**WhisperJAV Project - Complete Development Guide for Version 1.2**

**For Future Code Assistants:** This document contains everything needed to understand the current state of the WhisperJAV project and continue development. The project has recently undergone significant architectural enhancements.

**🎯 PROJECT OVERVIEW**

WhisperJAV is a production-ready, fully functional Japanese subtitle generation system. The core system has been hardened, and a flexible configuration system has been implemented to support advanced tuning and dynamic processing. The current development phase is focused on implementing the logic for optional enhancement modules.

**Project Status: Version 1.2**

* ✅ **Core system hardened:** The application's stability and usability have been improved.
  + The temporary directory logic now correctly uses the appropriate OS-specific temp folder (e.g., /tmp or AppData/Local/Temp) instead of a local ./temp folder. This was implemented in main.py using Python's tempfile library.
  + Temporary file cleanup is now fully implemented. The cleanup\_temp\_files method in the base pipeline now correctly removes intermediate files (extracted audio, scene chunks, etc.) after processing, unless the --keep-temp flag is used.
* ✅ **New Configuration System Implemented:** A powerful, hybrid configuration system has been implemented to allow for advanced tuning.
* ✅ **Pipeline Contracts Refactored:** The pipeline constructors have been refactored to accept their parameters from the new configuration system, making them more modular and maintainable.
* 🏗️ **Optional enhancement modules need implementation:** The stubs for segment\_classification.py, audio\_preprocessing.py, and srt\_postproduction.py are in place, and the system is ready for their logic to be built out.

**⚙️ NEW CONFIGURATION PHILOSOPHY (Hybrid Approach)**

The project has moved from a simple CLI-only approach to a flexible hybrid configuration model. This was done to support advanced, granular tuning without creating unwieldy command-line calls.

The configuration is loaded with the following order of precedence:

1. **Code Defaults:** The base parameters hardcoded in the application.
2. **config.json File:** A user can provide a JSON configuration file using the --config flag. Values in this file override the code defaults.
3. **Command-Line Arguments:** Any parameter set as a command-line flag (e.g., --vad-threshold) will always have the highest priority, overriding both the config file and the defaults.

**SYSTEM ARCHITECTURE: CONFIGURATION**

The architecture is now built around a **Module-Centric** configuration structure, defined in a user-provided config.json file.

* **components block:** This is a "library" of reusable parameter blocks for each module (e.g., scene\_detection, asr\_stable\_ts, post\_processing). Parameters are structured hierarchically (e.g., separating model\_load\_params from transcribe\_method\_params) for maintainability.
* **pipelines block:** This block acts as a "blueprint." It defines which components from the library each pipeline mode (faster, fast, balanced) should use.
* **actionable\_profiles block:** This enables dynamic, per-scene parameter changes. When a scene is classified (e.g., as "music"), the pipeline can look up the "music" profile and apply a different set of component settings for that scene only.

The entire loading and merging logic is handled in main.py. The final, consolidated parameters are passed to the pipeline constructors, which then configure the "worker" modules they manage.

**🚀 IMMEDIATE NEXT STEPS**

The previous development phases (UI enhancement, hardening, config system implementation) are now **complete**. The project is now ready to begin implementing the logic for the optional enhancement modules.

The immediate priority is to build out the **segment\_classification.py** module.

**Development Plan:**

1. **Implement SegmentClassifier:** Flesh out the SegmentClassifier class in segment\_classification.py. It should take a scene's audio file as input and return a classification label (e.g., "music", "dialogue", "noise").
2. **Integrate into Pipelines:** In fast\_pipeline.py and balanced\_pipeline.py, call the classifier after scene detection if the --adaptive-classification flag is enabled.
3. **Implement Actionable Logic:** Use the classification label to look up a profile in the actionable\_profiles section of the loaded configuration.
4. **Apply Dynamic Parameters:** If a profile is found, the pipeline should dynamically use the component settings from that profile for processing that specific scene. For example, if a scene is "music," it might apply the settings from the "audio\_enhancement\_vocal" component.

**📚 KEY FILES FOR UNDERSTANDING**

* config.template.json: **(New and Critical)** Defines the entire structure of the new configuration system. Understanding this file is essential.
* whisperjav/main.py: **(Heavily Modified)** Contains all the core logic for argument parsing, configuration loading/merging, and pipeline orchestration.
* whisperjav/pipelines/base\_pipeline.py: **(Modified)** Contains the now-implemented cleanup\_temp\_files logic.
* whisperjav/pipelines/fast\_pipeline.py & balanced\_pipeline.py: **(Modified)** Their \_\_init\_\_ methods show how they are constructed from the new configuration dictionaries.
* whisperjav/modules/segment\_classification.py: The next file to be implemented.

Sources