SpeechQC-Agent: A Natural Language Driven Multi-Agent System for Speech Dataset Quality

Anonymous ACL submission

Abstract

We introduce **SpeechQC-Agent**, a natural language–driven, multi-agent framework for automated verification of large-scale, multilingual speech-text datasets. Our system leverages a central Large Language Model (LLM) to interpret user-specified verification prompts and orchestrate a set of specialized agents that perform audio, transcript, and metadata quality checks. Each prompt is translated into a structured, dependency-aware workflow graph, executed through a combination of dynamically generated and pre-defined tools. To support evaluation, we release **SpeechQC-Dataset**, a synthetic yet realistic benchmark covering 15.5 hours of Hindi dialogue across diverse speakers, domains, and error types. Experiments across two verification stages-QC1 (audio and metadata) and QC2 (transcript and content), show that ChatGPT-based agents outperform open-weight LLMs in planning accuracy and execution robustness. We further adapt recent agentic evaluation protocols to measure workflow fidelity via subsequence and subgraph metrics. Our framework enables scalable, reproducible, and instruction-driven speech dataset verification, laying the foundation for high-quality speech corpus creation in low-resource settings.

1 Introduction

India is the epicenter of linguistic diversity, with the Census of India (2001) reporting 30 languages spoken by more than a million native speakers. Yet, despite this diversity, even widely spoken languages such as Hindi remain under-resourced in the context of publicly available speech-text datasets. Building speech technologies such as Automatic Speech Recognition (ASR), Text-to-Speech (TTS), and Speech Translation (ST), etc for these languages is critically dependent on the availability of large-scale, high-quality, and diverse speech datasets. However, curating such datasets is a slow, labor-intensive process fraught with several challenges. The creation of speech datasets often involves collaboration with multiple vendors, each adhering to different conventions for audio encoding formats, transcript formatting, and metadata structure. This heterogeneity makes it difficult to design unified processing pipelines. Beyond format inconsistencies, ensuring quality and diversity requires extensive manual validation. This includes verifying transcript accuracy, detecting corrupted or low-quality audio, and ensuring linguistic and demographic balance in speaker representation. The task becomes more complicated when high-quality dataset is needed, either low-quality data are discarded, or the process of validating good data introduces significant delays. Several initiatives, such as those of AI4Bharat and the Vaani collaboration, have attempted to address this problem by building open datasets for Indian languages(et al, 2024) However, these efforts remain constrained by the scalability of human-in-the-loop verification processes. Recruiting, training, and managing large teams of annotators and evaluators is both logistically and financially challenging, especially for low-resource languages.

Recently, there has been significant progress in using Large Language Models (LLMs) as agents across various domains, demonstrating competitive performance in tool use, planning, and decision-making tasks. However, these advancements have largely bypassed the domain of speech dataset quality control. The scarcity of specialized models and benchmarks in this area stems from two key limitations: the absence of comprehensive, high-quality datasets that cover diverse real-world edge cases, and the heterogeneity of speech-text data formats across languages and vendors. These factors complicate the development of robust agent workflows and hinder the transfer of generalization capabilities across tasks.

While some efforts in the agent community have

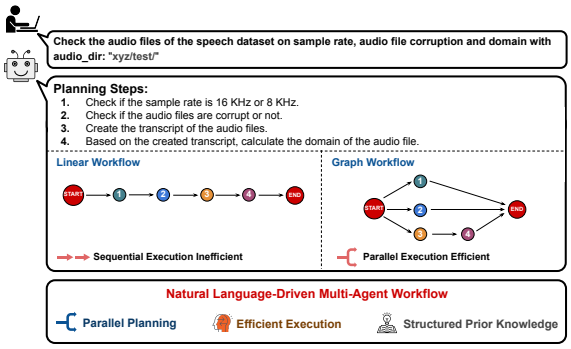

Figure 1: Our system leverages structured prior knowledge and parallel planning capabilities to generate efficient, self-managing task workflows for speech dataset verification.

focused on text-based data processing, they often encounter issues such as inconsistent environment configurations, difficulty adapting to novel data schemas, and poor performance in handling dataset diversity. Even with open-source text datasets, agents have been shown to hallucinate actions, repeat steps unnecessarily, or fail to produce meaningful outputs due to a lack of contextual grounding. These issues are magnified when dealing with multimodal datasets like speech, where alignment between audio, transcripts, and metadata is both critical and difficult to verify. Despite the promise of LLM-powered agents, their application to speech dataset curation remains underexplored and presents a novel set of challenges.

In this paper, we introduce SpeechQC-Agent, a Natural Language driven multi-agent system designed to automate the quality control and verification of large-scale speech datasets. The system is built around a centralized Large Language Model (LLM) that orchestrates a set of specialized sub-agents to carry out format normalization, transcript validation, audio quality checks, and verification decisions. By allowing users to issue natural language prompts (e.g., "Check the audio files of the speech dataset on sample rate, audio file corruption and domain"), the system dynamically generates task-specific workflows and tools, reducing human dependency and enabling scalable dataset processing as shown in Figure 1.

At the core of **SpeechQC-Agent** is a centralized LLM that interprets natural language task descriptions and orchestrates a set of modular sub-agents for data verification. The system accepts a speech dataset comprising raw audio, transcripts, and metadata, along with an instruction prompt (e.g., “Check sample rate, detect audio corruption, and validate transcript language”). This prompt is parsed into an *action list*, which is validated and transformed into a sequence of interdependent verification tasks.

Each task is represented as a node in a directed acyclic graph (DAG), where edges encode execution dependencies. The LLM-based planner identifies which nodes can be executed in parallel and assigns them to specialized agents (e.g., for audio format checking, transcript validation, language identification, or silence detection). Each agent uses either pre-defined tools or tools synthesized by the LLM to complete its task. An execution engine schedules and monitors node completion, ensuring retries in case of failure. The results are compiled into a structured verification report and a quality control dashboard, offering both automated and human-interpretable summaries of data quality (Figure 4).

This paper makes the following key contributions:

- **Natural Language-Driven Workflow Generation:** We introduce the first system to generate speech dataset verification workflows directly from natural language prompts using LLM-based planning, reducing reliance on manual scripting or rigid rule systems.

- **Modular Multi-Agent Execution Framework:** SpeechQC-Agent decomposes verification tasks into modular sub-agents, enabling task-level parallelism and structured dependency management across a graph-based workflow.

- **Tool Synthesis and Reuse:** Our architecture combines dynamically generated tools (via LLMs) with pre-defined, reusable components tailored for common speech processing tasks (e.g., VAD, Domain Identification, CTC scoring), supporting both generalization and efficiency.

- **First Application to SpeechQC-Dataset:** To our knowledge, this is the first end-to-end system that applies agentic workflow generation to real-world speech-text data quality control across multiple languages and vendor formats[1] .

[1]**Code and Dataset Availability:** https://anonymous.4open.science/r/Agents-Pipeline-1023

2 Related Work

Recent advances in LLMs and agent-based systems have led to the emergence of automated frameworks for task orchestration and tool-based reasoning[(Liu et al., 2023, 2024; Zhong et al., 2024a; Song et al., 2023; Zhu et al., 2024; Sun et al., 2024; Xie et al., 2024; Tang et al., 2023; Zhong et al., 2024b)]. Within this landscape, our work intersects with three major areas: (1) LLM-powered multi-agent collaboration, (2) automated agentic workflow generation and evaluation, and (3) modular and self-evolving agent systems. However, none of the existing work addresses the unique challenges of **speech dataset quality verification**, particularly in low-resource, multilingual settings.

Recent research has explored the scaling behavior and design of collaborative multi-agent systems using LLMs. MacNet (Qian et al., 2024) introduces a directed acyclic network topology to support reasoning among thousands of agents, showing that irregular collaborative topologies outperform regular ones. EvoMAC (Hu et al., 2024c) proposes a self-evolving multi-agent collaboration framework for software development, emphasizing requirement-level benchmarking. While these works focus on collaboration scaling, our system emphasizes *task-specialized agent decomposition and coordination*, with clear dependency resolution for audio/text validation workflows.

AFlow (Zhang et al., 2024b) formalizes agentic workflows as DAGs composed of LLM-invoking nodes and edges, establishing a principled representation of modular planning. WorfBench and its evaluation protocol WorfEval (Qiao et al., 2024) go further by proposing a benchmark for agentic workflow generation, utilizing subsequence and subgraph matching algorithms to quantify planning quality. We adapt these techniques to the speech domain by evaluating agent-planned verification pipelines using graph-based metrics, but unlike AFlow(Zhang et al., 2024b), our workflows are grounded in real-world *speech dataset curation tasks* and evaluated using domain-specific metrics like WER/CER and transcript alignment accuracy.

AgentPrune (Zhang et al., 2024a) addresses the issue of communication redundancy in multi-agent systems through message-passing graph pruning, optimizing communication overhead. In contrast, Multi-modal Agent Tuning (Gao et al., 2024a) introduces T3-Agent, a vision-language agent trained with MM-Traj(Gao et al., 2024b) for improved tool usage across modalities. Although both works emphasize tool efficiency, our work is domain-specific and *focuses on curating robust, reusable tools* (e.g., VAD, IndicLID(Madhani et al., 2023), CTC validators) either through LLM synthesis or predefined libraries, optimized for speech data rather than general tool usage.

