BatLab Developer Manual

1. Purpose of this project

BatLab is a Streamlit application for managing bat acoustic classification workflows. In the `feature/csv-results-export` branch, the app supports five main tasks:

1. Classifying `.wav` bat call files from a user-specified folder.
2. Exporting classification results as a CSV in a client-friendly format.
3. Managing detector records.
4. Managing species records.
5. Uploading training audio into the database and training subset models from database-backed data.

This manual is meant to help future developers understand how the codebase is organized, how data moves through the system, and where to make changes safely.

---

2. High-level architecture

The project is split into three main layers:

UI layer

- **Main entry point:** `app.py`
- Built with **Streamlit**.
- Handles login, tab navigation, user input, result display, CSV download, and subset-model training controls.

Machine learning layer

- Located under `src/ml/`
- Handles:
    - feature extraction
    - spectrogram generation
    - model definition
    - inference
    - full training
    - subset fine-tuning

Database layer

- Located under `src/db/`
- Handles reading and writing detector, species, and training-call information to MySQL.

Supporting assets

- `configs/default.yaml` stores model and audio-processing settings.
- `model_checkpoints/` stores the base model and locally trained subset models.
- `src/ui/` stores CSS for the interface.

---

3. Repository structure

A future developer should know these files first:

- `app.py`
  Main Streamlit app and the best starting point for understanding the user workflow.
- `configs/default.yaml`
  Central configuration for sample rate, mel settings, directories, and thresholds.
- `src/db/connection.py`
  Main database helper file for detectors, species, and `Call_Library` training data.
- `src/ml/classify_app.py`
  Main classification adapter used by the Streamlit app.
- `src/ml/training/inference.py`
  Low-level single-file inference logic.
- `src/ml/models/net.py`
  Neural network architecture (`BatClassifier`).
- `src/ml/scripts/train.py`
  CLI full-training entry point.
- `src/ml/training/subset_model_trainer.py`
  Fine-tunes a subset model from UI selections using database-backed training calls.
- `src/ml/data/`
  Audio preprocessing, feature extraction, manifest logic, and dataset helpers.

---

4. How the application works from the user side

After login, the app shows five tabs:

Tab 1: Classify

The user:

1. Selects a model.
2. Enters a source folder path containing `.wav` files.
3. Runs classification.
4. Reviews identified and unknown results.
5. Downloads a CSV of the classification results.
6. Optionally moves identified and unknown files into new folders.

Tab 2: Add Detector

The user creates and edits detector records.
These are stored in MySQL.

Tab 3: Add Species

The user creates and edits species records.
These are stored in MySQL.

Tab 4: Add Training Data

The user uploads training calls and attaches them to detector/species metadata.
These files are stored in the database so they can later be used for subset-model training.

Tab 5: Train New Model

The user selects detectors and species, then fine-tunes a subset model from the base model.
The trained subset model is saved locally and becomes available in the classification dropdown.

---

5. Main data flow for classification

This is the most important workflow in this branch.

Step 1: UI starts in `app.py`

The Classify tab manages the session state and user interaction.
Key state objects include:

- `known_data`
- `unknown_data`

- `export_data`
- `uploaded_files`
- `source_folder`

The app verifies a folder path, collects `.wav` filenames, and calls a helper that wraps those files into file-like objects.

Step 2: `process_audio_files()` in `app.py`

This helper builds file-like objects from disk and passes them to:

- `classify_uploaded_files()` in `src/ml/classify_app.py`

This function returns three DataFrames:

- known results
- unknown results
- export-ready CSV results

Step 3: `classify_uploaded_files()` in `src/ml/classify_app.py`

This file is the bridge between the Streamlit UI and the lower-level inference code.
It does the following:

- resolves the project root and config paths
- creates temporary cache directories for per-run processing
- writes uploaded file bytes to temp files
- calls `predict_file()` for each file
- converts raw prediction output into UI tables and CSV rows

This file is especially important because it defines the export schema used by the UI.

Step 4: `predict_file()` in `src/ml/training/inference.py`

This is the low-level inference call for one `.wav` file.
It loads the model and metadata, loads thresholds, prepares the audio/features, runs the network, and returns a prediction object.

Step 5: UI display + CSV export

Back in `app.py`, the returned DataFrames are displayed to the user.
The `export_data` DataFrame is converted to CSV bytes and made downloadable using `st.download_button()`.

---

6. CSV results export in this branch

This branch is centered around the CSV export feature, so future developers should understand it clearly.

