International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

# PrepForge AI: An AI-Powered Voice Mock Interview and Feedback System Using Job-Description-Grounded Questions and Rubric-Based Evaluation

Under the guidance of
**Jagdish Duppalli**
Professor
Department of Information Science and Engineering
Aditya College of Engineering and Technology
Bangalore, India

**VINODH M**
Department of Information Science and Engineering
Aditya College of Engineering and Technology
Bangalore, India
vinodhm0702@gmail.com

**ATHISH SHETTY**
Department of Information Science and Engineering
Aditya College of Engineering and Technology
Bangalore, India
athishshetty05@gmail.com

**SUJAN BA**
Department of Information Science and Engineering
Aditya College of Engineering and Technology
Bangalore, India
sujanbas2005@gmail.com

***Abstract*—** Candidates usually rehearse for interviews with generic question lists, nothing reacts to what they say, and no structured feedback follows. This paper presents PrepForge AI, a web-based system that turns any job description in PDF form into a personalised voice mock interview. The system extracts the requirements of the role, generates role-specific questions that each carry a rubric, and then runs the interview adaptively: the AI interviewer speaks a question, the candidate answers by typing or speaking, and a dialogue manager chooses between a follow-up and the next question. After the interview every answer is scored for technical accuracy, relevance, communication, grammar and vocabulary, and confidence of language, and a report with an overall score and quotations from the candidate's own answers is produced. The backend is written in Python with FastAPI and uses an open-weight language model and Whisper speech recognition through the Groq API, together with Piper for speech synthesis; the frontend is a React and Vite single-page application. Tests with two unrelated job descriptions, a backend engineer and a marketing manager, gave role-appropriate questions in both cases, and nine of twelve planned test cases passed. Facial analysis was left out on purpose because inferring confidence from facial expression is scientifically contested, so confidence is estimated from wording only. The system is intended for practice and coaching and not for employer screening.

***Keywords*—** *Mock Interview, Large Language Model, Speech Recognition, Text-to-Speech, Rubric-Based Evaluation, Job Description Analysis, Fast API, React.*

## I. INTRODUCTION

A job interview is one of the most important conversations in a person's working life, yet most candidates prepare for it alone. They read the job description once, look up lists of common questions and rehearse answers silently. This style of preparation has three weaknesses. The questions are generic and seldom match the real requirements of the role. Nothing responds to what the candidate actually says, so there is no follow-up and no pressure to be specific. And no one gives structured feedback afterwards, so the candidate cannot tell whether an answer was relevant, clear or convincing.

Three recent developments make a realistic practice tool possible. Large language models (LLMs) can read a long document, pick out its key points and

return well-formed structured output [4]–[6]. Speech recognition models such as Whisper transcribe many accents accurately without per-user training [1]. Neural text-to-speech engines now run on an ordinary computer and sound natural [13]. Combined, these parts allow a system that once needed a human interviewer to be built with modest resources.

PrepForge AI uses them to close the gap described above. A candidate uploads a job description (JD), the system extracts its real requirements, writes a tailored set of questions with a rubric for each, and holds a spoken, conversational interview. At the end the candidate receives a report with scores, labels and quotations taken from their own answers. The name reflects the idea of forging raw material, a plain JD and an unpolished answer, into something sharper.

**Problem statement.** Given only a job description, provide a candidate with an interview rehearsal that is specific to the role, conversational and adaptive, and evaluated against an explicit rubric so that the feedback names concrete strengths and gaps instead of giving generic advice.

The main contributions are listed below.

- A pipeline that converts a JD PDF into a structured analysis and a set of role-specific questions, for any profession and not only software roles.
- A rubric that is generated together with each question, so the standard of judgement is fixed before the candidate answers.
- An adaptive dialogue manager that decides, after each answer, whether to ask a follow-up, with a cap on follow-ups.
- A multi-dimensional scoring and reporting method that quotes the candidate's own sentences.
- An ethical design choice: no facial or body-language inference, and confidence estimated only from wording.

The rest of the paper reviews related work, describes the existing practice and the proposed system, explains the design and implementation, reports results and testing, and ends with limitations, conclusions and future work.

