A Comprehensive Technical Guide to the LLM-Powered Spoken Digit Classification Challenge

Section 1: Foundational Analysis and Data Ingestion Strategy

1.1 Deconstructing the Free Spoken Digit Dataset (FSDD)

A thorough understanding of the dataset is the bedrock upon which any successful machine learning project is built. The Free Spoken Digit Dataset (FSDD) is explicitly designed as an audio analogue to the famous MNIST dataset of handwritten digits, a fact that immediately signals its intended use as a clean, well-structured benchmark for classification algorithms.[1] This framing is crucial, as it informs the entire project strategy, steering focus away from arduous data cleaning and towards efficient modeling and workflow optimization.

The dataset's composition is straightforward and well-documented. It contains a total of 3,000 audio recordings of spoken digits from 0 to 9 in English.[1] These recordings are contributed by six distinct speakers, with each speaker providing 50 utterances for every digit, resulting in a balanced dataset of 300 samples per class.[4] This balance is advantageous as it prevents the model from developing a bias towards more frequently represented classes and simplifies the interpretation of accuracy metrics.

The audio files are provided in the WAV format, encoded as single-channel (mono) with a sampling rate of 8kHz.[2] While 8kHz is on the lower end of audio fidelity, it is well within the sufficient range for capturing the essential frequencies of human speech, making it a pragmatic choice that keeps file sizes small and processing computationally inexpensive. A key feature that greatly simplifies the preprocessing pipeline is that the recordings have been pre-trimmed to minimize silence at the beginning and end of each utterance.[2] This obviates the need for a custom voice

activity detection (VAD) or energy-based trimming step, which can be a common and time-consuming prerequisite in many audio-based projects.

Furthermore, the dataset employs a logical and descriptive file naming convention: {digitLabel}_{speakerName}_{index}.wav, with an example being 7_jackson_32.wav.[2] This structured naming allows for the trivial extraction of labels and metadata directly from the filenames, streamlining the creation of a data loading pipeline without the need for external annotation files. This deliberate simplicity in the dataset's design—its balanced classes, pre-trimmed files, and self-descriptive naming—is not an oversight but a strategic choice. It ensures that developers can rapidly move past data ingestion and focus on the core aspects of the challenge: selecting an appropriate model, justifying architectural choices, optimizing for performance, and demonstrating a modern, LLM-assisted development process. The data itself is a controlled variable; the developer's methodology is the experiment.

1.2 Data Loading and Splitting Strategy

With a clear understanding of the dataset's structure, the next step is to devise an efficient strategy for data ingestion and splitting. Three primary methodologies present themselves, each with distinct trade-offs between control, convenience, and speed of implementation.

1. **Hugging Face datasets Library:** This is the most highly recommended approach for this challenge due to its alignment with the goals of speed and reproducibility. The FSDD is readily available on the Hugging Face Hub, and their datasets library provides a high-level API, load_dataset, to download, cache, and access the data with a single line of code.[7] This method often provides the data in a pre-processed, tabular format (e.g., Parquet) with clearly defined splits, abstracting away the complexities of file parsing and directory traversal.[7] For a time-constrained challenge, this is the most direct path to obtaining model-ready data.
2. **Specialized PyTorch Libraries:** For developers working within the PyTorch ecosystem, libraries like torch-fsdd offer a compelling alternative. This package provides a dedicated torch.utils.data.Dataset wrapper for the FSDD, designed for seamless integration with PyTorch's DataLoader.[11] It handles the downloading and parsing of the dataset, presenting a clean, framework-native interface that is both convenient and efficient.

3. **Manual Loading and Custom Dataset Class:** The most fundamental approach involves cloning the original FSDD GitHub repository and implementing a custom data loading class.[2] This would typically involve using libraries like os and glob to traverse the file system, parsing the filenames to extract labels, and loading the audio waveforms using a library such as scipy.io.wavfile or librosa.[6] While this method offers the highest degree of control over the data pipeline, it is also the most time-intensive and error-prone, making it less suitable for a rapid prototyping challenge.

Regardless of the loading method chosen, adherence to the official train-test split is paramount for ensuring that the model's performance is comparable to established benchmarks. The FSDD repository explicitly defines this split based on the index number in the filename.[2] Recordings with an index from 0 to 4 (inclusive) are designated for the test set, constituting 10% of the data for each speaker-digit pair. The remaining recordings, with indices from 5 to 49, form the training set (90%).[13] Any data loading implementation must programmatically enforce this split to maintain the integrity of the evaluation process. This ensures that the model is tested on a consistent, unseen subset of the data, providing a fair and reproducible measure of its generalization capabilities.

Section 2: The Audio Preprocessing Pipeline: From Raw Waveform to Model-Ready Features

2.1 Standardization of Audio Signals

Before an audio signal can be processed by a neural network, it must be transformed into a standardized, numerical format. This involves two critical steps: standardizing the length and normalizing the amplitude. Neural networks, particularly Convolutional Neural Networks (CNNs), require inputs of a fixed size to maintain a consistent architecture. However, the audio recordings in the FSDD, despite being short, have variable lengths.[15] To resolve this, a uniform input length must be established. Analysis of the dataset suggests that a common length of 8192 samples is an appropriate choice. At a sampling rate of 8kHz, this corresponds to 1.024 seconds, a duration

sufficient to encompass any of the spoken digits without truncation.[14] Recordings shorter than this target length are symmetrically padded with zeros (silence), while longer recordings are truncated. This ensures that every input to the model has a consistent shape, a non-negotiable prerequisite for batch processing during training.

