Drummiez AI â   Friendly Step-by-Step Guide

Last updated: $(date +%Y-%m-%d)

This PDF is meant to feel like a patient tour guide through the Drummiez AI repository. Imagi
 I am walking next to you pointing at each file and line, explaining in plain words what it does, why it exists, and how the whole drum-reading machine works. Read it front-to-back once and you should be able to answer any "what is this?" question about the project.

---

1. Big Picture Story (Say It Like We Are Kids)

- **What is Drummiez?** It is a Python backend that takes a drum sheet (photo, PDF, or alread
clean MusicXML) and turns it into two things: (1) a neat list of drum hits with timing, and (
 a WAV file you can play.
- **How does it do that?** Think of a factory line:
  1. **Upload Bay** â   FastAPI receives a file.
  2. **Understanding Booth** â   either a deep-learning detector looks at the image or the OE
tool converts PDFs/images to MusicXML.
  3. **Music Brain** â   music21 reads MusicXML or detector notes, figures out drums and timi
  4. **Sound Forge** â   midi2audio + FluidSynth use a soundfont to turn the notes into real audio.
- **Who are the helpers?** Torch + torchvision supply the Faster R-CNN detector, music21/midi2audio/FluidSynth handle music conversion, and FastAPI exposes everything through
endpoints.

---

2. Architecture Overview

‘‘‘
User upload (PNG/JPG/BMP/TIFF/PDF/MusicXML)
      â
      â â â ” FastAPI ‘/parse_drumsheet/‘
      â   â â â ” Detector path (‘DrumOMRInference‘ + ‘detections_to_notes‘)
      â   â â â ” OEMER path (PDF/image â   MusicXML â   music21 parsing)
      â
      â â â ” FastAPI ‘/generate_drum_audio/‘
             â â â ” music21 stream â   MIDI â   FluidSynth â   WAV stream
‘‘‘

Environment variables (‘SOUNDFONT_PATH‘, ‘MODEL_WEIGHTS_PATH‘, ‘DRUM_LABEL_MAP_PATH‘, ‘MODEL_CONFIDENCE‘, ‘SKIP_MODEL_LOAD‘) configure which options are available at runtime.

---

3. File-by-File Map

Path	What it stores / does	Plain explanation

‘README.md‘	Marketing + quickstart doc	Gives outsiders the elevator pitch, endpoints, e vars, and roadmap.
‘main.py‘	FastAPI app + MIDI/audio helpers	Entry point; defines endpoints, loads models renders audio, glues everything.
‘model_inference.py‘	Torch-based detector wrapper + heuristics	Knows how to load Faster R-CNN weights and convert detections to playable notes.
‘prepare_dataset.py‘	JSON â CSV data prep script	Filters DeepScores-style annotations into a training CSV.
‘train_model.py‘	Training loop for detector	Builds dataset, trains Faster R-CNN, saves ‘drum_omr_model.pth‘.
‘run_parse_and_render.sh‘	Convenience shell helper	Activates venv, parses an image, renders WAV locally.
‘tests/‘	Pytest suite	Ensures detector heuristics and FastAPI endpoints behave.
‘requirements.txt‘, ‘Pipfile‘, ‘Pipfile.lock‘	Dependency manifests	Pin every library (FastAPI, music21, torch, etc.). Pipfile mirrors requirements for Pipenv users.
‘drum_omr_model.pth‘	Trained detector weights	Binary state dict loaded by ‘DrumOMRInference‘.
‘parsed_notes.json‘, ‘peaceful_take.wav‘, ‘Peaceful-Easy-Feeling-...png‘	Sample outputs/assets	Example run results and demo sheet music.
‘data/prepared_data.csv‘	Example prepared dataset	Output of ‘prepare_dataset.py‘; used ‘train_model.py‘.
‘process_explanation.txt‘, ‘understanding.txt‘, ‘todo6nov.txt‘, ‘review_todo.txt‘	Interna docs and checklists	Explain planning history, detailed architecture, and improvement to-dos
‘run_parse_and_render.sh‘	Script to test pipeline locally	Shows how to jump straight fr sheet image to parsed JSON + WAV.
‘tests/conftest.py‘	Adds repo root to ‘sys.path‘	Ensures pytest can import modules when run locally.
‘tests/test_model_inference.py‘, ‘tests/test_parse_endpoint.py‘	Test cases	Document expected behavior for detector heuristics and API flows.
‘requirements.txt‘	Python dependencies	Includes FastAPI, PyTorch, midi2audio, music21, etc.
Misc files (‘__pycache__‘, ‘Pipfile.lock‘)	Build artifacts / dependency locks	Not hand-edited; created by Python tooling.