## II. LITERATURE SURVEY

### *A. Interview Practice Tools*

Candidates today rely on question banks, books, video courses and, when they can find a mentor, human mock interviews. Some commercial simulators add recorded questions. Most of these resources address only part of the need: they are not built from the specific JD, they do not adapt to the answer given, or they depend on the availability of another person.

### *B. Automatic Speech Recognition*

Whisper was trained on a very large collection of weakly supervised audio and is robust to accents, background noise and technical vocabulary without fine-tuning [1]. This robustness is what makes spoken answers practical in a tool that cannot train a voice model for each user.

### *C. Large Language Models*

The Transformer architecture [4] made it practical to train language models at scale, and very large models were shown to perform new tasks from a few examples given in the prompt [5]. Fine-tuning with human feedback made them better at following written instructions [6]. These properties allow a model to act as an analyst that extracts requirements from a document, as a question writer, and as an interviewer that responds to an answer.

### *D. Language Models as Judges*

Zheng et al. found that strong language models can judge answers with a level of agreement with human preferences that is comparable to agreement between humans, but they also documented biases such as a preference for longer answers and for the answer shown first [2]. The lesson for an evaluation tool is that the judge needs explicit criteria. PrepForge AI supplies these through a rubric and uses plain rules wherever a rule is enough.

### *E. Neural Text-to-Speech*

Piper is a fast neural speech engine that runs locally on a CPU [13]. Running it on the server avoids a paid speech service and keeps the interviewer's voice independent of the language-model provider.

### *F. Nonverbal Analysis and Its Limits*

A systematic review by Barrett et al. concluded that facial movements do not map reliably onto inner emotional states, and that the same expression can mean different things in different people and settings [3]. Judging a candidate's confidence from the face would therefore be unreliable and could disadvantage neurodivergent candidates and people with

disabilities. For this reason the proposed system does not analyse faces or body language.

### *G. Risks of Language Models*

Bender et al. warn that language models can produce fluent text that is not grounded in fact and can reproduce biases from their training data [7]. In response, the JD analyser of the proposed system is told to extract only what appears in the document, and the system labels its confidence output as an estimate.

### *H. Web Technologies*

The system follows the REST architectural style [8]. FastAPI provides a typed Python web framework [9], React and Vite provide the single-page frontend [10], [11], and the Groq API serves the language and speech models [12].

### *I. Research Gap*

Existing tools rarely combine all of the following: questions grounded in a given JD, spoken and adaptive conversation, rubric-based scoring and feedback that quotes the candidate. The proposed system brings these into one application.

## III. EXISTING SYSTEM

Current preparation methods are fragmented. Table I lists the common options and what each lacks.

**TABLE I.** COMPARISON OF PREPARATION METHODS

| Method | Strength | Limitation |
|---|---|---|
| Question banks and books | Wide coverage of common questions | Generic; not tied to the actual JD |
| Video courses | Explain structure of good answers | No interaction; no feedback on the learner |
| Human mock interview | Realistic and adaptive | Needs a mentor; hard to repeat often |
| PrepForge AI | JD-specific, adaptive, rubric-based report | Depends on LLM quality; not calibrated yet |

As a result, candidates cannot easily rehearse often, cannot practise for a specific role, and cannot measure improvement from one attempt to the next.

## IV. PROPOSED SYSTEM

PrepForge AI automates the whole practice loop. A typical session proceeds as follows.

1. The candidate opens the web application, uploads a JD PDF and chooses how many tailored questions to generate (3, 5, 8 or 10).
2. The backend extracts the text and asks the language model for a structured analysis: job title, seniority, skills, responsibilities, qualifications and soft skills.
3. The model generates the requested questions, each with a category, a target requirement and rubric notes. A fixed self-introduction question is placed first.
4. Each question is converted to speech and played in the browser while its text is shown.
5. The candidate types or records an answer; recordings are transcribed by Whisper.
6. The dialogue manager decides whether to ask a follow-up or move to the next scripted question.
7. When the questions end, or the candidate presses Stop, every answer is scored and a report is shown.

Questions are classified into the categories of Table II, which follow the usual structure of interviews.

