only after both execute/fetchone calls; any exception thrown beforehand will bypass that line and leave the cursor unclosed.2 - The SQL condition uses "used_for_training IS NOT FALSE", which will include NULL-valued flags (NULL is not FALSE) and therefore can produce incorrect counts; the boolean comparison is ambiguous and should explicitly handle NULLs (e.g., IS TRUE, <> FALSE, or IS DISTINCT FROM FALSE) depending on the intended semantics. - proof: the WHERE clause contains (used_for_training IS NOT FALSE), which matches TRUE and NULL values, so rows with NULL used_for_training are counted.3 - The function only prints n1 and n2 and does not return these values for downstream programmatic use, preventing other checks or the training/retraining logic from consuming the measured metrics. - proof: there is no return statement in the function and results are only printed via print(...).
check_6_preview	verify_db_checklist.py	Function	129-158	1 - The function only prints query results and does not return the fetched rows or any structured data, which prevents other code from programmatically using the preview results (breaking expected synergy with downstream logic). - Proof: the function assigns r_rows/e_rows from cur.fetchall() but contains no return statement, so those results are never returned to callers and are only printed.2 - There is no exception handling around the database operations (get_cursor, execute, fetchall, close), so DB errors will propagate and may leave resources unhandled-this

Function Name	File	Type	Lines	Issue
				undermines robustness and proper interaction with calling code. - Proof: the function contains no try/except/finally and calls cur.close() only at the end, so any exception before that will skip closing the cursor and propagate.3 - The function does not validate the provided conn (e.g., check for None or closed connection) before calling get_cursor(conn), risking failures when integrated with calling code that may pass invalid connections. - Proof: the function directly calls get_cursor(conn) with no checks (None/closed) on conn, so an invalid connection will cause immediate failure in get_cursor or subsequent DB calls.
check_7_leaking	verify_db_checklist.py	Function	160-169	1 - The function counts confirmed cases with used_for_training = FALSE instead of TRUE, so it will not flag confirmed cases that were included in training (leakage). - proof: The SQL WHERE clause explicitly uses "used_for_training = FALSE", therefore rows with used_for_training = TRUE (the leaking cases) are excluded from the count.
check_8_time	verify_db_checklist.py	Function	171-181	1 - The function does not guard cursor use with try/finally or exception handling; if cur.execute or cur.fetchone raises, cur.close() will not run and the DB cursor/resource can be leaked. - cur.close() is only called after fetchone(); if cur.execute or cur.fetchone raises an exception control exits the function before cur.close(), so the cursor is not closed.2 - It only checks presence (NULL vs non-NULL) of segment_start_s but does not validate that segment_start_s is a proper datetime/timestamp, has consistent timezone/granularity, or that timestamps permit a time-based train/validation split (ordering, uniqueness, distribution), so it does not actually perform a complete "time-based split sanity" check. - The SQL only computes COUNT(*) and COUNT(segment_start_s), which only tests nullness and performs no type, ordering, timezone, uniqueness, or distribution validation.
check_10_summary	verify_db_checklist.py	Function	183-197	1 - The function does not guard against exceptions and may leak the DB cursor (no try/finally or context manager to ensure cur.close() runs on error). - Proof: cur.close() is only called at the end with no try/finally or with-statement, so any exception raised before that will skip close() and leave the cursor open.2 - The function only prints the summary and does not return any structured data, preventing programmatic consumption/reuse by downstream model training or retraining logic. - Proof: there is no return statement and all output is printed, so callers cannot access the fetched rows programmatically.
run_migration	db_migration.py	Function	5-80	1 - The inline comment says "We use Text instead of ENUM..." but the code adds mistake_target as VARCHAR(20), contradicting the stated intent and potentially limiting/stressing the expected flexibility of the column type. - proof: the executed SQL is "ADD COLUMN mistake_target VARCHAR(20)" in the migration, so the column is created as VARCHAR(20) not TEXT/ENUM as the comment states.2 - On any exception the function calls sys.exit(1), forcibly terminating the whole process instead of raising/propagating the error for the caller to handle - this breaks proper orchestration with downstream training/retraining logic. - proof: the except Exception as e: block ends with sys.exit(1), which exits the Python process rather than re-raising the exception.3 - The information_schema checks filter only by table_name and do not include table_schema (or use table_schema=current_schema()), so the presence checks can be wrong when the table is in a non-default schema or multiple schemas contain the same table name. - proof: both SELECTs query information_schema.columns with WHERE table_name='ecg_features_annotatable' AND column_name=... and contain no table_schema filter.