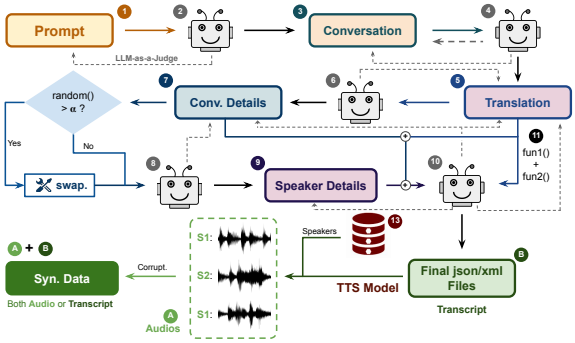

Figure 2: SpeechQC-Dataset generation pipeline. Each numbered step corresponds to an LLM or tool-based operation within the multi-LLM workflow.

The automated design of agentic systems (Hu et al., 2024a) and AutoAgent (Tang et al., 2025) propose zero-code or low-code frameworks to simplify LLM agent creation. Similarly, AgentSquare (Shang et al., 2024) abstracts LLM agents into a modular design space (planning, reasoning, tool use, memory), and introduces an agent search protocol for optimal configurations. While these works streamline agent generation, our focus lies in *end-to-end agentic verification of speech corpora*, leveraging modularity to accommodate heterogeneous formats, languages, and annotation inconsistencies, challenges not addressed in prior agent frameworks.

In contrast to these prior efforts, we introduce a *domain-specific, evaluation-aware agentic framework* tailored for **multilingual speech dataset verification**. We integrate natural language task parsing, structured workflow graph construction, tool invocation, and dashboard-based summarization into a single LLM-driven system. To our knowledge, this is the first work to apply agentic planning frameworks to real-world speech corpora curation and to evaluate agent performance using workflow graph metrics alongside speech-specific quality indicators.

3 SpeechQC-Dataset Pipeline

In this section, we will discuss the data pipeline for the evaluation of SpeechQC-Agent, including data

creation, and quality verification.

3.1 Data Creation Pipeline

We develop **SpeechQC-Dataset**, a synthetic dataset generation framework powered by multi-LLMs. As illustrated in Figure 2, our system simulates realistic conversational interactions, speaker diversity, and common ASR artifacts to generate structured, high-quality audio-text pairs annotated with rich metadata.

1. Prompt initialization: The pipeline begins with a carefully designed prompt (Figure 2, Step 1) that encodes task-specific intent, speaker roles, or domain constraints. This natural language prompt is provided to an LLM agent that orchestrates the conversation planning process.

2. Conversation generation: An LLM-based agent (Figure 2, Step 3) generates a multi-turn conversation from the prompt, simulating realistic human dialogue. We employ long-form in-context examples to improve discourse coherence and pragmatic diversity. A controller agent (Figure 2, Step 2) optionally invokes an LLM-as-a-Judge mechanism to monitor factuality or dialogue realism.

3. Translation: The conversation is translated into one or more low-resource languages using multilingual LLMs or specialized translation modules (Figure 2, Step 5). This ensures language coverage and enables cross-lingual generalization.

4. Metadata extraction: A conversation parsing module extracts fine-grained interaction metadata including speaker turn segmentation, utterance boundaries, intent types, and context tags (Figure 2, Steps 6 -7).

5. Probabilistic perturbation: A randomized perturbation function injects controlled variability by applying operations such as tag insertion (e.g., `<noise>`, `<html>`), token swaps, or word-level noise (Figure 2, Step 8). The perturbation decision is sampled based on a threshold parameter α, introducing structured data variation.

6. Speaker attribution: Synthetic speaker IDs are assigned to each utterance (Figure 2, Step 9) to simulate multi-speaker conversations. This speaker metadata is used to condition downstream audio synthesis, enabling voice diversity.

7. TTS-Driven audio synthesis: A TTS model takes the transcript and speaker annotations to generate speech audio (Figure 2, Step 13). The use of speaker-conditioned models ensures diversity in voice, accent, prosody, and gender.

8. Optional corruption module: To simulate real-world speech recognition challenges, both audio and transcripts may be selectively corrupted with typical ASR errors (e.g., dropped tokens, audio clipping). These corrupted examples support evaluation of the robustness of SpeechQC-Agent (left branch of "Syn. Data").

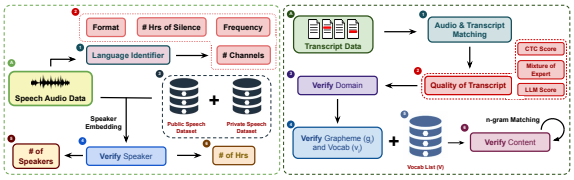

Figure 3: Overview of the SpeechQC-Dataset verification pipeline. (Left) QC1 verifies language, file-level audio properties, and speaker consistency. (Right) QC2 evaluates transcript alignment, quality, domain consistency, vocabulary coverage, and content duplication.

9. Structured conversion: The final outputs, including conversation text, speaker metadata, transcripts, and audio files, are exported in standardized formats `.json` or `.xml` (Figure 2, Step B) for compatibility with downstream evaluation and quality control tasks.

10. Post-processing and validation: Custom functions (e.g., `fun1()`, `fun2()`, Figure 2, Step 11) perform final cleaning, consistency checks, and metadata linking. An LLM-as-a-Judge LLM(Gu et al., 2024) can be invoked to check the generated samples in multiple stages (Figure 2, Steps 2, 4, 6, and 10) to prefer from data missing or hallucination of LLMs.

This pipeline allows precise control over synthetic dataset characteristics while incorporating the flexibility and creativity of LLM-based agents, resulting in a benchmark dataset suitable for evaluating and training multi-agent speech verification systems.

3.2 Data Quality Verification

To further understand the data quality and to thoroughly investigate the data errors in the speech-text dataset. We created SpeechQC-Dataset where we introduce a list of errors in the data using both rule-based and LLM-as-a-Judge approaches(Gu et al., 2024).

The performance of quality control agents, we designed a comprehensive verification suite covering both audio and transcript modalities. Our system is structured into two major verification stages: **QC1** (Audio and Metadata Verification) and **QC2** (Transcript and Content Verification). These checks are aligned with the multi-agent architecture of

QC Level	Check Name	Description
QC1	Language Identification (Audio)	Uses multilingual models to detect the spoken language, independent of metadata.
QC1	File Format	Validates that the audio file conforms to required encoding standards (e.g., WAV).
QC1	Corrupt File Detection	Detects corrupted or zero-length audio files.
QC1	Sample Rate Check	Verifies if the sample rate is 16kHz and above.
QC1	Silence Duration	Calculates total silence duration per audio file.
QC1	Upsampling Detection	Identifies audio upsampled from low fidelity (e.g., 8kHz to 16kHz).
QC1	Number of Speakers	Estimates the number of unique speakers in the batch.
QC1	Per-Speaker Duration	Measures cumulative speaking time per speaker to ensure speaker diversity.
QC1	Speaker Validity	Checks if a speaker is repeated across batches (e.g., same vendor, same voice reused).
QC2	Audio-Transcript Alignment	Aligns transcript with audio, regardless of format or file structure.
QC2	Timestamp-Based Segmentation	Segments long audio using provided transcription time-stamps for utterance-level alignment.
QC2	Script Consistency	Ensures transcript uses the correct native script, avoiding Romanized text unless intentional.
QC2	Code-Mixing Detection	Identifies code-mixed utterances (e.g., English-Hindi), which may require for diverse dataset.
QC2	MOE Score (Mixture of Expert)	Calculates WER and CER using multiple ASR models to quantify transcription quality.
QC2	CTC Score	Computes the Connectionist Temporal Classification (CTC) loss using wav2vec model.
QC2	LLM Score	Evaluates transcript coherence and fluency using LLM-as-a-Judge.
QC2	Transcript Normalization	Removes HTML tags, other tags, or other extraneous tokens from the transcript.
QC2	Transliteration Consistency	Checks Roman-to-native script transliteration for consistent representation.
QC2	Grapheme	Analyzes the distribution of different characters.
QC2	Vocabulary Coverage	Analyzes the distribution of rare words.
QC2	Domain Classification	Assigns each sample to domain labels (e.g., agriculture, etc) to ensure topic diversity.
QC2	Content Repetition Check	Flags duplicated or reused content within or across datasets, including public corpora overlaps.

Table 1: Overview of Data Quality Control (QC) Modules used by the SpeechQC-Agent for analyzing SpeechQC-Dataset and other speech datasets.

SpeechQC-Agent, enabling both modular and parallelized validation workflows.

QC1: Audio and Metadata Verification. As shown in the left panel of Figure 3, QC1 begins by applying a multilingual language identification model to determine the spoken language directly from the audio (Step 1). Subsequent checks validate the audio format, sampling rate, silence duration, frequency upsampling artifacts, and number of channels (Step 2). To handle speaker-related verification, we use speaker embedding-based clustering to identify unique speakers (Step 4) and validate whether speakers are reused across batches by comparing against known public and private datasets (Step 3). Additional statistics such as number of speakers (Step 5) and total speaking time per speaker (Step 6) are computed to assess speaker diversity and duration balance.

