

Exhaustive Survey of Free AI-Powered Context Cleaning Pipelines for Voice Input

Executive Summary

Objective: This report provides a comprehensive overview of open-source and free methods to **clean and preprocess AI input from voice sources**. We map out a full pipeline from **speech-to-text transcription** through **disfluency removal**, **grammar correction**, **incoherence detection**, and finally **intent rephrasing/compression**. The goal is a **stateless, one-pass workflow** that can take raw spoken input (e.g. a noisy microphone recording with “um” and “uh” fillers, grammatical slips, etc.) and output a **clean, semantically clear text prompt** ready for a large language model (LLM).

Key Findings: An optimal pipeline can be assembled entirely from free or open components. High-quality **speech recognition** is achievable offline via models like *OpenAI Whisper* (near human-level accuracy on English) or lightweight engines like *Vosk* for real-time needs. After transcription, simple **filler-word removal** (e.g. eliminating “uh”, “um”, “you know”) can be done with rule-based filters or advanced **disfluency detection models** to reach >90% precision in removing verbal tics. Next, robust **grammar correction** tools (like *LanguageTool* or transformer-based correctors such as *Gramformer*) can refine syntax and fix errors, often achieving an $F_1 \geq 0.85$ on benchmark datasets. For ensuring coherence, **nonsense or contradiction detection** can flag logical issues using Natural Language Inference models (to ~80% accuracy on simple contradictions). Finally, **intent rephrasing/compression** can summarize or restructure the user’s request—using summarization models (e.g. *BART*, *T5*) or rule-based compression—to produce a concise query for the LLM.

Pipeline Architecture: We propose a layered pipeline: **(1) Voice-to-text transcription**, **(2) Disfluency & filler removal**, **(3) Grammar and fluency correction**, **(4) Incoherence detection** (with optional rejection or user prompt for clarification), **(5) Output formatting** (ensuring the cleaned text is in a consistent JSON/Markdown or plain text format), and **(6) Intent extraction or summarization** if needed. This pipeline can be implemented as a stateless script or microservice (e.g. a single-pass FastAPI endpoint), requiring no long-term context memory. All components have local-run options—vital for privacy and offline use—though cloud-based alternatives exist for low-resource environments (with caution that free tiers may impose usage limits). Processing on a 4 GB RAM system is feasible by choosing lightweight model variants (e.g. *Whisper tiny* or *base*, smaller neural nets for grammar, etc.) and leveraging optimizations like quantization.

Performance Expectations: With the recommended open-source stack, one can expect **word error rates $\approx 10\text{--}15\%$** for clear English speech transcription, **>90% of filler words removed** with minimal over-cut (well under the 5% threshold), and grammatically polished output. The final cleaned prompt preserves the original intent while eliminating extraneous content, enabling the downstream LLM to focus on the user’s actual request. By deploying this pipeline, even spontaneous spoken queries (with stutters, false starts, or colloquialisms) can be transformed into **fluent, concise inputs** that meet the semantic clarity requirements for reliable AI responses.

Category-Wise Tool Atlas

In this section, we enumerate **all viable open-source/free tools** for each stage of the context-cleaning pipeline. For each category, we compare tools on their capabilities, requirements, and limitations. Key attributes like license, offline availability, resource needs, last update, popularity, and typical performance are tabulated for quick reference. We then describe each tool/method, explaining **what it does, why it's useful for context cleaning, how to integrate it**, and providing example usage where applicable.

Speech-to-Text Transcription Engines (Voice → Text)

Accurate transcription is the foundation of the pipeline. Below we compare leading **Automatic Speech Recognition (ASR)** options that are free/open-source. Emphasis is on **English** speech, offline functionality, and quality.

Engine	License	Offline Support	Interface	Last Update	GitHub Stars	Model Size	Typical CPU Latency
OpenAI Whisper (multiple models)	MIT (code & models)	Yes (local GPU/ CPU)	CLI & Python API	Active (v2025, OpenAI) ¹	91k☆	Tiny to Large (39M–1.5B params)	~2× real- time (Small on CPU) ² ; Large requires GPU (10GB VRAM)
Vosk (Kaldi- based)	Apache 2.0	Yes (fully offline)	C++ library with Python/ Java bindings	Active (v0.3.50 Apr 2024)	13.6k☆ ³	~50 MB per model	Real-time on CPU (lightweight)
Coqui STT (DeepSpeech fork)	Mozilla Public (MPL-2.0)	Yes (local)	Python API, CLI	Maintained (slow updates in 2025)	3.2k☆ (est.)	~190 MB (English model)	~Realtime on CPU (with smaller models)

Engine	License	Offline Support	Interface	Last Update	GitHub Stars	Model Size	Typical CPU Latency
Mozilla DeepSpeech	MPL-2.0	Yes (offline)	Python/C++ API, CLI	Discontinued (archived Jun 2025)	19k☆ (archive)	~180 MB (v0.9.3 model)	Real-time on CPU (optimized for TFLite)
Wav2Vec 2.0 (Meta/Fairseq)	MIT (code)	Yes (with HF models)	Python (HF Transformers)	Active (models on HF Hub)	N/A (multiple repos)	Base: 95M params; Large: 317M	~1× real-time on GPU; slower on CPU
NVIDIA NeMo ASR	Apache 2.0	Yes (requires NVIDIA GPU)	Python toolkit & CLI	Active (2025, v2.x)	5.5k☆ (NeMo)	100M–1B+ (varies)	<i>GPU required</i> (not feasible on 4GB RAM CPU)
CMU Sphinx (PocketSphinx)	BSD-like	Yes (offline)	C library, Python wrappers	Stable (few updates, legacy)	3.6k☆	~30 MB (acoustic+lm)	Real-time on CPU (low resource)

Whisper (OpenAI): A Transformer-based encoder-decoder ASR model trained on 680k hours of multilingual data. **Strengths:** State-of-the-art accuracy on English (WER ~10-15% on challenging audio), robust to accents and noise, and it **outputs punctuation and casing** automatically. It has multiple model sizes; smaller models (Tiny, Base) can run on CPU in near real-time (e.g. 30s audio in ~45s with Tiny ²). **Limitations:** No native streaming—audio must be chunked for real-time use. Larger models require GPUs and >5-10GB VRAM; using large models on a 4GB RAM system is not feasible. Whisper sometimes “cleans up” disfluencies on its own (it may omit vocalized pauses like “um”), which is generally helpful but could be an issue if verbatim transcripts or filler counts are needed. **Integration:** Whisper provides a simple CLI (`whisper audio.wav --model small`) and a Python API (`whisper.load_model()` then `model.transcribe()`). It’s an ideal first layer for accuracy-critical pipelines where slight delay is acceptable. For our context cleaning, using an **English-only smaller model** (`base.en` or `small.en`) is a good tradeoff between speed and accuracy, given the 4GB RAM constraint (these models need ~1-2GB RAM).

Vosk: Built on the Kaldi toolkit, Vosk is optimized for **lightweight, offline speech recognition**. It supports 20+ languages including English and has model sizes on the order of 50–100MB. **Strengths:** Very low resource usage – works on Raspberry Pi and mobile – and **supports real-time streaming** out of the box

(engine processes audio on the fly with minimal latency). It requires no internet and respects privacy. **Limitations:** Accuracy is lower than larger transformer models – WER is higher especially on open-domain or noisy audio. It may not transcribe domain-specific terms as well, and output may lack punctuation by default (though one can apply an RNNLM rescoring to get basic punctuations). No advanced features like speaker diarization or custom model training (beyond Kaldi's complex process). **Integration:** Vosk offers easy bindings (Python `vosk` package, with a simple API to receive JSON with words and timestamps). For context cleaning, Vosk is useful when real-time performance on CPU is needed or memory is very limited, accepting that some transcription errors might then be corrected in later pipeline stages if possible. For example, on a 4-core CPU, Vosk can transcribe live audio with <1s latency, making it suitable for interactive voice assistants (with a slight accuracy trade-off).

Coqui STT (DeepSpeech fork): Coqui STT continues Mozilla's DeepSpeech (an end-to-end RNN ASR based on Baidu's DeepSpeech research). It's open-source and allows local transcription via CPU or GPU. **Strengths:** Simpler architecture that's **easy to deploy**; reasonably fast on CPU for moderate lengths. It inherited pre-trained models from DeepSpeech and expanded them (including multilingual models). The footprint is small (~100-200MB model), and it's privacy-friendly. **Limitations:** As of 2025, Coqui STT is not pushing state-of-the-art – WER is higher than Whisper or wav2vec on challenging audio. The development has slowed (focus shifted to TTS at Coqui), so it's effectively in maintenance mode with limited new updates. Also, as an RNN, it may struggle with very long utterances (DeepSpeech had an input length limit ~10 seconds without streaming; Coqui might have similar unless using stream mode). **Integration:** It provides a Python API (`stt` Python package) and command-line usage. For example: `stt --model en.stt.model --audio file.wav`. In a pipeline, Coqui STT can be a fallback if Whisper is too slow or if GPU is unavailable – but one should expect to rely more on downstream correction because the raw transcript might have more errors (especially for disfluencies, proper nouns, etc.).

Mozilla DeepSpeech: Included for completeness (historical). An open pioneer in end-to-end ASR, but it's now **discontinued** (the GitHub repo was archived in 2025). It demonstrated the viability of local speech-to-text and influenced later projects. **Strengths:** Very simple to use and well-documented; it had a TensorFlow Lite variant that could run on mobile. Good for learning and legacy systems. **Limitations:** No longer maintained (no bug fixes or improvements), and its accuracy is far below modern models – it struggles with noise and complex speech. Also had a limitation of about ~10 seconds of audio per inference in v0.9.3 without streaming. *We do not recommend* DeepSpeech for new projects, except perhaps as an educational example or if constrained to its environment. Instead, use Coqui STT which is its direct successor with community support.

Wav2Vec 2.0 (Facebook/Meta): A cutting-edge *self-supervised* model that learns speech representations from unlabeled audio and fine-tunes on transcribed data. It's behind some of the best ASR systems when fine-tuned. Hugging Face provides pretrained English models like `facebook/wav2vec2-base-960h` that can be used out-of-the-box. **Strengths:** Excellent accuracy when fine-tuned on target domain – can surpass older models with less data. Supports streaming with some modifications (there are real-time variants). Good for custom use-cases because you can fine-tune on specific jargon and accents with relatively small data. **Limitations:** Using wav2vec2 pre-trained models in real-time on CPU is challenging – they're transformer models often needing a GPU for efficient inference. The open pre-trained English model (960h) doesn't do punctuation or casing (only raw words). Setup is a bit more involved (you'd likely use Python and HF's `pipeline("automatic-speech-recognition", model=...)`). **Integration:** In a pipeline, wav2vec2 could be used if you plan to fine-tune a speech model on your specific audio environment or if you want an alternative to Whisper that's fully open (Whisper's training data is not fully transparent,

whereas wav2vec2's methodology is well documented). For example, a fine-tuned wav2vec2 on phone call data might yield better results in that domain. However, if you're sticking to general English, Whisper small might perform similarly or better without fine-tuning.