file	IMPLEMENTATION_PLAN_FIX_SINUS.md	N/A	1-83	1 - The runtime "Double Safety" check is logically incorrect: it tests RETRAIN_CLASS_NAMES[label_idx] against EXCLUDED_PURE_SINUS, but RETRAIN_CLASS_NAMES was defined by excluding EXCLUDED_PURE_SINUS, so this check can never detect a leaked pure-Sinus label (and may be indexing the wrong label space). - Proof: RETRAIN_CLASS_NAMES is built as [c for c in CLASS_NAMES if c not in EXCLUDED_PURE_SINUS], yet the sanity check does if RETRAIN_CLASS_NAMES[label_idx] in EXCLUDED_PURE_SINUS, so nothing in RETRAIN_CLASS_NAMES can match EXCLUDED_PURE_SINUS and the condition cannot be true.2 - RETRAIN_CLASS_NAMES is built by direct string membership against EXCLUDED_PURE_SINUS without applying normalization. - Proof: The comprehension uses CLASS_NAMES values verbatim (no normalize_label call), so only exact-string matches to EXCLUDED_PURE_SINUS are excluded and normalized/whitespace/case variants could be missed.
write	train_balanced.py	Function	40-43	1 - The method flushes only the file log (self.log.flush()) and does not flush the terminal (self.terminal), which can leave console output buffered and out of sync with the log. - proof: the function

Function Name	File	Type	Lines	Issue
				calls self.log.flush() but contains no call to self.terminal.flush() after writing to terminal.2 - The method lacks error handling around the two writes; if self.terminal.write(message) raises, self.log.write(message) never runs (and vice versa), making logging brittle and potentially losing output. - proof: write executes self.terminal.write then self.log.write sequentially with no try/except, so an exception in the first call prevents the second from running and stops the flush.
forward	train_balanced.py	Function	66-70	1 - The function passes class weights into F.cross_entropy so pt becomes p_t ** weight, making the focal term use p_t**weight instead of p_t. - proof: F.cross_entropy applies class weight multiplicatively (ce = -w_c * log p_t), so pt = exp(-ce) = p_t**w_c and therefore ((1 - pt)**gamma) evaluates to (1 - p_t**w_c)**gamma rather than (1 - p_t)**gamma.
__getitem__	train_balanced.py	Function	188-203	1 - __getitem__ returns the cached numpy array by reference when self.augment is False (sig = self.signal_cache.get(...)) instead of returning a copy or a tensor; this allows downstream in-place ops to mutate the cache and cause data contamination/leakage across samples/epochs, which can break training/retraining correctness. - proof: The code sets sig = self.signal_cache.get(seg_id, ...) and when self.augment is False does not copy it before returning, so any in-place mutation of that numpy array will modify the cached object stored in self.signal_cache.
train_epoch	train_balanced.py	Function	233-259	1 - The function unconditionally divides by len(y_true) to compute the average loss; if the loader yields zero samples (empty loader / zero batches) this will raise a ZeroDivisionError - there is no guard or handling for an empty epoch. - proof: y_true is initialized as [] and only extended inside the loop, so if the loader yields no batches len(y_true) == 0 and total_loss / len(y_true) will raise a ZeroDivisionError.
calculate_metrics_report	metrics.py	Function	5-37	1 - classification_report is called with target_names=class_names while confusion_matrix is called with labels=range(len(class_names)); this inconsistent label handling will produce mismatched/incorrect metrics when y_true/y_pred are not encoded as 0..n-1 - proof: classification_report maps y labels to target_names but confusion_matrix is forced to look for integer labels 0..n-1, so if y_true/y_pred are string class names the confusion matrix will not match those labels (yielding incorrect/zero counts).2 - The function computes per-class sensitivity (sens) inside the loop but never returns or injects these sensitivity values into the output; the docstring promises sensitivity and specificity but sensitivity is not explicitly provided by the function - proof: sens is assigned each loop iteration but never stored in specificities or added to cr and the function only returns (cr, specificities), so the local sens values are unused.