QC2: Transcript and Content Verification. The right panel of Figure 3 depicts QC2, which starts by aligning the transcript with the corresponding audio using a timestamp-agnostic model (Step 1). Transcript quality is scored using three different metrics: (i) Connectionist Temporal Classification (CTC) loss from a pretrained wav2vec2.0 model, (ii) Mixture-of-Experts (MoE) relative WER/CER scores from multiple ASR models, and (iii) LLM-as-a-Judge scores evaluating fluency and coherence (Step 2). Domain labels (Step 3) are inferred to ensure topic coverage and diversity. Further checks analyze the distribution of graphemes (Characters) and vocabulary rarity (Step 4), using an internal vocabulary list (Step 5). Finally, content duplication is measured using n-gram and embedding-based overlap with both intra-dataset and public corpus references (Step 6).

Checklist Overview. Table 1 summarizes the verification checks included in both QC1 and QC2. Each verification step is designed to capture a different aspect of dataset quality-ranging from file integrity and speaker redundancy to transcription reliability, script consistency, and domain alignment.

4 Methodology

In this section, we describe our proposed SpeechQC-Agent, a natural language-driven, LLM-coordinated multi-agent framework designed specifically for speech dataset quality verification. The framework takes as input a batch of speech data (waveforms, transcripts, metadata) and a natural-language verification request. It then (i) decomposes the request into atomic checks, (ii) builds an executable directed acyclic graph (DAG) of those checks, (iii) instantiates or retrieves the required tools, and (iv) executes the graph while monitoring progress. All modules are themselves agents coordinated by a central planner LLM. Figure 1 illustrates the pipeline, which consists of the following stages:

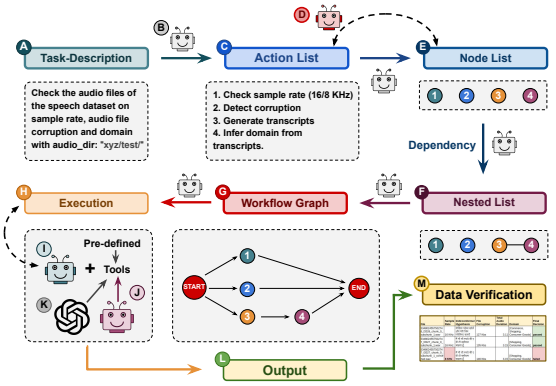

Figure 4: Architecture of the **SpeechQC-Agent** system. Given a natural language task description (A), a central planner LLM interprets the input and generates an ordered action list (C), which is validated (D) and mapped to a node list (E). Dependencies among nodes are resolved into a nested structure (F) and used to generate a topologically sorted workflow graph (G). Each node is executed (H) using dynamically synthesized (I) or pre-defined tools (J). Execution is monitored for completeness, and outputs (L) are aggregated into a structured Data Verification Dashboard (M) for final review.

4.1 Task Parsing and Action Generation

The SpeechQC-Agent: central planning agent (A-B) is the primary interface for interacting with the user. It receives tasks from the user, comprehends the tasks in natural language tasks description q, a central planning agent leverages an LLM to interpret and decompose it into a structured action list $A = [a_1, a_2, a_3, ..., a_n]$. Each action corresponds to a specific atomic quality check, such as `CheckSampleRate`, `DetectAudioCorruption`, or `ValidateTranscript` etc.

$$A \leftarrow PlannerLLM_\theta(q)$$

This planning process uses a combination of lexical mapping (e.g., "VAD" → `SilenceCheck`) and semantic prompting of an LLM to infer implied actions. Moreover, a lightweight rule-based LLM verifier ensures the consistency of A with known hard constraints. For instance, if q mentions silence or VAD, the agent appends $a_{silence}$ to A if missing by the central planning agent.

4.2 Node Generation and Dependency Graph Construction

We define an agentic workflow as a series of LLM invokes in which the action list is further transformed into executable nodes $V = \{v_1, v_2, \ldots, v_n\}$. Each node v_i represents a specific discrete verification subtask performed by an LLM. The dependencies between nodes are explicitly captured to construct a Directed Acyclic Graph (DAG):

$$G = (V, E)$$

where edges E represent dependencies between subtasks, which also govern the execution sequence.

While graph structures can represent workflow relationships W, they require complex extensions beyond basic DAGs to naturally express parallel execution and conditional logic(Hu et al., 2024b). Neural networks enable adaptive transitions but lack percise control over workflow execution(Liu et al., 2023). In contrast, code representation inherently support all the above relationships through standard programming constructs. Therefore, we adopt code(Zhuge et al., 2024) as our primary edge structure to maximize expressivity. Then, the nodes are first linearly sequenced using topological sorting, followed by the establishment of parallel or sequential execution relationships:

$$C(V) \Rightarrow TopologicalSort \Rightarrow G$$

4.3 Tool Synthesis and Retrieval

Each verification node v_i is associated with an executable tool T_i. Tools are selected or synthesized via: 1) Dynamic LLM-based tool synthesis T_{gen}: The agent prompts an LLM to generate the tools and callable functions. The tools are generated on-demand for new or customized tasks. and 2) Predefined tool repository T_{lib}: A curated set of robust tools pre-generated using ChatGPT-4o for stable performance when synthesis fails or confidence is low. The overall tool set is represented as:

$$T = T_{gen} \cup T_{lib}$$

The tool selection may be revised if the tool fails validation checks or runtime execution.

4.4 Workflow Execution and Monitoring

We execute the workflow graph G following the topological order with dependency-aware parallelism. A monitoring agent uses a separate LLM to track the execution status of each node, ensuring completeness and robustness. Let y^i be the output of node v_i. If $y^i = \emptyset$ or an exception is detected, the execution checker retries T_i up to r times. This guarantees completeness:

$$\forall v_i \in V, \quad \exists \hat{y}_i \neq \emptyset \vee fail(v_i)$$

Nodes that fail to execute are automatically retried or escalated for manual inspection.

4.5 Output Aggregation and Dashboard Generation

Upon completion, outputs from all executed nodes are aggregated into structured reports and visual dashboards. These dashboards enable users to interactively inspect and review quality metrics, error distributions, and execution logs for transparency and auditability. This enables both human analysts and downstream systems to filter or prioritize batches for data verification task.

4.6 Modularity and Extensibility

SpeechQC-Agent is modular by design, with each stage (action parsing, node building, tool generation, execution) being LLM-agent pluggable. New tasks can be added by: either by extending the action ontology or defining a new node schema or supplying new tool definitions or enabling auto-synthesis. This architecture generalizes across vendor schemas, speech domains, and languages without manual scripting, making it especially suited for multi-source, low-resource datasets. It also ensures adaptability and scalability to diverse speech dataset curation challenges.

5 Experiment

5.1 Dataset

To evaluate the performance of SpeechQC-Agent, we introduce the **SpeechQC-Dataset**, a synthetic speech-text dataset specifically designed to address the challenges of quality control in multilingual, low-resource language settings, with a focus on Indian languages using the Devanagari script.

The SpeechQC-Dataset was created through a multi-step pipeline using advanced large language models (LLMs) and Text-to-Speech (TTS) technologies to build a diverse corpus. Three LLMs, Llama 3.3 70B-versatile, GPT-4o + 4o mini, and DeepSeek-R1-distill Llama-70B, generated English conversations across 11 domains and 55 settings, ensuring varied dialogue styles encountered in everyday conversation. These were translated into Devanagari script for Indian linguistic relevance, formatted, and converted to audio via a TTS model, incorporating speaker-specific traits for voice diversity. Detailed metadata, including verbatim text, duration, scenario, speaker ID, native language, gender, and domain fields, was compiled into a CSV file for thorough analysis.

Model	File-Format	Corrupt	Sample-Rate	Domain
ChatGPT-4o-mini	100	100	100	28.17
ChatGPT-4.1-mini	100	100	100	60.32
deepseek-r1-distill-llama-70b	0	100	–	0
llama-3.1-8b-instant	0	100	0	0
llama-3.3-70b-versatile	100	100	100	7.54

Table 2: QC1 evaluation across four subtasks. Detect file format error, corrupt file error, sample rate error and domain identification error.

Model	Cost
gpt-4o-mini	< $0.01
gpt-4.1-mini	$0.08
llama-3.3-70b-versatile	$0.04
llama-3.1-8b-instant	$0.06
deepseek-r1-distill-llama-70b	$0.05
meta-llama/llama-4-scout-17b-16e-instruct	$0.04
meta-llama/llama-4-maverick-17b-128e-instruct	$0.05

Table 3: Inference cost (USD / 1K tokens) per 1,000 tokens for different LLMs.

In constructing this metadata and selecting speaker specific details, we utilized the LA-HAJA (Javed et al., 2024) dataset as a reference to extract speaker IDs and corresponding demographic details such as native language, gender, age group, and native state, while replacing their transcripts and audio files with our own synthetic conversations and normalized data to align with the goals of SpeechQC-Agent. This pipeline simulates realistic interactions and introduces controlled variability for ASR challenges, making it an ideal benchmark for systems like SpeechQC-Agent.

5.2 Quality Verification Framework

We evaluate our system across two stages: **QC1 (Audio & Metadata Verification)** and **QC2 (Transcript & Content Verification)**. Table 1 outlines the 22 specific checks performed by SpeechQC-Agent. QC1 validates language ID, file integrity, sampling rate, silence, speaker reuse, and upsampling. QC2 assesses transcript alignment (WER, CER), CTC loss, code-mixing, transliteration consistency, domain labeling, vocabulary coverage, and duplication. For both QC1 and QC2, the dataset is organized into subfolders: QC1-1, QC1-2, and QC1-3 for audio-related checks, and QC2-1, QC2-2, and QC2-3 for transcript-related checks. Here, '-1' represents individual QC transformations, '-2' indicates randomly paired transformations, and '-3' denotes cases where three or more QC transformations are applied.