(Yes, some entries appear twice intentionally to keep this map self-contained.)

---

4. Deep Dive Into Each Important File

4.1 ‘main.py‘ â The API Brain

Imports and global setup
- Top comments remind us which dataset inspired the project (DeepScores). Then Python imports roll in: FastAPI pieces, typing helpers, os/path utilities, subprocess for FluidSynth, tempfile/shutil for safe scratch handling, and uuid4 for output filenames.
- ‘music21‘ modules (‘converter‘, ‘instrument‘, ‘note‘, ‘stream‘, ‘tempo‘, ‘chord‘) plus Pillowâ s ‘Image‘ cover the music parsing and image IO needs.
- ‘model_inference‘ imports the detector class and utilities. An optional import block tries ‘import oemer‘; if it fails, ‘OEMER_RUNNER‘ stays ‘None‘, which later decides whether PDF

parsing is possible.

FastAPI app + logging
- ‘app = FastAPI()‘ instantiates the web app, and ‘LOGGER‘ grabs a namespaced logger (‘drummiez‘) for informative logs.

Drum MIDI dictionary (‘DRUM_MIDI_MAP‘)
- This big dictionary is the translation sheet from drum names (text) to MIDI note numbers. Example: "acoustic snare" â   38, "closed hi-hat" â   42. Itâ s referenced whenever we need figure out which MIDI pitch a note should become.

‘get_midi_pitch(n)‘ helper
- Input: a ‘music21‘ note/rest object.
- Steps:
  1. Try to read ‘instrumentName‘ either from the noteâ s instrument or its part. If it matc a key in ‘DRUM_MIDI_MAP‘, return that MIDI number.
  2. If the note is unpitched, inspect ‘displayStep‘ (letters like C/D/E) as a fallback mapping.
  3. If still lost, look at the notehead style. An "x" notehead usually means hi-hat.
  4. As a final safety, default to MIDI 35 (acoustic bass drum). This ensures the pipeline never crashes just because metadata was missing.

Configurable constants
- ‘SOUNDFONT_PATH‘, ‘MODEL_WEIGHTS_PATH‘, ‘MODEL_CONFIDENCE‘, ‘DRUM_LABEL_MAP_PATH‘, ‘SKIP_MODEL_LOAD‘ all read from environment variables with sensible defaults (FluidR3_GM soundfont, ‘drum_omr_model.pth‘, threshold 0.5, etc.).
- ‘SUPPORTED_IMAGE_EXT‘ and ‘OEMER_SUPPORTED_EXT‘ define which file extensions the detector a OEMER can handle.
- ‘VALID_ENGINES = {"auto", "detector", "oemer"}‘ restricts the ‘engine‘ query parameter to known values.
- ‘PERCUSSION_KEYWORDS‘ is a tuple of substrings ("drum", "snare", etc.) used later to guess a MusicXML part is percussive.

Loading optional label map + detector
- ‘LABEL_TO_MIDI‘ starts empty. If ‘DRUM_LABEL_MAP_PATH‘ exists, ‘load_label_mapping‘ turns JSON like ‘{ "1": 42 }‘ into ‘{1: 42}‘ and logs the success. Failures are caught and logged without crashing.
- ‘INFERENCE_RUNNER‘ is either:
  - ‘None‘ if ‘SKIP_MODEL_LOAD=1‘ or weights are missing.
  - A ready ‘DrumOMRInference‘ instance if weights load properly (the class handles device selection and evaluation mode). Exceptions become warnings so the API still responds for MusicXML/OEMER uploads.