The second standardization step is amplitude normalization. The raw amplitude of an audio signal can vary significantly based on recording volume and the speaker's proximity to the microphone. To prevent the model from misinterpreting these volume differences as meaningful features, the amplitude of each audio clip must be scaled to a consistent range.[17] A standard and effective technique is to normalize each recording by its maximum absolute value. This scales the entire waveform to a floating-point range of [-1, 1], ensuring that all audio samples are treated on an equal footing with respect to their amplitude, thereby improving training stability and model performance.[14]

2.2 Feature Extraction: A Critical Comparison

With the raw audio standardized, the next crucial decision is how to represent the audio data as features for the model. While the raw 1D waveform can be used directly, it is often more effective to transform it into a 2D time-frequency representation that makes the pertinent characteristics of the sound more explicit. The choice of feature representation is a foundational modeling decision that directly impacts the complexity and performance of the subsequent classifier.

- **Mel Spectrograms:** This is the recommended feature representation for this challenge, particularly when using a CNN. A Mel spectrogram is a 2D plot of time versus frequency, but with the frequency axis scaled according to the Mel scale, which is designed to mimic the non-linear perception of pitch in the human auditory system.[19] This transformation makes the representation more perceptually relevant for speech. Crucially, a Mel spectrogram can be treated as a single-channel image, making it an ideal input for a 2D CNN, which excels at learning hierarchical spatial patterns—in this case, patterns in the time-frequency domain.[15]
- **Mel-Frequency Cepstral Coefficients (MFCCs):** MFCCs are a highly compressed representation derived from the Mel spectrogram. The final step in calculating MFCCs involves applying a Discrete Cosine Transform (DCT) to the logarithm of the Mel spectrogram energies.[19] This DCT step serves to decorrelate

the features, which was historically advantageous for classic machine learning models like Gaussian Mixture Models (GMMs) and Support Vector Machines (SVMs) that perform better with less correlated inputs.[24] However, this compression and decorrelation process can discard valuable information that a powerful feature learner like a CNN could otherwise exploit.[27]

- **Raw Waveforms:** Using the 1D time-series data directly is a valid approach, often paired with a 1D CNN. This methodology tasks the network with learning all relevant features from the ground up, from basic frequencies to more complex phonetic structures.[21] While this end-to-end approach can be powerful, it typically requires a more complex and deeper model architecture to achieve the same level of performance as a 2D CNN operating on a spectrogram, which contradicts the "lightweight" constraint of the challenge.

The decision to use Mel spectrograms is not merely a matter of preference but a principled architectural choice. The fundamental strength of a CNN lies in its ability to learn local patterns from spatially correlated data. A spectrogram preserves the rich, correlated structure of audio in the time-frequency domain. The DCT step in MFCC generation actively works to reduce these correlations. Therefore, feeding MFCCs into a CNN is somewhat counter-intuitive, as it involves pre-processing the data to remove the very structure that the CNN is designed to leverage. By choosing Mel spectrograms, we are empowering the model to perform the feature learning it excels at, rather than relying on a traditional signal processing step that may discard useful information. This aligns with modern deep learning philosophy and provides the model with the richest possible input for the task.

The following table provides a clear, evidence-based justification for this selection, directly addressing the "Modeling Choices" evaluation criterion.

Feature	Information Density	Computational Cost	Suitability for Lightweight CNN	Rationale & Citations
Raw Waveform	Very High (unprocessed)	Low (to generate)	Low-to-Medium	Requires deeper 1D CNNs to learn hierarchical features, increasing model complexity and

				training time.[21]
Mel Spectrogram	High	Medium	**High**	Provides a rich 2D representation analogous to an image, ideal for 2D CNNs to learn spatial/spectral patterns effectively.[15]
MFCC	Medium (compressed)	Medium	Medium	The DCT step discards some information. While compact, this can limit a CNN's ability to learn nuanced features compared to a full spectrogram.[24] Historically better for linear models.[25]

Section 3: Architectural Blueprint: Designing a Lightweight and Responsive Classifier

3.1 Primary Architecture: A Lightweight 2D CNN

The core of this challenge is to build the "lightest effective solution." This directive necessitates a careful balance between model performance and computational complexity. The ideal architecture is not the most powerful one available, but rather the one that occupies the "knee of the curve" in the performance-versus-complexity

trade-off. For the task of classifying spoken digits from Mel spectrograms, a lightweight 2D Convolutional Neural Network (CNN) represents this optimal point. Extensive research and practical applications have demonstrated that even shallow CNNs, comprising just two to four convolutional layers, are remarkably effective for a wide range of audio classification tasks, including this one.[15]

The proposed architecture is designed for simplicity, speed, and efficacy. It treats the input Mel spectrogram as a single-channel image and applies a series of convolutional blocks to learn hierarchical features.