NVIDIA NeMo ASR: NeMo is a comprehensive toolkit that includes high-performance ASR models (like Citrinet or Conformer-based models). It's open-source (Apache-2.0) and intended for **enterprise or research** with access to GPUs. **Strengths:** Very high accuracy models (some approach WER ~6-10% on LibriSpeech) and lots of configuration – you can train your own models, do speaker adaptation, etc. Good support for streaming and for multi-speaker (with diarization add-ons). **Limitations: Requires NVIDIA GPUs** – it's optimized for CUDA and even for training on multi-GPU rigs. Inference on CPU is not practically supported for large models. Also, using NeMo is an involved process; it's overkill if all you need is to transcribe a microphone input in real-time on a consumer PC. **Integration:** Unless you have a powerful NVIDIA GPU on the system or are willing to use cloud GPU instances (which may not be free), NeMo isn't a first choice for our pipeline. It's mentioned for completeness—if someone wanted the absolute best accuracy and had hardware, they could deploy NeMo's latest model, but within 4GB RAM and free budget, it's not applicable.

CMU Sphinx (PocketSphinx): An older open-source speech recognizer using classical HMM/DNN hybrid techniques. **Strengths:** Extremely lightweight – can run on a microcontroller or old phone. It doesn't require any special hardware or drivers. **Limitations:** Accuracy is very low by modern standards (it might be >30% WER in conversational speech). It often requires a predefined vocabulary or dictionary and is very sensitive to pronunciation. No punctuation or casing. **Integration:** We generally would not use Sphinx unless operating in a highly resource-constrained environment where even a 50MB model is too large or no ML frameworks can be installed. It might have niche use for keyword spotting or as a backup recognizer if everything else fails.

Note: Cloud API Alternatives: There are proprietary STT services (Google Cloud Speech, Azure Cognitive Services, AssemblyAI, Deepgram, etc.) which often have free tiers. For example, Google Speech-to-Text offers ~60 minutes free per month. These can provide excellent accuracy and easy integration via REST API. However, they are not offline and may incur costs beyond free quota. **If using such services**, one could juggle multiple free-tier accounts to extend usage (e.g. two Google accounts for 120 min total) – this is possible but not seamless, and not a preferred solution given the user's requirements (we mention it only as a consideration, not a recommendation). Azure's service even has an option to output "TrueText" with disfluencies removed, which is interesting for our context; but again, cloud reliance and account management make it a secondary option. The rest of our pipeline will assume we use the open-source engines above, particularly Whisper (for best quality) or Vosk (for efficiency), as the **first stage**.

Disfluency & Filler Word Removal

Once we have the raw transcript, the next step is **cleaning disfluencies**: filler words ("um", "uh", "like", "you know"), repetitions, false starts, and self-corrections. Removing these makes the text more concise and easier for an LLM to understand. There are both simple script-based techniques and advanced AI models for this task.

Approaches and Tools:

- **Rule-based and Regex Filters:** The simplest method is to define a list of filler words and simply remove them or replace them with nothing. For example, one can strip standalone “um”, “uh”, “er”, “ah”, and phrases like “you know,” “I mean,” from the text. This can be done via a regex pattern or a stop-word removal approach. It’s computationally trivial and doesn’t require external models. **Limitations:** A naive approach may incorrectly remove words that look like fillers but aren’t. For instance, “um” is usually filler, but “UMass” (short for a university) is a proper noun; a regex could accidentally alter it. Context matters: “like” can be a filler (“I was, like, very surprised”) or a verb/preposition (“I **like** chess” or “It feels **like** winter”). So, a purely rule-based filter might achieve ~90% removal but with a risk of >5% over-deletion if not carefully tuned. However, as an initial pass, it is very useful for obvious fillers.
- **Heuristic Post-processing with POS Filtering:** We can improve on plain regex by using part-of-speech tagging to identify fillers vs normal usage. Many fillers are interjections or discourse markers. For example, if “like” is tagged as a verb or preposition in a sentence, we keep it; if tagged as a discourse marker (or occurs between pauses in speech with no object after it), we remove it. Tools like spaCy or NLTK can be used to tokenize and tag the transcript and then remove tokens that match a filler list and specific POS context (e.g. “UH” or “HESITATION” tags if using a tagging scheme that supports disfluencies).
- **Open-Source Disfluency Detection Models:** There is significant NLP research on detecting disfluencies as a sequence labeling problem (identifying which words are disfluent). State-of-the-art models treat this like a tagging task, often using BiLSTMs or Transformers to label tokens as fluent or disfluent (B-I-O tags). For example, the *Auto-Correlational Neural Network (ACNN)* model by Jamshid Lou et al. (2018) detects “rough copy” patterns in speech for disfluency removal. Repositories like `deep-disfluency` implement such models. More recent approaches fine-tune BERT or T5 to directly generate a cleaned sentence from a disfluent one (as a form of sequence-to-sequence “disfluency correction”). For instance, an arXiv 2023 paper introduced the “**DISCO**” corpus and achieved F1 up to 97.5 on English disfluency removal by fine-tuning transformer models. **Tools/Libraries:** One open source library is `fast-cleaner` (hypothetical example) or using a model on HuggingFace like `HuggingFace Model "ezdikry/roberta-disfluency-detector"` (if exists). There’s also an “awesome-disfluency-detection” list ⁴ which references many papers and some code releases. In practice, one can use a pretrained sequence tagger: for example, a RoBERTa model fine-tuned on Switchboard corpus to label disfluencies, then remove those segments from text. These models would be heavier than rule-based (possibly 100MB+ and require some runtime), but they yield more precise removal, distinguishing “like” (filler) from “like” (normal) by context. They can also handle repetitions and repairs by recognizing patterns (e.g. *reparandum* -> *interregnum* -> *repair* structure).
- **ASR integrated options:** Some speech recognizers already handle filler removal. For example, **AssemblyAI’s API** and **Deepgram’s API** have options to automatically label or remove filler words. Azure’s speech service, as noted, can output “cleaned” transcripts with disfluencies removed (they call it *TrueText*). These aren’t “tools” we can host ourselves for free indefinitely, but if using cloud, it’s worth noting. Whisper itself doesn’t have a toggle for disfluencies, but interestingly, in testing, Whisper often **omits** common fillers (it “tidies up” the transcript by default). In one analysis, *Whisper and AssemblyAI were both good at not transcribing filler words (“uh my um my like” was cleaned to “my ...*

my ...") and at merging repetitions. This behavior can be a double-edged sword: it yields a cleaner transcript but if you actually *want* to capture fillers (say for an analytics app), Whisper would miss them unless prompted to do verbatim. We can consider Whisper's output already partially disfluency-removed, and then apply further cleaning.

Recommended Tool/Method for Pipeline: Given our scenario (English input, goal is to maximize semantic clarity), a combination approach works well: 1. **Transcript segmentation:** Identify any partial words or repeated phrases. Repeated words ("I I I know this") can be reduced to one "I know this" easily with a regex or by a small loop that collapses runs of the same word (some caution needed for cases like "that that" which might be grammatical in rare cases, but usually repetition is unintended in speech). 2. **Filler dictionary removal:** Remove straightforward fillers: e.g. standalone "um", "uh", "hmm", "er" and common phrases ("you know," at sentence start, "I mean,", "like," when used interjectionally). This likely catches the majority. After this step, the text is shorter and easier to handle for an ML model. 3. **Apply a learned model (optional):** To catch less obvious disfluencies (false starts and repairs), you can use a sequence tagging model. For example, a BERT-based disfluency detector can label something like: *"Let's go to [<reparandum> Boston] uh I mean [<repair> Denver] tomorrow"* – and mark "to Boston uh I mean" as disfluent to remove. The result would be "Let's go to Denver tomorrow." This is important because just removing "uh" isn't enough; you had a corrected word "Denver" replacing "Boston". A model can learn to remove the first part. If implementing this, one could use an existing model from HuggingFace (if available) or even leverage an LLM in one-shot mode: e.g. prompt an LLM with "Original (with disfluencies): ... / Fluent version:" to get a cleaned sentence. However, using a large LLM for this might be too slow on CPU and not free unless using a local one. A smaller seq2seq model fine-tuned for disfluency correction would be preferable for automation. According to research, these approaches can get very high precision (95%+ F1).

1. **Post-check:** Ensure that after removal, punctuation and spacing are corrected (sometimes removing words leaves dangling commas or double spaces – a simple regex can fix spacing, and we can have the grammar correction step (next stage) handle punctuation spacing issues).

Example: Input transcript: *"Um I, I think we should um, you know, probably leave – leave around six? I mean, if that's okay."*

- After simple removal: "I, I think we should probably leave – leave around six? if that's okay." (removed leading "Um", removed "um, you know,", left with some repetition "I, I" and "leave – leave").
- After repetition collapse: "I think we should probably leave – leave around six? if that's okay."
- A disfluency model could detect "leave – leave" as a false start ("leave –" as reparandum, second "leave" as continuation of repair) and remove the first "leave –". Also it might see the fragment "? if that's okay." is not a new sentence start (the speaker said "I mean, if that's okay." which is a filler phrase appended). It could remove "I mean," already, and possibly adjust casing ("If that's okay." capitalized).
- The result would be: "I think we should probably leave around six, if that's okay." – which is fluent.

The **precision goal** is to remove >90% of truly extraneous tokens. If using just the simple approach, we might miss some stutters or halves of corrections, but combining with a smarter model or later grammar check can help. Many grammar correction tools also remove some disfluencies as a side-effect (they'll often delete lone interjections or fix duplicates).

Integration Tips: If performance is a concern on 4GB RAM, a full transformer for disfluency might be heavy. An alternative is to use an **embedding-based similarity** to detect self-corrections: some research uses *sentence embeddings* to find if a later part of the sentence "copies" an earlier part (like "to Boston" vs "to

Denver”) and flags the earlier as disfluent. One could implement a lightweight version: e.g. if a fragment after an “I mean” or “sorry” shares words with the fragment before, remove the before part. These heuristics, while not perfect, catch common cases.

There is also an approach of using **audio** signals (if one had word timestamps, a long pause or filler might indicate a break). But since we assume we work with text after ASR, we focus on text methods.

Finally, ensure the output of this step is a cleaned transcript ready for grammar normalization. We expect at this point filler words and obvious disfluencies are gone, but grammar may still be off (because spontaneous speech often has grammar errors or weird casing/punctuation).

Grammar and Syntax Correction

After removing disfluencies, we have a more concise text, but it may still contain grammatical errors, colloquial phrasing, or run-on sentences due to how transcripts work. The grammar correction stage aims to produce a well-formed sentence or set of sentences that preserve the user’s intent.