Endpoint: ‘GET /‘
- ‘read_root()‘ simply returns a JSON welcome message. Handy for health checks.

Endpoint: ‘POST /parse_drumsheet/‘
Signature: ‘parse_drumsheet(file: UploadFile, bpm: Optional[int]=100, engine: str="auto")‘

1. **File saving** â   The uploaded file is read asynchronously and dumped into a temporary f (preserving extension) so downstream libraries can open it like a normal file.

2. **Engine validation** â   Extension is lower-cased, ‘engine‘ is normalized, and we compute flags: ‘is_musicxml‘, ‘is_supported_image‘, ‘can_use_detector‘, ‘can_use_oemer‘.
3. **Engine enforcement** â   If the client explicitly asks for ‘detector‘ but the detector cannot run, respond with HTTP 503. Same for ‘oemer‘ when the OEMER module or extension is missing.
4. **MusicXML shortcut** â   When the upload already is ‘.xml‘ / ‘.musicxml‘, decode the byte string right away; ‘music21‘ parsing happens later.
5. **Detector path (images)** â   If we can run the detector and ‘engine‘ is ‘auto‘ or ‘detector‘, ‘_parse_image_with_model‘ is called. That function uses ‘INFERENCE_RUNNER.predict_path‘ and ‘detections_to_notes‘ to build ‘parsed_notes‘. On success the endpoint returns immediately with ‘source: "detector"‘.
6. **Auto fallback** â   If the detector path raises an ‘HTTPException‘ while in ‘auto‘ mode OEMER is available, the code logs the failure and falls back to ‘_run_oemer‘.
7. **OEMER path** â   For PDFs or when forced, ‘_run_oemer‘ launches ‘oemer.run‘ against the file, gathers the generated ‘.musicxml‘, and returns its text. ‘source_label‘ becomes ‘"oemer"‘ so the response explains where notes came from.
8. **Unsupported case** â   If none of the above succeeded, the function raises HTTP 501 tell the user to upload a known format.
9. **MusicXML parsing** â   ‘converter.parse‘ from music21 reads the MusicXML string and yiel ‘score‘. We iterate through ‘score.parts‘, skip non-percussion parts via ‘_part_is_percussion‘, and walk every note/rest:
   - Rests become ‘midi_pitch=0‘ entries (duration + offset copied over).
   - Chords are expanded into individual notes.
   - ‘get_midi_pitch‘ assigns MIDI numbers to actual drum hits.
10. **Response assembly** â   Build a JSON dict with filename, bpm, status, notes, and option source label. The ‘finally‘ block deletes the temporary file no matter what happened.

Endpoint: ‘POST /generate_drum_audio/‘
Signature: ‘generate_drum_audio(background_tasks, parsed_notes: dict, bpm: Optional[int]=100)

1. **Soundfont check** â   If ‘SOUNDFONT_PATH‘ does not point to a real file, abort with HTTP explaining how to set it.
2. **music21 stream creation** â   A ‘stream.Stream‘ is created, a tempo mark is added, and a percussion ‘Part‘ is inserted.
3. **Rebuilding notes** â   Iterate over ‘parsed_notes["parsed_notes"]‘. For each entry, crea ‘note.Rest‘ when ‘midi_pitch == 0‘, else a ‘note.Note‘ with its ‘midi‘. Duration and offset are set to match the JSON.
4. **MIDI export** â   Write the stream to a temporary ‘.mid‘ file.
5. **WAV rendering** â   Create a placeholder ‘.wav‘ file, delete it immediately, then call ‘_render_with_fluidsynth(midi_file_path, wav_file_path)‘ which shells out to the ‘fluidsynth‘ binary.
6. **Streaming response** â   Wrap the WAV bytes in a generator ‘audio_stream‘ that yields chunks. Feed it to ‘StreamingResponse‘ with ‘audio/wav‘ media type and a random filename.
7. **Cleanup** â   ‘BackgroundTasks‘ is used to delete the WAV file once FastAPI finishes streaming it. The MIDI file is deleted immediately in the ‘finally‘ block.