Proposed Lightweight CNN Architecture:

1. **Input Layer:** Accepts a single-channel Mel spectrogram of a fixed size (e.g., 64x128, corresponding to the number of Mel bands and time frames).
2. **Convolutional Block 1:**
    - Conv2D (e.g., 16 filters, 3x3 kernel, same padding)
    - ReLU activation function
    - BatchNorm2D for stabilizing and accelerating training
    - MaxPool2D (e.g., 2x2 pool size) for downsampling and feature invariance
3. **Convolutional Block 2:**
    - Conv2D (e.g., 32 filters, 3x3 kernel, same padding)
    - ReLU activation
    - BatchNorm2D
    - MaxPool2D (e.g., 2x2 pool size)
4. **Convolutional Block 3:**
    - Conv2D (e.g., 64 filters, 3x3 kernel, same padding)
    - ReLU activation
    - BatchNorm2D
    - MaxPool2D (e.g., 2x2 pool size)
5. **Flatten Layer:** Converts the 2D feature maps into a 1D vector.
6. **Dense Block (Classifier Head):**
    - Linear layer (e.g., 128 units)
    - ReLU activation
    - Dropout (e.g., p=0.5) for regularization
    - Linear output layer with 10 units (one for each digit)
    - LogSoftmax activation for producing log-probabilities suitable for the NLLLoss function.

This architecture is strategically chosen. The convolutional layers with small kernels are efficient at learning local patterns in the spectrogram, such as formants and

phonetic transitions. The pooling layers progressively reduce the spatial dimensions, creating a more abstract and robust feature representation while keeping the parameter count low. The final dense layers act as a classifier based on these learned features. This design is small enough to train in minutes on a modern GPU (or even a CPU), and its inference latency will be minimal, directly satisfying the "lightweight" and "responsiveness" criteria of the challenge.[28] This is not merely a "simple" choice; it is a strategically optimal one that demonstrates an understanding of engineering trade-offs, a key indicator of senior-level development capability.

3.2 Analysis of Alternative Models

To fully justify the selection of a lightweight CNN, it is essential to consider and critique viable alternative architectures in the context of the challenge's specific constraints.

- **Recurrent Neural Networks (RNNs/LSTMs):** RNNs, and their more advanced variant, Long Short-Term Memory (LSTM) networks, are designed to model sequential data by maintaining an internal state that evolves over time.[25] They are highly effective for tasks where long-range temporal dependencies are critical, such as continuous speech recognition or language modeling.[32] However, for this task of classifying short, isolated, and pre-trimmed audio clips, their strengths become liabilities. The entire acoustic context of the digit is available at once within the spectrogram. The sequential processing of an RNN, which processes one time-step at a time, is computationally less efficient and can introduce higher latency compared to the highly parallelizable nature of a CNN.[25] Thus, for this problem, the complexity and sequential nature of an RNN are unnecessary and would work against the goals of low latency and rapid development.
- **Support Vector Machines (SVMs):** As a classic machine learning algorithm, an SVM is a strong baseline for many classification tasks.[33] To apply an SVM to this problem, one would typically first extract handcrafted features, with MFCCs being a common and effective choice.[34] While an SVM model itself can be very lightweight and fast at inference time, its primary limitation is that it does not perform feature learning. Its performance is entirely dependent on the quality of the pre-extracted features. A well-trained CNN, by contrast, can learn features directly from the spectrogram that are optimally discriminative for the specific task, and is therefore likely to outperform an SVM-based approach on this type of perceptual data.[36]

- **Scattering Transforms:** The 1D scattering transform is a powerful, wavelet-based feature extraction technique that produces representations that are stable to deformations and invariant to time shifts.[14] When paired with a simple linear classifier, it can achieve excellent results on audio classification tasks.[14] However, the conceptual overhead and implementation complexity of scattering transforms are significantly higher than those of a standard CNN. Given the strict time constraint of "a couple of hours," delving into an advanced signal processing framework like this would be an inefficient use of development time, especially when a simpler, more common architecture like a CNN is known to be highly effective for the task.

In summary, while these alternative models have their merits in other contexts, the lightweight 2D CNN provides the best-fit solution for the specific intersection of requirements defined by this challenge: high effectiveness, low computational footprint, minimal latency, and rapid implementation.

Section 4: The LLM-Powered Development Workflow

A central and explicit component of this challenge is the evaluation of how a developer collaborates with a Large Language Model (LLM). The most effective use of an LLM in a development context is not as a black-box code generator, but as an interactive "pair programmer" and an on-demand "technical consultant." This approach transforms the LLM from a simple tool into a strategic partner that can accelerate development, deepen understanding, and improve the quality of the final product. A developer's ability to craft precise, context-aware prompts is a direct reflection of their own expertise and problem-solving methodology.

The development process can be broken down into distinct phases, with the LLM playing a tailored role in each.

4.1 Phase 1: Scaffolding and Boilerplate Generation

In the initial phase, the LLM's primary function is to eliminate the tedium of writing boilerplate code, allowing the developer to focus immediately on the core logic. By

providing a high-level description of the project, a significant portion of the initial file structure and class definitions can be generated automatically.

- **Example Prompt:** *"I am building an audio classification system in PyTorch for the Free Spoken Digit Dataset. The goal is to classify spoken digits from 0 to 9. Please generate a Python script that includes the following components:
    1. A torch.utils.data.Dataset class that can load .wav files from a directory structure where subdirectories are named by class.
    2. A simple CNN model defined as a torch.nn.Module subclass. The model should contain three 2D convolutional layers, each followed by ReLU activation, batch normalization, and max pooling.
    3. A basic training loop that iterates over a DataLoader, performs forward and backward passes, and updates model weights.
    4. Include placeholders for data preprocessing, loss function definition, and optimizer instantiation."*

This type of prompt delegates the creation of the project's skeleton, saving valuable time that would otherwise be spent on repetitive coding tasks. The generated code serves as a solid foundation that can then be refined and customized.