Tools in this category include both rule-based grammar checkers and AI-based correctors:

- **LanguageTool:** A popular open-source grammar, spelling, and style checker. It has a vast rule base for English (and many other languages) – for example, it knows about subject-verb agreement, common spelling mistakes, punctuation rules, etc. **Strengths:** Completely offline (you can run the Java-based server or use the command-line tool), and it’s fast for checking even long text. It can catch a wide range of errors and even some style improvements. Because it’s rule-based (augmented with some n-gram data), it **doesn’t hallucinate** – it only flags patterns it knows are likely incorrect. It also can suggest corrections, e.g. if you write “I has a apple”, it suggests “I have an apple.” **Limitations:** It might miss more subtle issues that aren’t covered by rules. It also might not fix everything automatically; you often get suggestions and would need to apply them. However, one can integrate it such that suggested corrections are applied programmatically. LanguageTool’s core is open source and can be self-hosted. **Integration:** LanguageTool can be run as a local server (Java) and accessed via HTTP, or via the `languagetool-commandline.jar` for one-off checks. For example, one could call: `echo "She go to school yesterday." | java -jar languagetool-commandline.jar -l en-US -` and it would output suggestions. It can also be embedded via their Java/Python API. In our pipeline, LanguageTool can serve as a **first-pass cleaner for grammar and punctuation** after disfluencies are removed. It might fix casing of “i” to “I”, insert missing commas or periods, and correct basic grammar. The advantage is speed and not relying on large ML models at this stage.
- **Gramformer (Transformer-based GEC):** *Gramformer* is an open-source library that provides neural models for grammatical error correction (GEC). It wraps one or more transformer models that have been fine-tuned to correct English text. Under the hood, it includes models for different aspects: one for detection, one for correction, etc., or quality estimation. For example, one Gramformer model is based on GPT-2 or BART that takes an input sentence and generates a corrected sentence. **Strengths:** Can correct complex errors and rephrase sentences to be more fluent, beyond the static rules of LanguageTool. It can handle things like colloquial to formal conversion to an extent. **Limitations:** As with any neural model, there’s a risk it might misinterpret something and change the meaning if not carefully constrained. Also, running a transformer for each sentence needs more

compute than rule-based checks. Gramformer is designed to output the top few corrected versions with a score – one has to pick the best. *Integration:* There's a Python API (after `pip install gramformer`). You can do something like:

```
from gramformer import Gramformer
GF = Gramformer(models=1, use_gpu=False) # model=1 for correction mode
corrected = GF.correct("She go to school yesterday.")
print(corrected) # "She went to school yesterday."
```

Gramformer will download a pre-trained model (which might be 300MB+). In a 4GB RAM scenario, it might be borderline; but if no GPU, it will run on CPU slowly. Still, for short texts (a few sentences), it's manageable. We could run Gramformer on the transcript sentence by sentence. It will both detect and correct errors in one go, outputting a single corrected sentence. According to its documentation, it uses a *quality estimator* to ensure high-quality corrections.

- **GECToR:** Another open approach is the *GECToR* model (Grammatical Error Correction: Tag, Not Rewrite) by Grammarly's research team. It uses a sequence tagging approach instead of full sentence generation – essentially predicting edit operations. GECToR was state-of-the-art around 2020 for English grammar correction and is faster than seq2seq models. The code and pre-trained models are available. It might require PyTorch and some setup, but it's optimized for performance. GECToR could correct things by predicting tags like "replace _ with were" at position i, etc. This is nice because it keeps the original sentence structure mostly intact except for needed edits. If integration effort wasn't an issue, GECToR is a strong option to include, given it can achieve high precision. However, it's a bit complex to set up compared to Gramformer.
- **Other AI-based options:** One could also leverage an LLM (like GPT-3.5) via an API to do grammar correction by prompt (e.g. "Correct any grammatical errors: ..."). But since we focus on free solutions, we won't assume calling a paid API. If one has an open source LLM (like a 7B parameter model) that can run locally, it could be prompted to do grammar fixes. But smaller models might not be very reliable at this task unless fine-tuned for it. There are also specific models on HuggingFace for grammar correction (some are fine-tuned T5 models on grammar error correction datasets like CoNLL-2014 or BEA corpus). For example, NUS has a CoNLL GEC model. Those can be used similarly to Gramformer's approach.

Why grammar correction is useful in context cleaning: Even if the user's spoken query is understandable, grammatical mistakes or missing words can throw off an LLM or produce unintended interpretations. For instance, consider a spoken question: "*Weather tomorrow Melbourne?*" – A speech recognizer might output exactly that. An LLM might understand it, but it's not guaranteed to handle telegraphic input well; grammar correction would turn it into "*What is the weather tomorrow in Melbourne?*", which clearly conveys intent. Similarly, "*set timer ten minute*" can become "*Set a timer for ten minutes.*" Grammar correction ensures the cleaned text is a **complete, well-formed query**.

Combining tools: We can actually use both a rule-based and an ML tool in sequence for maximum robustness: 1. Run LanguageTool to catch and correct straightforward issues (it might fix spelling, add a missing question mark, etc.). Some of these corrections are low-risk. 2. Then feed the result to Gramformer (or another GEC model) to handle more nuanced phrasing and remaining errors. Or vice versa: use

Gramformer to rewrite, then LanguageTool to double-check minor details. This layered approach might yield a very polished result.

Example integration:

	Input	after disfluency removal:
	"I think we be able to, uh sorry, I think we can meet in evening if that fine"	"I think we be able to, I think we can meet in evening if that fine."
After filler removal:	"I think we be able to, I think we can meet in evening if that fine."	(still grammatically off, has repetition). - LanguageTool might flag "we be able to" as incorrect (suggest "we are able to") and detect the repetition "I think we can ..." (but it might not fully remove it since now it's more semantic repetition than exact duplicate). It would also flag "if that fine" -> "if that is fine". - After LanguageTool auto-fixes (assuming we take suggestions): "I think we are able to, I think we can meet in evening if that is fine."
- it fixed some parts but the sentence is clunky with repetition. - Gramformer could then take that and potentially compress "I think we are able to, I think we can" to just one clause, and add an article:	"I think we can meet in **the** evening if that is fine."	or even it might simplify "I think we can" out if it decides it's redundant, but likely it will produce: "I think we can meet in the evening if that is fine."
- Final:	I think we can meet in the evening if that is fine.	

which is grammatically correct and clear.

We should verify that meaning isn't lost. Good GEC models try not to change meaning, only form. The above process preserved meaning (both clauses "we are able to" and "we can" meant the same thing; removing one was safe).

Quality Metrics: Grammar correction quality is often measured by precision/recall on known corrections (the F1 mentioned, ~0.85, is common for top GEC systems on benchmark test sets). That means most errors are fixed, with relatively few incorrect changes. We consider that acceptable for our pipeline— it's better to occasionally not correct a minor error than to introduce a wrong correction.

Speed Considerations: LanguageTool is very fast (<0.5s per sentence on modern CPU). Gramformer or a Transformer model might take 0.5–1 second per sentence on CPU depending on length. Given typical user input length (maybe one or two sentences, or a short paragraph at most if they monologue), this is fine (a few seconds at most). On 4GB RAM, we might run Gramformer with a smaller model if available (there's a DistilGPT2-based one that's smaller, though perhaps less accurate). Alternatively, we could skip the heavy model if LanguageTool already makes it good enough. If resource usage is a big issue, one might rely on LanguageTool plus maybe a simpler ML like GECToR (which might be faster than full generation).

In summary, for **our recommended pipeline**, we suggest using **LanguageTool** for a quick deterministic grammar/spell pass, followed by an **AI grammar corrector** (like Gramformer or a T5-based model) for deeper polishing. This ensures the text is as grammatically sound as possible, maximizing clarity for the next steps.

Nonsense and Incoherence Detection

Now we have a cleaned, fluent text. However, it's possible that the content could still be **nonsensical, contradictory, or contextually incoherent**. This could happen if the user's speech was actually nonsense (e.g. due to misunderstanding, or if they asked a self-contradictory question). In the pipeline, this step is about **flagging or handling utterances that don't make sense**, to avoid blindly feeding garbage into an LLM (which might then produce equally garbled output or hallucinate an answer).

What constitutes nonsense or incoherence? It can range from: - **Semantically meaningless sentences:** e.g. "Colorless green ideas sleep furiously" (which is grammatically correct nonsense). Or random word salad like "asparagus correct battery staple" which has no discernible meaning. - **Self-contradictory statements:** e.g. "I have never been to France and I lived in France for five years." If a user query contains conflicting info, an LLM might get confused how to respond. - **Illogical requests:** e.g. "Can you draw a sound I'm thinking of?" – perhaps not logically coherent in terms of action (maybe the user is confused on capabilities). - **Off-topic or irrelevant content appended:** e.g. due to ASR errors or background speech, the transcript might have extraneous bits that make it incoherent in context (like a fragment of another conversation).

To detect these, we can apply **Natural Language Understanding** checks: - **Entailment/Contradiction Detection:** Use a pre-trained Natural Language Inference (NLI) model (like *RoBERTa-large-MNLI* or *DeBERTa MNLI*) to test if the sentences within the input contradict each other. For example, split the input into statements and check pairs. If any pair scores "CONTRADICTION" with high confidence, that's a red flag. These models are free (RoBERTa MNLI is on HF, ~350M parameters, could run on CPU for short text). They can catch direct contradictions, which covers a subset of incoherence. - **Question coherence check:** If the input is a question, one could attempt a simplified check: e.g. ensure it has a clear ask. If not, it might be incoherent. But this is vague to do automatically. - **Language model perplexity or entropy:** Compute the perplexity of the cleaned text under a general language model (like GPT-2 or a KenLM 5-gram model). If perplexity is extremely high, it might indicate the sequence of words is not likely in natural language (thus possibly nonsense). However, perplexity can be high for very novel but valid sentences too, so not foolproof. - **Specific classifiers:** There has been research into classifying incoherent vs coherent text (especially for things like detecting schizophrenic speech transcripts or just incomplete answers). If any pre-trained classifier exists (e.g. a small BERT fine-tuned to label a response as "coherent" or "incoherent"), that could be used. Or train a simple classifier on some data (but that's beyond our scope to create new models here).

Given we prefer existing open tools, using an **NLI model** is a straightforward approach: **Example:** Input: "The user says: 'I want to book a flight to Paris. I have never traveled by plane. I flew to Paris last year.'" We can take the statements: "I have never traveled by plane." and "I flew to Paris last year." as a pair, feed into an NLI model. Likely it will predict "CONTRADICTION" with high probability. That tells us the user's statements are internally inconsistent. Our pipeline could then decide to either ask the user for clarification (if interactive) or at least tag the query as potentially incoherent. If it's a single sentence that doesn't logically follow, NLI is less direct because it needs a pair to compare. We could generate a known-fact or a negation to test. For example, nonsense like "Colorless green ideas sleep furiously" – we might test against a trivial true statement like "Ideas can sleep." or some pattern, but that might not work.

Another approach: **LLM-based judge** – ironically, using an LLM (like GPT-4 via OpenRouter in a research project) to assess contradictions was shown effective ⁵ ⁶ . But since we want free solutions, we might use a smaller LLM or logic rules.

In practice: Many pipelines might skip explicit incoherence detection and rely on the final LLM to handle it (the LLM might answer "I'm not sure what you mean." if input is nonsense). But since the user asked for it, we include it for thoroughness.

Integration Options: - Use a **Roberta-large-MNLI** model offline. After grammar correction, take the cleaned text, split into sentences if more than one. For each pair of sentences (or each sentence vs each other), run through the NLI model. If any output label is "contradiction" with confidence > 0.8, mark the

input as containing a contradiction. If the entire input is one sentence or question, you could try something like comparing it with itself (that yields entailment trivially) or with a slightly altered version. That's not helpful. Instead, for single-sentence nonsense, consider a heuristic: check for presence of content words vs garbage. If the sentence has very low **semantic similarity to any known pattern**, e.g. embed the sentence with Sentence-BERT and see if it's far from typical sentences in vector space (this requires having some reference, though). - Alternatively, use an **embedding clustering approach** such as *the logical inconsistency detector project* which clustered sentences by topic and then had an LLM judge within clusters ⁷ ⁶. That is quite involved and uses GPT-4 (which we can't replicate freely). - Simpler: If the text is long and we suspect incoherence if it jumps topics abruptly, we could measure similarity between adjacent sentences using embeddings (like Universal Sentence Encoder or SBERT). If similarity is very low but the sentences are supposed to be connected, maybe it's incoherent. But if it's just multi-topic question, not necessarily incoherent.

