Prototyping Methodology of End-to-End Speech Analytics Software

Oleh Romanovskyi [1], Ievgen Iosifov [1,2], Olena Iosifova [1], Volodymyr Sokolov [2], Pavlo Skladannyi [2] and Igor Sukaylo [2]

[1] *Ender Turing OÜ, ½ Padriku str., Tallinn, 11912, Estonia*
[2] *Borys Grinchenko Kyiv University, 18/2 Bulvarno-Kudriavska str., Kyiv, 04053, Ukraine*

Abstract

This paper presents the prototype of end-to-end speech recognition, storage, and postprocessing tasks to build speech analytics, real-time agent augmentation, and other speech-related products. Moving ASR models from the dev environment into production requires both researcher and architectural knowledge, which slows down and limits the possibility of companies benefiting from speech recognition and NLP advances for fundamental business operations. This paper proposes a fast and flexible prototype that can be easily implemented and used to serve ASR/NLP-trained models to solve business problems. Various software solutions' compatibility problems were solved during the experimental setup assembly, and a working prototype was built and tested. An architectural diagram of the solution was also shown. Performance, limitations, and challenges of implementation are also described.

Keywords [1]

Natural Language Processing, NLP, Automatic Speech Recognition, ASR, speech analytics.

1. Introduction

The rise of speech technologies created a demand for the newest software architectures applicable for real business solutions on a scale. According to Gartner, by 2025, 40% of all world call centers will use speech-to-text technology to handle incoming communication. Automotive giants like Porsche Group, Volkswagen AG, Mercedes-Benz Group, and others invest hundreds of millions of USD in creating new experiences involving voice communication between a driver and their car. Many more industries jump onto natural language voice communication between a human and a machine: gaming, medical devices, industrial machines, and others. This shift in utilizing science in the real world created a demand for higher-level software frameworks with pre-built architectures rather than low-level ASR and TTS frameworks. The new level of frameworks should include parts that ensure easy integration in existing IT infrastructures, fault tolerance, data security, and others. We have built the prototype for a system that handles both recoded voice processing and real-time voice stream handling [1–3].

When it comes to business cases, more is needed to have an overwhelming number of building blocks, which is okay for proof of value. It should be an end-to-end research-friendly solution that is flexible and reliable.

[1]MoMLeT+DS 2022: 4th International Workshop on Modern Machine Learning Technologies and Data Science, November, 25-26, 2022, Leiden-Lviv, The Netherlands-Ukraine
EMAIL: or@enderturing.com (O. Romanovskyi); ei@enderturing.com (I. Iosifov); oi@enderturing.com (O. Iosifova); v.sokolov@kubg.edu.ua (V. Sokolov); p.skladannyi@kubg.edu.ua (P. Skladannyi); i.sukailo.asp@kubg.edu.ua (I. Sukaylo)
ORCID: 0000-0003-3420-5621 (O. Romanovskyi); 0000-0001-6507-0761 (I. Iosifov); 0000-0001-6203-9945 (O. Iosifova); 0000-0002-9349-7946 (V. Sokolov); 0000-0002-7775-6039 (P. Skladannyi); 0000-0003-1608-3149 (I. Sukaylo)

2. Related works

There are many ASR toolkits on the market, like NeMo: a toolkit for building AI applications using Neural Modules [4], ESPNet [5], and Kaldi [6]. The limitation of these frameworks is that they are very focused on ASR-related tasks and are only one building block for business applicability.

NVIDIA put a lot more effort than others and developed the NVIDIA TAO Toolkit [7] and NVIDIA [8], which also contains pre-/post-processing components for speech-related tasks. NVIDIA Triton Inference Server is an open-source inference serving software that simplifies inference serving for an organization by addressing the above complexities. Triton provides a single standardized inference platform which can support running inference on multi-framework models, on both CPU and GPU, and in different deployment environments such as datacenter, cloud, embedded devices, and virtualized environments. Also Triton requires NVIDIA GPU and proprietary (can't be modified) but nice features like pipelines and dynamic batching [9]. Still, there need to be more high-quality implementations of prototypes to build software based on.