4.2 Phase 2: Implementation, Parameter Tuning, and Debugging

Once the project structure is in place, the LLM transitions into the role of a domain-specific expert and a debugging assistant. It can provide recommendations for library usage, explain the rationale behind certain parameters, and help diagnose errors.

- **Example Prompt (Parameter Tuning):** *"I am using the librosa library to generate Mel spectrograms from audio with a sampling rate of 8000 Hz. For a speech recognition task, what are appropriate values for the parameters n_fft, hop_length, and n_mels? Provide the librosa.feature.melspectrogram function call with these parameters and briefly explain why these values are suitable for speech."* [37]
- **Example Prompt (Debugging):** *"I am encountering a RuntimeError: mat1 and mat2 shapes cannot be multiplied in my PyTorch model's forward pass. Here is the code for my model's __init__ and forward methods: [...code...]. The input tensor shape reaching the first linear layer is ``. Explain what is causing this shape mismatch and modify the code to correctly flatten the tensor before the linear*

layer."

These prompts leverage the LLM's vast knowledge base to solve specific, contextual problems. This is far more effective than generic web searches, as the LLM can provide tailored solutions and explanations, accelerating the development cycle and reducing time spent on troubleshooting.

4.3 Phase 3: Conceptual Deepening and Architectural Refinement

Perhaps the most sophisticated use of an LLM is as a tool for deepening one's own conceptual understanding. By asking "why" questions, a developer can validate their architectural choices and ensure they are not just implementing patterns by rote, but are making informed decisions. This demonstrates intellectual curiosity and a commitment to engineering excellence, directly addressing the "Creative energy" evaluation criterion.

- **Example Prompt (Conceptual Understanding):** *"In the context of feeding a Mel spectrogram into a 2D CNN, explain the role and benefit of using BatchNorm2D after each convolutional layer. How does it operate on the time and frequency dimensions of the spectrogram, and what impact does this have on training stability and the model's ability to generalize?"*
- **Example Prompt (Code Refactoring):** *"Here is my current training loop: [...code...]. It works, but it's monolithic. Refactor this code into two separate functions: train_one_epoch() and validate_one_epoch(). The training function should handle the training steps and return the average training loss, while the validation function should run in torch.no_grad() mode and return the average validation loss and accuracy. This will make the main training loop cleaner and more modular."*

This level of interaction shows a developer who is not just a consumer of code, but an active, thoughtful engineer. The recorded LLM sessions for this challenge should ideally reflect this progression from broad scaffolding to specific implementation and finally to deep conceptual inquiry. This dialogue-based approach provides the strongest possible evidence for the "LLM collaboration" criterion, showcasing a partnership where the developer's strategic direction guides the LLM's tactical execution.

Section 5: Performance Evaluation and Model Fortification

5.1 Defining a Robust Evaluation Protocol

A model is only as good as the metrics used to evaluate it. To provide a comprehensive assessment of the classifier's performance, it is necessary to go beyond a single accuracy score and employ a suite of evaluation tools that offer a more nuanced view. This rigorous approach is essential for understanding the model's strengths and weaknesses and for making informed decisions about potential improvements.

- **Accuracy:** As the primary metric for a balanced classification task, accuracy provides a straightforward measure of the model's overall correctness. It is calculated as the proportion of correctly classified samples in the test set and serves as the main indicator of performance.[9]
- **Confusion Matrix:** This is an indispensable tool for diagnosing class-specific errors. A confusion matrix is a table that visualizes the performance of the classifier, showing the number of correct and incorrect predictions for each class.[40] By examining the off-diagonal elements, one can identify which digits are being confused with one another (e.g., a high value at the intersection of the '1' row and '7' column would indicate that the model frequently misclassifies '1' as '7'). This level of detail is crucial for targeted model improvement.[14]
- **Inference Time (Latency):** The challenge explicitly requires a "responsive" solution with "minimal delay." Therefore, measuring the average inference time is a direct and critical evaluation of this requirement. This metric should be calculated as the average time, in milliseconds, required for the model to process a single audio sample on a CPU, as this most closely simulates a real-world deployment scenario on a standard device.
- **Statistical Significance:** When comparing the performance of different model configurations (e.g., a baseline model versus one trained with data augmentation), it is important to determine if an observed improvement is genuine or simply a result of random statistical variation. A paired t-test, performed on the accuracy scores obtained from multiple runs or

cross-validation folds, can be used to calculate a p-value. A p-value below a chosen significance level (e.g., 0.05) provides statistical evidence that the improvement is significant and not due to chance.[41] While a full cross-validation procedure may be beyond the scope of a two-hour challenge, acknowledging the importance of statistical rigor demonstrates a mature approach to model evaluation.

5.2 Data Augmentation for Robustness

To satisfy the "Creative energy" criterion and demonstrate a desire to push beyond the baseline requirements, incorporating data augmentation is a highly effective strategy. Data augmentation artificially expands the training dataset by creating modified versions of the existing audio samples. This exposes the model to a wider variety of conditions, which acts as a form of regularization, helping to prevent overfitting and improve the model's ability to generalize to unseen data, such as recordings with background noise.