LLM Variant	Roman Script	Mean WER	# HTML Tags	# EN Tokens
ChatGPT-4.1-mini	99.64	0.094	99.96	100
deepseek-r1-distil-70B	85.2	0.334	0	0
Llama-3.3-70B-versatile	99.46	0.120	100	78.70
ChatGPT-4o-mini	66.40	1.151	98.33	68.17
Llama-3.1-8B-instant	0	-	0	91.34

Table 4: QC2 evaluation across four subtasks. Lower WER, Roman script, HTML Tags and English Tokens the accuracy calcuated based on detection.

Model	Accuracy (%)	Time (sec)
gpt-4o-mini	75	354.207
gpt-4.1-mini	75	136.545
llama-3.3-70b-versatile	62.5	64.056
llama-3.1-8b-instant	25	80.326
deepseek-r1-distill-llama-70b	25	72.693
llama-4-scout-17b-16e-instruct	37.5	50.920
llama-4-maverick-17b-128e-instruct	37.5	117.039

Table 5: LLM performance on QC1 instructions. Each model was evaluated using 3 audio samples each from QC1-1, QC1-2, and QC1-3. The number of missed tasks and total execution time (in seconds) are reported.

5.3 Baselines and LLM Variants

We compare multiple LLM variants as the planning and execution agents, including: *ChatGPT-4o*, *ChatGPT-4.1*, *DeepSeek-R1-distill*, *LLaMA-3.3-70B*, and *LLaMA-3.1-8B*. Evaluation covers: (1) task execution accuracy, (2) hallucinated steps, (3) cost per 1K tokens (Table 3), and (4) runtime.

6 Results

Table 2 summarizes performance across four key QC1 checks: file format, sample rate, file corruption, and domain inference. ChatGPT-4.1-mini and ChatGPT-4o-mini achieved perfect accuracy on format, corruption, and sample rate checks, with ChatGPT-4.1 scoring highest on domain inference (60.32%). In contrast, LLaMA-3.1 and DeepSeek failed to complete most tasks, indicating weak grounding or poor tool invocation capabilities. Table 5 further details execution accuracy and time, showing that ChatGPT-4o-mini completed all tasks correctly within 354 seconds.

Table 4 shows QC2 task performance: ChatGPT-4.1-mini achieved the lowest WER (0.094), high Roman script fidelity (99.64%), and zero HTML artifacts. Table 6 highlights per-task accuracy and hallucination. LLaMA-3.3 exhibited strong performance on domain and vocabulary checks (QC2-2, QC2-3), while ChatGPT-4o-mini excelled on alignment and LLM-score metrics (QC2-1, QC2-2). Table 10 reveals partial metric logging across baselines, suggesting room for improvement in reproducible evaluation pipelines.

As shown in Table 3, ChatGPT-4.1 incurs higher cost ($0.08/1K tokens) compared to LLaMA-3.3 ($0.04), but offers better planning quality and reliability. Table 5 presents task execution times, with LLaMA-3.3 being the fastest (64s) among accurate models, suggesting an accuracy-efficiency tradeoff. While ChatGPT models were slower, they produced fewer hallucinated nodes and avoided redundant actions.

Model	Accuracy	Hallucinated Tasks	Time (sec)
gpt-4o-mini	3	0	347.397
gpt-4.1-mini	3	0	298.261
llama-3.3-70b-versatile	2	0	37.380
llama-3.1-8b-instant	1	0	60.542
deepseek-r1-distill-llama-70b	12.5	2	65.314
llama-4-scout-17b-16e-instruct	0	6	80.937
llama-4-maverick-17b-128e-instruct	37.5	8	86.163

Table 6: Evaluation of LLMs on QC2 instructions. Each model was prompted to execute 8 quality verification tasks (language check, WER/CTC computation, normalization, etc.) for 3 audio samples. Models were evaluated based on task completion accuracy and hallucination rate.

7 Conclusion

We introduced **SpeechQC-Agent**, a natural language–driven, LLM-coordinated multi-agent framework for automated verification of multilingual speech datasets. By interpreting natural language prompts, the system generates structured, dependency-aware workflows and executes them using both synthesized and pre-defined tools. To evaluate system performance, we constructed **SpeechQC-Dataset**, a diverse synthetic benchmark reflecting real-world vendor and language variability. Experiments across audio-level (QC1) and transcript-level (QC2) checks show that commercial LLMs like ChatGPT-4o and 4.1 consistently outperform open-weight models in planning accuracy and execution robustness. We further adapt workflow evaluation metrics, such as subsequence and subgraph F1, to the speech domain, enabling reproducible assessment. Overall, this work represents the first application of agentic workflow systems to speech dataset curation, offering a scalable and traceable alternative to manual quality control pipelines.

Limitations

While SpeechQC-Agent presents a promising framework for automating speech dataset quality control, several limitations remain:

Instruction Following in Open-Weight LLMs: Open-source models such as LLaMA-3.1 and

DeepSeek exhibit weak grounding in complex prompts, often failing to decompose tasks accurately or invoking incomplete workflows. This hinders reliable performance in real-world applications without careful prompt engineering or model fine-tuning.

Tool Generation Hallucinations: Despite structured planning, LLMs sometimes hallucinate tools or invoke modules irrelevant to the task. While our fallback to pre-defined tools mitigates this, fully robust on-the-fly tool generation remains an open problem.

Metric Logging Gaps: Certain verification metrics, such as silence duration via SoX, CTC loss distributions, and LLM-human agreement scores, are not yet logged in a reproducible format across all baselines. This limits the ability to audit and compare agent decisions post hoc.

References

Tahir Javed et al. 2024. Indicvoices: Towards building an inclusive multilingual speech dataset for indian languages. In *Annual Meeting of the Association for Computational Linguistics*.

Zhi Gao, Bofei Zhang, Pengxiang Li, Xiaojian Ma, Tao Yuan, Yue Fan, Yuwei Wu, Yunde Jia, Song-Chun Zhu, and Qing Li. 2024a. Multi-modal agent tuning: Building a vlm-driven agent for efficient tool usage. *arXiv preprint arXiv:2412.15606*.

Zhi Gao, Bofei Zhang, Pengxiang Li, Xiaojian Ma, Tao Yuan, Yue Fan, Yuwei Wu, Yunde Jia, Song-Chun Zhu, and Qing Li. 2024b. Multi-modal agent tuning: Building a vlm-driven agent for efficient tool usage. *ArXiv*, abs/2412.15606.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. 2024. A survey on llm-as-a-judge. *ArXiv*, abs/2411.15594.

Shengran Hu, Cong Lu, and Jeff Clune. 2024a. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024b. Automated design of agentic systems. *ArXiv*, abs/2408.08435.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024c. Self-evolving multi-agent collaboration networks for software development. *arXiv preprint arXiv:2410.16946*.

Tahir Javed, Janki Nawale, Sakshi Joshi, Eldho George, Kaushal Bhogale, Deovrat Mehendale, and Mitesh M Khapra. 2024. Lahaja: A robust multi-accent benchmark for evaluating hindi asr systems. *arXiv preprint arXiv:2408.11440*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024. A survey of nl2sql with large language models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109*.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2023. A dynamic llm-powered agent network for task-oriented agent collaboration.

Yash Madhani, Mitesh M. Khapra, and Anoop Kunchukuttan. 2023. Bhasa-abhijnaanam: Native-script and romanized language identification for 22 indic languages. In *Annual Meeting of the Association for Computational Linguistics*.

Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2024. Scaling large-language-model-based multi-agent collaboration. *arXiv preprint arXiv:2406.07155*.

Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. 2024. Benchmarking agentic workflow generation. *arXiv preprint arXiv:2410.07869*.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. 2024. Agentsquare: Automatic llm agent search in modular design space. *arXiv preprint arXiv:2410.06153*.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2998–3009.

Yiyou Sun, Junjie Hu, Wei Cheng, and Haifeng Chen. 2024. Dfa-rag: Conversational semantic router for large language model with definite finite automaton. In *International Conference on Machine Learning*.

Jiabin Tang, Tianyu Fan, and Chao Huang. 2025. Autoagent: A fully-automated and zero-code framework for llm agents. *arXiv e-prints*, pages arXiv–2502.

Nan Tang, Chenyu Yang, Ju Fan, and Lei Cao. 2023. Verifai: Verified generative ai. *ArXiv*, abs/2307.02796.

Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. 2024. Haichart: Human and ai paired visualization system. *ArXiv*, abs/2406.11033.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. 2024a. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. *arXiv preprint arXiv:2410.02506*.

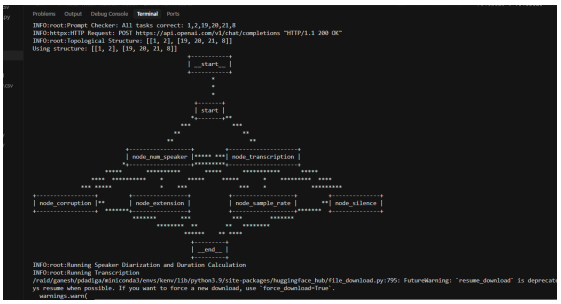

Figure 5: Complete audio processing summary for directory xyz/test/ including speaker diarization metrics, Voice Activity Detection (VAD) silence analysis, and comprehensive quality control validation

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. 2024b. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024a. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Annual Meeting of the Association for Computational Linguistics*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024b. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

Yizhang Zhu, Shiyin Du, Boyan Li, Yuyu Luo, and Nan Tang. 2024. Are large language models good statisticians? *ArXiv*, abs/2406.07815.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*.