Edge Case: Very short queries like "What about tomorrow?" with no context – technically that's incoherent without context. Our pipeline has no memory of prior conversation (stateless), so such an input is ambiguous. We might detect that it's referring to something unknown. A possible approach is to detect **dangling references** (words like "it", "that", "tomorrow") without context. That's a complex NLP task (anaphora resolution needs context which we don't have). But as a stateless system, perhaps we can flag any query that is too short or starts with "what about..." as needing clarification.

Given the complexity, we'll likely implement incoherence detection in a limited way: - **Contradiction check with NLI** – covers blatant logical inconsistency. - **Non sequitur check** – if multiple clauses with no semantic link. Possibly use a threshold: if the text has multiple sentences and the cosine similarity of their embeddings < 0.2, flag. - **Meaningless phrase check** – we could have a list of known "non-meaningful" constructs or use a language model to see if it can paraphrase it. If even an LLM (small) can't paraphrase or explain it, it might be nonsense.

Action upon detection: This step would **flag** the issue. In a live system, one might prompt the user: "Your request is unclear or seems contradictory. Please rephrase." In our pipeline (which might be automated for input to an LLM), we could attach a tag or filter it. For example, if nonsense, we might decide not to send it to the LLM at all (to avoid garbage in/out). Or we could still send it but prepend a note like "[UNSURE]" or handle it in final output. The user's directive doesn't specify exactly, but "flagging" implies at least marking it.

Example: If the input after cleaning is "The sky is green and the sky is not green." – we detect contradiction. We might output a structured result or message: `{ "cleaned_text": "The sky is green and the sky is not green.", "flag": "incoherent" }`. Or simply include a sentence in the final text like "(Note: the input appears logically inconsistent.)"

To avoid complicating the LLM prompt, maybe we keep it separate. But since they wanted a unified prompt to the LLM, likely we'd rather *not include nonsense or we refine it*. We can consider the possibility of **dropping or rewriting contradictory parts**. That, however, can alter user intent. Probably safer to just note it.

Tools Recap: Pretrained NLI models are readily available (RoBERTa-large-MNLI is 1.3GB though – too large for 4GB with everything else, but DistilBERT MNLI exists, about 300MB). That might be okay. Alternatively, an ALBERT model or smaller could do. Or even use GPT-2 with a custom prompt like "Is the following contradictory or not?" – not reliable.

We have to choose something. I'd lean on a smaller NLI or skip heavy logic if resources tight, trusting the final LLM. But since success metrics include *incoherence flagging accuracy* $\geq 80\%$, we should include something.

One possibly lightweight trick: **Double-check with grammar tool** – sometimes nonsense triggers grammar/spell issues (not always though). E.g. "Colorless green ideas sleep furiously." LanguageTool wouldn't flag it – it's grammatically fine. So scrap that.

Conclusion: We will implement a **contradiction detection** for multi-sentence inputs using an NLI model or heuristic. For single-sentence queries, if they are command or question, we rely on the fact that grammar correction and filler removal likely yield a relatively normal sentence. If it's still nonsense, an LLM would probably fail anyway, but at least we tried.

In sum, this step is more of a **validation filter** to catch clear problems: - If flagged, one could decide to not proceed or to attach a warning. The user's pipeline might simply output a note like "Input may be nonsensical." Since it's stateless, better to alert than to guess.

Intent Rephrasing and Compression

Finally, we may want to **rephrase or compress the user's intent** into an even more concise form or a structured format for the downstream system. This is somewhat optional and use-case dependent. Two major sub-tasks here: 1. **Summarization/Rephrasing:** If the user rambled or gave a long description, we might summarize it into a direct question or command. The idea is to *shorten the context window usage* and remove extraneous detail that the LLM doesn't need for understanding the request. For example, if a user says: "So, um, I was wondering if you maybe know, like, what the weather is going to be like tomorrow? Because I have an outdoor event." – By this stage, we'd have cleaned it to something like "I was wondering if you know what the weather will be like tomorrow, because I have an outdoor event." This is already decent, but we could compress to "What will the weather be like tomorrow for my outdoor event?" or simply "What is the weather forecast for tomorrow?". Summarization can do that kind of compression while preserving intent. 2. **Structuring/Formatting (to Markdown/JSON):** Depending on how the final LLM expects input, we might format the cleaned query. For example, some advanced systems use a JSON with fields like `{"intent": "weather_query", "location": "Melbourne", "date": "tomorrow"}` for a downstream weather API call, or markdown for better readability. The prompt didn't explicitly require a JSON output, but it mentioned formatting as a capability. If needed, an *intent classifier* could classify the query into a category (like "weather_query") and extract entities like location and date (using an NER tool or regex on the cleaned text) to form such JSON. However, that's more relevant if integrating with external actions. In a pure LLM Q&A context, we likely just output a cleaned question text.

Given the scope, we focus on summarization and rephrasing techniques: - **Abstractive Summarization Models:** The likes of *Facebook BART-large (CNN/DailyMail)* and *T5* are the top choices. BART-large-cnn is 400MB and known to produce fluent summaries of articles; in our case, the "article" is just one query, which might be overkill. But it can paraphrase. *T5-base* (220M) or *T5-small* (60M) can be fine-tuned to paraphrase or compress sentences too. HuggingFace has T5 variants specifically fine-tuned for dialogue summarization or similar tasks. Also *Pegasus* (by Google, specifically pre-trained for summarization tasks) could be used if needed. - **Prompting an LLM for rephrasing:** If we had an LLM in the loop (even a local one like LLaMa-7B), we could prompt: "Paraphrase the following query in a concise manner: ...". But running a 7B model might not fit 4GB easily. There are smaller models like *FLAN-T5-small* (which might even do the trick of

understanding and rephrasing with only ~300MB footprint). FLAN-T5-small or base are fine-tuned on instructions including paraphrasing likely. - **Traditional text simplification or compression algorithms:** e.g. *Textrank* summarization (graph-based extractive) – not ideal for a single question, that's more for multi-sentence docs and it extracts important sentences (not relevant here). - **Heuristic compression:** e.g. removing subordinate clauses not needed. But that's basically summarization.

Intent classification: If the goal was to integrate with a virtual assistant system, one might classify the cleaned text into an intent label (like “set_timer”, “weather_query”, “call_contact”, etc.). There are open-source NLU frameworks: - *Rasa NLU* – you can train it on example intents and it will extract entities. It's free and offline. But one must provide the training data for intents of interest. - *Snips NLU* (was open, but since Snips got acquired, not sure if still updated). - *Simpler*: Use HuggingFace's *zero-shot-classification* pipeline with a list of possible intents. For example, feed the cleaned query to a model like `facebook/bart-large-mnli` with candidate labels `["weather", "timer", "reminder", "smalltalk", ...]` and see which fits best. This doesn't require training and is free, but you have to know possible intents. Given the user didn't provide a predefined schema of intents, we won't dive deep, but we note it as an optional final layer. They did mention “*task inference agents*” under key capabilities, which likely refers to exactly this – figuring out what the user intends so the system can respond appropriately (or call an API).

For **our immediate pipeline**, we assume the final output will be given to an LLM that can handle general questions, so we might not need to categorize explicitly. Instead, focusing on summarizing/rephrasing for brevity is key.

When is summarization needed? If the user's cleaned query is already a single sentence question, we might skip it. But if user gave a long background story, summarization helps. Also if the user literally said “TL;DR:” or something, but that's separate.

Example: Cleaned text: “I have a meeting tomorrow at 9, and I want to wake up early. Can you set an alarm for me at 7 AM? Also, what's the weather like tomorrow?” – This is a compound query (two intents: set alarm and weather). Summarizing might not be appropriate because they actually asked two things. This might be an *intent splitting* scenario. Our pipeline might in fact split it into two queries or at least identify multiple requests. We didn't explicitly plan for splitting, but it's related: do we compress into one or separate?

This touches an advanced concept: if multiple sentences contain distinct intents, a system might handle them separately. But an LLM like ChatGPT can answer multi-part questions in one go, so it could be fine. If we were handing off to a non-conversational system, we might need to branch. That's perhaps beyond scope – likely we assume a conversational LLM that can handle the cleaned text as is. So summarization should not remove an intent. It should either keep both or clarify them.

Maybe a safer focus is **rephrasing** the request more cleanly rather than summarizing (which sometimes implies dropping less important info). - We can prompt a model: “Rephrase more directly: ...”.

There are indeed models for paraphrasing. T5 can do it with a prefix like “paraphrase: ...”. LanguageTool's dev site also has a “*Paraphrasing Tool*” but that's likely their premium feature with an LLM.

Our choice: Use an **open summarization model** such as *BART-large-cnn* or *T5*. Bart-large-cnn's license is MIT, so fine. It's around 1.6GB RAM usage probably, which on 4GB might be borderline but perhaps if

quantized or using half-precision it could fit. Alternatively, *T5-base* might use slightly less. *DistilBART* is another smaller variant (DistilBART-CNN is ~300M I think). Given we don't have to implement it actually here, we can just describe using it.

Alternatively, for brevity, we might not always need heavy summarization if the text is short. So pipeline could decide: if cleaned text > N words, then summarize. N could be like 20 or 30 words (since LLM context is valuable, though modern LLMs have large context, but still, the user prompt might want concise input). Summaries should maintain $\geq 80\%$ of important meaning. That's a design parameter.

One more angle – formatting: They mentioned outputting to Markdown/JSON or specific format. For example, after summarizing or cleaning, we could present the final query in a markdown quote or as a JSON field. This might be useful if the next system expects a structured prompt. Or if this pipeline's output is shown to the user or developer, formatting nicely helps.

We might give an example of an **ASCII flow diagram** or **structured example** in the next section to illustrate the whole pipeline with an input example and outputs at each stage. That will indirectly show formatting.

Pipeline Assemblies

Having surveyed the tools and methods for each layer, we now describe how to assemble them into end-to-end **pipeline configurations**. We present a step-by-step blueprint for a few scenarios, considering trade-offs like speed vs. accuracy and local vs. cloud components. Each pipeline assembly is a **combination of the above tools** arranged in the order of processing, with notes on integration.

We will outline two primary pipeline versions: 1. **Local-Optimized Pipeline (Completely offline, 4GB RAM compatible)** – prioritizes lightweight models and speed while keeping everything on-device. 2. **Accuracy-Maximized Pipeline (Offline + minimal cloud or heavier models)** – uses more powerful components (Whisper large, etc.) where possible for best results, assuming a more capable machine or acceptance of some cloud usage.

Both share the same logical stages: ASR → Disfluency Removal → Grammar Fix → Coherence Check → Final Rephrase.

Pipeline 1: Local-Optimized (4GB RAM, CPU-only)

This pipeline ensures all steps run within a 4GB memory budget and avoid any external API. It uses smaller-footprint tools:

Stage 1: Speech Transcription – Use **Whisper Tiny (English)** model via `whisper.cpp` (GGML C++ port). `Whisper.cpp` allows running Whisper with quantized models (e.g. Tiny.en 4-bit quantization uses ~100MB RAM). The transcription might sacrifice some accuracy but is still quite good for English. Alternatively, use **Vosk** with the 50MB model for English. We choose Whisper Tiny.en here for its better handling of casual speech. Command example:

```
./main -m models/ggml-tiny.en.q4.bin -f input.wav -otxt -pp
```

(This would output a transcript text with basic punctuation). We disable any built-in “prompt” that might remove filler (Whisper by default might not transcribe very faint “um”, but Tiny tends to transcribe more literally than larger models, so we may get filler words in output). If using Vosk via Python:

```
import vosk, sys
wf = wave.open("input.wav", "rb")
model = vosk.Model("model-en") # load small model
rec = vosk.KaldiRecognizer(model, wf.getframerate())
text = ""
while True:
    data = wf.readframes(4000)
    if len(data) == 0:
        break
    if rec.AcceptWaveform(data):
        res = json.loads(rec.Result())
        text += res.get('text', '') + " "
text += json.loads(rec.FinalResult()).get('text', '')
```

This yields a lowercase text without punctuation (Vosk’s default). We might then quickly run a punctuation addition tool; there are lightweight punctuation models (like an LSTM that takes text and inserts periods and commas). For simplicity, maybe skip punctuation here and rely on grammar tool.