file	MODEL_INTENT.md	N/A	1-29	1 - The document lacks any code-level interface details showing how SmallCNN outputs are consumed by TransformerEncoder (no tensor shapes, API/function calls, embedding dimensions, or example call sequence), so it does not check whether the functions are called correctly or interoperate as required. - Proof: The Markdown only gives high-level roles for SmallCNN and TransformerEncoder and contains no tensor shapes, embedding dimensions, API/function calls, or example call sequence. 2 - There is no description of model training or retraining logic (no loss functions, optimizers, learning rate schedule, training loop, checkpointing, evaluation metrics, or retraining triggers), so the requirement to verify training/retraining logic is not met. - Proof: The document discusses architecture and rationale but includes no training details such as loss, optimizer, LR schedule, training loop, checkpoints, metrics, or retraining triggers. 3 - The document contains no verification or testing information demonstrating that the components work synergistically in practice (no unit/integration tests, expected runtime behavior, or empirical evidence of the hybrid architecture's correctness). - Proof: The Markdown provides only design intent and rationale and contains no unit/integration tests, runtime expectations, benchmarks, or empirical results.
file	IMPLEMENTATION_PLAN_REMOVE_COMPOSITES.md	N/A	1-66	1 - The plan's Core Constraints claim maps "Sinus + X" to "Sinus" (dropped) but the code sets base_rhythm = parts[0] and returns get_retrain_label_idx(base_rhythm), so the implementation uses the literal base token (e.g., "Sinus Bradycardia") rather than the normalized "Sinus" - proof: Core Constraints text states "Sinus + X -> 'Sinus'" while the get_retrain_label_idx code clearly extracts and recurses on parts[0]. 2 - The implementation and verification reference RETRAIN_CLASS_INDEX (and CLASS_NAMES / EXCLUDED_PURE_SINUS) but the snippet only shows building RETRAIN_CLASS_NAMES and does not define or show how RETRAIN_CLASS_INDEX is constructed or kept in sync - proof: get_retrain_label_idx uses RETRAIN_CLASS_INDEX and the verification asserts it, yet no RETRAIN_CLASS_INDEX assignment is present in the provided Markdown. 3 - The

Function Name	File	Type	Lines	Issue
				document modifies data loader and verification assertions but does not show any updates or tests for training/retraining pipeline components (training loop, loss, checkpoints, class weights, dataset splits, evaluation) so end-to-end propagation is unverified - proof: Proposed Changes and Verification sections only touch data_loader.py and verify_retrain_setup.py with no references to training code or downstream artifacts.
evaluate	evaluate_detailed.py	Function	176-250	1 - The code redirects sys.stdout to the report file but does not guarantee restoration on exception (no try/finally or use of contextlib.redirect_stdout). If print_evaluation_report or file IO raises, sys.stdout will remain redirected, causing global side effects. - Proof: the code does original_stdout = sys.stdout; then inside with open(...): sys.stdout = f; print_evaluation_report(...); and only after the with-block does it do sys.stdout = original_stdout, so any exception inside the with prevents the restoration line from running.2 - The function writes the evaluation report to BASE_DIR/.../logs/... without ensuring the parent directories exist. If the "outputs/logs" directory is missing, open(report_file, 'w') will raise and the evaluation will fail. - Proof: report_file is built as BASE_DIR/"models_training"/"outputs"/"logs"/..., and the code calls open(report_file, 'w') without any mkdir(parents=True) or existence check, so Python will raise FileNotFoundError if the parent directory does not exist.
file	BUG_FIXES_APPLIED.md	N/A	1-234	1 - The patient-level split snippet uses patient_ids but calls train_test_split(unique_patient_list, ..., stratify=patient_labels) where unique_patient_list and patient_labels are never defined in the snippet - the snippet shows patient_ids earlier but no definitions for unique_patient_list or patient_labels. 2 - The __getitem__ example accepts idx but calls self.signal_cache.get(seg_id, ...) while seg_id is never derived from idx or defined in the method - the snippet shows idx in the signature and only seg_id in the body. 3 - The Markdown declares the training script "production-ready" but the document contains no training loop, checkpointing, optimizer/state handling, or any training/validation execution code to substantiate that claim - none of those artifacts appear in the provided content.
