Story2Audio Microservice for NLP Course Project

Story2Audio: Project Breakdown

1. Project Overview

In simple terms, the **Story2Audio** project is a microservice that takes a written story and produces an audio narration (an "audio book") that reflects the story's tone and sentiment. For example, a sad scene should sound mournful, an exciting scene upbeat, etc., rather than a flat monotone reading. The service must be self-contained: it will run as a standalone server with a gRPC /generate endpoint. Clients send a JSON payload with the story text, and the server replies with an audio file (or audio data) and status code. Internally, the system will use open-source models and libraries (e.g. LLaMA/Phi models, Bark TTS) to process the text and synthesize expressive speech. The deliverables include source code, a Dockerfile, and a simple front-end demo (e.g. Gradio) along with documentation and test cases.

Key Requirements: The assignment specifies a gRPC API with one core endpoint (such as /generate), asynchronous/concurrent handling of requests, minimal error handling, JSON responses with status codes, comprehensive testing, and demonstration via Postman and a Gradio/Streamlit UI. The audio must match the input text's tone (not just monotonic speech). All components must use freely available models (e.g. Phi, LLaMA, Bark) and be containerized with deployment scripts (e.g. Docker + GitHub Actions).

2. Core Components

To meet these requirements, the system should consist of the following components:

- **gRPC Service Layer:** A Python-based gRPC server (using grpcio or grpc.aio) that defines a protobuf schema with one RPC method (e.g. GenerateAudio) accepting the input text and returning a response (including audio data or a URL plus a status code). This layer handles request parsing, invokes the NLP/TTS pipeline, and formats the JSON response. (Deliverable: a .proto file and server implementation.)

- **Concurrency Handling:** Configure the gRPC server to handle multiple requests in parallel. By default, gRPC Python uses a thread-pool so each incoming request can run on its own threadstackoverflow.com. For more scalable async handling, one can use grpc.aio.server() (available in gRPC ≥1.32) and define async def service methodsstackoverflow.com. In either case, ensure the server can process overlapping calls (e.g. by increasing threadpool size or using asyncio) so that concurrent generate calls do not block each other.

- **Input Validation & Error Handling:** Before calling the model, validate the input (e.g. non-empty string, reasonable length). If the input is invalid, the server should return an error status (e.g. gRPC INVALID_ARGUMENT) and a JSON error message. gRPC has built-in status codes for this purposegrpc.io. Wrap model calls in try/catch and use gRPC's status codes (e.g. context.abort(grpc.StatusCode.INTERNAL, "error message")) to signal failuresgrpc.io. The response JSON should include a status field (e.g. "success":false or an HTTP-like code) and an error message when appropriate.

- **Text Processing (NLP):** (Optional) Analyze the input story text to determine its sentiment, emotion, or style. For example, run a simple sentiment classifier or use an open LLM (Phi, LLaMA, etc.) to detect mood. This information can guide the TTS model to select an appropriate speaking style. (Some TTS models allow conditioning on emotion or style tokens.) However, even without an NLP step, one can rely on the TTS model's natural expressiveness.

- **Expressive TTS Pipeline:** The core component is a Text-to-Speech model that produces **expressive** audio. A promising choice is Suno AI's **Bark** model (open-source, MIT-licensed). Bark is a text-to-audio transformer that generates highly realistic, multilingual speech (including inflections and nonverbal cues like laughter)github.com. The pipeline for Bark (and similar models) generally works in stages: it encodes the input text into semantic tokens, then generates coarse and fine acoustic codebooks, and finally decodes those codebooks into a raw waveformdocs.openvino.ai. (See diagram below.) Other viable TTS models include **Coqui TTS** variants (Tacotron 2, FastSpeech, VITS, etc.) and **X-TTS** (Coqui's advanced model capable of voice cloning and emotional style transfer)bentoml.com. The chosen model(s) should be loaded once at server startup and reused for each request to optimize throughput.