**TABLE II.** QUESTION CATEGORIES

| Category | What it probes |
|---|---|
| Technical | Knowledge of tools, methods or facts named in the JD |
| Behavioural | Past behaviour as evidence of future behaviour |
| Situational | Reasoning on a hypothetical problem |
| Role-specific | Understanding of the particular duties of the role |
| Follow-up | Created during the interview to probe a vague or interesting answer |

### *Key Design Decisions*

**Modular core package.** All logic is in one Python package with a folder for each phase: JD processing, dialogue, answer analysis and scoring. The API and the command-line test scripts call the same functions, so each phase can be tested without a server.

**Single LLM gateway.** Every model call goes through one function. The project was first built on a self-hosted model and later moved to a hosted one; because of the gateway, only that module and the configuration file changed.

**Extraction-only analysis.** The analyser may not add skills that the document does not mention, which keeps every question traceable to a real requirement.

**Rubric created with the question.** The rubric describes what a strong answer should cover. The scoring step later reads the same rubric, so feedback can point to specific missing points.

**Text-first development.** Generation, dialogue and scoring were built and tested with typed answers and a mocked model before audio was added, which isolated failures to one layer.

**Hosted model.** The first plan was a self-hosted model so that data stayed local. In practice the first request took about 64 seconds, around 60 of them spent loading the model, which made interaction impractical. A hosted service was therefore chosen; Table III compares the two options.

**TABLE III.** SELF-HOSTED VERSUS HOSTED LANGUAGE MODEL

| Aspect | Self-hosted (8B model) | Hosted (120B model) |
|---|---|---|
| Data location | Stays on the machine | Sent to the provider |
| Start-up delay | About 60 s on first call | None |
| Hardware | Large download, memory, ideally GPU | Network connection only |
| Output quality | Small model; format errors likelier | Large model; reliable JSON |
| Cost | No per-call cost | Free tier with token-per-minute limit |

## V. SYSTEM DESIGN AND METHODOLOGY

### *A. Architecture*

The system has four layers, shown in Fig. 1. The React frontend has three components: a JD uploader, an interview box and a report view. The FastAPI layer exposes the routes, keeps in-memory sessions and serves generated audio. The core package holds the business logic. External services sit at the bottom: the Groq language model, Groq Whisper and the Piper engine that runs on the server. Information moves down through function calls and up through return values; the frontend never calls the core package directly, and the core package knows nothing about HTTP.

### *B. Data Flow*

Data moves through the system as a chain of typed objects, shown in Fig. 2. A PDF becomes plain text, then a JDAnalysis, then a QuestionSet. During the interview each question and its answer form a CandidateAnswer. Afterwards each answer becomes an AnswerScore, the scores are merged into a ScoreBreakdown, and the breakdown with per-question feedback and a conclusion forms the InterviewReport. Typed objects, rather than loose dictionaries, make the expected fields explicit and let mistakes be caught early.