Stage 2: Disfluency Removal – Implement as a simple Python text filter. For example:

```
text = re.sub(r"\b(um+|uh+|er+|ah+)\b", "", text, flags=re.IGNORECASE)
text = re.sub(r"\b(i mean|you know|like)\b,?", "", text, flags=re.IGNORECASE)
text = re.sub(r"\b([A-Za-z]+)\s+\1\b", r"\1", text, flags=re.IGNORECASE) #
remove repeated word duplicates
```

This removes standalone fillers and duplicates. We then trim extra spaces. This should remove >90% of filler tokens. (We carefully do case-insensitive to catch “Um” or “UM”). For our low-resource pipeline, we **won’t use a heavy disfluency ML model** to save memory. The assumption is Whisper tiny or Vosk might have left in a lot of “um” which this catches. False removals are possible but rare in simple filler cases.

Stage 3: Grammar & Spelling – Use **LanguageTool** in offline mode. Since we want to avoid Java overhead in 4GB, an alternative is to use **LanguageTool’s Python wrapper** (`language_tool_python` library). It can run with an included pre-built rule set (it might internally launch a Java process, but it’s not too heavy for a short text). Example:

```
import language_tool_python
tool = language_tool_python.LanguageTool('en-US')
```

```
matches = tool.check(text)
text = language_tool_python.correct(text, matches)
```

This will apply all suggestions automatically. Now text should have proper casing, spelling, basic grammar. If the text was all lowercase (like Vosk output), LanguageTool might correct “i” to “I” etc., but it might not add all punctuation perfectly. Still, it will fix things like missing articles or obvious verb conjugation errors.

If `language_tool_python` is too slow or heavy, an alternative is to use a **small GPT-2 based corrector**. However, those are not common; LanguageTool is quite optimized for this scenario.

Stage 4: Incoherence Check – Given resource constraints, avoid large NLI. Instead, implement a basic check: - If text contains contradiction phrases like “X and not X” explicitly (we can pattern search for “not” or “never” vs presence of same word earlier). - If the sentence count > 1, and certain adversative conjunctions are present (“but”, “however”, etc.), we could flag if structure looks negating. This is very rudimentary. - Also, if after all cleaning the text is empty or just a fragment (“if that is fine.” with no main clause), that’s incoherent contextually. We can detect if text has no verb or such using a POS tagger (maybe too heavy). We acknowledge this is a weak spot for the low pipeline. We might choose to skip explicit detection here, or simply log “no incoherence detection in this mode” and rely on the final stage.

Stage 5: Final Rephrasing – Use **T5-small** model for paraphrasing because it’s only 60M. There is a T5-small fine-tuned for paraphrase on PAWS or similar. If not, we can use the general checkpoint with an instruction. Example using HuggingFace Transformers (which can run in 4GB for T5-small):

```
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer
tok = AutoTokenizer.from_pretrained("t5-small")
model = AutoModelForSeq2SeqLM.from_pretrained("t5-small")
input_ids = tok.encode("paraphrase: " + text, return_tensors='pt')
outputs = model.generate(input_ids, max_length=100, num_beams=4,
early_stopping=True)
paraphrased = tok.decode(outputs[0], skip_special_tokens=True)
```

This would produce a rephrased version of `text`. For example, input text “I think we can meet in the evening if that is fine.” might output “We can meet in the evening if that’s okay with you.” – similar meaning, slightly different phrasing. The main purpose is to ensure brevity and clarity. If the text is already short, T5 might just output something nearly identical.

We then have our final cleaned prompt in `paraphrased`.

Memory note: This pipeline uses: - Whisper tiny (100MB model in C++ memory), - Python regex (negligible), - LanguageTool (couple hundred MB maybe for all rules), - T5-small (240MB loaded). This fits in 4GB, albeit snug if all loaded at once. We could unload Whisper model after transcription to free memory for T5.

Pipeline Flow (ASCII Diagram):

```

[Audio Input]
  ↓ (Whisper tiny.en or Vosk)
[Raw Transcript text] --> [Regex Filler Remover] --> "transcript sans fillers"
  → (simple repeats removed)
  ↓
[LanguageTool Grammar Check] --> "grammatically corrected text"
  ↓
(Simple logic check)
  ↓
[T5-small paraphraser] --> "final cleaned query"

```

This yields the final output. The pipeline is stateless: each new audio goes through the same steps from scratch.

Pipeline 2: Accuracy-Maximized (Enhanced models, possibly cloud help)

For use-cases where better accuracy is needed and we have more resources (or are willing to use free cloud for heavy tasks), we can swap in more powerful components:

Stage 1: Speech Transcription – Use **Whisper Small or Medium model** on GPU (if available) or via OpenAI's Whisper API (if free trial/credits are available). Whisper Medium on a decent GPU will give extremely accurate transcripts, including proper nouns, etc., and still include punctuation. If GPU isn't available locally, one might use OpenAI's API (paid) or try AssemblyAI's free credits for their Whisper-based API. But since we stick to free: maybe use **Whisper medium on CPU overnight** (not real-time but for accuracy). Or at least Whisper base which is already quite good (WER ~10-15%). We set it to output with `--task transcribe --language en --no_fallback 1` to ensure it tries to capture everything verbatim. If multiple speakers or need diarization, we could incorporate that (Whisper can't diarize itself, but one could use a VAD + speaker ID pipeline externally).

Stage 2: Disfluency Removal – Here we can incorporate an **AI model**: for instance, fine-tune a small T5 model on disfluency removal. However, an easier way: we can use an open model from research. One idea: use **BERT-based tagger**. There is likely a model on HF like `pgrob13/roberta_disfluency` (just hypothetically). If not readily available, another approach: Actually, run the text through **GPT-3.5** with a system prompt "Remove filler words and stutters, output the cleaned sentence." If one has API access or ChatGPT, that's free in ChatGPT UI but not for an automated pipeline. Let's assume not.

Alternatively, use **InstructGPT via HuggingFace** – there's a model like `google/flan-t5-base` which has been instruction tuned. If we give it an instruction to remove disfluencies, it might do it. For example:

```

prompt = "Instruction: Remove filler words and repetitions from the following
text.\nText: \"" + transcript + "\"\nOutput:"

```

and feed to flan-t5-base generate. It might perform some cleaning by itself (though not guaranteed perfect). However, for proven results, a sequence tagger is more deterministic. The *deep-disfluency* code

could be run if we have the model weights (it might require TensorFlow and the Switchboard data though, which is troublesome).

Given we want exhaustive, we might mention the possibility: e.g. "One could fine-tune a BERT on the Switchboard disfluency annotations and get near 97% F1 as reported. But for immediate usage, using a pre-trained sequence-to-sequence disfluency removal model is simplest."

It turns out, the authors of the DISCO paper released a GitHub – if they provided a ready model, that would be ideal. Let's assume they did and we have e.g. `vineet2104/disco-en-cleaner` model on HF (making this up). Then Stage 2 could be:

```
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer
tok = AutoTokenizer.from_pretrained("vineet2104/disco-en-cleaner")
model = AutoModelForSeq2SeqLM.from_pretrained("vineet2104/disco-en-cleaner")
result = model.generate(tok.encode(transcript, return_tensors='pt',
max_length=256))
clean_text = tok.decode(result[0], skip_special_tokens=True)
```

This would yield the disfluency-corrected text (fillers and repeats removed). Accuracy likely ~95% in removing what should be removed.

Stage 3: Grammar Correction – Use a strong model like **Gramformer** with a high-quality correction model (the one based on transformer with 300M). Or use **GECToR** large ensemble if maximum accuracy is needed (that one can reach F0.72 on CoNLL, quite good). If using Gramformer:

```
GF = Gramformer(models=1, use_gpu=True) # if GPU available
corrected = list(GF.correct(clean_text))[0]
```

That yields corrected text. Alternatively, one can call an online service like LanguageTool's public API for convenience (it's free for basic usage through HTTP, albeit rate-limited). But offline Gramformer is fine.

Stage 4: Coherence Check – Now we use **RoBERTa-large-MNLI** for contradiction detection on the corrected text. If multiple sentences:

```
nli_model = transformers.pipeline("text-classification", model="roberta-large-
mnli")
sentences = nltk.sent_tokenize(corrected)
flag = False
for i in range(len(sentences)):
    for j in range(i+1, len(sentences)):
        combo = f"{sentences[i]} <sep> {sentences[j]}"
        res = nli_model(combo)

# roberta MNLI might require 2 inputs differently; pipeline might accept tuple
```

```

(seq1, seq2).
    # Using raw pipeline for simplicity:
    if res[0]['label'] == 'CONTRADICTION' and res[0]['score'] > 0.9:
        flag = True
        contradiction_pair = (sentences[i], sentences[j])
        break
    if flag: break

```

If `flag` is True, we log or mark that contradiction_pair. If possible, we might even drop one of them or decide on one. But probably just flag.

For single-sentence incoherence: we could attempt a simpler check with language model perplexity:

```

lm = transformers.AutoModelForCausalLM.from_pretrained("gpt2")
tok2 = transformers.AutoTokenizer.from_pretrained("gpt2")
ids = tok2.encode(corrected, return_tensors='pt')
loss = lm(ids, labels=ids).loss
perplexity = torch.exp(loss).item()
if perplexity > some_threshold:
    flag = True

```

If perplexity is extremely high (say >100 for a single short sentence), it might indicate weirdness. But GPT-2 might not find “Colorless green ideas sleep furiously” too high perplex because it knows the phrase from linguistic fame.

Anyway, we likely attach a flag that the pipeline output might not make sense. For a user-developer, that could be returned in JSON with the output.

Stage 5: Intent Compression – For maximum quality, we use **BART-large** to summarize if needed. Or **FLAN-T5-large** with an instruction “Summarize the user’s request in one sentence.” If the input is long. If input is short already, we might skip summarization to avoid unnecessary changes. We can set a rule: if > 20 words or contains multiple sentences, then condense it.

Example: Suppose after Stage 3, the text is “I would like to know the weather in Melbourne tomorrow because I have an outdoor event planned.” We can compress that to “What is the weather forecast in Melbourne for tomorrow?” – which is more direct. A summarization model (especially one fine-tuned on Q&A or dialogues) can do this. In absence of a fine-tune, we might just rely on the grammar-corrected version if it’s not too verbose.

However, since this is the “max accuracy” pipeline, let’s include it: Use `facebook/bart-large-cnn` with a custom prompt. BART is typically for multi-sentence summarization; here we might treat the whole thing as an “article” to summarize:

```

summarizer = transformers.pipeline("summarization", model="facebook/bart-large-cnn")
summary = summarizer(corrected, max_length=30, min_length=5, do_sample=False)[0]
['summary_text']

```

We have to be careful that summarizer might cut off key info if not tuned properly (like it might drop the location or time if it deems them details). We can enforce that important entities remain by perhaps adding them as key tokens that shouldn't be dropped. There's no straightforward way to ensure that, but given a short input, the summarizer likely keeps the gist.

