ChatGPT

AI-Driven Adaptive Technical Interview Bot: Design and Implementation

1. Overall System Architecture

A production interview bot is typically built as a **distributed, microservices-based system** rather than a single monolith [1] [2]. This decoupled design allows independent scaling and updates of components. For example, separate services can handle speech-to-text (ASR), natural language understanding (NLU), question retrieval, answer evaluation, and state management. Each service can be containerized (e.g. Docker) and orchestrated via Kubernetes or similar tools [2] [3]. APIs (REST/JSON or gRPC) connect these microservices: the frontend/UI (web or voice app) calls backend endpoints to ask questions or submit answers. Large models (LLMs, embeddings) run in dedicated inference services (e.g. using TorchServe, FastAPI, or cloud endpoints). A **vector database** (Milvus, Weaviate, Pinecone, etc.) stores question embeddings for semantic search [4] [2]. An orchestration layer (e.g. Apache Airflow, Temporal, or Step Functions) can coordinate multi-step pipelines (e.g. logging a turn, triggering model inference, updating knowledge state) [3].

In practice, the system supports **real-time** and **asynchronous** modes. In real-time mode, the candidate speaks or types answers in an ongoing session: audio streams to ASR (e.g. Whisper) and text flows to Q-selection and scoring modules immediately. Latency must be low (e.g. sub-second) so the bot feels responsive. In asynchronous mode, the candidate's responses (voice or text) are submitted offline and processed later (for batch scoring or analysis). Backend queues (e.g. Kafka, RabbitMQ) can buffer tasks between services. Continuous integration/deployment pipelines ensure models (and data schemas) are versioned and updated safely [2].

- **Microservices vs. Monolith:** A microservices design splits functionality into independent services, which can be scaled and updated without redeploying the whole system [5] [1]. While a monolith is simpler to start, microservices (with container orchestration) are preferred for enterprise-grade systems [5] [2].
- **APIs & Model Serving:** Each AI component (ASR, embeddings, LLM) exposes an API. For example, a FastAPI endpoint may load a Transformer model and serve inference requests. This separates model runtime from business logic.
- **Vector Databases:** Questions are embedded into semantic vectors (via a sentence or cross-encoder) and indexed in a vector DB [4]. At runtime, the bot can find semantically similar questions or answers by performing nearest-neighbor searches in the vector space [4].
- **Orchestration:** A workflow engine manages interview state transitions. For instance, it might enforce: "After scoring an answer, update the candidate's skill profile and then select the next question." AI-specific cloud-native designs treat models as first-class services [2] [3], allowing CI/CD and monitoring (e.g. automated drift detection, model versioning, logging).

2. Skill-Based Interview Initialization

Upon starting an interview, the bot parses the candidate's self-reported skills and experience level. Skills are normalized (e.g. "ML" → "Machine Learning") and matched against a **skill taxonomy or ontology** [6]. A skill taxonomy is a *hierarchical classification* of related skills [6], so "Python" might map to sub-

skills like “OOP,” “Data Structures,” “Async Libraries,” etc. In practice, the system can use a combination of rule-based matching and embedding similarity: for each input skill string, look up canonical names (e.g. using an internal mapping or pretrained NER model) and expand to related concepts via the taxonomy.

Next, each skill is calibrated for **difficulty** based on the candidate’s level (fresher, mid, senior). For example, a senior Python developer implies deeper topics (metaprogramming, concurrency) than a fresher. This sets the initial parameters of the interview policy (e.g. starting difficulty score, topic choice). The system might use a simple rule (e.g. `if experience >5y then difficulty += 2`) or an ML regression on profile features. In summary, the bot builds an *initial knowledge profile* that maps each declared skill to an estimated mastery (or baseline difficulty), to seed the adaptive questioning.