- **Simulating Microphone Noise:** One of the most common real-world challenges for an audio classifier is background noise. This can be simulated by adding Gaussian white noise to the raw audio waveforms before they are converted into spectrograms. A small amount of randomly scaled noise can make the model significantly more robust without distorting the underlying speech signal.[44] This is a simple yet powerful technique to improve real-world performance.
- **SpecAugment:** A more advanced and highly effective augmentation technique is SpecAugment, which operates directly on the 2D spectrograms.[46] This method consists of three components: time warping (which is often omitted in practice), and, most importantly, the random masking of frequency channels and time steps. By randomly hiding horizontal or vertical bands of the spectrogram during training, SpecAugment forces the model to learn more robust and distributed feature representations. It cannot rely on any single, specific time-frequency feature, as it might be masked out. Implementing SpecAugment is a clear demonstration of familiarity with state-of-the-art techniques in audio deep learning and a commitment to building a truly robust model.[47]

The results of these experiments should be systematically captured and presented in the project's README.md file. A clear, data-driven table provides the most effective

way to communicate the impact of these fortification strategies.

5.3 Model Performance Benchmark Template

Model Configuration	Test Accuracy (%)	Avg. Inference Time (ms)	Key Observation
Baseline CNN	96.5	5.2	Strong baseline performance on the clean test data, establishing a solid foundation.
+ Gaussian Noise Aug.	97.1	5.3	A modest but statistically significant improvement in accuracy, indicating better regularization and robustness to noise.
+ SpecAugment	98.2	5.3	A significant accuracy gain, demonstrating the model's enhanced ability to generalize by learning from incomplete spectral information.

This table provides a concise and powerful narrative of the model development process, showing a clear progression from a strong baseline to a fortified, more robust final model. It provides concrete evidence of the work performed and makes the results easily digestible for the evaluator.

Section 6: Advanced Implementation: Real-Time Microphone

Integration

Successfully completing the optional challenge of integrating live microphone input elevates the project from a simple batch-processing task to a real-time, interactive application. This demonstrates a broader set of engineering skills, including handling I/O streams, managing application state, and considering user-facing issues like latency. An implementation of this nature is a microcosm of a production MLOps pipeline, showcasing the ability to build and deploy a functional AI system.

6.1 Technical Architecture for Real-Time Processing

The core of a real-time audio processing system is a non-blocking input stream that continuously captures audio from the microphone without halting the main application thread. For this, Python libraries such as sounddevice or PyAudio are excellent choices. The sounddevice library, in particular, offers a clean, callback-based API that is well-suited for this type of application.[48]

PyAudio is another robust and widely used alternative.[50]

The architectural pattern involves setting up an input stream and registering a callback function that is automatically executed whenever a new chunk of audio data is available from the hardware buffer.

Real-Time Processing Loop:

1. **Stream Initialization:** An audio input stream is initiated using sounddevice.InputStream. It must be configured with the same parameters used for training data: a sampling rate of 8000 Hz and a single (mono) channel. A fixed block size (e.g., 1024 samples) is specified, which determines how frequently the callback function is invoked.
2. **Buffering in Callback:** A persistent buffer (e.g., a NumPy array or a Python deque) is maintained outside the callback function. Inside the callback, each new incoming audio chunk is appended to this buffer.
3. **Triggering Inference:** A simple condition checks the size of the buffer. Once the buffer has accumulated enough data to form a complete analysis window (e.g., 8192 samples, matching the standardized length used in training), the inference

process is triggered.

4. **Prediction Pipeline:** The collected audio data in the buffer is passed through the exact same preprocessing pipeline developed for the training data: amplitude normalization, Mel spectrogram generation, and conversion to a PyTorch tensor with the correct dimensions. The trained model then performs inference on this tensor to predict the digit.
5. **Output and Buffer Management:** The predicted digit is displayed to the user in the console. Afterward, the buffer must be managed to prepare for the next prediction. A common strategy is to clear the buffer or to use a sliding window approach, where only the oldest portion of the buffer is discarded, allowing for overlapping analysis windows.

6.2 Handling Real-World Challenges

Moving from clean, pre-recorded files to a live microphone stream introduces new challenges that must be addressed to create a usable application.

- **Latency:** The perceived responsiveness of the application is primarily governed by the end-to-end latency, which is the time from a digit being spoken to the prediction being displayed. This latency is a sum of several components: the time to buffer the audio, the time for preprocessing, and the model's inference time. The buffer size presents a direct trade-off: a smaller buffer (e.g., 0.5 seconds) results in lower latency but provides the model with less acoustic context, potentially reducing accuracy. A larger buffer (e.g., 1.5 seconds) provides more context but increases the delay. The choice of a ~1-second buffer (8192 samples) strikes a reasonable balance for this task. The lightweight nature of the proposed CNN ensures that the inference time component remains minimal.
- **Ambient Noise and Activation Threshold:** In a real environment, a microphone is always capturing sound, including background noise or silence. A naive implementation would continuously run inference on this noise, leading to spurious predictions and wasted computation. A simple and effective solution is to implement an energy-based activation threshold. Before triggering the full inference pipeline, the root mean square (RMS) energy of the audio in the buffer can be calculated. If this energy is below a certain threshold, the audio is considered to be silence or background noise, and the inference step is skipped. The speech_recognition library contains a useful utility, recognizer.adjust_for_ambient_noise, which can be used to automatically

calibrate this threshold by listening to the environment for a brief period upon startup.[52]

Successfully implementing this real-time system demonstrates a significant leap in capability, showcasing an understanding of the architectural patterns required to build responsive, event-driven AI applications that bridge the gap between offline model training and practical deployment.