A Compute Infrastructure

Compute details: For all our pre-training and fine-tuning experiments, we used two NVIDIA A100-SXM4-80GB GPUs. Each training requires 4-48 hours.

Software and Packages details: We implement all our models in PyTorch[2]

B Dataset Composition

To evaluate the robustness of our audio and transcript quality control mechanisms, we constructed a synthetic dataset with intentional flaws from the LAHAJA dataset and custom-generated data. The dataset comprises four subsets to test specific quality control aspects across diverse error profiles and sources.

- **Vendor A (Audio-Specific):** Applied QC1 transformations (e.g., File Format Conversion, Corrupt File Simulation, Sample Rate Reduction) to 3,000 LAHAJA entries (1,000 individual, 1,000 paired, 1,000 multiple QC1).
- **Vendor B (Transcript Quality):** Applied QC2 transformations (e.g., Audio-Transcript Misalignment, Script Inconsistency, Transcript Denormalization) to 3,000 LAHAJA entries (1,000 individual, 1,000 paired, 1,000 multiple QC2).
- **Vendor C (Mixed Flaws):** Applied both QC1 and QC2 transformations to 100 random LAHAJA entries for combined audio-transcript testing.
- **Vendor D (Synthetic Data):** Generated an independent dataset using LLMs and TTS models for synthetic audio and transcripts with controlled quality parameters.

C Further Analysis

Table 9 presents a comparative analysis of LLM performance on QC1 tasks involving audio and metadata verification. Among all evaluated models, `ChatGPT-4o-mini` exhibited the most reliable behavior, successfully completing all five tasks including file format validation, corruption detection, sample rate checking, speaker duration estimation, and speaker validity matching. `ChatGPT-4.1-mini` also performed well in most categories but failed to correctly handle valid speaker identification. In contrast, `llama-3.3-70b-versatile` completed all core tasks but introduced unnecessary operations, indicating weaker task-grounding. Notably, `llama-3.1-8b-instant`, while the fastest model, failed to execute most tasks and struggled with topological reasoning and task mapping. `deepseek-r1-distill-llama-70b` demonstrated partial success in speaker tasks but did not engage with other checks and required more iterations. These results highlight the trade-offs between speed, instruction-following capability, and task reliability across model families, reinforcing the need for instruction-grounded evaluation in speech data quality workflows.

Table 10 reveals substantial metric-coverage gaps: none of the QC-1 audio-metadata checks (format integrity, silence detection, up-sampling, language ID, speaker diversity or reuse) are logged,

[2] https://pytorch.org/

Table 7: Domains and Settings with LLM Attribution

Domain	Setting	LLM Used
Indian Agri.	Village farm on crops	GPT Models
	Agri fair innovations	GPT Models
	Rural sustainable workshop	Llama Model
	Farmers' crop tips	Llama Model
	School farm trip	DeepSeek
Indian Law	Mock court basics	GPT Models
	Library governance talk	GPT Models
	Civic rights discussion	Llama Model
	Town hall governance	Llama Model
	Constitution lecture	DeepSeek
Indian Finance	Budgeting workshop	GPT Models
	Digital banking expo	GPT Models
	Savings community chat	Llama Model
	Loan process at bank	Llama Model
	Banks' role in class	DeepSeek
Indian Sports	Sports event at park	GPT Models
	Movie event planning	GPT Models
	Fitness benefits in gym	Llama Model
	Dance prep in area	Llama Model
	Cinema fan club	DeepSeek
Indian Military	Fitness drills camp	GPT Models
	Military history talk	GPT Models
	Veterans' community event	Llama Model
	Defence awareness seminar	Llama Model
	Armed forces career fair	DeepSeek
Indian Politics	Democracy school talk	GPT Models
	Political history session	GPT Models
	Civic duties debate	Llama Model
	Voting cultural event	Llama Model
	Civic podcast	DeepSeek
Indian Edu.	Rural learning school	GPT Models
	Student science fair	GPT Models
	Exam study group	Llama Model
	Parent-teacher engagement	Llama Model
	University education day	DeepSeek
Indian Science	Tech innovation exhibit	GPT Models
	Basic coding workshop	GPT Models
	Tech future school club	Llama Model
	Eco-tech startup hub	Llama Model
	Digital tools outreach	DeepSeek
Indian Rural Dev.	Infrastructure village meet	GPT Models
	Sanitation campaign	GPT Models
	Amenities workshop	Llama Model
	Renewable energy event	Llama Model
	Model village project	DeepSeek
Indian Business	Entrepreneurship fair	GPT Models
	Small business seminar	GPT Models
	Trade at marketplace	Llama Model
	Supply-demand class	Llama Model
	Financial planning	DeepSeek
Indian Art	Art evolution exhibit	GPT Models
	Cultural fair performance	GPT Models
	Modern art club	Llama Model
	Architecture history	Llama Model
	Heritage preservation	DeepSeek

Model	Task	Accuracy	Hallucination
ChatGPT-4o-mini	QC2-1	92.34	0
	QC2-2	91.49	0
	QC2-3	47.99	0
ChatGPT-4.1-mini	QC2-1	1	57.21
	QC2-2	1	10.89
	QC2-3	98.07	6.35
deepseek-r1-distill-llama-70b	QC2-1	0	0
	QC2-2	0	0
	QC2-3	0	0
llama-3.1-8b-instant	QC2-1	27.64	10.57
	QC2-2	18.02	0
	QC2-3	0	0
llama-3.3-70b-versatile	QC2-1	0	0
	QC2-2	95.74	
	QC2-3	91.58	0

Table 8: Evaluation of different LLMs on quality control tasks (QC2-1 to QC2-3) measuring Accuracy and Hallucination rate in percent.

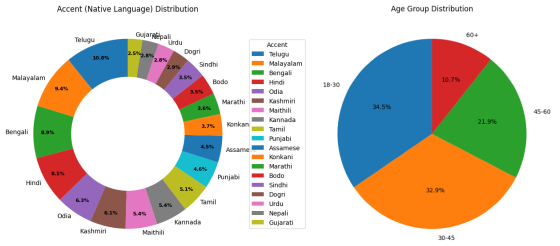

Figure 6: Distribution of speakers in the **SpeechQC-Dataset** by native language accent (left) and age group (right). The dataset exhibits broad linguistic diversity, with representation from 19 native languages, and covers a wide range of age groups, ensuring demographic balance for robust speech technology evaluation.

and several critical QC-2 dimensions, segmentation accuracy, CER, LLM-as-a-judge agreement, transliteration accuracy, vocabulary diversity, domain match, and duplication detection, remain unpopulated or only partially captured. Closing these gaps will require integrating raw SoX/ffprobe diagnostics, diarisation statistics, IndicTrans comparisons, lexical-entropy measures and embedding-based duplication scores, enabling a truly end-to-end, metrics-complete evaluation pipeline for future batches.

D Additional Data Information

It includes 15.51 hours of Hindi speech data from 110 unique speakers, with a balanced gender split of 54 female and 56 male. Speakers cover age groups of 18-30, 30-45, 45-60, and 60+, and represent 19 native languages, led by Telugu, Malayalam, Bengali, Hindi, and others (Fig 6). The data set spans 11 domains, such as agriculture and science and technology, in 55 conversational settings

LLMs	File Format	Corrupt File	Sample Rate	Speaker (Duration)	Valid Speaker	Remarks
`llama-3.1-8b-instant`	Not performing	Not performing	Not performing	Not performing (2/3)	Not performing	Fastest model, but failed to recognize tasks, construct valid workflows, or execute code reliably.
`llama-3.3-70b-versatile`	Completed	Completed	Completed	Completed	Completed	Completed all tasks, but performed additional actions not requested in the instruction.
`deepseek-r1-distill-llama-70b`	Not performing	Not performing	Not performing	Completed	Not performing	Needed more iterations to complete tasks and exhibited unnecessary tool usage.
`ChatGPT-4.1-mini`	Completed	Completed	Completed	Completed	Failed	Executed all required tasks with good flow, but failed in valid speaker verification.
`ChatGPT-4o-mini`	Completed	Completed	Completed	Completed	Completed	Completed all tasks accurately with coherent planning and no unnecessary operations.

Table 9: Performance of LLMs on QC1 verification tasks. Each model is evaluated on its ability to execute file format validation, corruption detection, sample rate checks, speaker duration analysis, and valid speaker matching. Remarks provide qualitative insights into model behavior during task execution.