Helper: ‘_is_supported_image(extension)‘
Returns ‘True‘ when the extension lives inside ‘SUPPORTED_IMAGE_EXT‘ (PNG/JPG/JPEG/BMP/TIFF). Tiny guard but keeps logic readable.

Helper: ‘_can_process_with_oemer(extension)‘

Checks two things at once: the OEMER module actually imported, and the file extension is eith
a supported image or PDF.

Helper: ‘_run_oemer(source_path)‘
1. Ensure OEMER exists; if not, raise HTTP 503.
2. Create a temporary directory (‘tempfile.mkdtemp‘).
3. Call ‘OEMER_RUNNER(source_path, output_path=output_dir)‘.
4. Collect generated ‘.musicxml‘ files, error if none exist.
5. Open the first MusicXML file as UTF-8 text and return it.
6. Always delete the temporary dir via ‘shutil.rmtree‘.

Helper: ‘_parse_image_with_model(image_path)‘
1. Confirm ‘INFERENCE_RUNNER‘ is available; otherwise 503 with instructions.
2. Run ‘predict_path‘ to get detections. If empty, raise HTTP 422.
3. Open the image with Pillow to read its height.
4. Call ‘detections_to_notes(detections, img.height, label_to_midi=LABEL_TO_MIDI or None)‘.
5. Return the parsed notes list.

Helper: ‘_part_is_percussion(part)‘
- Tries ‘part.getInstrument()‘; if it returns ‘instrument.Percussion‘, weâ re done.
- Otherwise gather candidate names from ‘instrumentName‘, ‘partName‘, ‘fullName‘, and ‘id‘, a
check if any of them contain keywords like "snare" or "tom".
- Returns ‘True‘ if the part looks percussive. This keeps OEMER outputs useful even when metadata is incomplete.

Helper: ‘_render_with_fluidsynth(midi_path, wav_path)‘
- Locates the ‘fluidsynth‘ CLI via ‘shutil.which‘.
- Builds a command array ‘[’fluidsynth’, ’-ni’, ’-F’, wav_path, ’-r’, ’44100’, SOUNDFONT_PATH midi_path]‘.
- Runs it with ‘subprocess.run(check=True)‘ to capture errors cleanly.
- Wraps failure into HTTP 500 with stderr messages. This custom runner avoids argument-order quirks inside the ‘midi2audio‘ helper.

‘if __name__ == "__main__"‘
Running ‘python main.py‘ will launch ‘uvicorn‘ on ‘0.0.0.0:8000‘, so the script doubles as bo
module and executable.

4.2 ‘model_inference.py‘ â   Detector Utilities

Module docstring
- Immediately states the purpose: wrap the trained Faster R-CNN and turn detections into drum notes.

Imports and logging
- ‘dataclasses‘, ‘statistics.median‘, typing hints, and ‘import_module‘ are used for type safety and lazy torch loading. PILâ s ‘Image‘ is needed for reading input images.

‘Detection‘ dataclass
- Holds ‘bbox‘, ‘score‘, and ‘label‘. Using a dataclass keeps code tidy and self-documenting.