Hence, we decided to put effort into building a modular prototype that can be used to develop any business software. At the ASR level, we put our models trained with an automated pipeline [10] which can be trained and served with any of [4–6] toolkits. As postprocessing, we have put punctuation [11] and emotion recognition [12].

3. Prototyping

Prototyping stage includes major uncertainties in both technical implementation and client's business requirements. At the same time, we need to evolve it into reliable and scalable solution to recognize, store, and post-process speech-related tasks, without rewriting it from scratch. Thus we decided to focus on three main areas:

- Loose coupling. We should be able to replace each functional block either without changes to the rest of the system, or with minimal changes. For example, using a different tool or a different service for speech recognition should not require backend or UI changes.
- Pipeline flexibility. During prototyping we don't know which processing steps customer needs (like punctuation or emotion detection). So processing pipeline should be easy to add/remove steps or to change execution graph.
- Interoperability: the solution should be easy to integrate with existing enterprise software (like CRM, Customer Support system, etc.).

3.1. Defining loosely-coupled components

We've defined the following requirements to achieve loose coupling:

- Data formats and protocols between components are independent of data-structures of underlying frameworks
- Asynchronous processing for non-interactive tasks
- Being able to deploy and update components independently

We also needed to select protocols and frameworks to receive and process audio for the prototyping.

After evaluation, we decided that to satisfy pipeline scalability requirements, we will implement the solution in microservice architecture [13–15] to add new services. We also decided to divide ASR and postprocessing tasks for real-time processing; there is no need for most postprocessing time, and any additional time consumption is critical for real-time tasks [16].

3.2. Protocol selection

One of the decision for prototyping was protocols selection. Criteria for protocols was: easy to implement and use on different programming languages; generic, so we can implement adapters for

other protocols later as separate services selection to accept the audio signal and provide results. We evaluated and compared real-time and non-real-time approaches and protocols to build prototypes in Table 1.

Table 1
Real-time and non-real-time comparison

Connection type	Complexity for prototyping	Applicable cases range	Computational requirements
Real-time	Hard	Low	High
Non-real-time	Medium	High	Medium

After evaluation, we decided to approach prototyping in two steps, starting from non-real-time because of its easiness of prototype and broad application, and after that, enlarging the prototype to Version 2 with the support of real-time audio processing.

3.3. Non-real-time protocol selection

For non-real-time, we decided to implement the asynchronous method using REST API. This will be the most flexible approach to building complex integrations with enterprise systems like Customer Relationship Management, Business Intelligence, email and chat system, etc.

REpresentational State Transfer (REST) is a software architectural style that describes a uniform interface in a client-server architecture [17]. An Application Programming Interface (API) that complies with some or all of the six guiding constraints of REST is considered to be RESTful [18]. An API establishes a connection between programs so they can transfer data.

A program with an API implies that some parts of its data are exposed for the client to use. The client could be the front end of the same program or an external program.

3.4. Real-time protocol selection

The decision was more complicated for real-time processing because various methods exist, like Websocket, MQTT over WebSocket, gRPC, etc. We decided to go with the most widely used and simplest one—WebSocket for the real-time prototype [16, 17].

WebSocket is a computer communications protocol providing full-duplex communication channels over a single TCP connection. The main advantage of WebSocket is simplicity and prevalence [18].

Real-time alternatives are GRPC or MQTT.

3.5. Server-side frameworks selection

As the main idea behind it is to build flexible ready in terms of integration prototype, was to prepare prototype as natural for researchers as possible, that is why we selected Python as the main language because of popularity in research community. After evaluating REST API Python frameworks (Django, Flask, FastAPI) [19], we decided to go with FastAPI running under Uvicorn as one of the fastest Python frameworks available. FastAPI is a modern, fast (high-performance) web framework for building APIs [20].

Their API-first approach will best integrate future solutions with any enterprise solutions.

3.6. Speech recognition serving framework

Server-side framework decision not only affects how we serve speech recognition tasks but also creates limitations that speech recognition models should be trained using the same framework. Hence decision on the server side Speech recognition framework was the most important during prototyping. We were chosen between two commonly used frameworks: Kaldi and Nemo.