Protocol Block	Metric	In Sheet?	Missing Evidence / Action
QC1 - Audio & Metadata			
File format & corruption	Accuracy	2	Log SoX/ffprobe checks per file
Silence hours	Precision/Recall	2	Duration histograms with silence detector
Upsampling (8→16 kHz)	Binary accuracy	2	FFT-based up-sampling flag per file
Language ID (MMS)	Accuracy	2	MMS predictions + meta-tags
Speaker hours / diversity	Completeness, SDI	2	Diarisation output, per-ID hours
Speaker reuse detection	Match-rate	2	Embedding match vs. public pools
QC2 - Transcript & Content			
Audio-text alignment	WER, CER	1	CER still missing
Segmentation by timestamps	Seg. accuracy	2	Gold vs. predicted boundaries
Script validity	Script-match %	3	Need total-token denominator
CTC quality score	Avg. CTC	1	-
LLM-as-Judge rating	1-5 score,	2	Per-utt. ratings + agreement
Normalization noise	HTML-error rate	1	Tag counts → rate per K tokens
Transliteration match	Accuracy	2	IndicTrans vs. transcript tokens
Vocab / grapheme diversity	Diversity score	2	Entropy or TTR statistics
Domain verification	Domain-match	3	Need gold domain labels
Duplication detection	Dup. score	3	Embedding-similarity counts

Table 10: Coverage of the full QC-metric suite. 1 = logged, 2 = partially logged, 3 = not present.

(Table 7). Its strength lies in its extensive demographic and linguistic diversity, paired with broad domain coverage, making it a vital tool for inclusive speech technologies.

E Future Work

Future work will explore fine-tuning LLMs on speech-specific reasoning tasks, integrating real-world vendor datasets, and extending the system to correction tasks (e.g., ASR post-editing) and multilingual alignment.

F Discussion

Our experiments demonstrate that **SpeechQC-Agent successfully operationalizes natural language-driven agentic workflows** for speech dataset verification. Key findings include:

- ChatGPT-4.1 and 4o variants consistently outperform open-weight LLMs in execution grounding, especially on QC2 tasks involving complex judgment (e.g., transcript fluency, transliteration).
- Modular agent architecture and topological planning enable robust execution and parallelism, particularly useful in large datasets with heterogeneous error profiles.
- SpeechQC-Dataset offers a diverse, controllable, and reproducible benchmark to evaluate speech QC pipelines-something not previously available for the community.

Nevertheless, limitations remain. Instruction-following in open-weight LLMs remains brittle, and hallucination handling during tool generation needs reinforcement. Metric logging in production still requires integration of raw ffprobe/SoX logs and alignment modules. These insights set the stage for further exploration into fine-tuning LLMs for speech quality workflows, zero-shot error correction, and cross-lingual transfer.

G Prompts

Task Selection Prompt

Prompt:
You are given the following functions:
1. ASR Transcription
2. Number of Speakers calculation and duration per speaker
3. Quality of Transcript
4. Graphene or character calculation
5. Vocab calculation
6. Language identification
7. Audio length calculation
8. Silence calculation (using VAD)
9. Sample rate check
10. CTC score calculation
11. Upsampling Check
12. Check if speakers are new or old
13. Check the domain of the speech dataset
14. Map transcriptions to audio files using forced alignment
15. Language identification using ASR transcriptions and IndicLID
16. Normalization by removing HTML and other tags from transcriptions in JSON or XML files
17. Evaluate transcript coherence and fluency using LLM-as-a-Judge and score out of 10
18. Transliteration - Convert Roman script words to Native script using Transliteration for a specified file and language

Based on the prompt, reply with task numbers that have to be done without any explanation or reasoning.

Input:
Prompt: {user_prompt}

Output Format:
Example: 1,3,5

Topological Sorting Prompt

Prompt:
You are given the following functions:
1. ASR Transcription using audio files
2. Number of Speakers calculation and duration per speaker using audio files
3. Quality of Transcript using transcriptions
4. Graphene or character calculation using transcriptions
5. Vocab calculation using transcriptions
6. Language identification using transcriptions
7. Audio length calculation using audio files
8. Silence calculation (using VAD) using audio files
9. Sample rate check using audio files
10. CTC score calculation using audio files and transcriptions
11. Upsampling Check using audio files
12. Check if speakers are new or old using the results from number of speakers calculation
13. Check the domain of the speech dataset using transcriptions from ASR
14. Map transcriptions to audio files using forced alignment, using ground truth transcriptions
15. Language identification using ASR transcriptions and IndicLID, using transcriptions from ASR
16. Normalization by removing HTML and other tags from transcriptions in JSON or XML files
17. Evaluate transcript coherence and fluency using LLM-as-a-Judge and score out of 10, using transcriptions from ASR
18. Transliteration - Convert Roman script words to Native script using Transliteration, using a specified file and language code from the prompt

We have to do tasks: {resp_1}.

Make a Topological sorting for what is the best way to proceed with these tasks, sequentially and concurrently.

Guidelines:
- We can do tasks concurrently if they are independent of each other.
- Task 12 depends on task 2.
- Task 13 depends on task 1.
- Task 14 depends on the ground truth conversion process.
- Task 15 depends on task 1.
- Task 17 depends on task 1.
- Task 18 is independent.

Output Format:
Example: [[1,3], [5], [8]] (this means do 1 and 3 concurrently, then do 5, and finally do 8)

Finally, give me the topological sorting for the tasks: {resp_1} without any explanation or reasoning.

Input Source Determination Prompt

Prompt:
Determine the source of the following inputs for task {task_id}:
{json.dumps(required_inputs, indent=2)}

Parameters:
Possible sources:
- User prompt: {state.get('user_prompt', '')}
- Previous task outputs in CombinedStateDict: {json.dumps(k: v for k, v in state.items() if k not in ['folder_path', 'user_prompt', 'execution_log', 'task_inputs', 'topological_sort'], indent=2)}
- Default: folder_path={state.get('folder_path', '')}

Output Format:
Return a JSON object mapping each input to its source value or an error message if not found.

Corruption Check Prompt

Prompt:
You are given a folder with audios at this path: {state['folder_path']}.

Write a Python script to:
- Attempt to open and read each audio file.
- If a file fails to load or raises an error, mark it as corrupted and capture the error message.

Save a CSV listing all files and their status ("Corrupt" or "Valid") as audio_validity.csv in the same directory.

Finally, Respond with "Success" if all files are valid, otherwise "Invalid".

Audio Extension and Format Check Prompt

Prompt:
You are given a folder with audios at this path: {state['folder_path']}.

Write a Python script to:
1. Confirm that each file except {file_path} has a valid audio extension (only .wav or .mp3). Ignore files with extensions: .csv, .xml, and .json (do not process, validate or flag them).
2. For audio files, also check if they are in WAV format by attempting to read them using a library like wave or librosa.
3. Create a CSV with columns: Filename, Valid_Extension, `Is_WAV_Format`, Status
4. Status should be "Pass" only if both extension is valid and format is WAV.
5. Save the CSV as `audio_format_check`.csv in the same directory.

Respond with "Success" if all files pass, otherwise "Invalid".

Sample Rate Check Prompt

Prompt:
You are given a folder with audio files at this path: {state['folder_path']}.

Write a Python script to:
1. Check each audio file's sample rate
2. Create a CSV with columns: Filename, Sample_Rate, Status
3. Store "Pass" in Status if sample rate is 16000 Hz, otherwise "Fail"
4. Save the CSV as `sample_rate_check`.csv in the same directory

Use libraries like librosa, soundfile, or wave to check the sample rate.

Ground Truth File Conversion Prompt

Prompt:
You are given a file of ground truths of audios {state['folder_path']} at {file_path}.

1. Get the structure of the txt, csv, json, xml file.
2. Identify the element/column that contains the filename and transcriptions (ground truth). If there is no such column, return "Invalid".
3. Convert the file to CSV with added columns of Filename and Transcription.
4. Save the updated CSV with the new column to the same directory as new_transcriptions.csv.

Finally, Respond with "Success" if all steps are done, otherwise "Invalid".

Conversation Generation Prompt

Prompt:
You are a conversation generator tasked with creating realistic dialogue between exactly two speakers in English.
Topic: {topic}
Setting: {setting}
Speakers: {speaker1} and {speaker2}

Requirements:
- The conversation must be rich in content related to the specified topic and reflect the given setting.
- Generate a long conversation with approximately 100 dialogue exchanges.
- Format the output strictly as:
{speaker1}: sentence1
{speaker2}: sentence2
{speaker1}: sentence3
...and so on.
- Do not include any explanations, actions, or additional text outside the conversation format.
- Ensure the conversation flows naturally and is meaningful with detailed exchanges relevant to the setting and topic.

Output:

Translation Prompt

Prompt:
Translate the following sentence into {language} while maintaining realism and natural flow.
Guidelines:
- The conversation should primarily be in {language}, but preserve certain English words commonly used by {language} speakers.
- Enclose all preserved English words within <eng>...</eng> tags.
- Randomly and sparsely insert conversational effect tags such as [babble], [bg-speech], [laugh], [music], [no-speech], [noise], [overlap], or [silence].
- Use <initial>...</initial> tags for any initials or abbreviations.
- Avoid overusing English words and tags; include them only when contextually appropriate.
- Output only the translated sentence without any explanation.

Input:
Sentence: {content}

Output Format:
Translation: [Translated sentence will be provided here in the specified format with appropriate tags.]

Conversation Metadata Prompt

Prompt:
Generate conversation metadata based on the provided conversation content.
Input:
Conversation: {translated_content}

Output Format:
Generate conversation metadata in the following JSON format:
{"domain":"<domain>","topic":"<topic>","language":"{language}","conversation_name":"{conv_id}-GPT"}

Instructions:
- Determine the "domain" and "topic" based on the conversation content.
- Set "language" to the predominant language of the conversation.
- Use the provided "conversation_name" as is.
- Provide only the raw JSON string without any explanation or formatting wrappers.