3. Question Bank & Knowledge Representation

The **question bank** is a core data component. Questions are stored in a structured database with fields like: `{id, text, solution, skill_tags, difficulty, format, ...}`. Each question’s *text* is indexed in two ways: (1) traditional indexing (full-text or key tags), and (2) **semantic embeddings**. We use a sentence encoder (e.g. Sentence-BERT) to convert each question into a fixed vector, then store these in a vector database [4]. At runtime, to find a question on a particular topic or similar content, the bot generates a query vector and retrieves nearest neighbors by cosine similarity [4]. This enables recall of semantically related questions even if keywords differ.

Questions are also organized by **topics and prerequisites** via a knowledge graph. In a knowledge graph approach, each question is linked to concept nodes, and edges indicate prerequisite relationships [7]. For example, “implementing a LRU cache” might depend on “hash tables” and “doubly linked lists.” This graph lets the bot avoid asking a question if a prerequisite hasn’t been covered. It also helps sequence questions: e.g. teach basic sorting before asking about algorithm optimizations. As Kop et al. describe, knowledge-graph methods explicitly model *domain structure* and concept dependencies [7], which guides content selection.

There are two ways to generate questions: **static** and **dynamic**. Static questions are hand-crafted and vetted (stored as above). Dynamic question generation uses an LLM or template engine to create novel questions on the fly. For example, one could prompt a GPT to “generate a medium-difficulty Python OOP question,” then check the output. Dynamic generation expands the question pool but requires careful filtering to ensure correctness and uniqueness. In both cases, the system tracks which questions have been asked (by ID or by semantic hash) to **avoid repetition**. Before re-asking or generating a similar question, the bot checks the candidate’s history (using the vector store) to ensure new questions. This prevents leaks or duplicates: the interview won’t accidentally present the same question wording twice.

4. Dataset Design & Training Data

Building and training the AI components requires diverse data. Key dataset types include:
- **Interview Transcripts:** Text (or audio) logs of real or mock interviews. These contain the questions asked and the candidate’s answers. Transcripts are used to train dialogue management and response evaluation models.
- **Question–Answer Pairs:** Curated or synthetic Q/A pairs (including solutions and multiple example answers) are needed to teach the system what correct and incorrect answers look like. These can come from textbooks, forums, or be generated by large LMs.
- **Candidate Performance Logs:** Records of candidate actions (e.g. time per question, code

submissions, answer correctness) help in modeling pacing and scoring.
- **Rubrics & Scoring Annotations:** Each question may have an expected answer rubric. Human experts annotate model answers or actual candidate answers with scores and rationales. This labeled data trains evaluation models.

Datasets must be **labeled** for difficulty and content. For example, each question can be tagged as "easy/medium/hard," and answers can be annotated with concept coverage or depth. In practice, labeling is often performed by domain experts using guidelines (rubrics) to ensure consistency. For instance, experts might rate answers 1–5 for correctness or identify which key points are covered. As Leong et al. note, creating comprehensive rubrics and calibrating raters is essential to minimize subjective bias [8].

Since real interview data is scarce, **synthetic data** generation is often used. One can use LLMs (GPT-4, PaLM, etc.) to automatically create large volumes of Q/A examples. For instance, prompts can generate variations of a coding problem or plausible candidate answers. These can bootstrap initial training; later replaced with real data as it comes in. An MVP might start with a few thousand Q/A pairs and a few dozen transcripts; a full system may require tens of thousands of questions and hundreds of hours of labeled interviews.

Bias and fairness are critical. Human interviewers (and therefore their annotations) often exhibit unintended biases. Studies show factors like age, gender, or race can skew human hiring decisions [9]. We must actively mitigate this. In practice, that means diversifying the training corpus and auditing model performance across demographic groups. All candidate data must be handled per privacy laws. For example, GDPR requires consent for recording interviews and secures PII [10].

5. Deep Learning Models Used

Different AI models serve different pipeline stages. Common choices include:

- **Skill Parsing & Embedding:** A text encoder (e.g. BERT/RoBERTa) can embed the candidate's skill list or resume summary. These embeddings are matched against a skill taxonomy to normalize input. For example, Sentence-BERT could encode "Python, Web Dev" and identify nearest skills in an ontology.