run_migration_ectopy	db_migration_ectopy.py	Function	5-55	1 - Hardcoded database credentials (host, database, user, password) are embedded directly in the function instead of being loaded from configuration or environment variables. - The function creates conn_params with literal "host", "database", "user", and "password" string values defined inline.2 - The migration sets a default value of the literal string 'None' for ectopy_label instead of using NULL or an appropriate sentinel, which can break downstream logic/model training that expects NULL/absent values. - The ALTER TABLE statement explicitly adds "ectopy_label VARCHAR(50) DEFAULT 'None'", i.e., the literal string 'None' is used as the default.3 - The function calls sys.exit(1) inside its exception handler, forcibly terminating the process rather than raising the exception or returning an error for the caller to handle. - The except block prints the error and calls sys.exit(1), which will exit the process on any caught exception.
evaluate_rhythm	evaluate_rhythm.py	Function	19-94	1 - The checkpoint loading logic only checks for "model_state" and otherwise passes the entire checkpoint dict to model.load_state_dict; it does not handle the common "state_dict" key (or other checkpoint wrappers), which will cause load_state_dict to fail or receive an unexpected structure for many typical checkpoints. - Proof: The code does state = torch.load(...); model.load_state_dict(state["model_state"] if "model_state" in state else state) so a checkpoint with "state_dict" will fall to the else branch and pass the whole wrapper dict (not state["state_dict"]) into load_state_dict, leading to key mismatches or errors.2 - The code calls y.numpy() when aggregating labels (y_true.extend(y.numpy())) without ensuring y is a CPU tensor; if collate_fn or the dataset returns y on the GPU or returns a non-tensor type, this will raise an exception. - Proof: Inside the loop for x, y, _, _ in loader the code directly does y_true.extend(y.numpy()) with no .cpu() or type check, so a CUDA tensor will cause a RuntimeError and a non-tensor will cause AttributeError.
file	IMPLEMENTATION_PLAN_REMOVE_SINUS.md	N/A	1-120	1 - SQL vs Python filtering mismatch: the proposed SQL excludes abbreviations ('SR','SB','ST') but RETRAIN_CLASS_NAMES only filters names containing "Sinus" (and optionally "Artifact"), so abbreviation labels could remain in the class list while being removed by SQL - the SQL NOT IN clause includes 'SR','SB','ST' but the RETRAIN_CLASS_NAMES comprehension does not filter those strings. 2 - Label-normalization / mapping inconsistency: get_retrain_label(...) is defined but the processing loop uses normalize_label(...) then directly checks RETRAIN_CLASS_INDEX membership, so the plan never ensures normalize_label's outputs align with RETRAIN_CLASS_INDEX keys - the snippet shows

Function Name	File	Type	Lines	Issue
				get_retrain_label unused while the loop relies on normalize_label.
file	IMPLEMENTATION_PLAN_SPLIT_MODELS.md	N/A	1-86	1 - Contradictory Bigeminy mapping between Model B classes and mapper: Model B classes list Bigeminy under both "PVC" and "PAC", while get_ectopy_label_idx explicitly maps "Bigeminy/Trigeminy" to "PVC", proving a contradiction. 2 - Incomplete Rhythm SQL_FILTER: the document shows "WHERE arrhythmia_label != 'Sinus Rhythm'..." with an ellipsis, so the full WHERE clause/filter logic is not specified.
__init__	models.py	Function	60-91	1 - Uses a mutable default argument for cnn_channels (cnn_channels=[32, 64, 128]) in the __init__ signature, which is a potential bug (shared mutable state across instances). - proof: the __init__ signature uses a list literal as a default (cnn_channels=[32, 64, 128]); that list is created once at definition time and would be shared if mutated across instances.2 - The initializer does not create or register any positional encoding / temporal embedding for the Transformer; without positional information the Transformer cannot use sequence order, which breaks the intended CNNTransformer temporal synergy. - proof: __init__ defines no positional embedding module and forward passes z straight into self.transformer_encoder (z is not augmented with any positional encodings).3 - The class-level TARGET_LEN (2500) is not used or validated in this __init__; there is no enforcement or handling of expected input length which can lead to mismatches during training/inference. - proof: TARGET_LEN is declared on the class but neither __init__ nor forward reference or check TARGET_LEN, so input length is neither validated nor enforced.