Speaker Details Prompt

Prompt:
Generate speaker information for two speakers based on the provided conversation content.
Input:
Conversation: {translated_content}

Output Format:
Generate speaker information for {speaker1} and {speaker2} in the following JSON format:

```
{
"{speaker1}": {
"speakers": [
{
"gender": "<male or female>",
"speakerId": "<alphanumeric ID>",
"recorderId": "<alphanumeric ID>",
"nativity": "{language}",
"ageRange": "<age range like 25-34>"
}
]
},
"{speaker2}": {
"speakers": [
{
"gender": "<male or female>",
"speakerId": "<alphanumeric ID>",
"recorderId": "<alphanumeric ID>",
"nativity": "{language}",
"ageRange": "<age range like 35-44>"
}
]
}
}
```

Instructions:
- Follow the exact JSON structure shown above with all opening and closing braces properly matched.
- Randomly assign values for "gender" (choose either "male" or "female").
- For "speakerId", use a format like "S-XXXXX" where X is a digit.
- For "recorderId", use a format like "RXXX" where X is a digit.
- Set "nativity" to exactly "{language}" as provided.
- For "ageRange", use one of these formats: "18-24", "25-34", "35-44", "45-54", "55-64", "65+".
- Ensure the JSON is properly formatted and valid - all quotes, commas, and braces must be correctly placed.
- Provide only the raw JSON string without any explanation, markdown formatting, or code blocks.

Transcription Function Prompt

Prompt:
Transcribe audio files from a specified folder and return the transcription output in CSV format. This task assumes that all audio files are in Hindi.
Input:
- A folder path containing audio files.
- The folder must exist and be a valid directory.
- All audio files should be in Hindi.

Output Format:
A dictionary with the following structure:
{
"A" [where A is node in the node graph]: "<CSV transcription result or error message>",
"audio_dir": "<Path to the input folder>"
}

Instructions:
- Validate that the provided folder path exists and is a directory.
- If invalid, return the error message: `"A": "Error: Invalid audio directory"`.
- If valid, perform transcription of all audio files in the folder.
- Use the `transcribe_folder_to_csv()` function for transcription.
- Assume the source language is "Hindi".
- Log the transcription process using appropriate logging levels (info and error).
- Return the transcription results in the key "A" along with the input directory.

Silence Detection Prompt

Prompt:
Perform silence detection on all audio files within a specified directory and return the result.
Input:
- A directory path containing audio files to be processed.
- The folder must exist and be a valid directory.

Output Format:
A dictionary with the following structure:
{
"D": "<Silence detection result or error message>"
}

Instructions:
- Check if the provided audio directory exists and is valid.
- If the directory is invalid or not found, return the error message: `"D": "Error: Invalid audio directory"`.
- If valid, apply silence detection to all audio files in the directory using the `process_folder_vad()` function.
- Log the beginning of the detection process with an info-level message.
- Return the result under the key "D".

Vocabulary Extraction Prompt

Prompt:
Extract unique words (vocabulary) from the transcriptions in a CSV file and save them into a new column. Output the updated CSV with the extracted vocabulary.
Input:
- A directory containing a CSV file, typically named `indicconf_hypothesis.csv`.
- The CSV must have a column named `Transcription` or `Ground_Truth` (case-insensitive).

Output Format:
A dictionary in the following format:
{
"vocab_output": "<Path to vocab_list.csv or error message>"
}

Instructions:
- Locate the CSV file using the key `"A"` in state, or fallback to `audio_dir/indicconf_hypothesis.csv`.
- If the file doesn't exist, return: `"vocab_output": "Error: CSV file <path> not found"`.
- Within the CSV, identify the transcription column by searching for 'Transcription' or 'Ground_Truth' (case-insensitive).
- For each row, extract a list of **unique words** from the transcription.
- Store the list in a new column named `vocab_list`.
- Save the updated CSV as `vocab_list.csv` in the same directory.
- Return `"vocab_output": "CSV saved at: <path>"` if successful.
- If the agent fails to complete the task or the file is not created, return an appropriate error message.
- Handle and log all exceptions clearly.

Character Extraction Prompt

Prompt:
Extract unique characters from each transcription in a CSV file and save them into a new column. Output the updated CSV with the extracted characters.
Input:
- A directory containing a CSV file, typically named `indicconf_hypothesis.csv`.
- The CSV must have a column named `Transcription` or `Ground_Truth` (case-insensitive).

Output Format:
A dictionary in the following format:
{
"character_output": "<Path to character_list.csv or error message>"
}

Instructions:
- Locate the CSV file using the key `"A"` in state, or fallback to `audio_dir/indicconf_hypothesis.csv`.
- If the file doesn't exist, return: `"character_output": "Error: CSV file <path> not found"`.
- Identify the transcription column by searching for 'Transcription' or 'Ground_Truth' (case-insensitive).
- For each row, extract a list of **unique characters** from the transcription.
- Store the list in a new column named `character_list`.
- Save the updated CSV as `character_list.csv` in the same directory.
- If the script completes successfully and the file is created, return: `"character_output": "CSV saved at: <path>"`.
- If the agent fails or the output file is not found, return an appropriate error message.
- Log any exceptions during processing clearly and accurately.

Audio Length Calculation Prompt

Prompt:
Calculate the duration of each audio file in a given folder and save the results in a CSV file.
Input:
- A valid directory path containing audio files.

Output Format:
A dictionary in the format:
{
"audio_length_output": "<Result of operation or error message>"
}

Instructions:
- Check if the `audio_dir` exists and is a directory. If invalid, return: `"audio_length_output": "Error: Invalid audio directory"`.
- Write a Python script that performs the following tasks:
1. Iterate over all audio files in the directory.
2. Calculate the duration of each audio file in seconds.
3. Store the filename and corresponding duration in a CSV with columns: `Filename`, `Audio_length`.
4. Save the resulting CSV as `audio_length.csv` in the same folder.
- Execute the script using the `[python_repl]` tool.
- Return the script's output message under the key `"audio_length_output"`.
- In case of failure or exceptions, return an appropriate error message.
- Log errors clearly to aid debugging.

Devanagari Script Verification Prompt

Prompt:
Verify whether each transcription in a CSV file is written in the Devanagari script using Unicode checks.
Input:
- Path to a CSV file (e.g., `indicconf_hypothesis.csv`) with a column containing ground truth text.

Output Format:
A dictionary in the format:
{
"language_verification_output": "<Result of operation or error message>"
}

Instructions:
- Load the CSV file and identify the transcription column (case-insensitive: 'Ground_Truth', 'Transcription', etc.).
- For each row:
1. Remove whitespace and punctuation from the transcription.
2. Check if all remaining characters fall within the Unicode range U+0900–U+097F (Devanagari script).
3. If they do, set `Is_Devanagari` to `True`; otherwise `False`.
4. If the transcription is empty or only punctuation, set `Is_Devanagari` to `False`.
- Add a new column `Is_Devanagari` to the CSV.
- Save the output file as `language_verification.csv` in the same directory.
- Ensure the final CSV includes: `Filename`, `Transcription`, `Is_Devanagari`.
- Use the `[python_repl]` tool to execute the script.
- On success, return `"Success"`; else provide an error message.
- Handle edge cases and log any errors encountered.

CTC Score Computation Prompt

Prompt:
Compute Connectionist Temporal Classification (CTC) alignment scores from audio-transcription pairs and classify alignment quality.
Input:
- A directory containing audio files (`audio_dir`)
- A CSV file (e.g., `indicconf_hypothesis.csv`) with aligned transcripts, identified via key `'A'`

Output Format:
A dictionary in the format:
{
"ctc_score_output": "<CSV output path or error message>"
}

Instructions:

- Load the CSV and audio directory.
- For each audio file, compute alignment scores using the transcriptions in the CSV.
- Use `process_audio_directory()` to return segment-wise alignment with scores and timestamps.
- Aggregate results by:
  - Grouping by filename.
  - Combining the segment labels into a full transcript (`Aligned_Transcript`).
  - Taking the average CTC score as `CTC_Score`.
  - Serializing segment-level details (label, start, end, score) into JSON under `Aligned_Segments`.
- Classify the score using:
  - `Good` if score > 0.7
  - `Medium` if score > 0.5
  - `Poor` otherwise
- Save the final CSV with columns: `Filename`, `Aligned_Segments`, `Aligned_Transcript`, `CTC_Score`, `CTC_Status`.
- Output the result to `ctc_scores.csv` in the same directory as the input CSV.
- Log and report errors appropriately.

Valid Speaker Verification Prompt

Prompt:
Analyze speaker presence across files to determine whether a speaker is "New" or "Old" based on repetition across files.
Input:
- A directory containing a CSV named `num_speakers.csv` with columns:

  - `File Name`
  - `Number of Speakers`
  - `Speaker Durations` - JSON object mapping speaker IDs to durations

Output Format:
A dictionary:
{
"valid_speaker_output": "<CSV output path or error message>"
}
Instructions:

1. Load `num_speakers.csv`.
2. Build a dictionary to track how many files each speaker appears in.
3. For each row:
   - Skip if `Number of Speakers` == "Error".
   - If only one speaker and `SPEAKER_00` is reused across files, mark as `Old`.
   - If multiple speakers and any speaker is reused across files, mark as `Old`.
   - Otherwise, mark the speaker as `New`.
4. For each row, populate:
   - `Filename`
   - `Speaker_Status` (`New` or `Old`)
   - `Common_File` (the current file name if status is `Old`, else empty)
5. Save the result to `valid_speaker.csv` in the same directory.
6. Respond with "Success" if the script runs without errors and file is saved. Otherwise, return "Invalid".