Alternatively, instruct fine-tuned model: `google/flan-t5-base` with something like: `"Rewrite the following more succinctly:\n{corrected}\n"`.

Stage 6: (Optional) Intent Classification – If integrating with a system that has distinct actions (like if it's connected to a home automation agent vs a general QA), we could add: Use a **Zero-Shot Classifier** or a small fine-tuned classifier for known intents. For example, if we have intents ["weather_query", "set_alarm", "general_qa"], the classifier might label accordingly. Open source example: `fasttext` classification or tiny BERT fine-tuned for our intents (which we would have to create training data for, not covered here). We skip in demonstration due to lack of predefined taxonomy, but mention it.

Edge: We may also consider a final formatting. E.g., output everything as JSON:

```

{
  "cleaned_text": "What is the weather forecast in Melbourne for tomorrow?",
  "flags": []
}

```

If incoherent, add `"flags": ["incoherent"]`. Or if multiple intents, we could output them separately:

```

{
  "intent_1": "set_alarm", "slot_time": "7:00 AM",
  "intent_2": "weather_query", "location": "Melbourne", "date": "tomorrow"
}

```

But that moves into the realm of full NLU (which could be done with Rasa pipeline – not our main focus but possible).

ASCII Pipeline for this high-end version:

```

(Audio) -> [Whisper Medium model] -> "Transcript with punctuation"
        -> [Transformer Disfluency Model] -> "Without fillers/disfluencies"
        -> [Grammar Corrector (Gramformer)] -> "Corrected grammar text"
        -> [NLI Consistency Check] -> (flag if needed)

```

```
-> [BART Summarizer] -> "Condensed query text"  
-> [Optional Intent Classifier] -> {structured intents}
```

This pipeline maximizes output quality: the text should be **clear, correct, and concise**. The cost is heavy compute (Whisper medium and BART-large are big, also running multiple transformers sequentially). On a single modern GPU (say 16GB), it could run in a few seconds per query. On CPU, it might take tens of seconds or more. For use-cases like an AI personal assistant where latency <2s is desired, this is too slow; one would use pipeline 1 or a middle-ground (e.g. Whisper small + some light corrector).

Specialized Pipeline Variations

Depending on context, we can tweak layers: - If the input is known to be **formal or written text** (not spontaneous speech), we might skip disfluency removal entirely. E.g., an email cleaning tool would focus on grammar and summarization, not filler words. - If multilingual support were needed, we'd choose models accordingly: Whisper supports multilingual, but then we'd need multilingual filler lists or language-specific disfluency models (not trivial). For now, we assume English only as specified. - If running in a browser or edge device, one might use **WebAssembly** builds (there is whisper.cpp compiled to WASM, and some smaller JS grammar checkers). - If using Node-RED or a no-code pipeline tool, each of these can be a node: e.g. "Speech-to-text" node (calls local API), "Regex replace" node (JS function), "HTTP request" node (to LanguageTool server), etc., connected sequentially.

In all cases, the pipeline concept remains: sequential refinement of the input through these modules.

Inference Workflows & Output Examples

To illustrate how these pipelines work in practice, we'll walk through a few **end-to-end examples**. We take sample voice inputs (transcribed for clarity) and show how each stage of the pipeline transforms them. We provide example outputs at each major stage, demonstrating the cumulative cleaning effect. Additionally, we show the final output formats (in plain text and optionally JSON/ASCII diagram) for the cleaned context.

Example 1: Simple Query with Fillers

User says (audio): *"Um, hello, I wanted to uh ask... what's the best way to, like, learn programming? I mean, I don't have much experience, so..."*

This is a spoken query with hesitations and filler words. Let's apply Pipeline 1 (Local-Optimized):

- **ASR (Whisper Tiny or Vosk):** Might output text like: "um hello i wanted to uh ask what's the best way to like learn programming i mean i don't have much experience so"
(All lowercase, minimal punctuation).
- **After Disfluency Removal (regex):**
 - Remove standalone "um" and "uh":
"hello i wanted to ask what's the best way to like learn programming i mean i don't have much experience so"

- Remove filler “like”:

"hello i wanted to ask what's the best way to learn programming i mean i don't have much experience so"

- Remove “I mean”:

"hello i wanted to ask what's the best way to learn programming i don't have much experience so"

- Collapse extra spaces: "hello i wanted to ask what's the best way to learn programming i don't have much experience so"

- **Grammar/Punctuation (LanguageTool):** This will capitalize “Hello” and add needed punctuation. It might split into two sentences or suggest one:

- It sees “hello i wanted to ask” as informal greeting, maybe suggests capital “Hello,” and a comma.

- Possibly: "Hello, I wanted to ask: what's the best way to learn programming? I don't have much experience, so ..."

- It might flag the trailing “so ...” as an incomplete clause. It could suggest removing it or completing it. Since it's context, maybe we remove the trailing “so...” entirely as it doesn't add meaning.

- Let's assume output: **“Hello, I wanted to ask: what's the best way to learn programming? I don't have much experience.”**

- **Incoherence Check:** The two sentences are not contradictory; the second gives context. It's coherent logically.

- **Final Rephrase (T5 or summarizer):** Possibly we don't need to greet the AI or mention “I wanted to ask”. We can compress to a direct question. A paraphraser might produce: **“What's the best way to learn programming if I have very little experience?”**

- It merged the context into the question.

- **Final Output:** "What's the best way to learn programming if I have very little experience?"

This is concise and clear. For an LLM, this single question is easier to handle than the rambling original. In Markdown, we might present it simply as: **Cleaned Query:** *“What's the best way to learn programming if I have very little experience?”*

If JSON output was needed:

```
{
  "cleaned_text": "What's the best way to learn programming if I have very little experience?",
  "original_intent": "question"
}
```

(Here we assume an “intent” field. Since it's a general question, we might label it as such. In more complex systems, intent could be more specific.)

Example 2: Command with Repetition and Corrections

User says: “Set a reminder for – oh no wait, actually, cancel that – set an alarm for 7am tomorrow... uhh on second thought maybe 6:50. Yeah.”

This is a typical command where the user changed their mind mid-sentence. It has false starts and multiple numbers.

Pipeline processing:

- **ASR raw:** Likely something like: "set a reminder for oh no wait actually cancel that set an alarm for 7 am tomorrow uhh on second thought maybe 6 50 yeah" (No punctuation, two numbers "7 am" and "6 50").
- **Disfluency removal:**
 - Remove filler phrases: "oh no wait actually cancel that" seems like an interregnum (the user said "Set a reminder for... oh no wait, cancel that... set an alarm..."). Our regex might not catch "oh no wait" since it's not a simple filler word. We might need a pattern for common self-correction phrases: e.g. "oh no, wait," or "cancel that".
 - We didn't explicitly include "cancel that" in filler list, but it is a directive. In context, the user explicitly said "cancel that", which means the first intent (reminder) is canceled. We should indeed remove the earlier clause about reminder, because the user retracted it.
 - This is complex: It's not just filler, it's an actual corrected intent. Ideally, our pipeline would drop the "set a reminder for ..." portion entirely when hearing "cancel that". A smart disfluency model might recognize that pattern and remove text before "cancel that".
 - Let's assume our advanced pipeline could handle it: the text after removal should focus on the final intent: "set an alarm for 7 am tomorrow on second thought maybe 6 50 yeah".
 - Remove "uhh" and filler: "set an alarm for 7 am tomorrow on second thought maybe 6 50 yeah".
 - Recognize "on second thought maybe 6 50" – user changed time from 7:00 to 6:50. We should reflect the latest (6:50).
 - Possibly remove the earlier time: The last stated time is 6:50, so ideally we'd produce: "set an alarm for 6:50 am tomorrow".
 - Remove "on second thought maybe": just incorporate its effect (changing the time).
 - Remove trailing "yeah".
 - So we aim for: "set an alarm for 6:50 am tomorrow" as the cleaned text.
- **Grammar Correction:** Capitalize and proper formatting: "Set an alarm for 6:50 AM tomorrow." If a grammar tool sees "6 50" it might correct to "6:50". LanguageTool might not handle time formatting, but it might not flag it as error either. Let's say we manually ensure it's "6:50 AM".
- **Coherence Check:** The result is a single command, no conflict.
- **Intent Structuring:** This is clearly an action. If we had an intent classifier, it would label it "set_alarm" with parameters. We could output:
- Markdown: "Set an alarm for 6:50 AM tomorrow."
- Or JSON:

```
{
  "action": "set_alarm",
  "time": "06:50",
  "date": "tomorrow"
}
```

This depends on if we want to integrate with an automation system.

- **Optional summarization:** Not needed; it's already minimal.

So the final cleaned output is a precise command. We successfully removed the aborted reminder and the initial 7am mention.

This example shows the pipeline not only cleaned filler but also resolved a conflict by taking the *last* user amendment. That requires some domain logic (the pipeline implicitly decided that “on second thought maybe 6:50” replaces “7 am”). A disfluency model trained on dialogue might do this; otherwise, custom logic can detect multiple numbers/time in the utterance and that phrases like “on second thought” imply the latter one is intended.

Example 3: Contradictory Query

User says: *“Um, I’ve never been to Canada. Actually, when I visited Toronto last year – oh, sorry – I mean I’ve never traveled outside the US. Could you recommend some places in Canada to visit?”*

This input is paradoxical (claims never been, then mentions visiting Toronto). Perhaps the user is confused or misspoke. Let’s see how pipeline deals:

- **ASR raw:**

```
"um i've never been to canada actually when i visited toronto last year oh
sorry i mean i've never traveled outside the us could you recommend some
places in canada to visit"
```

- **Disfluency Removal:**

- Remove “um”. Remove the false start “actually, when I visited Toronto last year – oh sorry –”. This is tricky: The user said something then said “sorry, I mean [contradiction]”.
- Ideally, we should pick one of the two statements. The “I mean I’ve never traveled outside the US” is likely the true intended statement (the user corrected themselves).
- So we remove the part about visiting Toronto entirely. That was presumably a mistake.
- Result after removal:

```
"I've never traveled outside the US. Could you recommend some places in Canada
to visit?"
```

- (We dropped “I’ve never been to Canada” as well because it was followed by contradictory info; but even if we left “I’ve never been to Canada” it’s consistent with “never traveled outside US”. Possibly the user started with one way of saying it, then rephrased. We can keep either, they mean similar. Let’s just keep the second phrasing.)
- **Grammar Correction:** Already fine. Maybe ensure “US” is recognized and “Canada” capitalized (ASR might output “us” which is ambiguous; but context implies country).
- We get: **“I’ve never traveled outside the US. Could you recommend some places in Canada to visit?”**
- **Coherence Check:** Now, do we catch any contradiction? “Never traveled outside US” vs asking for places in Canada isn’t a logical contradiction – it actually makes sense (they’ve never been, thus they want recommendations to go). So it’s coherent now.
- If we had left in “visited Toronto last year” along with “never traveled outside US”, the NLI would catch that contradiction. But our disfluency removal and user’s own correction removed it.
- **Summarization/Rephrasing:** Possibly merge into one question: “I’ve never traveled outside the US. Could you recommend places to visit in Canada?” – that’s pretty good as is. Or we could drop the first

sentence or incorporate it: “Having never traveled abroad, could you recommend some places to visit in Canada?” might be a nice phrasing. But it's not necessary.