- **Question Selection Policy:** We can use deep learning here in multiple ways. A **policy network** (e.g. an RL agent) can learn to pick questions given the current state. Alternatively, a pair of encoders could rank candidate questions by relevance: one encoder for state+history, another for question text, scoring via dot-product. Reinforcement-learning models (DQN, policy gradient) can be trained to maximize a long-term reward (such as candidate mastery). For instance, Lin and Liu formulate adaptive testing as an MDP and find that a DQN-based selector outperforms traditional item-response methods [11].

- **Answer Evaluation:** Transformer models are used to judge answers. Options include: 1) **Encoder-only** models to embed the candidate's answer and compare it to the solution (e.g. compute cosine similarity between SBERT embeddings of answer vs. reference); 2) **Decoder-only** (LLM) models to generate a score or feedback via prompting; 3) **Encoder–Decoder** models (T5/BART) fine-tuned on Q→score. In practice, all three are used. For example, one approach is to extract a BERT embedding of the answer and feed it to a regression/classifier (as shown by Xie et al. achieving QWK≈0.817 on essay scoring with a frozen BERT+contrastive regressor [12]). Another is to fine-tune a GPT- or Llama-based model end-to-end on answer-scoring labels [13]. Hybrid methods (e.g. using LLM prompting plus retrieved exemplars) can also be applied.

- **Follow-up Generation:** After an answer, the bot may need to probe deeper. A generative LLM (GPT, Claude, etc.) is used here. For instance, given the question, answer, and current context, an LLM can be prompted to "ask a clarifying question" or "give gentle hints." Large chat models (with system prompt guidance) excel at such conversational Q/A.

- **Knowledge Tracing Models:** To maintain the candidate's skill state, we can use models like Bayesian Knowledge Tracing (BKT) or Deep Knowledge Tracing (DKT). **BKT** treats each skill as a hidden binary state (mastered or not) and updates its probability with each answer [14]. It uses parameters for initial mastery, learning, slip (careless error), and guess [15]. **DKT** replaces this with an RNN (LSTM) that ingests the sequence of (question, correctness) pairs and outputs skill mastery vectors [16]. Piech et al. showed DKT can learn complex patterns and outperform BKT by modeling latent state with neural nets [16]. In practice, we may choose BKT for its simplicity or DKT for its flexibility, or even hybrid.

- **Curriculum and Bandit Learning:** The bot's policy can be refined by curriculum learning (start easy, then gradually harder) and multi-armed bandit methods. A bandit algorithm might treat each topic as an "arm" and select questions to balance exploitation (test known strengths) vs. exploration (probe uncertain areas). Over many interviews, a contextual bandit could learn which sequences yield fastest mastery.

6. Answer Evaluation & Scoring

The bot must evaluate answers **semantically**, not just by keywords. Techniques include:

- **Semantic Similarity:** Compute the similarity between the candidate's answer and the reference solution. For example, encode both with a Transformer (SBERT, etc.) and score by cosine similarity. This captures meaning even if different wording is used.

- **Concept Coverage & Partial Credit:** Answers often deserve partial credit. One approach is to detect presence of key concepts. For instance, the rubric might list "use of loop" and "proper variable naming." The model checks which rubric points appear. Neural models can be trained (supervised on human-scored answers) to regress to a partial score.

- **Reasoning Depth:** To assess depth, we might prompt an LLM to judge if the candidate showed step-by-step reasoning. For example, chain-of-thought prompting can produce an explanation or score. A well-tuned model can assign higher scores to answers that include logical justifications.

- **Handling Wrong or Vague Answers:** If an answer is off-topic or nonsensical, we assign a low score. Techniques like perplexity (how well the answer fits a language model) or NER/ontology checks (does it mention relevant entities) help detect garbage. We may also check for **hallucinations**: if the answer contains factual claims, we could use a retrieval step to verify them or flag unlikely assertions.

- **Confidence & Calibration:** Model outputs (softmax probabilities or regression uncertainty) should be calibrated. For instance, if the model is 90% confident an answer is correct, that should empirically match a 90% success rate. Calibration techniques or an auxiliary confidence predictor can be applied.