Domain Checker Prompt

Prompt:
You are a Hindi language expert. Analyze the following normalized Hindi transcript and determine the general domain of the speech dataset.
Instructions:

- Return the domain as a **single word** (e.g., `News`, `Call Center`, `Interview`, `Conversation`, `Education`).

Input:
A CSV file `indicconf_hypothesis.csv` located inside a directory, containing a column named `transcriptions` with normalized Hindi transcripts.
Expected Output:
A new column `domain` added to the CSV, representing the predicted domain of each transcription. The final output is saved as `domain_check.csv` in the same directory.
Agent Behavior:

1. Validate the input directory and CSV.
2. Iterate over each row in the `transcriptions` column.
3. For each transcript, send a prompt to the language model to classify the domain.
4. If the LLM fails, label the domain as `Unknown`.
5. Save the resulting DataFrame with the new `domain` column to `domain_check.csv`.

IndicLID Language Identification Agent Prompt

Prompt Objective:
Identify the language of each transcript using the IndicLID model.
Input Description:

- A folder containing a CSV file (default name: `indicconf_hypothesis.csv`).
- The CSV should include a column named `transcriptions` and optionally `Filename`.

Instructions:

1. For each row in the CSV:
    - Extract the transcript and filename.
    - If the transcript is empty or NaN, assign `Language_Code = Unknown`, `Confidence = 0.0`, `Model_Used = IndicLID`.
    - Otherwise, use the IndicLID model to perform language identification.
2. If language identification fails for a transcript, mark it with `Language_Code = Error`.
3. Store all results in a new DataFrame with columns: `Filename`, `Transcription`, `Language_Code`, `Confidence`, `Model_Used`.
4. Save the output as `indiclid_language_identification.csv` in the same directory.

Expected Output:
A CSV file containing language identification results for each transcript, with confidence scores and the model used (`IndicLID`).

Text Normalization and Tag Removal Agent Prompt

Prompt Objective:
Normalize transcription text by cleaning ground truth data in a CSV file.
Input Description:

- A directory containing a CSV file named `indicconf_hypothesis-gt.csv`.
- The file should have a column named `Transcriptions` or `ground_truth` (case-insensitive).

Instructions:

1. Read the CSV file and identify the transcription column (`Transcriptions` or `ground_truth`).
2. Clean each transcript using the following rules:
    - Remove HTML tags like and .
    - Remove any text enclosed in square brackets (e.g., `[START]`).
    - Remove symbols such as #, $, and %.
3. Add a new column named `normalized_transcripts` with the cleaned text.
4. Save the updated CSV as `normalized_list.csv` in the same directory.

Expected Output:
A new CSV file with the original columns and an additional `normalized_transcripts` column saved as `normalized_list.csv`.

LLM-Based Transcription Quality Scoring Agent Prompt

Prompt Objective:
Evaluate the fluency and coherence of ASR-generated transcriptions using a Language Model (LLM) and assign scores and comments.

Input Description:

- A directory path containing a CSV file named `indicconf_hypothesis.csv`.
- The CSV contains:
  - `Filename` column (case-insensitive).
  - One of `ground_truth` or `transcriptions` columns (case-insensitive), containing ASR outputs.

Instructions:

1. Load the CSV file.
2. For each transcription:
   - Analyze sentence fluency and meaning very strictly.
   - Score each transcription from **0 to 10**:
     - 10: Highly meaningful and fluent Hindi sentence.
     - 0: Nonsensical or contains language other than Hindi.
     - Gradually decrease score based on fluency degradation.
   - Provide a brief `Evaluation_Comment` justifying the score.
3. Create a new CSV file with the columns: `Filename`, `Transcription`, `LLM_Score`, and `Evaluation_Comment`.
4. Save the output as `llm_scores.csv` in the same directory.
5. Handle errors gracefully during execution.

Expected Output:
A CSV file named `llm_scores.csv` containing scored and reviewed transcriptions.

English Word Count Agent Prompt

Prompt Objective:
Determine the number of English words present in each line of a normalized transcript using an LLM.

Input Description:

- A directory path that contains a CSV file named `normalized_list.csv`.
- The CSV must have a column named `ground_truth`, containing the transcription text.

Instructions:

1. Load the `normalized_list.csv` file.
2. For each row in the `ground_truth` column:
    - Construct a prompt asking a language expert to count the number of English words (case-insensitive) in the given text.
    - Extract the integer response.
    - If the LLM fails, assign `-1` for that row.
3. Append the count as a new column called `english_word_count`.
4. Save the updated CSV as `english_word_count.csv` in the same directory.

Prompt Template:

> You are a language expert. Count and return only the number of **English words** (case-insensitive) in the following text.
>
> Text:
> `{ground_truth_text}`
>
> Respond with just the number.

Expected Output:
A CSV named `english_word_count.csv` containing an additional column `english_word_count` with English word frequencies per row.

Utterance Duplicate Checker Agent Prompt

Prompt Objective:
Identify and report duplicate utterances across all text-based columns in a CSV.
Input Description:

- A directory containing a CSV file named `normalized_list.csv`.

Instructions:

1. Load the `normalized_list.csv` file.
2. Iterate through each column of the DataFrame.
3. For columns with text (`dtype == object`):
    - Detect duplicated utterances (preserve all duplicates using `keep=False`).
    - For each unique duplicated utterance, count the number of occurrences.
    - Record the column name, the duplicated utterance, and the count.
4. Save the results in a new CSV called `duplicate_utterances.csv` containing:
    - `column_name`, `utterance`, and `count`
5. If no duplicates are found, return a message indicating that.

Expected Output:

- A CSV file named `duplicate_utterances.csv` if duplicates exist.
- Otherwise, a message stating `"No duplicate utterances found."`

WER Computation Agent Prompt

Prompt Objective:
Compute Word Error Rate (WER) between normalized reference transcriptions and predicted hypotheses.
Input Description:

- A directory containing two CSV files:
  - `normalized_list.csv` with the column `normalized_transcripts`.
  - `indicconf_hypothesis.csv` with the column `transcriptions`.

Instructions:

1. Ensure both CSVs exist and contain the same number of rows.
2. For each row, compute the Word Error Rate (WER) between:
   - `Reference` ← `normalized_transcripts`
   - `Hypothesis` ← `transcriptions`
3. Use the `jiwer` library for WER calculation.
4. Handle exceptions on a per-row basis to ensure continuity even if some rows fail.
5. Save the output in a CSV named `wer.csv` with columns:
   - `Reference`, `Hypothesis`, and `WER`

Expected Output:

- A CSV file named `wer.csv` saved in the same directory.
- Each row shows the WER score for the respective transcription pair.

Graph Builder Agent Prompt

Prompt Objective:
Construct a ‘StateGraph‘ from a structured list of task groups while filtering by a valid task set.
Input Description:

- `structure`: A list of lists where each sublist represents a group of task IDs that can be executed in parallel.
- `valid_tasks`: A set of valid task identifiers (as strings). Only these will be included in the final graph.

Instructions:

1. Filter the `structure` to retain only task IDs present in `valid_tasks`.
2. If the resulting structure is empty but `valid_tasks` is non-empty, use all numeric valid tasks as a fallback.
3. Add each valid task as a node in the graph using `node_map`, which maps `task_id` to a tuple: `(node_name, function, description)`.
4. Add a dummy `start` node and connect it to the first group.
5. Connect each group to the next group, allowing fan-in/fan-out connections.
6. Connect the last group to the terminal `END` node.

Expected Output:

- A compiled `StateGraph` object that respects the dependency structure implied by the groupings and task validity.
- An error is raised if no valid tasks remain after filtering.

Prompt Checker Agent Prompt

Prompt Objective:
Analyze a user's natural language prompt to determine whether the currently selected task IDs are appropriate, and update the task list if any are missing based on defined task descriptions and selection rules.

Input Description:

- `user_prompt`: A natural language prompt provided by the user describing the task they want to perform.
- `selected_tasks`: A comma-separated string of task numbers (e.g., "1,2,5") that have been initially selected for execution.

Task Descriptions:

- Contains 24 predefined task definitions, ranging from ASR transcription to WER computation.

Selection Rules:

- Uses keyword and semantic rules (e.g., "if prompt mentions 'Vocab calculation', include task 5") to guide inclusion.
- Tasks 1 and 15 are linked if language identification is mentioned.
- Certain tasks (e.g., 9, 23, 24) trigger the inclusion of dependent tasks (e.g., task 16).

Instructions to the LLM:

1. Analyze the `user_prompt` and determine which tasks are required based on semantic understanding and rules.
2. Compare the determined tasks with `selected_tasks`.
3. If any tasks are missing, return `Status: Missing`, with task IDs and an explanation.
4. If all are correct, return `Status: Correct` and the list of tasks.
5. Format the output as:

```
Status: <Correct|Missing>
Tasks: <comma-separated task IDs>
Explanation: <why tasks were added (if Missing)>
```

Execution Loop:

- Repeats for a maximum of 3 iterations to ensure task completeness.
- Dynamically updates task list with each LLM feedback.
- Calls `select_tasks()` if new insights are needed.

Output:

- Returns the final list of task IDs as a comma-separated string.