Section 7: Final Deliverables: Code Architecture and Documentation

The final evaluation of this challenge rests not only on the functional correctness of the code but also on its quality, organization, and documentation. A professional project structure and a comprehensive README.md file are not mere formalities; they are critical components that demonstrate foresight, clarity of thought, and respect for the evaluator's time. They serve as "passive documentation," making the project easy to understand, navigate, and evaluate.

7.1 Recommended Project Structure

Adopting a clean, modular, and extensible code architecture is a hallmark of professional software engineering. The structure should separate distinct concerns of the project—data handling, model definition, training logic, and application code—into different modules. This approach, inspired by MLOps best practices, enhances readability, maintainability, and reusability.[53]

Proposed Project Structure:

```
spoken-digit-classifier/
├── README.md           # The main project documentation and report
```

```
├── requirements.txt      # List of Python dependencies for reproducibility
├── config.yaml           # Centralized configuration for paths, hyperparameters, etc.
├── data/                 # Directory for raw and processed data (should be in.gitignore)
├── notebooks/            # Jupyter notebooks for exploration, visualization, and
prototyping
├── src/                  # Main source code for the model pipeline
│   ├── __init__.py
│   ├── data_loader.py    # Contains the PyTorch Dataset and DataLoader definitions
│   ├── model.py          # Definition of the lightweight CNN architecture
│   ├── preprocess.py     # Functions for audio processing and feature extraction
│   ├── predict.py        # Script for running inference on a single audio file
│   └── train.py          # Main script for training and evaluating the model
├── app/                  # (Bonus) Source code for the real-time application
│   └── realtime_mic.py   # Script for live microphone classification
└── saved_models/         # Directory for storing trained model checkpoints
(in.gitignore)
```

This structure clearly delineates the different logical parts of the project. A config.yaml file is used to store hyperparameters, file paths, and other settings, preventing hard-coded values from cluttering the source code and making experiments easy to configure and reproduce.

7.2 The README.md Template: A Narrative of Excellence

The README.md is the most important document in the repository. It is the project's front page, user manual, and technical report all in one. It must be crafted to be comprehensive, clear, and persuasive, guiding the evaluator through the project's narrative and preemptively answering their questions.[53]

README.md Template Outline:

1. **Project Title:** Spoken Digit Classification with PyTorch & LLM Collaboration
2. **Overview:** A concise, one-paragraph summary of the project. It should state the goal (lightweight spoken digit classification), the core technology (PyTorch, lightweight CNN on Mel spectrograms), and the key outcome (a high-accuracy, low-latency model with a real-time demo).
3. **Key Results:** This section should immediately present the final performance

metrics in a clear, tabular format, as designed in **Section 5.3**. This allows the evaluator to see the project's success at a glance.

4. **LLM Collaboration Showcase:** This is a critical, dedicated section that directly provides evidence for the corresponding evaluation criterion. It should briefly explain the strategy of using the LLM as a pair programmer. It should then feature one or two of the most compelling, anonymized prompt/response examples from the development process that demonstrate scaffolding, debugging, or conceptual deepening. This makes the evidence explicit and easily accessible.
5. **Architecture and Methodology:**
    - **Data Preprocessing:** Detail the steps taken to prepare the audio data: standardization of length (padding/truncation to 8192 samples), amplitude normalization, and feature extraction into Mel spectrograms.
    - **Model Architecture:** Present the lightweight CNN architecture, perhaps with a simple diagram or a torchsummary output. Justify the choice of a CNN over alternatives like RNNs for this specific task.
    - **Training and Fortification:** Describe the training process (optimizer, loss function, number of epochs). Detail the data augmentation techniques (Gaussian Noise, SpecAugment) used to improve model robustness.
6. **Setup and Usage:** Provide clear, step-by-step instructions for a user to replicate the project.
    - **Installation:** git clone..., pip install -r requirements.txt.
    - **Running the Code:** Provide the exact command-line instructions to run the training script (python src/train.py), and to perform inference on a new file (python src/predict.py --file <path_to_wav>).
    - **(Bonus) Real-Time Demo:** Provide the command to launch the live microphone classification demo (python app/realtime_mic.py).
7. **Project Structure:** Briefly explain the layout of the repository, referencing the directory tree to help the evaluator navigate the codebase.

By investing in this level of structure and documentation, the submission transcends being a mere collection of scripts and becomes a polished, professional engineering project. It signals a mature development process and a deep understanding of what it takes to build and communicate a successful machine learning solution.

Works cited

1. Free Spoken Digit Dataset (FSDD), accessed on August 12, 2025, https://datasets.activeloop.ai/docs/ml/datasets/free-spoken-digit-dataset-fsdd/
2. Free Spoken Digit Dataset (FSDD) - Kaggle, accessed on August 12, 2025, https://www.kaggle.com/datasets/joserzapata/free-spoken-digit-dataset-fsdd