Where the export schema lives

The export columns are defined in `src/ml/classify_app.py`.
The current export schema is:

- `IN FILE`
- `DATE`
- `TIME`
- `AUTO ID*`
- `MATCHING`
- `Fc`
- `Dur`
- `Fmax`
- `Fmin`
- `SC`

Where the export table lives in UI state

In `app.py`, the export results are stored in:

- `st.session_state.export_data`

Where the CSV file is created

Still in `app.py`, `convert_df_to_csv()` converts the DataFrame into UTF-8 CSV bytes for download.

Where to change export behavior

If a future developer needs to:

- add a new column
- rename a column
- compute new acoustic measurements
- change how unknown rows are represented

then they will likely need to edit both:

1. `src/ml/classify_app.py`
2. `app.py`

A good rule is:

- put measurement generation and row construction in `classify_app.py`
- keep download/display behavior in `app.py`

Safe extension strategy

If you add columns, update all of these together:

1. `EXPORT_COLUMNS` in `classify_app.py`
2. default `export_data` initialization in `app.py`
3. any fallback row builders for unknown or failed predictions
4. any documentation or user-facing CSV labels

If you only update one of these, the UI may still run but the export table can become inconsistent.

---

7. Model architecture

The neural network is defined in `src/ml/models/net.py` as `BatClassifier`.

It is a multi-input model that combines three information sources:

A. Mel spectrogram branch

A CNN processes the spectrogram input.
The convolution stack uses repeated Conv2d, BatchNorm, ReLU, and pooling blocks, then ends with `AdaptiveAvgPool2d((4, 4))`.

B. Location embedding branch

A learned embedding represents detector/location information.
This allows the model to use location as a signal during classification.

C. Numeric feature branch

A small fully connected subnetwork processes numeric audio features.

Final classifier

The outputs from the CNN, location embedding, and numeric features are concatenated and passed through fully connected layers to produce the final class logits.

Why this matters to future developers

Any change to input features, metadata, or the saved checkpoint format may affect:

- training
- inference
- subset fine-tuning
- backward compatibility with saved checkpoints

When changing the model, test both:

1. full training from CLI
2. UI classification with old and new checkpoints if backward compatibility matters

---

8. Database layer

The database logic lives mainly in `src/db/connection.py`.

What it stores

The DB layer supports at least three major concepts:

- detector/location data
- species data
- training calls in `Call_Library`

Important functions

Future developers will likely touch these functions:

- `load_detectors_from_db()`
  Loads detector records for the UI.
- `save_detectors()` / `delete_detectors()`
  Used when editing detector entries.
- `load_species_from_db()` / related save-delete helpers
  Used when editing species entries.

- `save_training_data()`
  Stores uploaded training `.wav` data in `Call_Library`.
- `load_training_records_df()`
  Returns training-record summaries for UI display.
- `get_call_library_data()`
  Returns database training calls as a DataFrame with columns like `file`, `bat`, and `location` for subset training.

DB environment variables

The project expects MySQL connection settings from environment variables:

- `MYSQL_HOST`
- `MYSQL_PORT`
- `MYSQL_USER`
- `MYSQL_PASSWORD`
- `MYSQL_DATABASE`

Development advice

Do not bypass `connection.py` unless there is a very good reason.
This file is already acting as the contract between the app and the database. Centralizing queries here makes maintenance much easier.

---

9. Full training workflow

The CLI full-training path starts in:

- `src/ml/scripts/train.py`

This script is used for training from a data root on disk rather than from database-selected subsets.

At a high level it:

1. parses CLI arguments
2. loads config
3. builds or loads a manifest
4. creates train/validation/test splits
5. trains the model
6. calibrates the model using temperature scaling

7. writes the checkpoint and metadata to the model directory

This is the best place to look when a developer wants to understand the original, non-UI training pipeline.

---

10. Subset model training workflow

Subset fine-tuning is handled in:

- `src/ml/training/subset_model_trainer.py`

This is one of the most important advanced features in the app.

What it does

It takes detector/species selections from the UI, filters the database-backed training calls, loads the base model, swaps the classifier head to match the subset class count, freezes earlier layers, and trains a new subset model.

Why it exists

The idea is to start from a stronger base model and create smaller, specialized models tied to selected detectors and species.

Where subset models are stored

Subset models are saved under:

- `model_checkpoints/local/<subset_name>/`

Why future developers should be careful here

This file touches all of the following at once:

- metadata shape
- label mapping
- checkpoint saving
- transfer learning behavior
- UI integration

A small change can break the training tab, the model discovery dropdown, or inference for subset models.

Practical warning

If you change saved metadata keys, location mappings, or class-index behavior, re-test classification on both the base model and newly trained subset models.

---

11. Session state and UI behavior

Because the app is built in Streamlit, a lot of behavior depends on `st.session_state`.

Why that matters

Streamlit reruns the script often. Session state is what keeps the app from losing everything after each interaction.