forward	models.py	Function	93-100	1 - The sequence embeddings are passed to the TransformerEncoder with no positional encoding added (no learned or fixed positional embeddings), so the transformer has no explicit information about token/time order. - proof: forward permutes z and directly calls self.transformer_encoder(z) with no positional additions, and __init__ defines no positional embedding/encoding module.
file	IMPLEMENTATION_PLAN_VALIDATION_STRATEGY.md	N/A	1-35	1 - The plan adds segment_start_s to each sample tuple but does not specify updating all code paths that unpack/use sample tuples (e.g., ECGRawDatasetSQL.__getitem__, collate_fn, dataloader consumers). Changing the tuple structure will break existing function calls that expect the old schema. - The document explicitly changes the tuple shape to include segment_start_s but contains no instructions to update any unpacking sites or consumers that would rely on the previous tuple layout.2 - The document triggers the time-based branch only when len(unique_patients) <= 1, but the Objective explicitly calls out "single-patient (or few-patient) scenarios." The plan therefore fails to handle "few-patient" cases (small patient counts >1) where random patient-level splitting may still be unsafe. - The proposed check is exactly len(unique_patients) <= 1, so datasets with 2+ patients will not use the time-based split despite the stated goal.3 - The plan mandates "No Shuffling" but omits explicit instructions to disable all downstream shuffling (e.g., DataLoader shuffle=True, Samplers, or other train/val sampling utilities). Without those changes, shuffling could be reintroduced and re-enable leakage. - The proposal sorts and assigns splits but contains no guidance to set DataLoader/Sampler shuffle flags to False or otherwise prevent downstream re-shuffling.
file	IMPLEMENTATION_PLAN_MISTAKES_AND_WEIGHTS.md	N/A	1-52	1 - The plan omits any validation / holdout / evaluation procedure (metrics, validation set, or test set) for retraining - it only mentions a "Dry-run to ensure loss stability," which is insufficient to detect overfitting or measure generalization when training on a much smaller, mistake-only dataset. - Proof: The only Verification step shown is "Log the new dataset size... Dry-run to ensure loss stability," and the document contains no mention of validation/test splits, evaluation metrics, or holdout procedures.2 - The document instructs clamping class weights (`class_weights = torch.clamp(class_weights, max=5.0)`) but does not specify recomputing those class weights after applying the mistake-only filter - clamping precomputed weights from the original distribution would be incorrect for the new training set. - Proof: Change 4.B prescribes torch.clamp in retrain_model(), and Change 5 introduces a mistake-only SQL filter, but nowhere does the document state that class weights are recomputed on the filtered dataset.3 - Change 5 mandates excluding all correct predictions and training strictly on mistakes/borderline samples without any mitigation (e.g., reintroducing a controlled fraction of representative correct/normal examples or other calibration) - this creates a clear class-prior / distribution-shift risk and no countermeasure is specified. - Proof: The SQL filter uses annotation_type IN ('FALSE_POSITIVE','FALSE_NEGATIVE','BORDERLINE') and the text explicitly says "Correct predictions ('REDUNDANT' data) are now strictly excluded," with no subsequent mitigation or calibration strategy described.

Function Name	File	Type	Lines	Issue
get_rhythm_label_idx	data_loader.py	Function	96-111	1 - The function contradicts its docstring for ectopy labels (e.g., "PVCs"): it can return a valid RHYTHM_INDEX for labels like "PVCs" instead of dropping them (returning None). This happens because RHYTHM_INDEX (built from RHYTHM_CLASS_NAMES) includes ectopy class names, so get_rhythm_label_idx(normalize_label("PVC") / "PVCs") will yield an index rather than None. - proof: CLASS_NAMES contains "PVCs" and the RHYTHM_CLASS_NAMES filter does not exclude it so RHYTHM_INDEX maps "PVCs" to an index, and LABEL_MAP maps "PVC" -> "PVCs", hence the function will return that index.