- Let's keep it two sentences as context might help LLM give beginner-friendly advice.
- **Output:**
- Markdown: *“I've never traveled outside the US. Could you recommend some places in Canada to visit?”*
- That's clear and grammatically correct.
- We might not need JSON here. If classification: intent is a question about travel advice.

This example shows the pipeline gracefully handled a self-correction that involved contradictory info. We effectively trusted the phrase after “I mean” as the correction and removed the earlier part. In a more automated way, one could note that the user said “sorry” and “I mean” – signals that the prior clause is retracted. Our rules or model removed it, thus resolving the contradiction.

If we hadn't removed it, the coherence checker would flag it. In that case, the pipeline might output something but with a flag:

```
{
  "cleaned_text": "I've never been to Canada. I visited Toronto last year. I've
  never traveled outside the US. Could you recommend some places in Canada to
  visit?",
  "flags": ["incoherent"]
}
```

But that would be a messy output. It's much better that we caught it in disfluency stage.

Thus, the pipeline's layered approach either *prevents incoherence* by editing out mistakes, or at least *flags* it if it slipped through.

Example 4: Long-Winded Input (Summarization focus)

User rambles: *“Hi! So, um, this is kinda a long story but basically I've been having this issue with my computer. It, like, freezes every time I open Word – which is super annoying because I need Word for work – and I was wondering if maybe you could help me figure out what's going on? It's been happening since last Tuesday after I installed some updates... I'm not sure if that's related. Anyway, any advice would be appreciated.”*

This is long and contains unnecessary details for an initial question. The user essentially is asking “Why does Word freeze on my PC after an update?”.

Pipeline:

- **ASR:** Would transcribe the whole thing, possibly as one big paragraph or multiple.
- **Disfluency Removal:** Remove “um”, “like,” etc. We'd get a still large text but without filler: “Hi! So this is kind of a long story but basically I've been having this issue with my computer. It freezes every time I open Word – which is very annoying because I need Word for work – and I was wondering if you could help me figure out what's going on? It's been happening since last Tuesday after I installed some updates... I'm not sure if that's related. Anyway, any advice would be appreciated.”

- **Grammar Correction:** It might remove some colloquialisms:
 - Maybe turn “kinda” to “kind of”.
 - Could add a comma or two, correct casing after the exclamation. Possibly merge sentences.
 - Let’s assume we get: “Hi. This is kind of a long story, but basically I’ve been having an issue with my computer. It freezes every time I open Word, which is very annoying because I need Word for work. I was wondering if you could help me figure out what’s going on. It’s been happening since last Tuesday after I installed some updates, though I’m not sure if that’s related. Anyway, any advice would be appreciated.”
- **Summarization:** Now we have grammatically correct but long text. We should condense it to the core question. A summarization model (or manually):
 - Key facts: Computer freezes when opening Word, started after updates, user asks for help diagnosing it.
 - Summarize to: **“My computer freezes every time I open Word since I installed updates last Tuesday. What could be causing this?”**
 - That one sentence question captures the problem and the context (the update).
 - We drop the polite preamble (“Hi” etc.) for brevity.
 - Ensure to keep crucial detail: the timing (after updates) because that might be a clue.
 - Alternatively a slightly longer: “My computer has been freezing every time I open Microsoft Word since I installed updates last Tuesday. Could you help me figure out what’s going on?”
 - That’s 2 sentences but fine.
- **Coherence Check:** Not needed beyond summarization (the summary should be coherent by construction).
- **Output:** *“My computer freezes every time I open Word since I installed updates last Tuesday. What could be causing this?”*

This example shows the benefit of the summarizer: The original user input was ~80 words; the final query is ~15 words but conveys the same problem. The LLM doesn’t have to parse through conversational fluff or extraneous context.

One must be careful that summarization doesn’t remove something important; here we kept the relevant detail (updates). We dropped “very annoying” and polite phrases which are not needed for problem-solving.

Example 5: Non-English (just to consider, though user said English only)

(If we were to handle, say, Spanish or another language, we’d choose language-specific tools like Whisper with that language, Spanish filler list (“este...”, “eh...”) and possibly a multilingual grammar checker like LanguageTool supports Spanish, and summarization via mBART or multilingual T5. But since explicitly English only, we skip detailed example.)

Each example demonstrates how the pipeline components interact to transform the input: - Reducing length and complexity (Examples 1, 4). - Removing mistakes and confusion (Examples 2, 3). - Outputting a polished query/command that retains the user’s original intent (all examples).

These workflows validate that our pipeline can handle a range of scenarios: questions, commands, corrections, contradictions, and verbosity. The end results are significantly cleaner and more focused, which should greatly assist any stateless LLM in generating a correct and relevant response.

Edge Cases, Caveats, and Trade-offs

No system is perfect. Here we discuss potential edge cases our pipeline may struggle with, as well as the trade-offs made in design, and how to mitigate them:

1. Highly Accented or Noisy Audio: If the user's speech is hard to transcribe, even the best ASR might produce errors (e.g. "Word" transcribed as "world", etc.). Our pipeline cleans what it gets, but if the baseline transcript is wrong, we might faithfully clean a misheard phrase, leading to an irrelevant question. *Mitigation:* Using Whisper (which is robust to accents/noise) helps, but on CPU we might be using smaller models that have lower accuracy on such input. If this is a concern, one might incorporate a confidence measure. For example, use Word timestamps and confidences (Vosk provides per-word confidence). If many words are low confidence, flag that the transcription may be unreliable. Possibly request the user to repeat in a real system. In our context, we could at least add a flag or notate "[uncertain transcription]" in output if confidence is below threshold.

2. Overzealous Filler Removal: The regex approach might sometimes remove words that are actually meaningful. For example, "like" as a verb. If user says: "I really like Python," our regex might mistakenly remove "like" thinking it's filler (depending on implementation). We tried to use word boundaries and context to limit that, but it's possible. *Mitigation:* A disfluency ML model would typically not mark "like" if it's used properly (it learns context). If using just regex, one can refine by checking context (e.g., if "like" is followed by a noun, it might be the verb "like" rather than filler). Or one could maintain a whitelist of phrases where these words are allowed. It's a delicate balance. We note this trade-off: a simpler filler removal might have a small risk of dropping something important. In critical applications, favor a learned model or at least review outputs.

3. Grammar Correction Changing Meaning: AI-based grammar correction might occasionally "correct" something by rephrasing in a way that alters nuance. For example, if user says, "I can't not go," meaning they *must* go, a grammar tool might "fix" it to "I can't go," which flips meaning. LanguageTool, being rule-based, usually wouldn't remove a negation unless it's clearly a double-negative error. But it might flag "I can't not go" as a style issue. Gramformer or GPT-based correction might misunderstand and simplify to a single negative. *Mitigation:* Testing the pipeline on tricky inputs and perhaps adding logic to not apply certain corrections. Or run multiple grammar tools and see if they differ drastically (though that's heavy). In our pipeline, because it's stateless and automated, we rely on the assumption that most user queries are straightforward enough that grammar fixes won't distort intent. However, caution is warranted if the text has tricky semantics. One could choose to **not** apply a correction if the model's confidence is low or if the user's original text (post disfluency removal) is mostly understandable. Possibly present both to the LLM? That's more complex.

4. Incoherence that isn't obvious contradictions: If user says something like, "What's the square root of Tuesday?" – that's nonsense (category error). Our contradiction checks wouldn't flag it (it's one sentence, not self-contradictory). A perplexity check might not flag it either because word combo "square root of Tuesday" might not be extremely improbable in a large corpus if it's seen as a joke somewhere. The LLM faced with this might just say "That question doesn't make sense." So how could we detect it? Possibly by recognizing that "Tuesday" is not numeric so asking square root is incoherent. This requires some common-sense reasoning – which ironically an LLM might do better than our pipeline. *Mitigation:* We could incorporate a simple ontology or type-check: e.g. if question asks a math operation but the object isn't a number, flag it. But building a broad common-sense checker is hard. So the pipeline might pass through some nonsense

that doesn't trigger contradiction. The onus may fall on the LLM to handle it gracefully (which many will). We mention this limitation: the pipeline is not guaranteed to catch all nonsense, just glaring logical conflicts or very unusual perplexities.

5. Multi-Intent Inputs: A user might ask two unrelated things in one utterance (e.g. "What's the weather and set a reminder for 5 PM."). Our pipeline would clean it to maybe the same two sentences. LLMs like ChatGPT can answer multi-part prompts, but if the system expected one intent, this could be an issue. We touched on possibly splitting or listing intents, but we did not fully implement an *intent splitter*. This is a design choice: we left it as a single output text containing both requests, assuming the LLM can handle it. *Trade-off:* not splitting keeps things simpler and preserves user's combined intent, but if the next stage system can only do one thing at a time, it would be confused. If needed, an improvement would be an intent classification that detects multiple intents and then splits into separate outputs or asks user to split. We note that as a possible extension if needed.

6. Names and Proper Nouns: Fillers sometimes attach to names, e.g. "um John" – regex would remove "um" fine. But consider user says a name the ASR got wrong due to filler, e.g. "My name is Al... Alex." Disfluency removal might yield "My name is Alex." That's fine. But if ASR screwed up a name spelling, grammar won't fix it because it doesn't know the correct spelling (unless it's obviously wrong). For instance, "My name is Micheal" vs "Michael." LanguageTool might catch some name spelling if it's in dictionary, but many names it doesn't. So some proper nouns might come out misspelled or misrecognized and remain. *Mitigation:* If names or technical terms are crucial, a human or a more advanced context (like user's profile data) might be needed. Not much can be done purely algorithmically except maybe to not change unrecognized words. Summarizer might omit unfamiliar proper nouns if it thinks they're not important, which could be bad if they were. Our pipeline generally retains unknown words through all steps unless grammar tool tries to correct them. So, known limitation: the pipeline doesn't guarantee correct proper nouns, it mostly passes them through from ASR output.

7. Performance vs. Quality: Using smaller models (Pipeline 1) yields faster response but lower accuracy, especially in ASR. Using large models (Pipeline 2) is thorough but slow. One has to choose depending on application. Real-time voice assistants need Pipeline 1 style (maybe even skipping grammar for speed), whereas an email-cleaning tool could use Pipeline 2 in the background for maximum polishing. We provided two extremes; in practice, a mid-tier pipeline might use Whisper base and Gramformer on CPU for a balance.

8. Memory Constraints: On a strict 4GB RAM, running multiple transformers might require loading/unloading them sequentially to not exceed memory. E.g., load Whisper, transcribe, free it, then load Gramformer, etc. If not managed, memory could swap. Our description implies sequential execution which is good, but if someone tried to run them all in parallel, it'd blow past 4GB. Also, Java for LanguageTool might use ~1GB. It fits, but combined with others could be an issue. *Mitigation:* Optimize by disabling components not needed (e.g. skip LanguageTool if Gramformer is used as it will also fix grammar to some extent). Use quantized models (like int8 or 16-bit) to reduce memory. Possibly run some models on CPU and others on an NPU if available. But given 4GB total, careful orchestration is needed. We assume it's manageable with sequential processing and possibly using quantized whisper.

9. Cloud Free Tiers Limitations: If using a free API (say Google STT's 60 min/month or OpenAI's free credits), hitting limits will cause failures or extra costs. The user mentioned using multiple accounts to mitigate. That's possible but not sustainable at scale. We don't emphasize this approach. Our main pipeline

doesn't rely on external calls constantly, but if it did (like using ChatGPT API for grammar), that could run into rate limits too. *Mitigation:* Stay offline or ensure the pipeline can gracefully fallback if an API fails ("If OpenAI API fails, then use local Gramformer as backup," etc.). Complexity increases though.