‘DrumOMRInference‘ class
- ‘__init__(weights_path, detection_threshold=0.5, device=None)‘:
  - Validates the path.
  - Calls ‘_load_torch()‘ which lazily imports PyTorch. If torch isnâ t installed, it raises ‘RuntimeError‘ early.
  - Chooses device (‘cuda‘ if available, else CPU).
  - Builds the Faster R-CNN architecture via ‘_build_model‘. Notice ‘weights=None‘ so the backbone isnâ t preloaded; we expect to load our own state dict.
  - Loads weights from disk, moves model to device, switches to ‘eval()‘.
- ‘_build_model(num_classes=2)‘ recreates the training-time architecture: ResNet-50 FPN backbone + new ‘FastRCNNPredictor‘ with 2 classes (background + drum glyph).
- ‘predict_image(image)‘:
  1. Converts PIL image to tensor via ‘torchvision.transforms.functional.to_tensor‘.
  2. Runs the model in ‘no_grad‘ mode.
  3. Extracts ‘boxes‘, ‘scores‘, ‘labels‘. Handles ‘None‘ cases gracefully.
  4. Applies the detection threshold and returns a list of ‘Detection‘ objects.
- ‘predict_path(path)‘ simply opens the file, converts to RGB, and forwards to ‘predict_image

‘detections_to_notes(...)‘
Parameters: detections iterable, ‘image_height‘, optional ‘duration‘, optional ‘label_to_midi mapping.

1. Sort detections by left-most x coordinate to determine play order.
2. If no detections, return empty list.
3. Validate ‘image_height‘ (avoid divide-by-zero by forcing ‘>=1‘).
4. Estimate drum staff bounds via ‘_estimate_staff_bounds‘ â   uses 5th and 95th percentiles centers to ignore stray marks.
5. Compute x-center spacing median to understand beat separation (‘_estimate_spacing‘).
6. For each detection:
   - Use label-based MIDI mapping when provided (‘label_to_midi[det.label]‘).
   - Otherwise map vertical position to hi-hat/snare/kick via ‘_midi_from_relative_position‘ ‘_midi_from_vertical_position‘.
   - Estimate note duration based on distance to the next detection, quantized to sixteenth notes (‘_quantize‘).
   - Track ‘current_offset‘ so every note knows when it should play.
7. Build dictionaries with ‘midi_pitch‘, ‘duration‘, ‘offset‘, ‘confidence‘, and ‘label‘.

‘load_label_mapping(json_path)‘
- Opens JSON, expects a dict.
- Keys are coerced to ints; values can be direct ints or nested dicts containing ‘"midi"‘.
- Returns ‘{label_id: midi}‘; raises ‘ValueError‘ for malformed entries. This is how ‘main.py can override heuristics with precise instrument mappings.

Helper functions
- ‘_midi_from_vertical_position(y_center_norm)‘ â   simple threshold mapping (<0.33 hi-hat, < snare, else kick).
- ‘_estimate_staff_bounds(detections, image_height)‘ â   percentile-based top/bottom to avoid outliers.
- ‘_midi_from_relative_position(y_center, staff_bounds)‘ â   normalizes absolute y coordinate into ‘[0,1]‘ and feeds ‘_midi_from_vertical_position‘.

- ‘_percentile(sorted_values, pct)‘ â   returns percentile even for short lists.
- ‘_estimate_spacing(x_centers)‘ â   median spacing between neighbors (minimum 1.0 pixel) to guess beat length.
- ‘_quantize(value, step)‘ â   snaps to nearest multiple of ‘step‘.
- ‘_load_torch()‘ â   wraps ‘import_module("torch")‘ so import errors surface as friendly run exceptions.

4.3 ‘prepare_dataset.py‘ â   Filtering Raw Annotations

1. Imports: ‘json‘ and ‘csv‘, because we read DeepScores JSON and emit a CSV.
2. ‘prepare_dataset(json_path, output_csv_path)‘:
   - Opens the JSON, expects ‘images‘, ‘annotations‘, ‘categories‘ top-level keys.
   - Defines ‘drum_categories‘, a whitelist of percussion-friendly annotation classes (variou noteheads, rests, dynamics, articulations, beams, ties, etc.).
   - For every image, iterate through its ‘ann_ids‘, fetch the annotation, then through ‘ann['cat_id']‘ to see all categories assigned to that annotation.
   - If the category name is in ‘drum_categories‘ and the bounding box is valid (width/height 0), append a dict with filename, bbox, and category.
   - Finally, write the list to CSV with headers ‘filename‘, ‘bbox‘, ‘category‘.
   - Returns the prepared data list so other scripts/tests can reuse it.
3. CLI entry point (‘if __name__ == '__main__'‘): calls the function on ‘data/ds2_dense/deepscores_train.json‘ and prints how many rows landed in ‘data/prepared_data.csv‘.