3. FSDD Dataset - Papers With Code, accessed on August 12, 2025, https://paperswithcode.com/dataset/fsdd
4. Free Spoken Digit Dataset (FSDD) Digital Recognition Audio Dataset, accessed on August 12, 2025, https://hyper.ai/en/datasets/33094
5. Free Spoken Digits Dataset (FSDD) - Kaggle, accessed on August 12, 2025, https://www.kaggle.com/datasets/jackvial/freespokendigitsdataset
6. 音声・オーディオ版MNIST "FSDD (Free Spoken Digit Dataset)" の紹介と，メルスペクトログラム算出 - Wizard Notes, accessed on August 12, 2025, https://www.wizard-notes.com/entry/asp/fsdd
7. silky1708/Free-Spoken-Digit-Dataset · Datasets at Hugging Face, accessed on August 12, 2025, https://huggingface.co/datasets/silky1708/Free-Spoken-Digit-Dataset
8. Getting Started with Data Loading in Python: A Beginner-Friendly Guide to the Hugging Face load_dataset | by Emanuele | Medium, accessed on August 12, 2025, https://medium.com/@emanueleorecchio/getting-started-with-data-loading-in-python-a-beginner-friendly-guide-to-the-hugging-face-0d8f04a96cbb
9. Audio classification - Hugging Face, accessed on August 12, 2025, https://huggingface.co/docs/transformers/tasks/audio_classification
10. mteb/free-spoken-digit-dataset at main - Hugging Face, accessed on August 12, 2025, https://huggingface.co/datasets/mteb/free-spoken-digit-dataset/tree/main
11. eonu/torch-fsdd: A utility for wrapping the Free Spoken Digit Dataset into PyTorch-ready data set splits. - GitHub, accessed on August 12, 2025, https://github.com/eonu/torch-fsdd
12. pytorch_ood.dataset.audio.fsdd — pytorch-ood documentation - Read the Docs, accessed on August 12, 2025, https://pytorch-ood.readthedocs.io/en/v0.1.8/_modules/pytorch_ood/dataset/audio/fsdd.html
13. Jakobovski/free-spoken-digit-dataset: A free audio dataset ... - GitHub, accessed on August 12, 2025, https://github.com/Jakobovski/free-spoken-digit-dataset
14. Classification of spoken digit recordings — kymatio 0.3.0 ..., accessed on August 12, 2025, https://www.kymat.io/gallery_1d/plot_classif_torch.html
15. Digit Recognition from Sound | sound-mnist, accessed on August 12, 2025, https://adhishthite.github.io/sound-mnist/
16. Spoken Digit Recognition with Custom Log Spectrogram Layer and Deep Learning - MATLAB & Simulink - MathWorks, accessed on August 12, 2025, https://www.mathworks.com/help/signal/ug/spoken-digit-recognition-with-custom-log-spectrogram-layer-and-deep-learning.html
17. Preprocessing the Audio Dataset - GeeksforGeeks, accessed on August 12, 2025, https://www.geeksforgeeks.org/data-analysis/preprocessing-the-audio-dataset/
18. Should we normalize audio before training a ML model - Bioacoustics Stack Exchange, accessed on August 12, 2025, https://bioacoustics.stackexchange.com/questions/846/should-we-normalize-audio-before-training-a-ml-model
19. Mel-frequency Cepstral Coefficients (MFCC) for Speech Recognition -

GeeksforGeeks, accessed on August 12, 2025, https://www.geeksforgeeks.org/nlp/mel-frequency-cepstral-coefficients-mfcc-for-speech-recognition/
20. Using Python to classify sounds (with PyTorch) | by Martin - Medium, accessed on August 12, 2025, https://medium.com/@mlg.fcu/using-python-to-classify-sounds-a-deep-learning-approach-ef00278bb6ad
21. How can convolutional neural networks (CNNs) be applied to audio data? - Milvus, accessed on August 12, 2025, https://milvus.io/ai-quick-reference/how-can-convolutional-neural-networks-cnns-be-applied-to-audio-data
22. Audio Classification with PyTorch's Ecosystem Tools - Edge AI and Vision Alliance, accessed on August 12, 2025, https://www.edge-ai-vision.com/2021/08/audio-classification-with-pytorchs-ecosystem-tools/
23. Mel-frequency cepstrum - Wikipedia, accessed on August 12, 2025, https://en.wikipedia.org/wiki/Mel-frequency_cepstrum
24. Difference between mel-spectrogram and an MFCC - Stack Overflow, accessed on August 12, 2025, https://stackoverflow.com/questions/53925401/difference-between-mel-spectrogram-and-an-mfcc
25. aniruddhapal211316/spoken_digit_recognition: Speech Recognition on Spoken Digit Dataset using Bidirectional LSTM Model in PyTorch. - GitHub, accessed on August 12, 2025, https://github.com/aniruddhapal211316/spoken_digit_recognition
26. spoken-digit classification using artificial neural network - ResearchGate, accessed on August 12, 2025, https://www.researchgate.net/publication/368915694_SPOKEN-DIGIT_CLASSIFICATION_USING_ARTIFICIAL_NEURAL_NETWORK
27. Why do Mel-filterbank energies outperform MFCCs for speech commands recognition using CNN? - Stack Overflow, accessed on August 12, 2025, https://stackoverflow.com/questions/60439741/why-do-mel-filterbank-energies-outperform-mfccs-for-speech-commands-recognition
28. LD-CNN: A Lightweight Dilated Convolutional Neural Network for Environmental Sound Classification, accessed on August 12, 2025, https://web.pkusz.edu.cn/adsp/files/2015/10/ICPR_final.pdf
29. BUILDING LIGHT-WEIGHT CONVOLUTIONAL NEURAL NETWORKS FOR ACOUSTIC SCENE CLASSIFICATION USING AUDIO EMBEDDINGS Technical Report Bo - DCASE, accessed on August 12, 2025, https://dcase.community/documents/challenge2021/technical_reports/DCASE2021_Kim_35_t1.pdf
30. An Audio Classification API with CNN + FastAPI | by Camilla Nawaz | Aug, 2025 - Medium, accessed on August 12, 2025, https://medium.com/@camillanawaz/an-audio-classification-api-with-cnn-fastapi-cd59e429b709
31. Recurrent neural networks as neuro-computational models of human speech