Common state variables

Important state variables include:

- `known_data`
- `unknown_data`
- `export_data`
- `uploaded_files`
- `training_file_bytes`
- `detectors`
- `species`
- `logged_in`
- `source_folder`
- form-visibility flags for moving files

Developer advice

If you add a new workflow in the UI, define its state keys near the session initialization section at the top of `app.py`.
Do not scatter first-use state initialization deep inside tabs if you can avoid it.
That makes the app harder to debug.

---

12. File movement behavior after classification

The UI includes options to move identified and unknown files into new folders.
This is handled directly in `app.py` using file-system operations.

Why this matters

This means classification does not only read files. It can also move them.
That is convenient for users, but it creates side effects.

Developer caution

Before changing this logic, think about:

- whether files should be copied or moved
- whether folder collisions matter
- what should happen if a destination already contains a file with the same name
- whether rollback behavior is needed after a partial failure

Right now, this is a user-convenience feature, but it is also a place where accidental data movement bugs could happen.

---

13. Config and checkpoints

Config

`configs/default.yaml` controls the audio and model-processing defaults.
Typical settings include:

- sample rate
- mel spectrogram parameters
- cache directories
- threshold file locations
- model directories

Checkpoints

The project uses two main checkpoint areas:

Base model

- `model_checkpoints/colab/`

This contains the base model used by default for inference and as the starting point for subset fine-tuning.

Local subset models

- `model_checkpoints/local/`

This contains one subfolder per subset model trained from the UI.

Developer advice

Treat checkpoint metadata as part of the API.
If you change checkpoint structure, make sure all checkpoint readers are updated too.

---

14. Best places to make common changes

If you want to change the UI layout

Start in `app.py`.

If you want to change classification result columns

Edit `src/ml/classify_app.py` and then sync the matching session-state/DataFrame setup in `app.py`.

If you want to change the neural network

Edit `src/ml/models/net.py`, then re-test training and inference.

If you want to change inference logic

Edit `src/ml/training/inference.py`.

If you want to change training from raw folders

Edit `src/ml/scripts/train.py` and related helpers under `src/ml/training/` and `src/ml/data/`.

If you want to change subset training from the UI

Edit `src/ml/training/subset_model_trainer.py`.

If you want to change detector/species/training-call persistence

Edit `src/db/connection.py`.

If you want to change CSS or appearance

Edit files in `src/ui/`.

---

15. Known design limitations and technical debt

Future developers should know that some parts of the app are practical but not fully production-hardened.

Hardcoded login

The login uses a very simple username/password check inside `app.py`.
This is fine for a school project or demo, but not for real deployment.

Large `app.py`

`app.py` contains a lot of UI logic in one file.
This makes it easy to follow at first but harder to scale.
A future refactor could split each tab into separate UI modules.

File-system path entry in UI

The classification workflow asks the user to type a source folder path.
This works locally, but it is less portable for hosted deployments.

Tight coupling between export schema and UI state

The CSV format depends on multiple matching definitions across files.
It works, but future developers should keep them synchronized.

Database contract sensitivity

Subset training depends on `get_call_library_data()` returning the right shape.
If DB schema or aliases change, training may fail.

Checkpoint compatibility risks

Any model or metadata change can break older checkpoints.
There is no dedicated migration layer yet.

---

16. Recommended future refactors

If a new team inherits this project, these are the highest-value improvements:

1. Split `app.py` into smaller modules by tab or feature.
2. Replace the simple login with environment-backed or real authentication.
3. Centralize the export schema into a shared constant used by both UI and classifier code.
4. Add automated tests for:
    - CSV row generation
    - detector/species CRUD helpers
    - subset model discovery
    - checkpoint loading
5. Add a clear model metadata schema version field.
6. Improve file-move safety with collision handling and better error reporting.
7. Add developer scripts for environment setup and DB bootstrapping.

---

17. Suggested onboarding path for a new developer

A new developer should read files in this order:

1. `README.md`
2. `app.py`
3. `src/ml/classify_app.py`
4. `src/ml/training/inference.py`
5. `src/ml/models/net.py`
6. `src/db/connection.py`
7. `src/ml/training/subset_model_trainer.py`
8. `src/ml/scripts/train.py`

That order usually gives the clearest picture of how the project works end to end.

---

18. Final practical advice for future maintainers

When you make changes in BatLab, always think in terms of the full chain:

UI input → session state → DB or ML helper → model/data output → UI tables/export

Most bugs in this codebase will happen because one part of that chain changed and another part did not.

If you are changing anything related to classification output, model metadata, or training-call structure, test all of these afterward:

- app startup
- classification on a small folder of `.wav` files
- CSV download
- detector/species editing
- training data upload
- subset model training
- new model appearing in the dropdown

That one checklist will catch a lot of integration bugs before they become a bigger problem.