normalize_label	data_loader.py	Function	186-216	1 - The function does not guarantee the returned label is a member of CLASS_NAMES: the composite path builds strings like "Atrial Fibrillation + PVCs" that do not match the entries in CLASS_NAMES, so downstream code that expects labels to be exactly in CLASS_NAMES (e.g. index lookups) can fail. - proof: normalize_label("AF+PVC") splits and normalizes parts to "Atrial Fibrillation" and "PVCs" (via LABEL_MAP), returning "Atrial Fibrillation + PVCs" which is not present in CLASS_NAMES.2 - Inconsistent canonicalization for PVC: LABEL_MAP maps "PVC" -> "PVCs" (plural) but composite class names in CLASS_NAMES use "PVC" (singular). This mismatch causes composite outputs to differ from the canonical CLASS_NAMES entries and breaks label synergy/indexing. - proof: LABEL_MAP contains '"PVC": "PVCs"' while CLASS_NAMES contains entries like "Atrial Fibrillation + PVC" (singular), so normalize_label will produce "PVCs" where CLASS_NAMES expects "PVC".3 - The function uses "Sinus Rhythm" as the silent default for None or any unknown/uncaught label which will silently reassign unknown or invalid labels to a normal class instead of flagging them. - proof: the code returns "Sinus Rhythm" immediately when label is None and also returns "Sinus Rhythm" as the final fallback for any unmatched input.
__init__	data_loader.py	Function	229-243	1 - The __init__ method only checks Path.exists() but does not verify Path.is_dir(); it can accept a file path (or non-directory) and proceed, causing unexpected behavior when listing JSONs and breaking downstream data loading robustness. - Proof: the code only calls self.data_dir.exists(), so an existing file path passes that check, but self.data_dir.glob("*.json") will yield no files (or fail for non-dir), causing len(self.files)==0 and a RuntimeError("No JSON files found..."), demonstrating the issue.
_resample_and_fixlen	data_loader.py	Function	305-333	1 - The function does not guard against orig_fs == 0 (or negative), so the division len(sig) * TARGET_FS / orig_fs can raise a ZeroDivisionError; because the fallback interpolation recomputes the same division outside the try/except, that ZeroDivisionError will propagate and crash the loader. - proof: If orig_fs == 0, int(orig_fs) == 0 so the initial resample attempt raises ZeroDivisionError (caught) but the except block recomputes int(len(sig) * TARGET_FS / orig_fs) unguarded, causing an unhandled ZeroDivisionError.2 - The interpolation fallback path is not itself wrapped in error handling. If the initial resample attempt fails, the fallback can still raise exceptions (e.g., due to invalid orig_fs or other edge cases) and there is no catch, so the method can crash instead of returning a safe padded/truncated signal. - proof: The except block directly calls np.linspace and np.interp without any try/except, so any exception raised there (ZeroDivisionError, ValueError, etc.) will propagate out of the method.3 - The code does not validate the computed new_len (int(len(sig) * TARGET_FS / orig_fs)) to ensure it is >= 1 before calling resample or using it as the num in linspace; passing new_len <= 0 to resample/linspace is invalid and leads to exceptions. - proof: new_len can evaluate to 0 (e.g., very large orig_fs relative to len(sig)) and the code calls resample(sig, new_len) or uses that value in np.linspace without checking, which will raise errors.
__getitem__	data_loader.py	Function	340-382	1 - The function uses CLASS_INDEX (y = CLASS_INDEX.get(label_norm, 0)) but CLASS_INDEX is not defined in the module/globals, causing a NameError and breaking label-to-index mapping. - Proof: The file globals list contains CLASS_NAMES, RHYTHM_INDEX, ECTOPY_INDEX but no CLASS_INDEX, so referencing CLASS_INDEX will raise a NameError at runtime.