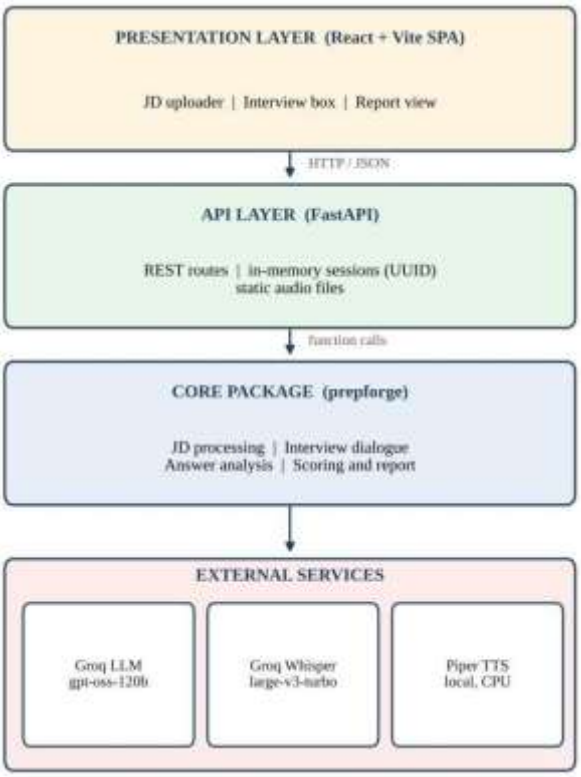



**Fig. 1.** Layered architecture of PrepForge AI.

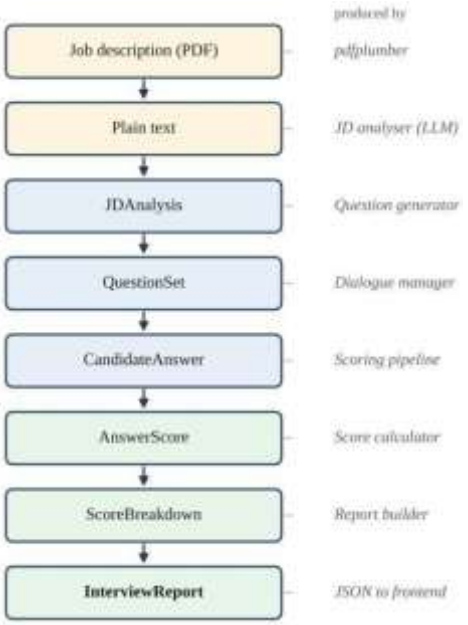



**Fig. 2.** Chain of data objects from job description to report.

### *C. Interview Flow*

The complete control flow is given in Fig. 3. The loop between speaking a question and receiving an answer continues until either the scripted questions are exhausted or the candidate presses Stop.

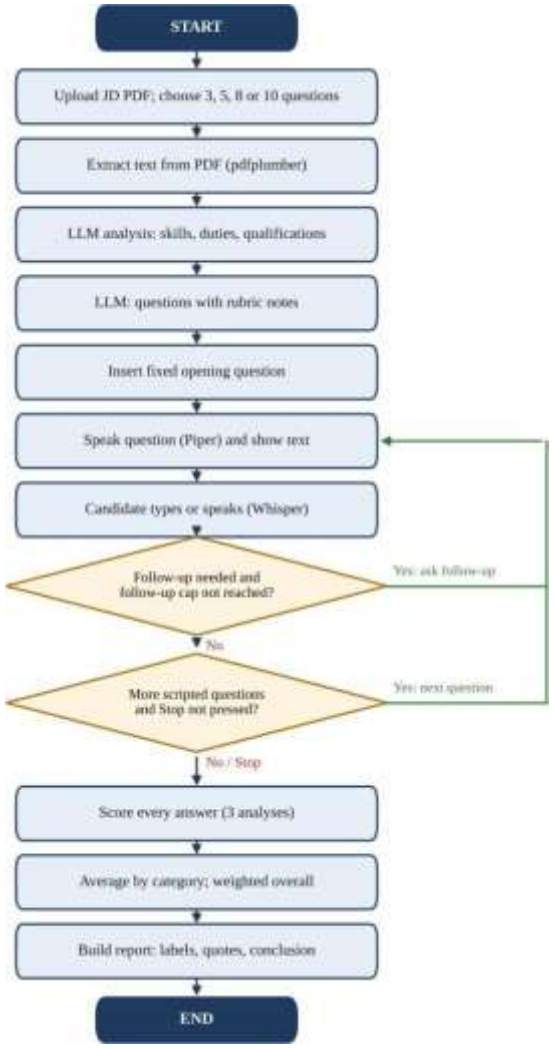



**Fig. 3.** Flowchart of the mock interview from JD upload to report.

### *D. JD Parsing, Analysis and Question Generation*

The parser extracts text page by page with the pdfplumber library. If no text is found, for example in a scanned PDF, an error is raised and the API answers with HTTP 400. The analyser then sends the text to the language model with a strict instruction to extract only what is present, and loads the reply into a JDAnalysis object with empty lists as defaults. The question generator receives this analysis and the requested number of questions and returns, for each question, its text, category, target requirement and rubric notes (ideal_answer_notes) as JSON. A category outside the four allowed values is mapped to role-specific. The opening question is inserted by the program itself, so it costs no model call and is always consistent.

### *E. Dialogue Manager*

The dialogue manager decides what the interviewer says next. If no answer has been recorded, the first scripted question is returned. Otherwise, if the follow-up cap for the current question has not been reached, one model call receives the question, its rubric and the answer, and returns whether a follow-up is needed and its text. A follow-up is requested when the answer is vague or incomplete against the rubric, or when the candidate mentioned something concrete, such as a project, tool or decision, that a curious interviewer would pursue; the second rule was added after testing showed that interviews felt mechanical. If the call fails, the manager falls back to the scripted list so that the interview never stops. Follow-ups inherit the target and rubric of the question they probe. Because the decision uses only the latest answer, a detail mentioned earlier is not revisited; passing the whole history is left as future work.