3.6.1. Nemo framework

NVIDIA Nemo [4] is a conversational AI toolkit built for researchers working on automatic speech recognition (ASR) [2, 3, 21–23], natural language processing (NLP), and text-to-speech synthesis (TTS) [24], freely available under the Apache License v2.0.

The main advantages:

- Comparatively, easy model finetuning.
- No need for grapheme to phoneme models.
- Ready to go server-side framework to serve models.

3.6.2. Kaldi framework

Kaldi [6] is an open-source speech recognition toolkit written in C++ for speech recognition and signal processing, freely available under the Apache License v2.0.

The main advantages:

- The Acoustic Model (AM) is not biased for the Language Model (LM) task (AM of end-to-end frameworks absorbs some part of AM inside AM).
- Fewer input audios need to train the same accuracy model.

After careful consideration, we have decided to build a prototype with the Nemo framework as it already has a server side.

4. Implementation

After consideration, we split the prototype into two versions. The first version should be simple and fast, supporting only non-real-time audio processing; because it is more stable, tasks can be repeated, postponed, and use lower resources. For the second version of the prototype, we decided to dedicate to real-time processing, indexing, and architecture comprehension to be more production ready.

To get production-ready results, we need to guarantee stability and provide end-to-end results, which means that we need not only to recognize speech but also separate speakers for mono audio where speakers are in one channel and apply punctuation and inverse text normalization to convert words into easily readable numbers and signs (e.g., @, #, etc.).

Almost all postprocessing tasks should be done after speech recognition results are obtained. Hence, we implemented a pipeline.

1. Speech recognition.
2. Diarization (mono-to-stereo).
3. Normalization (word-to-number conversion).
4. Punctuation.

4.1. First version of prototype

For the first version of the prototype, we decided to go with the minor functionality to check end-to-end work. We designed a straightforward algorithm that takes the file into processing and returns the future ID of the result immediately. Hence, after some time using the GET method, the customer can get results. The main limitation of this algorithm is that users do not know when exactly the results will be ready and may need to check a few times and get just the ‘IN PROGRESS’ status. The algorithm for prototype Version 1 is shown in Fig. 1.

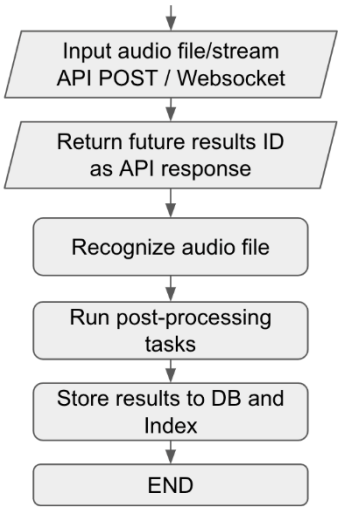

Figure 1: Prototype algorithm (Version 1)

To implement offered algorithm, we develop the most straightforward architecture shown in Fig. 2.

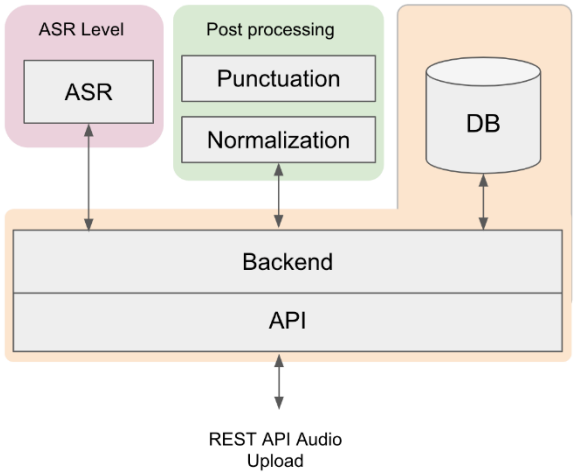

Figure 2: Prototype structure (Version 1)

The main building blocks of Version 1 architecture is:

1. ASR level: to accept audio file/stream and return raw text results.
2. Post-processing level: to implement additional tasks based on recognized text, e.g., Punctuation, Classification, Diarization, etc.
3. Storage level: where a database can store all results to serve them back to the user on requests.
4. API level: serving as the main gateway interface for customers to send audio files to and get results back. It is also a logic center to decide which steps to take and store results in the database at the end of file processing.