- **Audio Post-processing:** After generation, the raw waveform may need normalization or conversion (e.g. to a compressed format). Store the audio as a byte array or file (e.g. WAV or MP3) that can be returned. The response JSON might include a link or base64-encoded audio. Ensure the audio output respects the input style (including pauses or emphasis as needed).

- **Frontend Demo & Postman:** Develop a minimal user interface (Gradio or Streamlit) that calls the gRPC service. For example, a Gradio app with a text input box and an "Generate" button that streams the audio output. Also prepare a Postman collection with sample requests to test the API. These demos help demonstrate the microservice's functionality.

3. Expressive Text-to-Speech Models

*Bark's text-to-audio pipeline (illustrative): input text → semantic tokens → coarse/fine acoustic codebooks → decoded waveform*docs.openvino.ai*.* Bark is a state-of-the-art text-to-audio model that produces very realistic, emotionally-inflected speech. As shown above, Bark first converts the input sentence into a sequence of **semantic tokens**, then generates hierarchical acoustic **codebooks** (coarse then fine), and finally synthesizes the raw audio waveformdocs.openvino.ai. This multi-stage approach allows Bark to capture subtle prosody and even add non-speech sounds (laughter, sighs) for expressivenessgithub.com.

Recommended models for expressive TTS include:

- **Suno AI's Bark:** An open-source Transformer-based TTS model that generates *highly realistic, multilingual* speechgithub.com. Bark supports a variety of "speaker prompts" and can produce nonverbal cues (e.g. "laugh", "cry" tokens) to make narration engaginggithub.com. It is well-suited for storytelling because it naturally varies intonation.

- **Coqui's X-TTS (XTTS-v2):** A recent open model with strong voice-cloning and emotional control. X-TTS can replicate not only a speaker's voice but also *emotional tone and speaking style*, yielding very natural and expressive speech synthesisbentoml.com. This makes it good for matching the mood of the text. (Note: the open-source version may have licensing restrictions.)

- **Coqui TTS Library:** Contains Tacotron-2, FastSpeech, Glow-TTS and VITS models. These neural TTS models can produce high-quality speech and can be trained or fine-tuned on emotive data. Coqui's toolkit is fully open and supports many languages. While Tacotron-based models tend to require more inference time, they offer fine-grained control over prosody.
- **Others:** Models like **Tortoise TTS** (by Suno) and **Mozilla TTS** also provide expressive voices, though they may not be as straightforward to deploy. Given the course constraints, Bark and Coqui-based models are usually sufficient and easy to integrate.

By choosing one or more of these models, the service can generate engaging audio that matches the input story's emotions. For example, for a sentence "She gasped in horror," Bark might produce a startled tone, whereas a flat TTS would not. Ensuring the model is configured with appropriate style/emotion tokens (if available) is key to maximizing expressivenessgithub.combentoml.com.

4. Building the gRPC API

- **API Definition:** Define a Protocol Buffer schema with one RPC method (e.g. Generate). The request message contains the input text (string) and possibly metadata (e.g. language or requested voice). The response should include a status code (e.g. integer or enum) and either the raw audio bytes or a link/identifier for the generated audio, plus an optional error message. Although gRPC typically uses protobuf, you can include a JSON field or simply document that clients interpret the response as JSON. Ensure every response includes a status indicator so clients know if the generation succeeded. (The project spec explicitly requires JSON responses with status codes, so you might wrap your data in a JSON-like proto message or use gRPC-Web.)
- **Asynchronous Handling:** To support concurrency, either rely on gRPC's built-in threading or use the async API. By default, **gRPC Python** creates a thread for each incoming call, so many requests can be processed in parallelstackoverflow.com. For greater control, use grpc.aio.server(): define your service handlers with async def and use await when calling the TTS model. gRPC v1.32+ supports asyncio nativelystackoverflow.com. Make sure to set a sufficiently large thread pool or parallelism so that long-running TTS calls don't block new requests.
- **Error Handling:** Use gRPC's status codes for errors. For invalid input (e.g. empty text) return grpc.StatusCode.INVALID_ARGUMENT; for server errors return INTERNAL or UNAVAILABLE. The official gRPC guide notes: *"If an error occurs, gRPC returns one of its error status codes, with an optional message describing what happened"*grpc.io. In practice, inside your service implementation catch exceptions and do context.set_details("reason") and context.set_code(grpc.StatusCode.X). Also validate input before processing and abort early if needed. The JSON response (or proto response fields) should include a status field and message so clients can display errors.
- **Concurrency and Timeouts:** Configure appropriate timeouts/deadlines for the RPC so that very long generations (e.g. excessively long stories) don't hang forever. Consider limiting story length or chunking it. For example, break very long text into paragraphs and generate each audio in sequence (or stream chunks). gRPC supports streaming RPCs, but for simplicity a single-response RPC is fine as long as payload size is manageable.
- **API Usability:** Follow best practices: document the endpoint, use clear field names, and return consistent JSON (or protobuf) structures. A well-designed API will get higher marks.