The length of an interview follows from the number of tailored questions n and the follow-up cap c. Since one opening question is added,

$$T_{max} = (n + 1)(1 + c) \quad (1)$$

### *F. Answer Analysis*

Each answer passes through three independent analyses. In the grammar and clarity analysis, rules count filler words such as “um” and “basically” and hedging phrases such as “I think” and “maybe”. With N words in an answer, the ratios are

$$\rho_m = N_m / N, \quad \rho_h = N_h / N \quad (2)$$

where $N_m$ and $N_h$ are the numbers of filler and hedging words. The hedging ratio is converted to a confidence-language score between 0 and 100,

$$C = g(\rho_h), \ g \text{ decreasing} \quad (3)$$

so that fewer hedges give a higher score. The language model additionally scores grammar and vocabulary, lists issues and gives one corrected rephrasing with an explanation, because showing only a polished replacement would teach imitation instead of understanding. In the relevance analysis the model rates how directly the answer addresses the question and returns a short explanation with a quotation from the answer. In the technical and communication analysis the model acts as a judge against the rubric and returns scores, strengths, gaps, a quoted example and one concrete suggestion.

### *G. Scoring and Report*

Fig. 4 shows how the analyses are combined. Per-answer results are grouped into report categories and averaged. With category means $\bar{S}_k$ and weights $w_k$

defined in a single configuration file, the overall score is

$$O = \Sigma\, w_k\, \bar{S}_k\,, \quad \Sigma\, w_k = 1 \tag{4}$$

on a scale of 0 to 100. Each category receives a label such as Strong or Needs improvement and an example quotation. The report builder asks the model for a short conclusion that names what actually happened, for instance the strongest answer and the habit that most needs work. Keeping the weights in one file makes the scoring rule visible and easy to adjust.

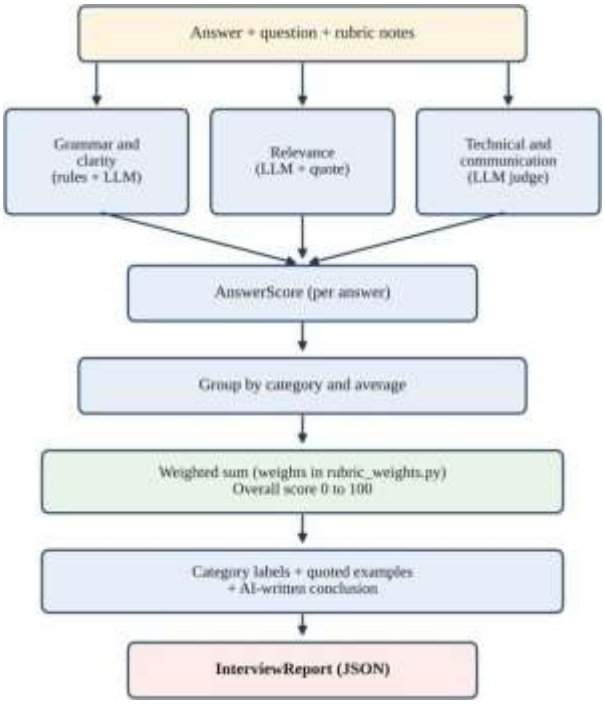



**Fig. 4.** Scoring and report generation.

### *H. REST API*

The API follows the REST style [8]. State that must persist between requests is held in an in-memory session identified by a UUID. Table IV lists the endpoints and Fig. 5 shows the sequence of calls in a session.