4.2. Second version of prototype

After evaluation of Version 1, which is described in detail in the results part, it was clear that we need to implement queue logic to build the Version 2 prototype because without overcoming of blocking of BE/API, it is clear that we cannot serve real-time requests.

As a result, we develop a prototype Version 2 algorithm shown in Fig. 3.

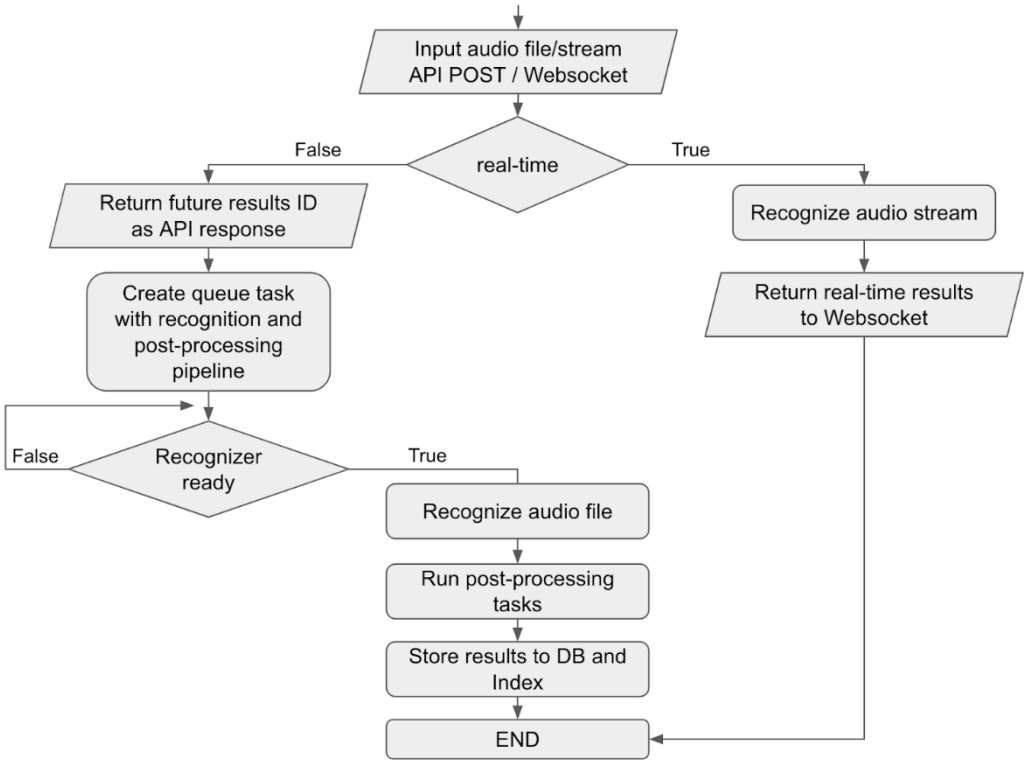

Figure 3: Prototype algorithm (Version 2)

To implement indicated algorithm, we designed the architecture shown in Fig. 4.