Why it matters: This script is how you curate the dataset that ‘train_model.py‘ expects. Without it the detector would have nothing to learn from.

4.4 ‘train_model.py‘ â   Training the Detector

Imports
- Torch + torchvision pieces, pandas for CSV reading, os/PIL for file IO.

‘DrumSheetDataset‘
- ‘__init__(csv_file, root_dir, transform=None)‘ stores annotations, image folder, and option transform.
- ‘__len__‘ returns number of rows.
- ‘__getitem__(idx)‘:
  1. Builds the absolute image path, opens it as RGB.
  2. Parses the bbox string from CSV, turning ‘"[x1, y1, x2, y2]"‘ into floats.
  3. Wraps the bbox into tensors shaped exactly how PyTorch detection models expect (‘boxes‘: ‘[N,4]‘).
  4. Uses placeholder labels (‘torch.ones‘) because training currently assumes binary classification.
  5. Applies transforms like ‘ToTensor‘ if provided.
  6. Returns ‘(image, target)‘ pair.

‘get_model(num_classes)‘

- Loads ‘fasterrcnn_resnet50_fpn(pretrained=True)‘.
- Replaces the ROI head with a ‘FastRCNNPredictor‘ sized to ‘num_classes‘. This is the same architecture the inference helper rebuilds.

‘main()‘ training routine
1. Define transforms (currently only ‘ToTensor‘).
2. Instantiate ‘DrumSheetDataset‘ pointing at ‘data/prepared_data.csv‘ / ‘data/ds2_dense/images‘.
3. Split into 80% train, 20% validation. Then take up to 100 samples from validation for quic evaluation.
4. Create ‘DataLoader‘s with ‘batch_size=2‘ and custom collate function ‘lambda x: tuple(zip(*x))‘, which is the recommended way for torchvision detection models.
5. Set ‘num_classes=2‘, instantiate the model, move it to GPU if available.
6. Define SGD optimizer (lr=0.005, momentum=0.9, weight decay=5e-4).
7. Training loop (currently ‘num_epochs=1‘):
   - ‘model.train()‘.
   - For each batch, move tensors to device, call ‘model(images, targets)‘ which returns a di of losses, sum them, backprop, and step the optimizer.
   - Print ‘Epoch: {epoch}, Loss: {loss}‘ for quick feedback.
8. Validation snippet:
   - ‘model.eval()‘ and disable gradients.
   - For each batch in validation loader, run predictions, then compute IoU between each predicted box and every ground-truth box via ‘calculate_iou‘. Keep the best IoU per predictio accumulate totals, and finally compute an average IoU.
9. After training: ‘torch.save(model.state_dict(), ’drum_omr_model.pth’)‘ so ‘main.py‘ can us the weights.

‘calculate_iou(boxA, boxB)‘
- Standard intersection-over-union math: compute overlap rectangle, area of each box, union area, and return ‘interArea / union‘. Adds ‘+1‘ padding to mimic pixel-inclusive coordinates.

CLI guard
- ‘if __name__ == ’__main__’: main()‘ lets you run ‘python train_model.py‘ to kick off training.

4.5 ‘tests/‘ â   Ensuring Behavior

‘tests/conftest.py‘
- Adds the repository root to ‘sys.path‘ so imports like ‘import main‘ work even when pytest changes directories.

‘tests/test_model_inference.py‘
- ‘test_label_mapping_overrides_vertical_mapping()‘ â   ensures ‘detections_to_notes‘ respect explicit label-to-MIDI maps.
- ‘test_vertical_mapping_used_when_label_missing()‘ â   ensures the hi-hat/snare/kick heurist fires in order.
- ‘test_staff_bounds_survive_large_image_height()‘ â   checks percentile logic still works on tall images.
- ‘test_horizontal_spacing_influences_duration_and_offset()‘ â   verifies the timing math rea

to horizontal spacing and quantization.