**TABLE IV.** REST API ENDPOINTS

| Endpoint | Purpose |
|---|---|
| POST /jd/upload | Upload JD PDF and question count; create session; return first question with audio URL |
| POST /interview/{id}/answer/text | Submit typed answer; return next question |
| POST /interview/{id}/answer/audio | Submit recording; return transcript and next question |
| POST /interview/{id}/stop | End early (needs at least one answer) |
| GET /report/{id} | Score the answers and return the report |
| GET /audio/{id}/{n}.mp3 | Serve the spoken question |

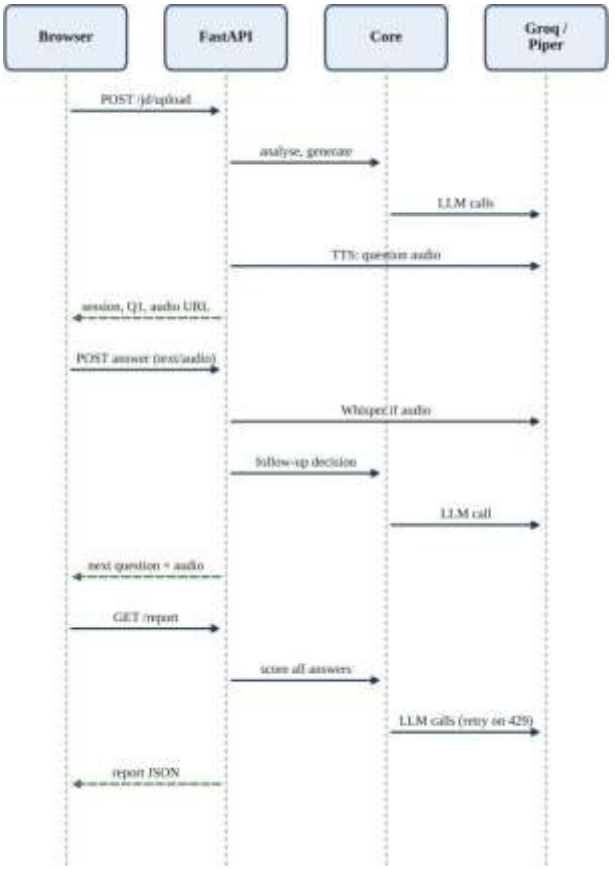



**Fig. 5.** Sequence of calls in a typical interview session.

### *I. Frontend and Error Handling*

The interface has three stages: upload, interview and report. In the interview stage the question is shown and played automatically; the candidate can type an answer or record one, and a Stop button shows the score at any time after the first answer. A status line with an animated indicator is shown while the slowest step, report scoring, is running.

External services fail in ordinary ways, so each failure has a defined response, summarised in Table V. The common principle is that a failure in an optional feature never ends the interview.

**TABLE V.** FAILURE MODES AND RESPONSES

| Failure | Response |
|---|---|
| Missing API key | Error message that points to the configuration file |
| Rate limit (HTTP 429) | Wait and retry up to three times with increasing delay |
| Malformed JSON from model | Strip code fences, parse, else report error with a sample |
| Scanned or unreadable PDF | Parse error; API returns 400 |
| Follow-up decision fails | Skip follow-up; continue with next scripted question |
| Speech synthesis fails | Return question without audio; show text only |
| Transcription fails | Return 502; candidate can switch to typing |

## VI. IMPLEMENTATION

The backend is written in Python with FastAPI [9]. PDF text is read with pdfplumber. The language model is the open-weight openai/gpt-oss-120b and speech recognition uses whisper-large-v3-turbo, both reached through the Groq API [12]. Speech synthesis uses the Piper engine [13] on the server. The frontend is a React [10] single-page application built with Vite [11]. API keys are kept only in an environment file, and the core package is installed in editable mode so that scripts and the server import the same code.

**LLM gateway.** A single function requests JSON from the model, strips any code fences, parses the reply and raises one error type on failure. When the service reports a rate limit it waits and tries again up to three times with increasing delay. This retry was added after a long interview exhausted the free-tier token limit during scoring, because scoring one answer makes several model calls.

**Speech synthesis.** The installed Piper version returns audio chunks, one per sentence, instead of writing to a file. The module therefore clips the floating-point samples to the range −1 to 1, converts them to 16-bit integers, joins the chunks and writes a WAV file, or encodes MP3 with a pure-Python encoder so that no external program is needed. An early version that wrote nothing produced a silent file of 835 bytes; after the fix a test sentence gave an audible file of about 56 KB.