__init__	data_loader.py	Function	390-402	1 - Hardcoded DB credentials in __init__ (dbname/user/password/host/port) create security and deployment inflexibility - proof: __init__ assigns conn_params = {"dbname": "ecg_analysis", "user": "ecg_user", "password": "sais", "host": "127.0.0.1", "port": "5432"} directly in the constructor.2 - __init__ performs heavy I/O by calling _load_all_data(limit), causing blocking side-effects and slow instantiation - proof: __init__ calls self._load_all_data(limit) and _load_all_data opens a DB connection, executes a query and fetches/processing rows during construction.3 - limit is forwarded unsanitized and interpolated into the SQL, enabling SQL injection

Function Name	File	Type	Lines	Issue
				/ malformed-query risk - proof: _load_all_data uses query += f" LIMIT {limit}" and then cur.execute(query), directly inserting limit into the SQL string.
_load_all_data	data_loader.py	Function	407-465	1 - Uses undefined global variable CLASS_INDEX (will raise NameError - CLASS_INDEX is not present in the provided globals) - Proof: the module globals shown do not define CLASS_INDEX while the function directly references CLASS_INDEX, which will raise a NameError at runtime.2 - Hardcodes TARGET_LEN = 2500 instead of using the existing global SEG_LEN / TARGET_FS constants - Proof: the file defines SEG_LEN = TARGET_FS * 10 (2500) but the function sets TARGET_LEN = 2500 literal rather than using SEG_LEN or TARGET_FS.3 - Builds the SQL LIMIT by string concatenation instead of using parameterized queries, creating a potential SQL injection / safety issue - Proof: the code appends f" LIMIT {limit}" directly into the query string without parameterization or validation, allowing an attacker-controlled limit value to inject SQL.
print_report	evaluate_simple.py	Function	68-139	1 - The function accepts a 'cm' (confusion matrix) parameter but never uses or prints it, so the confusion matrix passed to print_report is ignored. - Proof: 'cm' appears only in the function signature and is not referenced anywhere in the function body.
evaluate	evaluate_simple.py	Function	142-242	1 - When saving the evaluation report the code sets sys.stdout = f and only restores it after print_report; if print_report (or any code between) raises, sys.stdout is not restored (no finally block) - The code assigns original_stdout = sys.stdout; sys.stdout = f; then calls print_report and only afterwards restores sys.stdout, with no try/finally, so an exception during print_report will skip the restore and leave global stdout redirected to the file.
forward	calibration.py	Function	25-27	1 - The forward method returns scaled logits (logits / T) but does not apply softmax, contradicting the class docstring which states it applies to model probabilities (effectively softmax(logits / T)). - proof: temperature_scale simply returns logits / temperature and neither temperature_scale nor forward call softmax, so the output are logits not probabilities.
temperature_scale	calibration.py	Function	29-35	1 - The function does not enforce temperature > 0 (no positivity constraint); as an unconstrained nn.Parameter it can become zero or negative during optimization, causing division-by-zero or invalid calibration behavior. - Proof: temperature is declared as an unconstrained nn.Parameter(torch.ones(1)*1.5) and is used directly in logits / temperature with no clamp/abs/softplus, so an optimizer can set it to 0 or negative producing division-by-zero or sign flips. 2 - The implementation assumes logits are 2D (batch_size, num_classes) and expands temperature to (batch_size, num_classes); it will fail or behave incorrectly for logits with additional spatial dimensions (e.g., (N, C, H, W)) or other shapes. - Proof: code expands temperature to (logits.size(0), logits.size(1)) producing shape (N,C), which cannot broadcast against a logits tensor of shape (N,C,H,W) (mismatched trailing dims) and will raise a shape/broadcasting error.
set_temperature	calibration.py	Function	37-81	1 - Unconditionally calling self.cuda() instead of moving the module to the provided device (e.g., self.to(device)), creating possible device mismatches with criteria/inputs moved to device - proof: the code calls self.cuda() while nll_criterion/ece_criterion are .to(device) and inputs are input.to(device), so if device != default CUDA device a mismatch can occur. 2 - The function never sets the model to evaluation mode before collecting validation logits, leaving dropout/BatchNorm active and corrupting calibration - proof: logits are collected inside with torch.no_grad() by calling self.model(input) but there is no self.eval() or self.model.eval() call. 3 - The temperature parameter is optimized unconstrained as a raw nn.Parameter and can become non-positive (no reparametrization like optimizing log T) - proof: temperature is defined as nn.Parameter and optimizer = torch.optim.LBFGS([self.temperature]) is used with no constraints or exponentiation applied.