‘tests/test_parse_endpoint.py‘
- Pre-sets ‘SKIP_MODEL_LOAD=1‘ so torch doesnâ t initialize during testing.
- ‘_fake_png_bytes()‘ / ‘_fake_pdf_bytes()‘ generate in-memory upload payloads.
- ‘test_parse_endpoint_uses_detector()‘ monkeypatches ‘INFERENCE_RUNNER‘ and ‘detections_to_notes‘ to confirm the endpoint returns detector results when given a PNG.
- ‘test_parse_endpoint_uses_oemer_for_pdf()‘ fakes an OEMER run + ‘music21‘ parse to ensure P uploads take the OEMER path.
- ‘test_parse_endpoint_auto_falls_back_to_oemer()‘ ensures ‘engine=auto‘ tries OEMER when the detector raises ‘HTTPException‘.

These tests double as executable documentation â   by reading them you see exactly how the A is expected to behave.

4.6 Support Scripts and Assets

- **‘run_parse_and_render.sh‘** â   sample end-to-end script. Activates ‘.venv‘, sets ‘IMAGE_PATH‘/‘BPM‘, runs an inline Python block that:
  1. Checks the image and detector exist.
  2. Calls ‘main._parse_image_with_model‘ to produce ‘parsed_notes.json‘.
  3. Builds a music21 stream from those notes and saves a MIDI file.
  4. Calls ‘main._render_with_fluidsynth‘ to render ‘peaceful_take.wav‘.
- **‘parsed_notes.json‘ / ‘peaceful_take.wav‘ / ‘Peaceful-Easy-Feeling-....png‘** â   the out that script produced for the Eagles drum sheet example.
- **‘process_explanation.txt‘** â   narrative of how the author set up the project, libraries chosen, and next steps. Good for onboarding.
- **‘understanding.txt‘** â   very detailed internal architecture write-up. It mirrors much o this PDF but from the developerâ s perspective.
- **‘todo6nov.txt‘** â   prioritized to-do list covering OMR improvements, dataset prep, deployment, and frontend plans.
- **‘review_todo.txt‘** â   code review notes calling out current shortcomings (dataset splitting, label usage, fallback handling, tests).
- **‘requirements.txt‘** â   pins versions for FastAPI (0.120.1), PyTorch (2.6.0), torchvisio (0.21.0), music21, midi2audio, pytest, and many supporting libraries like numpy, Pillow, http
- **‘Pipfile‘ / ‘Pipfile.lock‘** â   allow Pipenv users to replicate the exact environment.
- **Binary model/data files** â   ‘drum_omr_model.pth‘ (weights) and ‘data/prepared_data.csv‘ (sample training CSV). You donâ t edit these manually; training scripts regenerate them.

---

5. Walking Through the Runtime Flow

1. **Startup** â   When FastAPI starts (via ‘uvicorn main:app‘ or ‘python main.py‘), environm variables decide whether the detector loads. If ‘SKIP_MODEL_LOAD=1‘ or weights missing, only the MusicXML/OEMER path works.
2. **User uploads file â   ‘/parse_drumsheet/‘**
   - File saved temporarily.
   - If itâ s MusicXML, skip to music21 parsing.

   - If itâ s an image: try detector first (unless ‘engine=oemer‘). Detector success returns JSON immediately.
   - If detector fails and ‘engine=auto‘, try OEMER; otherwise bubble up error.
   - OEMER or direct MusicXML path parse the XML into percussion notes using ‘get_midi_pitch‘ and ‘_part_is_percussion‘ heuristics.
   - JSON response includes ‘parsed_notes‘, ‘source‘, ‘bpm‘.
3. **Client optionally POSTs JSON â   ‘/generate_drum_audio/‘**
   - Confirms soundfont file exists.
   - Rebuilds a music21 stream from the JSON.
   - Writes MIDI, shells out to FluidSynth, streams WAV back in chunks.
4. **Cleanup** â   Temporary files deleted via context managers and ‘BackgroundTasks‘.