- **Rubric-based vs Neural:** A fully **rubric-based** approach explicitly encodes scoring rules. For example, the RATAS framework translates rubrics into a decision tree and uses an LLM to fill it

[17]. This yields interpretable scores (points per criterion). By contrast, **neural scoring** (embedding/regression or fine-tuning) learns end-to-end from data. In practice, hybrid solutions work well: e.g. an LLM can be prompted with the rubric to "explain which criteria are met" [17], and then a deterministic score is computed. As Patil et al. observe, purely neural methods can lack transparency, so combining them with rubric structures improves explainability [17].

- **Transformer Scoring Examples:** As reviewed by Kübler et al., transformer-based scoring can be done via zero-shot LLM prompts, embedding+regression, or end-to-end fine-tuning [12] [13]. For instance, Jiang & Bosch used GPT-4 prompting to score short answers (achieving a respectable QWK~0.68 with no training [18]). Embedding-based methods (frozen encoders + neural regressor) have reached top public benchmarks [12]. Fine-tuning a decoder model on labeled data also works well (Ormerod & Kwako achieve QWK~0.76–0.79 by fine-tuning Llama-family models [13]). In our system, we might use several of these in ensemble.

7. Adaptive Question Selection

Deciding *what to ask next* is the heart of adaptivity. In our framework, this is formulated as a **sequential decision problem** (often modeled with reinforcement learning or knowledge tracing). Key elements:

- **State Representation:** The bot's state captures what it "knows" about the candidate. Concretely, this could be a vector of estimated mastery probabilities for each concept or skill (from a BKT or DKT model), plus dialogue context (which questions have been asked, recent correctness). For BKT, the state is simply P(mastered) for each skill [14]. For DKT, the hidden RNN state encodes the entire interaction history [16]. We might also include meta-data like the candidate's declared experience, speed, and confidence signals.

- **Actions:** The action space is "select next question (or topic/difficulty level)." The bot could either pick a specific question ID or choose a high-level action (e.g. "increase difficulty" or "switch topic to X"). In an RL setting, each action corresponds to asking a particular question (from the pool compatible with the state).

- **Reward:** The reward function encourages learning and accuracy. A simple reward is +1 for a correct answer and 0 for wrong (or penalize wrong). More sophisticated rewards might consider the information gained (e.g. how much the candidate's mastery estimate changed) or long-term goals (e.g. maximize final score). Meng et al. (2024) formulate the interview path as an MDP: states = knowledge profile, actions = content selection, reward = learning outcome [19]. In practice, you could define a reward that is the square of the improvement in estimated skill, or tie it to the eventual candidate success metric.

- **Policy & RL:** With this setup, one can train a policy network by RL. For example, Lin & Liu reformulated computerized adaptive testing under RL and found a deep Q-network (DQN) strategy that outperformed traditional information-based item selection [11]. This means the bot could learn, via trial interviews, which questions maximize information about the candidate. Alternatively, policy-gradient methods or contextual bandits (where each "arm" is a topic) could be used. Over many iterations, the model learns to ask easier questions first, then harder ones as mastery increases (a form of *curriculum learning*).

- **Knowledge Tracing:** In tandem, we update the candidate's estimated mastery after each answer. BKT provides a closed-form Bayes update (accounting for slips/guesses) [14]. DKT updates the hidden state via its network. These estimates feed back into the state for the next

action. For instance, if after some questions the candidate's P(mastered) for "recursion" is near 1.0, the bot will shift to another topic.

- **Mastery vs Guessing:** BKT explicitly models guessing errors, but in RL terms we can also check mastery by asking multiple questions in a row. If a candidate gets two varying recursion problems correct, it's unlikely to be luck. We may decrease uncertainty only when answers are consistent. Some implementations add a "confirmatory question" action to verify mastery of a skill.