**Speech recognition.** A recording is posted as multipart form data to the transcription endpoint. Missing files, network errors and bad responses are all converted to one exception type. Accepted formats include mp3, mp4, m4a, wav and webm with a limit of 25 MB, far above the length of an interview answer.

**Frontend.** The interview component records audio in the browser, uploads it to the API and displays the transcript. A root script starts the backend and frontend together with a single command.

## VII. RESULTS AND DISCUSSION

### *A. JD-Grounded Question Generation*

The first test used a JD for a backend engineering role. The generated questions named items from the document, including FastAPI, PostgreSQL and AWS services (EC2, S3 and RDS). A query-optimisation question carried a rubric that expected EXPLAIN ANALYZE, indexing and a measured improvement. To check that the pipeline does not depend on software roles, a second test used a JD for a marketing manager. All questions were marketing-specific, as shown in Table VI, and none contained programming content.

**TABLE VI.** QUESTIONS GENERATED FROM A MARKETING JD

| Category | Topic of generated question |
|---|---|
| Behavioural | Self-introduction and fit for the role (fixed opener) |
| Technical | Analytics goals and attribution to measure return on a multi-channel campaign |
| Role-specific | Designing an integrated campaign across email, social and paid channels |
| Behavioural | Coaching an underperforming marketing coordinator |
| Situational | Diagnosing leads that miss the sales team's qualification criteria |
| Role-specific | Keeping brand voice consistent across touchpoints |

### *B. Speech Input and Output*

A recorded self-introduction of about 30 seconds was transcribed correctly, with correct punctuation and the speaker's name. After the synthesis fix described above, the spoken question was clearly audible and the interface played it automatically. Recording spoken answers in the browser could not be tested because the development machine had no microphone; this item is reported as untested.

### *C. Adaptive Behaviour*

In a mocked test a deliberately vague answer triggered exactly one follow-up, a specific reply to it was accepted, and the interview moved on, respecting the cap. In a live session the interview ran longer than the tester expected. Equation (1) explains why: with an opening question added and two follow-ups allowed, a request for 10 questions can reach 33 turns, as plotted in Fig. 6. Returning the cap to one cut the maximum to 22 turns, and the Stop button was added so that the candidate can finish when they wish.

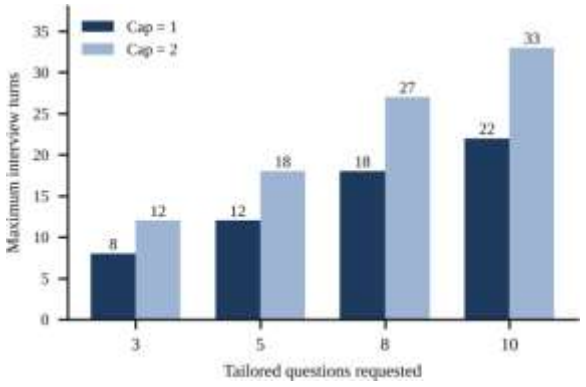



**Fig. 6.** Maximum number of turns for each setting, from Equation (1).

### *D. Scoring and Report*

In a mocked scoring test, a well-structured answer scored high on technical accuracy and confidence language, while an answer full of fillers and hedges scored low on both. The written conclusion referred to what had happened in the answers and not to generic advice, which is the purpose of attaching a rubric to every question and quoting the candidate. The rate-limit error found during live scoring was handled by the retry described in Section VI.

### *E. Performance Observations*

The main performance finding is the start-up delay of about 64 seconds for the first request to a self-hosted model, which led to the move to a hosted service. With the hosted service, question generation returns within an ordinary web-request time and speech synthesis for a question is short. The slowest step is the final report, because a twelve-answer interview needs a few dozen model calls. No formal latency or throughput benchmark was carried out, so no such figures are claimed; measuring each stage is part of future work.

## VIII. SYSTEM TESTING

Each phase was tested separately, first with typed answers and a mocked model, then with the live model and finally through the browser. Table VII summarises the twelve test cases. Nine passed, one was fixed and awaits a repeat test, one is implemented and awaits verification, and one could not be run for lack of a microphone.