recognition, accessed on August 12, 2025, https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1013244
32. Understanding LSTM for Sequence Classification: A Practical Guide with PyTorch - Medium, accessed on August 12, 2025, https://medium.com/@hkabhi916/understanding-lstm-for-sequence-classification-a-practical-guide-with-pytorch-ac40e84ad3d5
33. Content-Based Audio Classification and Retrieval Using SVM Learning - Microsoft, accessed on August 12, 2025, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/content_audio_classification.pdf
34. Content-Based Audio Classification Using Support Vector Machines and Independent Component Analysis - ResearchGate, accessed on August 12, 2025, https://www.researchgate.net/publication/220928088_Content-Based_Audio_Classification_Using_Support_Vector_Machines_and_Independent_Component_Analysis
35. Music Genre Classification using SVM - IRJET, accessed on August 12, 2025, https://www.irjet.net/archives/V5/i10/IRJET-V5I10197.pdf
36. (PDF) A Survey of Audio Classification Using Deep Learning - ResearchGate, accessed on August 12, 2025, https://www.researchgate.net/publication/374101086_A_Survey_of_Audio_Classification_using_Deep_Learning
37. librosa.feature.mfcc — librosa 0.11.0 documentation, accessed on August 12, 2025, https://librosa.org/doc/main/generated/librosa.feature.mfcc.html
38. librosa.feature.melspectrogram — librosa 0.11.0 documentation, accessed on August 12, 2025, https://librosa.org/doc/main/generated/librosa.feature.melspectrogram.html
39. Comparative analysis of audio-MAE and MAE-AST models for real-time audio classification, accessed on August 12, 2025, https://www.tandfonline.com/doi/full/10.1080/00051144.2025.2504749?scroll=top&needAccess=true
40. Train Spoken Digit Recognition Network Using Out-of-Memory Features - MATLAB &, accessed on August 12, 2025, https://www.mathworks.com/help/signal/ug/train-spoken-digit-recognition-network-using-out-of-memory-features.html
41. Statistical Tests for Comparing Machine Learning and Baseline Performance - Medium, accessed on August 12, 2025, https://medium.com/data-science/statistical-tests-for-comparing-machine-learning-and-baseline-performance-4dfc9402e46f
42. Statistical Significance Tests for Comparing Machine Learning Algorithms - MachineLearningMastery.com, accessed on August 12, 2025, https://machinelearningmastery.com/statistical-significance-tests-for-comparing-machine-learning-algorithms/
43. ML Series: Day 42 — Statistical Tests for Model Comparison | by Ebrahim Mousavi - Medium, accessed on August 12, 2025, https://medium.com/@ebimsv/ml-series-day-42-statistical-tests-for-model-com

parison-4f5cf63da74a
44. “Transforming Sound: Audio Augmentation in Python” | by Prerak Joshi | Medium, accessed on August 12, 2025, https://medium.com/@joshiprerak123/transforming-sound-audio-augmentation-in-python-89c1c08a836b
45. Enhancing Speech Recognition Accuracy with Data Augmentation Techniques - Medium, accessed on August 12, 2025, https://medium.com/@jesus.cantu217/enhancing-speech-recognition-accuracy-with-data-augmentation-techniques-1debcb54628d
46. SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition, accessed on August 12, 2025, https://arxiv.org/abs/1904.08779
47. Advanced Audio Data Augmentation - Number Analytics, accessed on August 12, 2025, https://www.numberanalytics.com/blog/advanced-audio-data-augmentation
48. bastibe/SoundCard: A Pure-Python Real-Time Audio Library - GitHub, accessed on August 12, 2025, https://github.com/bastibe/SoundCard
49. How to read realtime microphone audio volume in python and ffmpeg or similar, accessed on August 12, 2025, https://stackoverflow.com/questions/40138031/how-to-read-realtime-microphone-audio-volume-in-python-and-ffmpeg-or-similar
50. Realtime Audio Visualization in Python - SWHarden.com, accessed on August 12, 2025, https://swharden.com/blog/2016-07-19-realtime-audio-visualization-in-python/
51. How to do real-time audio signal processing using python, accessed on August 12, 2025, https://python-forum.io/thread-21674.html
52. Connecting to the Microphone and Processing Speech with Python | by Happy nehra | Medium, accessed on August 12, 2025, https://medium.com/@happynehra/%EF%B8%8F-connecting-to-the-microphone-and-processing-speech-with-python-a23b7f463467
53. kylebradbury/ml-project-structure-demo - GitHub, accessed on August 12, 2025, https://github.com/kylebradbury/ml-project-structure-demo
54. A best practice for deep learning project template architecture. - GitHub, accessed on August 12, 2025, https://github.com/L1aoXingyu/Deep-Learning-Project-Template
55. Structuring Your Machine Learning Project with MLOps in Mind | Towards Data Science, accessed on August 12, 2025, https://towardsdatascience.com/structuring-your-machine-learning-project-with-mlops-in-mind-41a8d65987c9/
56. How to Write a Good README File for Your GitHub Project - freeCodeCamp, accessed on August 12, 2025, https://www.freecodecamp.org/news/how-to-write-a-good-readme-file/