- **Exploration vs Exploitation:** An adaptive interview must balance exploiting known strengths (asking more questions on a topic already handled well) and exploring unknown weaknesses. This is analogous to the multi-armed bandit trade-off. In practice we might implement an ε-greedy or upper-confidence-bound scheme: e.g. with some probability, try a question from a less-tested skill to gather information. RL inherently learns some exploration (via its stochastic policy or added entropy bonus).

In summary, the bot uses a **policy-learning model** (potentially with RL) to decide each next question. The state includes the candidate's inferred knowledge (via BKT/DKT), and the reward is tied to eliciting informative answers. As Kop et al. explain, one can view the interview path as an MDP where "states represent learner knowledge profiles, actions correspond to content selection, and rewards derive from learning outcomes" [19]. For example, Lin & Liu show that a DQN-based strategy learns to pick items that reduce estimation error better than classic methods [11]. We apply these ideas to dynamically raise or lower difficulty, switch topics, or drill deeper based on the candidate's performance.

8. Conversation Flow & Context Management

Maintaining context over the interview is crucial. We employ both short-term and long-term memory strategies:

- **Short-Term Memory (STM):** Keep the last several turns (questions and answers) in context. This can simply be a buffer passed to the LLM as part of the prompt. For example, when generating a follow-up, we include the previous Q/A pair to maintain continuity. Techniques like Retrieval-Augmented Generation (RAG) could retrieve relevant past exchanges from a database if the context window is limited.

- **Long-Term State:** Beyond the immediate conversation, the bot stores the evolving **knowledge state** (mastery estimates, as in BKT/DKT above). This acts as longer-term memory of what's been covered. Additionally, non-critical facts (e.g. the candidate's name or self-descriptions) could be stored in a database so the bot doesn't re-ask basic info.

- **Avoiding Contradictions:** The bot checks consistency: if the candidate said earlier "I have no Java experience," it won't later ask for Java-specific details. State flags can mark which topics are active or "off-limits." Also, if a question was asked, it won't repeat it. For dialogue, if a user's answer is unclear, the bot may ask for clarification ("Could you elaborate on your approach?"). The LLM's natural language capabilities ensure follow-ups sound human-like, avoiding robotic repetition.

- **Natural Language Flow:** The system uses a conversational prompt style for the LLM (system and user messages) to keep the tone polite and professional. This includes greetings, guiding phrases, and acknowledgement ("Thank you for that answer"). It may even paraphrase a

candidate's answer back ("I understand you solved it with a loop and a dictionary.") to confirm understanding, using LLM summarization.

- **Clarifications & Follow-ups:** If the answer is incomplete or mentions an interesting idea, the bot can spontaneously generate a follow-up question ("You mentioned using a loop; could you explain why you chose that data structure?"). The LLM, given context, can craft these seamlessly.

Overall, the bot continuously updates a memory of the interview. Contradictions are avoided by checking stored state before generating a question. For example, if the candidate's transcript contains an answer, the bot cross-checks it before formulating any reference or next question. This combination of dialogue context and structured state prevents repetitive or contradictory prompts.

9. Training Strategy

The models are trained in stages:

- **Pretraining vs Fine-tuning:** We start with large pretrained language models (e.g. GPT-4, Llama 3, or specialized code models) rather than training from scratch. These models have broad knowledge. We then **fine-tune** them on our domain data (coding and interview transcripts). For instance, a GPT-like model can be fine-tuned on a corpus of technical Q/A to better suit interview dialogue. Similarly, encoder models can be fine-tuned on our labeled answers for better scoring accuracy.

- **Offline Training:** Initially, training is done offline using the collected dataset. For RL components, offline simulations can be run: e.g. simulate many candidate profiles and answers to train the Q-selection policy network by backpropagation through game episodes. Supervised models (skill parser, answer scorer) are trained on labeled examples using standard gradient descent.

- **Online Learning / Human-in-the-Loop:** After deployment, the system can continue learning. Human reviewers may correct model outputs or annotate new data. For example, if the bot's scoring disagrees with a human rater, that example is fed back to refine the model. A/B testing of different policies can be done online (e.g. compare two selection strategies). This enables *continual learning*—periodically re-training models on new interviews to adapt to changing norms or content.