For instance, include fields like request_id or model_version if relevant. Provide examples in the README. Since this is a gRPC service, also include a Postman (or similar) collection that shows how to call the endpoint and what responses to expect.

5. Test Case Design

Thorough testing is required. Design both **unit/integration tests** and **performance tests**:

- **Unit Tests:** Use a Python testing framework (pytest/unittest) to check individual components. For example, test that given a known short story, the TTS function returns non-empty audio data of reasonable length. You can mock the TTS model to test the API logic itself. Check edge cases: empty input should yield an error, overly long text should be handled or rejected, invalid JSON should return a parse error. Validate that the JSON output fields are correct (status code is 0 for success, audio field is present, etc.).

- **Integration Tests:** Write tests that actually start the gRPC server (perhaps on a test port) and use a gRPC client stub to call /generate. Verify end-to-end behavior. For example, use a short sample story and assert that the returned audio can be decoded or has certain properties (e.g. sample rate, duration > 0). Compare audio lengths to input text length (longer text ⇒ longer audio).

- **Auto-Generated Inputs:** You can leverage an LLM to create diverse test inputs. Recent research shows LLMs can generate effective test cases for code and APIsarxiv.org. For instance, use GPT-4 or an open LLM to "write a short children's story about a fox and a raven" or to produce edge-case sentences (very long, with unusual characters, multiple languages, etc.). These synthetic inputs help ensure robustness. The assignment even allows auto-generated inputs.

- **Concurrent/Load Testing:** Use a load-testing tool (e.g. **Locust**) to simulate multiple users hitting the /generate endpoint simultaneouslylocust.io. Locust is an open-source Python tool that can spawn many virtual users calling your API, which is ideal for generating performance graphslocust.io. Configure it to send concurrent requests (e.g. 10, 50, 100 at a time) and record response times. Plot *response-time vs. number-of-concurrent-requests* to evaluate throughput and latency. This aligns with the deliverable of performance evaluation graphs (# of concurrent requests vs. response time). Document any bottlenecks (e.g. GPU vs CPU inference time) and optimize (e.g. batching, caching).

- **Validation Criteria:** Each test should assert correctness (e.g. assert success status, non-empty audio, correct error code on bad input). Collect coverage metrics. The rubric explicitly awards points for *Testing and Fault Tolerance*, so aim for high coverage and include tests for unusual inputs (non-English text, special characters, extremely short or long inputs, etc.).

By using a combination of unit tests, LLM-generated story inputs, and load testing (Locust), you can thoroughly validate the microservice's functionality and performance.

6. Frontend, Containerization, and Deployment

- **Frontend (Gradio/Streamlit):** Create a minimal UI to demo the service. For example, a Gradio app with a text box and a "Generate Audio" button. Gradio is *designed* for this: "Gradio is the fastest way to demo your machine learning model with a friendly web interface so that anyone can use it"gradio.app. In the backend of the app, call the gRPC client stub to get audio, then play or download it in the browser. Streamlit is an alternative (text input + audio player), but Gradio typically requires less code. Ensure the UI clearly labels input/output and shows status codes or errors.

