Moderate-Level (30) Interview Answers

1. I implemented the Java logic for detecting pupil positions from camera frames and mapping them to UI actions. A tricky bug was inconsistent frame rates across devices causing jitter; I fixed it by normalizing timestamps and buffering coordinates to smooth movements.

2. I used sentence-scoring with frequency thresholds and a minimum sentence-length rule so short but important sentences aren't dropped. Additionally I preserved sentences that contained named entities or keywords detected via simple heuristics.

3. Tokenization sometimes split abbreviations or numeric ranges incorrectly, making sentence scores misleading. I added custom token rules and used regex-based exceptions to preserve meaningful tokens.

4. Use list comprehensions when you need the whole list in memory for fast indexing; choose generator expressions to save memory for large streams. Generators are lazy and avoid building large intermediate lists, improving memory usage.

5. I always use parameterized queries or ORM prepared statements so user text is never concatenated into SQL strings; this prevents SQL injection. Additionally, input length checks and escaping are applied where necessary.

6. Use an ORM for rapid development, maintainability, and object mapping; prefer raw SQL when performing complex queries or heavy JOINs where fine-tuned SQL and indexes outperform ORM-generated queries.

7. I apply a low-pass filter (moving average) and discard outliers beyond a configurable threshold. Combining short-term smoothing with a running calibration reduces jitter while keeping responsiveness.

8. Use ROUGE scores (ROUGE-N, ROUGE-L) for overlap-based evaluation and BLEU or human judgments for quality/usability. Also track user feedback (accept/reject) and average summary length vs. source length.

9. Denormalization helps when read-heavy queries demand fewer JOINs: e.g., caching summary meta (user name, snippet) inside a summaries table for faster listing. It trades some write complexity for faster reads.

10. Extractive summarization picks existing sentences (simpler and faster) while abstractive can paraphrase (more fluent but complex). For production, extractive is safer; abstractive requires ML models and more compute.

11. Analyze the query plan (EXPLAIN), add appropriate indexes on join columns, avoid SELECT *, and consider materialized views if joins are static and expensive. Also review access patterns and

denormalize where needed.

12. Break long text into chunks, summarize each chunk, and then create a meta-summary of chunk summaries. Stream processing and setting per-request time limits prevents memory exhaustion and timeouts.

13. Implement a revisions table that stores original and edited summaries with timestamps and user IDs. This allows undo, auditing, and rollbacks while preserving history without overwriting records.

14. Race conditions can occur when multiple requests attempt to update the same row; mitigate using transactions, row-level locking, or optimistic locking with version fields. Also use a job queue for heavy tasks.

15. Calibration asks the user to look at fixed points on screen to map raw coordinates; store per-user offsets and scale factors. Provide an accessible calibration UI and option to recalibrate when accuracy drops.

16. Use structured logging (JSON logs) and a centralized logging system (e.g., ELK stack) with alerts on error rates. Add metrics (latency, success rates) and set up health checks and exception tracebacks for triage.

17. Allow users to edit a generated summary and save as a new version; collect edits as labeled examples to fine-tune models or adjust heuristics. Implement an import pipeline that periodically retrains scoring weights.

18. CORS blocks cross-origin requests by default; enable CORS on the backend with appropriate allowed origins and restrict methods/headers. Alternatively use a proxy or same-origin deployment to avoid misconfiguration.

19. Precompute sentence features (term frequencies) and use inverted indices to compute scores faster. Also avoid recomputing shared counts per request by caching token frequencies across documents.

20. Sanitize inputs by stripping scripts, validating content types, and scanning for known malware patterns. Store raw content in a protected area with access controls and limit file sizes to prevent abuse.

21. Include keyboard navigability, ARIA labels, contrast ratios, and alternative input methods (e.g., keyboard shortcuts for eye-detection). Provide text-to-speech or high-contrast themes for accessibility.

22. Use limit/offset or keyset pagination for listing summaries; support search with indexed full-text fields (MySQL/MariaDB/InnoDB or dedicated search like Elastic). Return total counts and next/prev links.

23. I would use OpenCV for Java (JavaCV) to process camera frames and detect eye features because it provides robust image processing primitives and community-tested algorithms.

24. Write unit tests for scoring functions and edge cases such as empty input or extremely short sentences. An edge-case test: very long single-paragraph input where summarizer must not crash or exceed memory.

25. Client-side validation gives immediate feedback and reduces bad requests; server-side validation is mandatory for security and correctness. Both together improve UX and prevent malicious input.

26. Example endpoints: POST /summaries {text, userId} to create; GET /summaries/{id} to retrieve; GET /summaries?userId=&page=&q= for listing with pagination and search parameters. Use JSON bodies and proper status codes.

27. Use migration tools (Flyway, Liquibase, or ORM migrations) and write backward-compatible scripts; deploy schema changes in phases (add columns, backfill, then remove deprecated fields) with tests on staging.

28. Use normalized CSS resets, avoid vendor-prefixed behaviors, test on major browsers and screen sizes, and rely on Bootstrap responsive utilities rather than absolute fixed widths. Always test on real devices or emulators.

29. Profile with cProfile or line_profiler to find hotspots, inspect call graphs, and then optimize by algorithmic improvements, caching results, or rewriting heavy loops in vectorized form or using libraries.

30. Sensitive user text requires encryption at rest and in transit, role-based access control, and clear data retention policies (automatic deletion after defined period). Provide an option for users to delete their data and anonymize logs.