- **Curriculum Learning:** Training itself can use a curriculum. For example, the Q-selection policy might be trained first on easy question distributions, then gradually introduced to harder cases. This mirrors how the bot will adapt difficulty during interviews.

- **Evaluation Metrics:** We measure performance with both ML metrics and interview-specific KPIs. For scoring models, we might use Quadratic Weighted Kappa or Pearson correlation against human scores [12]. For the overall bot, metrics include the accuracy of knowledge estimation (does it predict a user's proficiency?), interview length, candidate satisfaction (through surveys), and fairness scores across demographics. Psychometric-style validation (e.g. whether model's scores agree with expert rubrics) is also applied [12]. We monitor for model drift (e.g. if scoring patterns gradually deviate from expected baseline) [20].

10. Production Concerns

Deploying a live interview bot brings many engineering and compliance issues:

- **Latency & Scaling:** Real-time interactions demand low latency. Models may be large, so we optimize inference (using quantization, distillation, or smaller model variants) to meet response-time budgets (typically <1–2 seconds). We use GPU/TPU for heavy inference (especially Whisper ASR and LLMs), and autoscale these services under load.

- **Cost Control:** Running large models continuously can be expensive. An open-source-first approach helps control costs (no per-query API fees). We may use proprietary APIs (OpenAI, Azure OpenAI, etc.) only in early prototyping or fallback when hosting isn't feasible. Caching common inferences and batching requests can save resources.

- **Monitoring & Drift Detection:** In production, we log all interactions. We set up automated alerts if model outputs deviate (e.g. scoring distribution shifts, embedding-based retrieval fails). Tools like Prometheus or custom analytics track key metrics. As InfoWorld notes, AI-native deployments include drift monitoring to flag performance degradation [20].

- **Security & Privacy:** The system handles sensitive candidate data. All transcripts and answers are securely stored with encryption at rest and in transit. Access controls ensure only authorized components/people see PII. We comply with GDPR and other data laws [10]: candidates must consent to being recorded, and have the "right to be forgotten." We anonymize or discard personally identifying data when not needed. For example, EU law classifies AI recruiting as "high risk," so we implement strict bias controls and data protection [21] [10].

- **Anti-Cheating Measures:** Because candidates might try to game the system (using external AI, plagiarizing answers, etc.), we build countermeasures. We randomize non-critical question order to thwart memorization. We analyze answer patterns (e.g. timing, code copy-paste detection) to flag suspicious behavior. We might include **honeypot questions** (that require creative answers). For coding answers, integration with a plagiarism-check service (like MOSS) can detect identical submissions. These measures reduce the risk of answer memorization or tool-assisted cheating.

- **Compliance & Safety:** Under regulations (e.g. EU AI Act, US EEOC guidelines), AI hiring tools must be fair and explainable. We log decision rationales (e.g. which rubric points were met) to enable audits. The system includes override buttons for human interviewers to step in if needed. As a legal source advises, "algorithms used by AI in recruitment [...] may inadvertently perpetuate bias" and must be carefully guarded [21]. We proactively test for protected-class biases in model outcomes and adjust training if any are found.