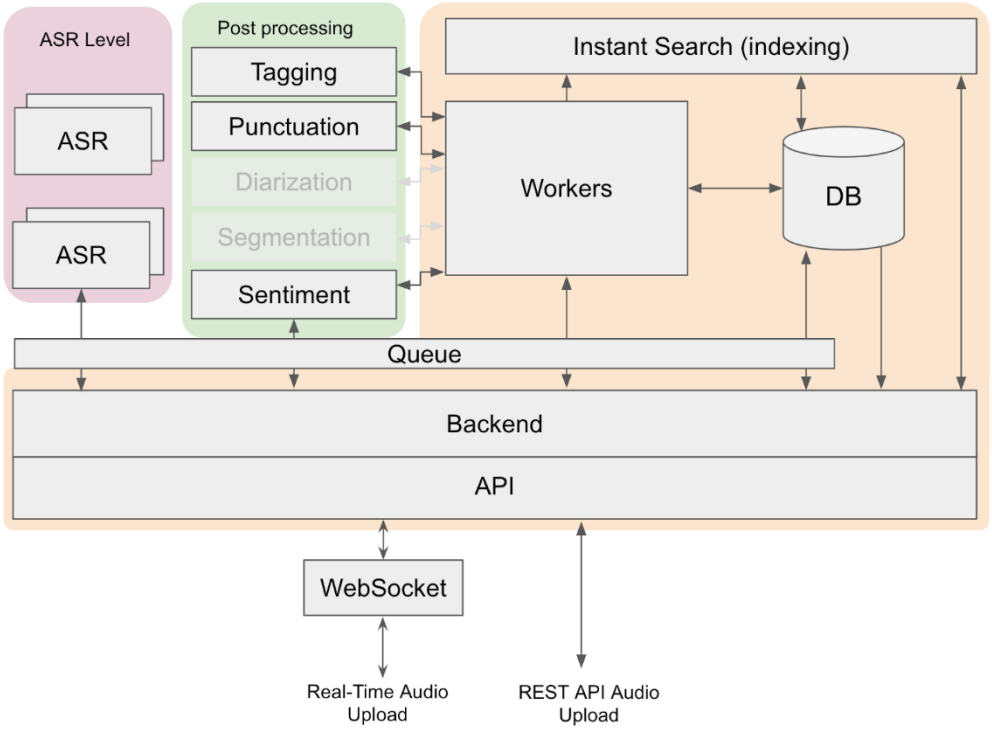

Figure 4: Prototype structure (Version 2)

The main building blocks of Version 2 architecture is:

1. All features from Version 1.
2. Queue level: where we can put non-real-time tasks for future processing when the CPU is ready and be sure the task will wait until processing.

3. Indexing level: for search functionality.

Code example of creating a basic API endpoint to accept audio file or streams can be found below and results:

```
@router.post("/audio-file")
def upload_audio_file(
  *,
  file: UploadFile = File(...),
  params: schemas.RecognizerEndpointParams = Depends(endpoint_parameters),
) -> Any:
  session_id = uuid.uuid4()
  asr_instances = get_asr_by_language(language)

  with tempfile.NamedTemporaryFile(
   prefix="uploaded_",
   delete=False,
   dir=TMP_DATA_DIR
  ) as fp:
    fp.write(file.file.read())
    fp.seek(0)
    tempfile_name = fp.name

  if params.realtime:
    return recognize_file(
      filepath=tempfile_name,
      asr_instances=asr_instances,
    )
  else:
    return queue_pipeline(
      session_id=session_id,
      audio_location=tempfile_name,
      asr_instances=asr_instances,
    )

def endpoint_parameters(
  language: str = None,
  realtime: bool = False,
) -> schemas.RecognizerEndpointParams:
  return schemas.RecognizerEndpointParams(
    language=language,
    realtime=realtime,
  )
```

In the above code snippet, we declare the URI path to server code at http(s)://{server_ip_address}/recognizer/audio-file with `@router.post("/recognizer/audio-file")`, and declare parameters to be accepted: audio file and dictionary parameters to parse as metadata.

During the processing of a request, we store the received audio stream into a temporary file `with tempfile.NamedTemporaryFile`. We do this to have the possibility to work with queueing mechanism. Hence, we can't keep all in the memory and need to exploit storage.

In the end, we examine metadata parameters to identify whether we should return results in real-time `return recognize_file()` or we can queue as a task and just return identification `return queue_pipeline()` to obtain results in the future by requesting the GET method.

The presented code will result in generating the API endpoint shown in Fig. 5.