---

6. Why the Project Is Good or Bad & How to Improve

Area	Why itâ s good / bad	Possible improvements
Detector integration	**Good:** Modular ‘DrumOMRInference‘ lazily loads torch, supports custom label maps, and plugs into FastAPI seamlessly. **Bad:** Training code still treats everything as one class, so multi-instrument detection is limited.	1) Update ‘DrumSheetDataset‘ + training loop to keep real category labels. 2) Train multi-class model s ‘label_to_midi‘ mappings shine.
MusicXML parsing	**Good:** ‘get_midi_pitch‘ + ‘_part_is_percussion‘ handle messy OEMER outputs and fallback defaults mean no crashes. **Bad:** Mapping is heuristic; hi-hats vs ride vs ghost notes all become the same few MIDI pitches.	1) Expand ‘DRUM_MIDI_MAP‘ and heuristi to read articulations/noteheads. 2) Add unit tests covering more MusicXML fixtures.
Audio rendering	**Good:** Uses proven FluidSynth CLI, streams audio to avoid huge memory usage. **Bad:** Fails hard if ‘SOUNDFONT_PATH‘ missing; no caching or streaming progress feedback.	1) Provide default bundled soundfont or friendlier instructions. 2) Consider caching repeated renders of the same note sequence.
File uploads	**Good:** Temp files + extension checks prevent memory blowups. **Bad:** No explicit size limits or virus scanning; OEMER dependency errors surface only at runtime.	1) Enforce max upload size via FastAPI ‘UploadFile‘. 2) Surface OEMER install instructions in ‘/ endpoint or README when missing.
Dataset prep + training	**Good:** Scripts are short and documented, enabling users to retrain. **Bad:** Each CSV row only has one bbox, so many annotations per image are ignored; train/val split can leak identical pages.	1) Re-architect dataset so each sample returns al boxes for the image. 2) Split train/val by image, not by row.
Testing	**Good:** Pytest suite covers detector heuristics and API fallbacks, using monkeypatch to avoid heavy dependencies. **Bad:** No tests yet for ‘prepare_dataset‘, trainin helpers, or ‘/generate_drum_audio‘.	1) Add fixture-driven tests covering dataset filtering audio rendering. 2) Integrate tests into CI so regressions are caught automatically.
Documentation	**Good:** README, process_explanation, understanding docs, and this PDF mak onboarding approachable. **Bad:** Info is scattered across multiple files, and some environme steps (FluidSynth install) could trip people up.	1) Consolidate docs into a MkDocs or Sphi site. 2) Add troubleshooting FAQ for OEMER, torch, FluidSynth issues.
Deployment story	**Good:** Pure Python stack works on CPU, so it fits cheap servers. **Bad:** No Dockerfile, no CI/CD, no frontend yet.	1) Containerize app (install FluidSynth soundfont). 2) Publish minimal React/CLI client once API stabilizes.

---

7. Recap Checklist (Use This to Verify Understanding)

- â I know every endpoint (‘/‘, ‘/parse_drumsheet/‘, ‘/generate_drum_audio/‘) and what they return.
- â I can describe how images flow through ‘DrumOMRInference‘ â ‘detections_to_notes‘ and MusicXML files get parsed via music21.
- â I understand why ‘get_midi_pitch‘, ‘_part_is_percussion‘, and ‘_render_with_fluidsynth‘ exist.
- â I can run ‘prepare_dataset.py‘ and ‘train_model.py‘ to retrain the detector, and I know where the weights are used.
- â I can explain what each test validates and how to run them (‘SKIP_MODEL_LOAD=1 pytest‘)
- â I know where support docs and sample assets live, and what improvements the TODO files suggest.

If you can tick all of those boxes, you officially understand Drummiez AI end-to-end. Happy drumming! ð ¥