11. Example Pseudo-Implementation

Below is a high-level pseudocode and data schema illustrating how the components interact during a live voice interview. This is not full runnable code but outlines the flow:

```
# Load models (Python example)
import whisper, numpy as np
from sentence_transformers import SentenceTransformer
```

```
asr_model = whisper.load_model("medium")   # OpenAI Whisper for ASR
embed_model = SentenceTransformer('all-MiniLM-L6-v2')
# Encoder for semantic retrieval
# (Assume pre-loaded: question_selector, answer_evaluator, state_manager)

# Data schemas (simplified)
class Question:
    def __init__(self, id, text, skills, difficulty, solution):
        self.id = id
        self.text = text
        self.skills = skills      # e.g., ["Python", "OOP"]
        self.difficulty = difficulty  # e.g., 0.7
        self.solution = solution  # reference answer or rubric

class InterviewState:
    def __init__(self, candidate_id, skills_mastery):
        self.candidate_id = candidate_id
        self.skills_mastery = skills_mastery  # dict skill -> P(master)
        self.asked_questions = []             # list of question IDs asked
        self.conversation_history = []        # list of (Q_id, answer_text,
score)
    def update(self, question, answer_text, score):
        # update mastery (e.g. via BKT/DKT)
        # pseudo: self.skills_mastery = update_mastery(self.skills_mastery,
question, score)
        self.asked_questions.append(question.id)
        self.conversation_history.append((question.id, answer_text, score))

# Example interview loop
state = InterviewState(candidate_id=123, skills_mastery={'Python': 0.5, 'ML':
0.3})
while True:
    # Select next question based on current state
    question = question_selector.select_next_question(state)  # could use RL
policy
    print("Bot:", question.text)

    # Capture candidate answer via microphone (here we simulate by loading a
file)
    audio = record_audio_from_microphone(duration=10)  # capture 10 sec audio
    asr_result = asr_model.transcribe(audio)
    answer_text = asr_result["text"]
    print("Candidate:", answer_text)

    # Evaluate answer
    score, confidence = answer_evaluator.evaluate(answer_text,
question.solution)
    print(f"Score: {score:.1f} (conf={confidence:.2f})")

    # Update interview state
    state.update(question, answer_text, score)
```

```
    # Check end condition (e.g. number of questions or time)
    if state.finished():   # e.g., 15 questions or all skills covered
        break
```

Flow Explanation: First, we transcribe the candidate's speech to text using Whisper. Next, the `question_selector` (which may use a neural policy or simple rules) picks the next `Question` object based on `state`. The question is asked (printed or spoken). We record the candidate's answer (audio), transcribe it, and pass the text to `answer_evaluator`. The evaluator could, for example, embed the answer and compare to the solution, or prompt an LLM to score it. We then update `state` (e.g. via a BKT update or neural network) to reflect the new mastery estimates. This loop continues until termination.

Model Flow Diagram (textual):
1. **Skill Initialization:** Candidate provides skills → NLP normalization → initialize `InterviewState` (skill masteries).
2. **Question Selection:** State → question selector model (possibly RL policy) → next question.
3. **Ask Question:** Question.text → spoken to candidate via TTS or displayed in UI.
4. **Receive Answer:** Candidate speaks → audio captured → Whisper ASR → answer text.
5. **Answer Evaluation:** Evaluate(answer text, question.solution) → numerical score + feedback.
6. **State Update:** Use score to update mastery probabilities (BKT/DKT) and conversation history.
7. **Loop:** Return to step 2 until interview end.

Data Interaction: The `Question` objects (from the question DB) come with precomputed embeddings stored in a vector DB. The `question_selector` may query this DB or use a policy network. The `InterviewState` may also be persisted per user in a database, enabling resume if interrupted. All interactions are logged for later analysis.

12. Industry Practices & Real-World Constraints

In practice, fully autonomous adaptive bots are still emerging. Some research prototypes (e.g. Anthropic's "Interviewer" [22]) use a multi-stage pipeline (planning a rubric, conducting interviews, then analyzing transcripts). Major hiring platforms today (HackerRank, Codility, etc.) mostly use static or scripted assessments (often with human oversight) rather than free-form conversational bots.

However, the hiring industry is rapidly evolving. Many companies now **expect AI usage in interviews**. For example, Rippling and Meta allow candidates to use AI tools during coding rounds [23]. A recent survey found **startups** adding AI-assisted interview stages and phasing out "hand-coding" questions [24]. This shift emphasizes *AI-collaboration skills*. At the same time, big companies remain cautious: they still heavily rely on algorithmic questions, though they are re-designing them to defeat simple LLM answers [25]. Interviewers are also changing their style (asking more follow-ups and engaging) to counteract cheating [26].