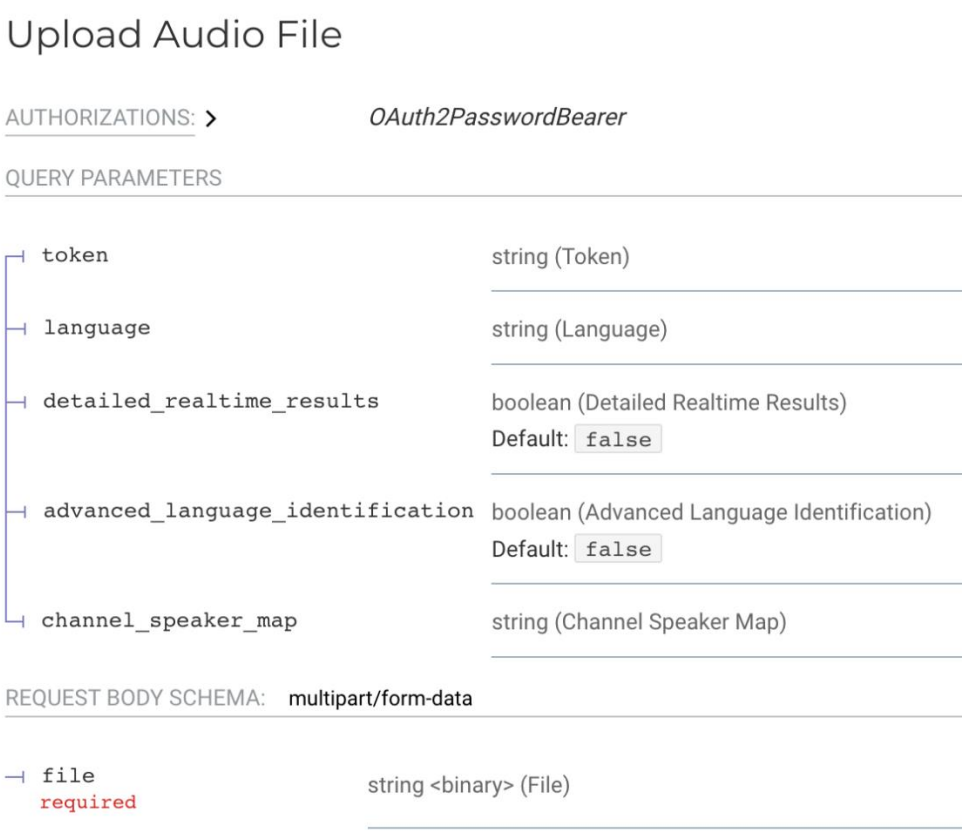

Figure 5: API endpoint example for audio file uploading

5. Comparison results

5.1. Prototype performance

Using Version 1 of the prototype on a server with 4×A100 GPUs, we were able to process 8000 hours of audio in 24 hours or 333 hours in 1 hour, which represents 0.003 Real-Time Factor which is far behind with state-of-the-art models [25, 26]. We will not go deeply into the limitations of Version 1 (described below) because the main goal is to develop a prototype that will align with the state-of-the-art models.

Using Version 2 of the prototype on a server with 4×A100 GPUs, we were able to process 45,000 hours of audio in 24 hours or 1875 hours in 1 hour, which represents 0.0006 Real-Time Factor which is in line with the state of art models [27, 28].

The main components of such a tremendous end-to-end performance are ASR and WEB frameworks [4, 19].

The main performance limitation is ASR prediction. The performance of ASR models serving on CPU was minimal, and with the big model with WER, 4–6% is almost one real-time factor per one vCPU. Hence, it makes sense to use CPU-based deployments only for limited (approximately up to 100 on the most significant AWS instance) concurrent sessions [29–31].

5.2. Difficulties with the first version implementation

The main limitation was API (back-end) as a logic center that decided which steps to take next and waited for the results of any current task. Hence, API/BE was blocked from accepting any new files before the current audio file was inside the pipeline. And while each file was 5–10 minutes long, API was blocked for approximately this period.

5.3. Difficulties with the second version implementation

With Version 2, the prototype became much more stable and predictable, but still, there were a lot of limitations and difficulties during prototype testing.

1. Prioritization of real-time: current prototype does not prioritize real-time tasks over queued ones, which means there is no confidence that real-time tasks will be accepted if queue and recognizers have a lot to do.
2. Stability: to support both streaming and batch file recognition (with one ASR server prototype) current prototype version is recognized through streaming of audio files from the worker directly to ASR, which is not recommended for a production-ready solution and should be refined.
3. Scalability: there is no load balancing opportunity in the current prototype (neither for API block nor for ASR and Post-processing block). This should be refined for the production readiness of the prototype.
4. Feedback and visibility: no feedback information is sent back to the user other than the ID of future results. This can be tricky if services are busy, as users will not know when exactly the task will be finished and will not get any estimations and progress info. Hence prototype should be refined with added visibility and webhooks for feedback when results are ready and in case of any issue with production readiness.

