**Wake Word Detection Engine Design Document**

**1. System Overview**

This document details a fully offline wake word detection engine for Windows using a lightweight CNN architecture. The system operates entirely on the local machine without requiring internet connectivity, making it suitable for privacy-conscious applications and reliable operation in all network conditions.

**1.1 Core Components**

1. **Audio Capture Module** - Continuously samples audio from system microphones
2. **Preprocessing Pipeline** - Converts raw audio into suitable features for ML inference
3. **CNN Model** - Performs wake word detection on processed audio features
4. **Trigger System** - Manages detection events and triggers configurable actions
5. **Training Toolkit** - Allows users to create custom wake words on consumer hardware

**2. Technical Architecture**

**2.1 Audio Capture Module**

Input Device → PyAudio Stream → Circular Buffer → Processing Queue

**Specifications:**

* Sample rate: 16 kHz (16-bit, mono)
* Frame size: 512 samples (32ms chunks)
* Circular buffer: 2 seconds total (to provide context before/after detections)
* Supports multiple audio input devices with device selection interface

**Implementation Details:**

* Use PyAudio's callback mode for efficient non-blocking audio capture
* Implement automatic audio device enumeration with descriptive naming
* Include automatic gain control for consistent input levels
* Add voice activity detection to filter silent periods

**2.2 Preprocessing Pipeline**

Raw Audio → Windowing → MFCC Extraction → Normalization → Feature Frames

**Specifications:**

* Window size: 25ms with 10ms stride (overlapping windows)
* MFCC features: 13 coefficients × 101 frames = 1313 features per sample
* Normalization: Per-feature mean and variance normalization
* Feature caching: Store computed features in memory to reduce redundant calculations

**Implementation Details:**

* Use librosa for MFCC extraction (or implement custom extraction for better performance)
* Implement sliding window feature extraction with 1-second context
* Apply batch processing where possible to leverage CPU vectorization
* Add feature quantization option for reduced memory footprint

**2.3 CNN Model Architecture**

Input → Conv1D → BatchNorm → ReLU → MaxPool → Conv1D → BatchNorm → ReLU → MaxPool → Dense → Dense → Sigmoid

**Specifications:**

* Input shape: [1, 13, 101] (channels, MFCC coefficients, time frames)
* Conv1D layer 1: 64 filters, kernel size 3, stride 1
* MaxPool layer 1: pool size 3, stride 2
* Conv1D layer 2: 64 filters, kernel size 3, stride 1
* MaxPool layer 2: pool size 3, stride 2
* Dense layer 1: 128 units with ReLU activation
* Dense layer 2: 1 unit with sigmoid activation
* Total parameters: ~100K (model size <1MB)

**Implementation Details:**

* Use PyTorch for both training and inference
* Implement model export to ONNX format for optimized inference
* Add confidence threshold parameter (default 0.85, adjustable)
* Include windowed averaging of predictions to reduce false positives

**2.4 Trigger System**

Detections → Confidence Filtering → Debouncing → Action Execution

**Specifications:**

* Minimum confidence threshold: User-adjustable (default 0.85)
* Debounce period: 3 seconds between consecutive triggers
* False positive reduction: Require 3 positive frames within 300ms
* Configurable actions: Run executable, keyboard shortcut, system event

**Implementation Details:**

* Implement thread-safe event queue for processing triggers
* Add user-configurable actions via JSON configuration file
* Include visual/audio feedback for successful detections
* Provide detection logging for troubleshooting

**2.5 Training Toolkit**

Audio Collection → Augmentation → Feature Extraction → Model Training → Validation → Deployment

**Specifications:**

* Minimum samples: 50 wake word examples, 30 minutes background audio
* Augmentation: Pitch shift (±10%), time stretch (±10%), amplitude variation (±6dB), noise addition
* Negative samples: Random snippets from background audio, common phrases
* Training regime: 50 epochs max with early stopping (patience=5)
* Validation: 5-fold cross-validation with precision/recall metrics

**Implementation Details:**

* Create guided recording UI for sample collection
* Implement one-click augmentation to generate 500+ training samples
* Add automatic negative sample generation from background recordings
* Include transfer learning option from pre-trained audio models
* Provide training progress visualization and final performance metrics

**3. Performance Considerations**

**3.1 Resource Usage Targets**

* CPU usage: <5% average on quad-core i5 (8th gen or newer)
* Memory footprint: <150MB total (including audio buffers)
* Disk usage: <20MB for application, <50MB per trained model
* Latency: <300ms from wake word end to action trigger

**3.2 Optimization Techniques**

* Use threading to separate audio capture, processing, and inference
* Implement feature caching to avoid redundant calculations
* Batch process features where possible for vectorized operations
* Utilize SIMD instructions through NumPy/PyTorch
* Apply model quantization for faster inference

**4. User Interface**

**4.1 System Tray Application**

* Minimal UI with system tray icon
* Quick controls: Enable/disable, sensitivity adjustment
* Status indicators: Active, processing, detected
* Settings access for detailed configuration

**4.2 Configuration Interface**

* Wake word model selection/management
* Audio device selection and testing
* Detection sensitivity adjustment with live feedback
* Action configuration (what happens on detection)
* Training interface access

**4.3 Training Interface**

* Step-by-step guided process for recording wake word samples
* Background noise recording tool
* One-click training with progress visualization
* Performance metrics display after training
* Model comparison and selection

**5. Dependencies and Environment**

**5.1 Primary Dependencies**

* Python 3.9+
* PyAudio 0.2.11+
* NumPy 1.20+
* PyTorch 1.9+ (CPU version)
* Librosa 0.8+ (for feature extraction)
* PyInstaller (for packaging)

**5.2 Packaging Strategy**

* PyInstaller single-file executable for main application
* User files (configurations, models) stored in %APPDATA%
* Silent installation option for enterprise deployment
* Automatic update checking (optional, user-configurable)

**6. Development Guidelines**

**6.1 Code Organization**

src/

├── audio/

│ ├── capture.py # Audio device handling and capture

│ ├── features.py # MFCC extraction and processing

│ └── vad.py # Voice activity detection

├── model/

│ ├── architecture.py # CNN model definition

│ ├── inference.py # Real-time inference engine

│ └── training.py # Training pipeline

├── ui/

│ ├── tray.py # System tray application

│ ├── config.py # Configuration interface

│ └── training\_ui.py # Training interface

├── utils/

│ ├── config.py # Configuration handling

│ ├── logger.py # Logging utilities

│ └── actions.py # Trigger action execution

└── main.py # Application entry point

**6.2 Testing Strategy**

* Unit tests for all core components
* Integration tests for end-to-end pipeline
* Performance benchmarks on reference hardware
* False positive/negative testing with standard datasets

**7. Future Extensions**

* Multi-word phrase detection
* Speaker verification option
* Adaptive background noise handling
* GPU acceleration for training on compatible systems
* Multiple wake word support with different actions

**8. Implementation Timeline**

1. Audio capture and feature extraction (2 days)
2. CNN model implementation and inference engine (3 days)
3. Training pipeline and toolkit (4 days)
4. User interface and system integration (3 days)
5. Testing, optimization, and packaging (3 days)

Total estimated development time: 15 working days