Feasibility vs. Theory: While the above architecture is theoretically sound, practical systems face pitfalls. LLMs can hallucinate or make subtle mistakes; relying on them for key decisions can be risky. Real-time voice transcription can fail on accents or noise. Maintaining rigorous fairness (per regulatory guidelines) is challenging. For instance, EU rules classify AI hiring tools as "high risk" [21], requiring strict bias audits and candidate consent. Moreover, automated systems must be scrutinized by legal teams – as Greenberg Traurig notes, AI recruiting must not violate anti-discrimination laws or GDPR [21] [10].

Common Failure Modes: Empirically, the bot might incorrectly assess an answer (overestimate or miss a key point), leading to a skewed difficulty trajectory. It might ask an inapplicable question if the skill state is wrong. Network or GPU outages can cause downtime. A poorly calibrated scoring model might demotivate candidates by unfair grading. These are mitigated by extensive testing, fallbacks (e.g. static questions if the bot "gets stuck"), and human-in-the-loop oversight.

In summary, the described system outlines a cutting-edge approach to adaptive technical interviewing, combining ASR (Whisper), transformer-based AI, RL policies, and modern MLOps. It leans on open-source building blocks (Whisper, open LLMs, sentence-transformers, vector DBs) for full control and on-demand fine-tuning. Proprietary services might be used early on (e.g. GPT-4 for bootstrapping plan generation), but the goal is an open, extensible system. By integrating these components with careful data design and compliance checks, one can build a highly technical, adaptive interview bot as described above.

Sources: Industry articles and academic research on AI-driven interviewing and tutoring inform these design principles [5] [2] [4] [6] [12] [13] [14] [16] [19] [11] [17] [9] [21] [22] [23] [25], ensuring the solution reflects real-world practices and cutting-edge methods.

---

[1] [5] Monolithic vs. Microservices Architecture | IBM
https://www.ibm.com/think/topics/monolithic-vs-microservices

[2] [3] [20] Understanding AI-native cloud: from microservices to model-serving | InfoWorld
https://www.infoworld.com/article/4111954/understanding-ai-native-cloud-from-microservices-to-model-serving.html

[4] Vector Databases: Architecture Deep Dive | Medium
https://medium.com/@nay1228/unveiling-the-inner-workings-of-vector-databases-a-technical-deep-dive-eac76f0b1779

[6] Skill Taxonomy 101: The Complete Guide to Building a Future-Ready Tech Workforce
https://prismforce.com/blog/skill-taxonomy

[7] [19] Deep knowledge tracing and cognitive load estimation for personalized learning path generation using neural network architecture - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC12246154/

[8] [9] vikramr.com
http://www.vikramr.com/pubs/ICCV_Interpreting_and_Explaining_Visual_Artificial_Intelligence_Models.pdf

[10] [21] Use of AI in Recruitment and Hiring – Considerations for EU and US Companies | Insights | Greenberg Traurig LLP
https://www.gtlaw.com/en/insights/2025/5/use-of-ai-in-recruitment-and-hiring-considerations-for-eu-and-us-companies

[11] An adaptive testing item selection strategy via a deep reinforcement learning approach - PubMed
https://pubmed.ncbi.nlm.nih.gov/39271633/

[12] [13] [18] Crash course in transformer-era automated scoring
https://psychometrics.ai/llm-based-automated-scoring-overview

[14] [15] Bayesian knowledge tracing - Wikipedia
https://en.wikipedia.org/wiki/Bayesian_knowledge_tracing

[16] stanford.edu
https://stanford.edu/~cpiech/bio/papers/deepKnowledgeTracing.pdf

[17] RATAS: A Generative AI Framework for Explainable and Scalable Automated Answer Grading
https://arxiv.org/html/2505.23818v1

22 Introducing Anthropic Interviewer \ Anthropic
https://www.anthropic.com/research/anthropic-interviewer

23 How Technical Interviews Are Evolving with AI
https://formation.dev/blog/how-technical-interviews-are-evolving-with-ai/

24 25 26 How is AI changing technical interviews at FAANG+? We have the data, and it's not much at all : r/leetcode
https://www.reddit.com/r/leetcode/comments/1nqc0r4/how_is_ai_changing_technical_interviews_at_faang/