6. Acknowledgements

The research team is grateful to Ender Turing OÜ for defining the business problem, comments, corrections, inspiration, and computational resources.

7. Conclusion and future works

The article presents the prototype of an end-to-end speech recognition framework with a storage database and results from indexing. For this framework and protocols for real-time and non-real-time were selected in the way to be ready to scale for a production-ready solution.

Various software solutions' compatibility problems were solved during the experimental setup assembly, and a working prototype was built. An architectural diagram of the solution was also shown.

As a result, it was tested that the prototype delivers stability if the number of processed audios is less than one hour of audio per one vCPU. Hence, managing and load balancing of connections and audios to process is a task to be solved in the following versions of the prototype, as well as prioritization of real-time processing against non-real-time tasks.

Future research will be focused on optimization issues, such as the scaling of speech recognizers, parallelization of the pipeline, fault tolerance, pipeline progress visibility, security, and webhooks implementation to inform result readiness and make it easier to deploy.

8. References

[1] I. Iosifov, et al., Natural Language Technology to Ensure the Safety of Speech Information, in: Proceedings of the Workshop on Cybersecurity Providing in Information and Telecommunication Systems 3187(1) (2022) 216–226.

[2] O. Iosifova, et al., Analysis of Automatic Speech Recognition Methods, in: Proceedings of the Workshop on Cybersecurity Providing in Information and Telecommunication Systems 2923 (2021) 252–257.

[3] O. Iosifova, et al., Techniques Comparison for Natural Language Processing, in: Proceedings of the Modern Machine Learning Technologies and Data Science Workshop 2631 (2020) 57–67.

[4] O. Kuchaiev, et al., NeMo: A Toolkit for Building AI Applications using Neural Modules, arXiv (2019) 36–44. doi:10.48550/arXiv.1909.09577.