**TABLE VII.** TEST CASES AND STATUS

| Test case | Status |
|---|---|
| PDF text extraction | Passed |
| JD analysis and questions, live model | Passed |
| Domain generality (marketing JD) | Passed |
| Speech-to-text, 30 s recording | Passed |
| Text-to-speech, one sentence | Passed |
| Follow-up logic (mock) | Passed |
| Scoring (mock) | Passed |
| API health check | Passed |
| Browser interview, typed mode | Passed |
| Rate-limit handling during scoring | Fixed; re-test advised |
| Early stop after one answer | Pending verification |
| Browser microphone recording | Not tested |

The defects found were mostly integration and configuration problems and not faults in the core logic, which supports the decision to test each phase separately. Table VIII lists the main ones.

**TABLE VIII.** SELECTED DEFECTS AND FIXES

| Defect | Corrective action |
|---|---|
| Circular import in a package file | Emptied the stray import |
| Model not found on the provider | Listed live models and chose an available one |
| Silent 835-byte audio file | Rewrote synthesis to collect chunks and write samples |
| 429 error while scoring | Automatic retry with increasing delay |
| Interview felt endless | Follow-up cap set to one; Stop button added |
| Backend exit on Windows script | Quoted the executable path in the script |

## IX. ADVANTAGES, LIMITATIONS AND ETHICS

The system generates role-specific questions for any text-based JD, works conversationally with spoken questions and optional spoken answers, and gives feedback that quotes the candidate and points to specific gaps. A typed mode makes it usable without a microphone, and the single-gateway design lets any external service be replaced without changing the rest of the code.

The limitations are clear. The scoring depends on a language model and has not been calibrated against human raters. The free tier limits throughput. Follow-up decisions use only the latest answer. Sessions are held in memory and are lost when the server restarts. Scanned PDFs are not supported because there is no OCR, and spoken input recorded in the browser still needs verification. When a hosted model is used, the interview text is sent to the provider and the provider's terms apply.

On ethics, the system is intended for practice and coaching and must not be used to screen or rank applicants. It does not analyse faces or body language, for the reasons given in Section II, and it presents its confidence measure as an estimate based on wording. Keys and secrets are kept out of shared files.

## X. CONCLUSION AND FUTURE SCOPE

PrepForge AI shows that a complete, role-specific voice mock interview can be assembled from a language model, a speech recogniser, a speech synthesiser and a small web application. The system extracts requirements from a job description, generates questions with rubrics, runs an adaptive interview with typed or spoken answers and produces a report that quotes the candidate. Isolating each

external service behind one module, testing on text before audio, and writing the rubric together with the question were the design choices that proved most valuable.

Future work includes giving the dialogue manager the full conversation history, storing sessions in a database so that progress can be tracked, calibrating the scoring against human interviewers and showing the reasoning behind each category score, accepting a resume so that questions can probe real experience, streaming transcription with automatic end-of-answer detection, OCR and DOCX support, measuring latency of each stage, and containerising the application with authentication and clear privacy controls.

## REFERENCES

[1] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust Speech Recognition via Large-Scale Weak Supervision," arXiv:2212.04356, 2022.

[2] L. Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," in Advances in Neural Information Processing Systems (NeurIPS), 2023.

[3] L. F. Barrett, R. Adolphs, S. Marsella, A. M. Martinez, and S. D. Pollak, "Emotional Expressions Reconsidered: Challenges to Inferring Emotion from Human Facial Movements," Psychological Science in the Public Interest, vol. 20, no. 1, pp. 1–68, 2019.

[4] A. Vaswani et al., "Attention Is All You Need," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

[5] T. B. Brown et al., "Language Models are Few-Shot Learners," in Advances in Neural Information Processing Systems (NeurIPS), 2020.

[6] L. Ouyang et al., "Training language models to follow instructions with human feedback," in Advances in Neural Information Processing Systems (NeurIPS), 2022.

[7] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?," in Proc. ACM Conf. Fairness, Accountability, and Transparency (FAccT), 2021.

[8] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[9] FastAPI documentation. [Online]. Available: https://fastapi.tiangolo.com

[10] React documentation. [Online]. Available: https://react.dev

[11] Vite documentation. [Online]. Available: https://vite.dev

[12] Groq API documentation. [Online]. Available: https://console.groq.com/docs

[13] Piper text-to-speech (OHF-voice). [Online]. Available: https://github.com/OHF-voice/piper1-gpl