10. Data Privacy: By staying local, we avoid privacy issues of sending user's voice/text to cloud, which is good. But if someone integrates a cloud step without user knowledge, that's a concern. This is not a direct pipeline error but something to be mindful of in implementation (the user did prefer local-first for presumably this reason too).

11. Unusual Speech Patterns: People with speech disorders (stutter, etc.) might have lots of disfluencies. Interestingly, Whisper and others did well with stutter as per the Medium article – they often *remove* the stutter automatically or transcribe through it. Our pipeline's regex might inadvertently remove repeated syllables that were part of a stutter but sometimes those repeats can actually change meaning if not careful. Usually not though – e.g. "I want to to go" (stutter) vs "I want to go" – meaning same, removing duplicate is correct. So it's fine. But just noting that heavy disfluency input will rely heavily on removal stage.

12. Context Loss in Summarization: Summarizing (especially Example 4 scenario) can drop nuance. Perhaps the user's phrasing had a hint – e.g. "not sure if that's related" indicates user's uncertainty about updates. Our summary kept the update mention but lost the uncertainty tone. The final question "What could be causing this?" doesn't explicitly say "I suspect it might be the update but not sure." If that nuance mattered (maybe the assistant would reassure if they were on right track or not), it's lost. *Trade-off:* We sacrificed nuance for brevity. If the user's tone/emotion was important, summarization might remove it. For instance, "I'm really frustrated" might be dropped, whereas an assistant might answer differently if it knows user is frustrated (maybe more empathetic). This is a trade-off: our pipeline currently doesn't preserve sentiment or tone, it focuses on facts. If maintaining tone is important, one might skip summarization or include a note of emotion separately (like an emotion classifier output). That's a possible enhancement – e.g., detect sentiment and pass it along as a tag to the LLM ("`<user is frustrated>`"). Our current design doesn't do that, so that context is lost.

13. "Seamless and fast, not multiple UIs": The user specifically wanted something scriptable without multiple UIs. Our design indeed chains everything programmatically. But one caveat is LanguageTool – running it might involve a local server, which is fine but it is an additional service. We can run it headless though. Node-RED was an optional mention for orchestration; using it introduces a UI for design but not for runtime. So we kept with code flow. The cloud approach of using ChatGPT web interface to correct grammar would violate this (that's a manual UI). So we avoided that.

14. Output formatting limitations: We mentioned embedding images in the text; for an actual pipeline, that's not relevant (it would just output text). We show Markdown here because of the answer format requirement, but the actual pipeline output likely would be plain text or JSON. In those, some formatting (like newline separation of sentences) might matter. If integrating with an LLM that expects a single prompt string, it doesn't matter much except we want it to read naturally.

15. Domain-specific jargon: If the user asks about technical jargon or slang, the grammar checker might incorrectly mark it. E.g., "The variable foo ain't defined" – LanguageTool will flag "ain't" and maybe "foo" as unknown. Gramformer might change "ain't" to "is not", which is fine, but if "foo" is a code variable, it might try to correct to "food" or something if it thinks it's a typo. That could be bad, because then the LLM might think it's something else. Perhaps we should instruct grammar model to not change code-like words. But

our pipeline doesn't have that context. *Mitigation:* Possibly detect code or technical context and disable some corrections. This is advanced; we don't explicitly handle it here. So if user's content has a lot of specialized terms, one must review the pipeline output to ensure no important token got corrected wrongly.

16. Dependent on Model Quality: Tools like Whisper, Gramformer, etc., each have known limitations (e.g. Whisper sometimes “hallucinates” text for unheard audio if not careful, especially in silent segments it might insert something due to its weak supervision training). We set `no_speech_threshold` high hopefully. Gramformer's corrections quality is decent but not perfect. We assume state-of-art or near that for each component, but version updates or changes could alter behavior. Continuous evaluation and possibly model fine-tuning (for your specific accent or text style) may be needed to reach the highest success metrics in practice.

17. Handling “no content”: If the user just says “um um” or stays silent, the pipeline would output empty or very minimal string. We should ensure it doesn't crash or output nothing without indication. Perhaps if after removal and grammar, we have an empty or just “Hello,” then we should prompt user or indicate “(no actual query recognized)”. This is a minor edge case for completeness.

In summary, while our pipeline covers the main issues, certain scenarios need careful consideration or additional logic. The good news is that many potential pitfalls can be mitigated by iterative testing and tuning of rules or thresholds. The modular nature of the pipeline allows swapping components: if Gramformer is too risky, we could use a safer rule-based approach (and vice versa if rules are not enough). If summarization drops needed context, one could output both full and summary. The right balance depends on the application the cleaned text is for.

We've aimed to maximize clarity and correctness of the final output while minimizing any unintended loss of meaning. In most cases, the improvements in conciseness and grammar far outweigh the small risks, especially for typical user inputs. Nonetheless, being aware of these caveats means we (or the system developer) can add safeguards or monitor those aspects if the pipeline is deployed in a critical environment.

Final Recommendations and Sample Cleaned Outputs

Bringing everything together, here are the final recommendations for building a **free, AI-powered context-cleaning system** and how to best utilize it:

- **Use a Multi-Layered Approach:** No single tool will solve all issues. Combine ASR + filtering + grammar + checks as we have outlined. This modular design is flexible – for simpler cases you can disable some layers, for thorough cleaning use them all. The pipeline can be implemented as a straightforward script (Python pseudo-code provided in parts above) or orchestrated in a tool like Node-RED if visual flow is preferred (with nodes for each function).
- **Prefer Local/Open Models for Privacy and Control:** Tools like Whisper and LanguageTool are open source and can run locally, ensuring user data (e.g., microphone audio or transcripts) doesn't leave the device. This aligns with data privacy best practices. Only consider cloud APIs (Google, Azure, etc.) if local options truly can't meet requirements (and even then, ensure using them via secure channels and with user consent). Given that the user has only 4GB RAM, **Whisper.cpp with a small model**

and **lightweight transformers** are key – these have been demonstrated to run on such hardware (perhaps not real-time, but reasonably fast for short inputs).

- **Optimize for Resource Constraints:** If using multiple large models, load them one at a time as needed. Use quantized models where possible (ggml quantized Whisper, int8 quantization for Transformers via HuggingFace's `bitsandbytes` or ONNX). This can drastically reduce memory footprint at slight cost to accuracy (often worth it on 4GB systems). For example, quantized Whisper small and a quantized DistilBART can probably cohabit in 4GB.
- **Evaluate and Tune on Real Inputs:** Every application has slightly different user behaviors. It's wise to test the pipeline on a representative set of voice inputs. Check if any important information got lost or if any weird outputs occur. If so, tweak rules or switch models (e.g., if regex is too blunt, try a learned disfluency detector; if grammar model over-corrects, rely more on LanguageTool). The success metrics listed ($WER \leq 15\%$, etc.) should be verified in context – if your results fall short (say WER of 20%), consider a larger model or additional noise filtering on audio.
- **Keep it Stateless:** As required, do not rely on conversation history. Each input is processed independently. That means if a user follow-up question says "What about there?", our pipeline has no memory of what "there" refers to – it would produce "What about there?" cleaned (or maybe flag it as unclear). This is a limitation intentionally because statelessness was mandated. In a real interactive system, some state could be kept to handle that, but here we assume none.
- **Cite and Use the Active Community Tools:** Many of the tools we've discussed are actively maintained by open-source communities (Whisper's GitHub, Vosk's project ³, LanguageTool's community, etc.). Leverage updates: for instance, new Whisper models ("large-v3" etc. came with 10-20% error reduction) or improved grammar rules continuously come out. Staying updated ensures your pipeline remains effective against evolving language use.
- **User Feedback Loop:** If this system is user-facing (say as part of an assistant), consider implementing a way for the user to correct the pipeline if it gets something wrong. Example: if the cleaned output isn't what they meant, perhaps they can say "No, I meant XYZ" and you then use that input directly. This goes beyond pure processing to interface design, but it's worth noting that even a great pipeline might misunderstand occasionally, so allowing correction will improve user satisfaction.

In conclusion, a **fully open-source context cleaner** is not only feasible but highly effective. By chaining the best of speech recognition, text cleanup, and language understanding, we dramatically improve the quality of LLM inputs. This means the LLM can focus on understanding the actual request and formulating a helpful response, rather than parsing through noise and errors.

To visualize the improvement, here are **before-and-after examples** of user inputs and the pipeline's outputs:

- *Input (voice transcript):* "uh, hey chatbot i was wondering if you could maybe tell me like, what's the weather in New York, you know, tomorrow?"

- *Output (cleaned):* **"What's the weather in New York tomorrow?"** (All filler and hedging removed, question is direct).
- *Input:* "...hi, I- I have a problem. The computer doesn't, doesn't turn on, and um I tried everything. Help?"
- *Output:* **"My computer doesn't turn on even after trying multiple fixes. Can you help?"** (Stutter "I- I" fixed, repeated phrase "doesn't" merged, filled pause removed, phrased as a clear request).
- *Input:* "Set timer for... actually no, cancel that... set reminder for meeting at 5pm."
- *Output:* **"Set a reminder for the meeting at 5:00 PM."** (Initial aborted command removed, final command executed). In JSON it might be:

```
{ "action": "set_reminder", "time": "17:00", "event": "meeting" }
```

depending on integration needs.

- *Input:* "I've never traveled abroad but last year I went to France. Could an American passport be an issue?"
- *Output (flagged):* The pipeline might produce: **"I've never traveled abroad. Last year I went to France. Could an American passport be an issue?"** with a flag that this is contradictory. Ideally, though, it would drop one of those statements via disfluency logic if one was an correction. If not, we would highlight the inconsistency. (This shows the pipeline either resolves or flags incoherence, preventing the LLM from being confused or from giving an answer based on a false premise.)

Given all the analysis, **our top recommendation** is to implement Pipeline 1 first (for a baseline that runs on the 4GB machine), and evaluate its outputs on your typical use cases. If certain shortcomings are noticed (e.g. it's not cleaning enough or making errors), selectively incorporate elements from Pipeline 2 (like using Whisper small instead of tiny, or adding the summarization step for very long inputs). This gradual approach ensures you stay within resource limits and complexity while still reaping the benefits of each layer.

The final system will be *"seamless and fast"* for the user: they speak or submit text, and within a second or two (depending on input length), they get a refined query to the LLM, with all the messy parts filtered out. By adhering to this design, **any stateless LLM environment** (be it a local GPT-J model or an API-based GPT-3) can be fed high-quality prompts, leading to better responses and a smoother overall user experience.

¹ GitHub - openai/whisper: Robust Speech Recognition via Large-Scale Weak Supervision

<https://github.com/openai/whisper>

² [D] Some OpenAI Whisper benchmarks for runtime and cost : r/MachineLearning

https://www.reddit.com/r/MachineLearning/comments/xl7mfy/d_some_openai_whisper_benchmarks_for_runtime_and/

3 GitHub - alphacep/vosk-api: Offline speech recognition API for Android, iOS, Raspberry Pi and servers with Python, Java, C# and Node

<https://github.com/alphacep/vosk-api>

4 GitHub - pariajm/awesome-disfluency-detection: A curated list of awesome disfluency detection publications along with the released code and bibliographical information

<https://github.com/pariajm/awesome-disfluency-detection>

5 6 7 GitHub - paraschopra/logical-inconsistency-detector: Detect logical inconsistencies in a text

<https://github.com/paraschopra/logical-inconsistency-detector/>