- **Docker Container:** Write a Dockerfile to containerize the microservice. The image should install the Python runtime, required libraries (grpcio, the TTS model library, etc.), and copy your service code. Expose the gRPC port (e.g. 50051). The Dockerfile must allow the service to run out-of-the-box (self-contained). Docker ensures the environment is reproducible (same OS, dependencies, models). This is explicitly required: deliverable #3 is a Dockerfile or deployment scripts.
- **CI/CD (GitHub Actions):** Set up a GitHub Actions workflow for continuous integration. For example, on each push, Actions can build the Docker image and run your test suite. This automates testing and ensures the container builds correctly. As suggested by the assignment, using GitHub Actions is recommended. You could also add a step to push the Docker image to Docker Hub or a container registry. Optionally, use Actions to deploy the Gradio app to Hugging Face Spaces or a cloud VM. (The spec even suggests Hugging Face Spaces for hosting demos.)
- **Reproducibility:** In your README/documentation, include instructions for setting up the environment, running Docker, and using the service. Provide versions for all tools (Python, model, libraries) so others can reproduce your results. Using Docker and a CI workflow already covers the key reproducibility criteria.
- **Performance Monitoring:** As part of deployment, you might add logging (e.g. response times) and health checks. This is optional but can demonstrate scalability. Also prepare performance graphs (requests vs latency) to include in the final report, as per deliverable #5.

7. Tips for High-Quality Submission

To maximize marks, align your work with the evaluation criteria:

- **Functionality (30%)**: Fully implement all required features. The API should *always* produce the correct output (audio) for valid stories and appropriate errors for bad input. The audio output must be intelligible and reflect the input sentiment. For example, demonstrate with test stories (happy vs sad) that your system conveys the difference. Handle edge cases (empty story, extreme length). Include any extra polish: for instance, split long stories into paragraphs and generate a single combined audio.
- **API Design and Usability (10%)**: Design a clean, well-documented interface. Use intuitive method/field names (e.g. GenerateAudio, text, audio_data, status_code). Stick to consistent JSON/proto formats. Provide clear error messages. Include a Postman collection showing example requests and responses. Make the gRPC schema easy to understand. Good API design will make your service easy to use by others.
- **Code Quality and Documentation (20%)**: Write clean, modular code (functions/classes for different tasks), and follow Python best practices (PEP8, meaningful variable names, docstrings). Use a linter (flake8) and formatter (black) if possible. Document your code where logic is complex. Provide a thorough README that explains setup, usage, dependencies, and architecture. (The rubric explicitly values documentation.)
- **Testing and Fault Tolerance (20%)**: As outlined above, include comprehensive automated tests. Achieve good code coverage. Show in your report how you tested concurrency and

error cases. Fault tolerance also means your service shouldn't crash on bad input or overload—handle exceptions gracefully. Include the Postman collection and any test scripts. Refer to your test results or outputs in the documentation. Demonstrating robust tests directly contributes to the *Testing* criteria.

- **Reproducibility and Deployment (20%)**: Use the Dockerfile and CI/CD workflow to ensure anyone can run your service. In your documentation, note exact model versions and provide links to any pretrained models used. Ensure that running docker build and docker run on a fresh system reproduces your results. The inclusion of a deployment script or GitHub Actions workflow (with automated build/test) will fulfill the deployment/reproducibility requirement.
- **Presentation**: Although not explicitly scored in the rubric percentages, the final delivery (README, slides, demo video) matters. Structure your report clearly (as done here), use visuals where helpful (e.g. pipeline diagrams, performance charts), and speak concisely in any demo. A short demo video (even optional) can impress evaluators by showing the system in action. For performance, include graphs (# of concurrent requests vs. response time) to highlight scalability.

By carefully covering all these aspects—correct functionality, clean API, readable code, thorough testing, easy reproducibility, and clear documentation—you will meet the requirements and score highly on each rubric category.

Sources: The above recommendations are informed by the course project specification and contemporary best practices for building ML microservices and TTS systemsgithub.comgradio.appstackoverflow.comgrpc.io.