[5] S. Watanabe, et al., ESPnet: End-to-End Speech Processing Toolkit, arXiv (2018) 1–5. doi:10.48550/arXiv.1804.00015.
[6] D. Povey, et al., The Kaldi Speech Recognition Toolkit, in: IEEE 2011 Workshop on Automatic Speech Recognition and Understanding (2011) 1–4.
[7] NVIDIA Developer, Endless Ways to Adapt and Supercharge Your AI Workflows with Transfer Learning (2022). URL: https://developer.nvidia.com/tao-toolkit-usecases-whitepaper/1-introduction
[8] S. Chandrasekaran and M. Salehi, Simplifying AI Inference in Production with NVIDIA Triton (Apr 12, 2021). URL: https://developer.nvidia.com/blog/simplifying-ai-inference-in-production-with-triton/
[9] NVIDIA Developer, NVIDIA Triton Inference Server (2022). URL: https://developer.nvidia.com/nvidia-triton-inference-server
[10] O. Romanovskyi, et al., Automated Pipeline for Training Dataset Creation from Unlabeled Audios for Automatic Speech Recognition, Advances in Computer Science for Engineering and Education IV 83 (2021) 25–36. doi:10.1007/978-3-030-80472-5_3.
[11] I. Iosifov, O. Iosifova, V. Sokolov, Sentence Segmentation from Unformatted Text Using Language Modeling and Sequence Labeling Approaches, in: Proceedings of the 2020 IEEE International Scientific and Practical Conference Problems of Infocommunications. Science and Technology; IEEE: Kharkiv, Ukraine (2020) 335–337. doi:10.1109/PICST51311.2020.9468084.
[12] I. Iosifov, et al., Transferability Evaluation of Speech Emotion Recognition Between Different Languages, Advances in Computer Science for Engineering and Education 134 (2022) 413–426. doi:10.1007/978-3-031-04812-8_35.
[13] J. Thönes, Microservices, IEEE Software 32(1) (2015) 113-115. doi:10.1109/MS.2015.11.
[14] N. Alshuqayran, N. Ali, and R. Evans, A Systematic Mapping Study in Microservice Architecture, in: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA) (2016) 44–51. doi:10.1109/SOCA.2016.15.
[15] N. Dragoni, et al., Microservices: Yesterday, Today, and Tomorrow, Present and Ulterior Software Engineering (2017) 195–216. doi:10.1007/978-3-319-67425-4_12.
[16] H. Vural, M. Koyuncu, and S. Guney, A Systematic Literature Review on Microservices, Computational Science and Its Applications (ICCSA) (2017) 203–217. doi:10.1007/978-3-319-62407-5_14.
[17] A. Neumann, N. Laranjeiro, and J. Bernardino, An Analysis of Public REST Web Service APIs, IEEE Transactions on Services Computing 14(4) (2021) 957–970. doi:10.1109/tsc.2018.2847344.
[18] A. Ehsan, et al., RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions, Applied Sciences 12(9) (2022) 4369. doi:10.3390/app12094369.
[19] TechEmpower, Web Framework Benchmarks (2022). URL: https://www.techempower.com/benchmarks/#section=test&runid=7464e520-0dc2-473d-bd34-dbdfd7e85911&hw=ph&test=query&l=zijzen-7
[20] V. Pimentel and B. G. Nickerson, Communicating and displaying real-time data with WebSocket, IEEE Internet Computing 16(4) (2012) 45–53. doi:10.1109/MIC.2012.64.
[21] J. Li, et al., Jasper: An End-to-End Convolutional Neural Acoustic Model, arXiv (2019) 1–5. doi:10.48550/arXiv.1904.03288.
[22] S. Kriman, et al., QuartzNet: Deep Automatic Speech Recognition with 1D Time-Channel Separable Convolutions, arXiv (2019) 1–5. doi:10.48550/arXiv.1910.10261.
[23] O. Hrinchuk, M. Popova, and B. Ginsburg, Correction of Automatic Speech Recognition with Transformer Sequence-To-Sequence Model, in: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (2020) 7074–7078. doi: 10.1109/icassp40776.2020.9053051.
[24] S. Beliaev and B. Ginsburg, TalkNet: Non-Autoregressive Depth-Wise Separable Convolutional Model for Speech Synthesis, Proc. Interspeech (2021) 3760-3764. doi:10.21437/Interspeech.2021-1770.
[25] A. Georgescu, et al., Performance vs. Hardware Requirements in State-of-the-Art Automatic Speech Recognition. J. Audio Speech Music 28 (2021). doi:10.1186/s13636-021-00217-4.

[26] A. Dutta, G. Ashishkumar, and Ch. V. R. Rao, Improving the Performance of ASR System by Building Acoustic Models using Spectro-Temporal and Phase-Based Features, Circuits, Systems, and Signal Processing 41(3) (2021) 1609–1632. doi:10.1007/s00034-021-01848-w.
[27] S. Gondi and V. Pratap, Performance and Efficiency Evaluation of ASR Inference on the Edge, Sustainability 13(22) (2021) 12392. doi:10.3390/su132212392.
[28] S. Li, et al., Improving Transformer-Based Speech Recognition Systems with Compressed Structure and Speech Attributes Augmentation, Interspeech (2019). doi:10.21437/interspeech.2019-2112.
[29] A. Rahmatulloh, I. Darmawan, and R. Gunawan, Performance Analysis of Data Transmission on Websocket for Real-Time Communication, in: 16th International Conference on Quality in Research (QIR) (2019) 1–5. doi:10.1109/QIR.2019.8898135.
[30] P. Murley, et al., Websocket Adoption and the Landscape of the Real-Time Web, in: World Wide Web Conference (WWW) (2021) 1192–1203. doi:10.1145/3442381.3450063.
[31] P. Bansal and A. Ouda, Study on Integration of FastAPI and Machine Learning for Continuous Authentication of Behavioral Biometrics, in: International Symposium on Networks, Computers and Communications (ISNCC) (2022) 1–6. doi:10.1109/ISNCC55209.2022.9851790.