Adaptive Authentication Framework: Interview Talking Points

1. Elevator Pitch (60 Seconds)

I built a risk-based authentication system that evaluates login attempts in real time and decides whether to require MFA or let the user through with just a password. It scores each login across four signals: IP reputation, device trust, impossible travel detection, and login time patterns. If the score crosses a threshold, the system forces an OTP challenge. Once the user passes that challenge, the device becomes trusted, so future logins from that device are frictionless. The whole thing is deployed live with a simulation panel so you can trigger different risk scenarios and see the engine respond. I built it with FastAPI on the backend, React on the frontend, and it is deployed on Railway.

---

2. Technical Walkthrough (3 Minutes)

The Problem

Most authentication systems treat every login the same. Either you require MFA every time, which frustrates users, or you skip it entirely, which is a security risk. Adaptive authentication solves this by evaluating the context of each login and adjusting the security requirements accordingly.

The Architecture

The backend is built with FastAPI and SQLAlchemy. When a user submits credentials, the request hits the login endpoint, which calls the risk engine before returning a response. The risk engine runs four independent checks and produces a cumulative score.

The four signals are:

- IP Reputation (+90 points): checks the source IP against a blacklist of known malicious prefixes. In production, this would call an external API such as AbuseIPDB.
- New Device (+105 points): queries the trusted devices table for the user's device fingerprint. If the fingerprint has not been verified through MFA previously, it is flagged.
- Impossible Travel (+150 points): uses the haversine formula to calculate the distance between the current login location and the last one. It divides by elapsed time to get the required travel speed. Anything over 1,000 km/h is flagged.
- Atypical Time (+30 points): compares the current login hour against the user's median login hour from the past 30 days. A deviation of more than 3 hours is flagged.

The Decision

If the score is below 100, the user proceeds with password-only authentication. If it reaches 100 or above, the system generates a 6-digit OTP, hashes it with SHA-256, and stores it with a 5-minute expiry and a 3-attempt limit. The user must verify the OTP to complete the login.

The Adaptive Part

Once the OTP is verified, the device fingerprint is added to the trusted devices table. On the next login from that same device, the new device signal scores zero, and the user is more likely to get through without MFA. The system learns which devices to trust based on successful verification.

The Frontend

The React frontend provides a clean login flow with registration, OTP challenge, and a post-login dashboard that shows the risk score, signal breakdown, and login context. There is also a simulation panel with preset scenarios so that anyone can test the system without needing to understand the backend.

Deployment

FastAPI serves the compiled React build as static files, so the entire application runs as a single service on Railway. SQLite is used for development, but SQLAlchemy abstracts the database layer, making it a single-line configuration change to switch to PostgreSQL.

---

3. Anticipated Questions and Answers

"Why did you build this?"

This started as a technical assessment for a company. I designed the architecture, submitted it, and they went silent. Rather than let the work go to waste, I decided to implement the full system myself. The original submission was a design document with workflow diagrams and pseudocode. The deployed version is a working full-stack application with a live demo.

"Did you use AI to build this?"

Yes, I used AI as a development tool throughout the build process. I designed the architecture and the risk scoring model myself, and I made the technical decisions around the stack, the signal weights, the threshold logic, and the deployment approach. AI assisted with scaffolding code, debugging, and accelerating the implementation. I can walk through any part of the codebase and explain why it works the way it does.

"Why FastAPI over Flask?"

FastAPI auto-generates OpenAPI documentation through Swagger UI. It reads the endpoint definitions and Pydantic schemas and produces interactive API docs without any additional configuration. It also provides built-in type validation and async support. Flask would have required additional libraries to achieve the same functionality.

"How does the impossible travel detection work?"

It uses the haversine formula, which calculates the great-circle distance between two latitude/longitude coordinates. The engine pulls the most recent login with location data, calculates the distance to the current login location, and divides by the elapsed time. If the resulting speed exceeds 1,000 km/h, which is faster than commercial aviation, it flags the login. The threshold accounts for the fact that no legitimate user can travel faster than a commercial flight between logins.

"How does device trust work?"

Device trust is earned, not assumed. When a user logs in from a new device, the system flags it and requires MFA. Once the user successfully verifies the OTP, the device fingerprint is written to the trusted devices table. On subsequent logins from that device, the new device signal returns zero points, reducing the overall risk score. The trust relationship is tied to the combination of user ID and device fingerprint.

"What would you change for production?"

Several things. The IP reputation check currently uses a static prefix list, which would be replaced with a call to an external threat intelligence API such as AbuseIPDB. The OTP delivery currently prints to the console and displays in the UI for demo purposes, but in production it would integrate with an email or SMS service such as SendGrid or Twilio. The device fingerprint currently uses the browser user agent string, which is not robust enough for production. A proper implementation would use a fingerprinting library that accounts for screen resolution, installed fonts, timezone, and other browser attributes. Additionally, the database would be migrated from SQLite to PostgreSQL for concurrent connection handling.

"Why is the new device weight set to 105?"

The threshold is 100. Setting the new device signal to 105 ensures that any login from an unrecognized device will always require MFA, regardless of whether the other signals fire. This was a deliberate design choice. The first time a user logs in from any device, they should be required to verify their identity. Once they do, the device is trusted and future logins are frictionless.

"How do you handle OTP security?"

The OTP is a 6-digit randomly generated code. Before storing it, the system hashes it with SHA-256, following the same principle as password storage. The plaintext OTP is never persisted in the database. The pending authentication record also includes a 5-minute expiration and a maximum of 3 verification attempts. After 3 failed attempts, the OTP is invalidated and the user must request a new one.

"What is the haversine formula?"

It calculates the shortest distance between two points on the surface of a sphere given their latitude and longitude coordinates. It is commonly used in geospatial applications to determine the distance between two locations on Earth. The output is in kilometers, which the engine then uses to calculate the travel speed required between two logins.

"How does the simulation panel work?"

The simulation panel is a set of demo endpoints that allow you to override the login context. Each button sends a request to the backend with preset parameters, such as a blacklisted IP address or Moscow coordinates. The risk engine evaluates these parameters the same way it would evaluate a real login. The seed button populates realistic login history from Milwaukee so that the time-based and travel-based signals have data to work with. It also auto-creates the demo user account if one does not exist.

"What was the hardest part of building this?"

The timezone handling. SQLite stores timestamps as naive datetimes without timezone information, but the risk engine compares them against timezone-aware UTC timestamps. This caused comparison failures in both the OTP expiration check and the impossible travel calculation. I had to add timezone-awareness checks throughout the codebase to handle both naive and aware datetime objects consistently.

---

4. Career Fair Quick Reference

If someone asks about your background and you want to steer toward this project:

"I spent two and a half years at CDW as an IAM Engineer, configuring Okta and Entra ID for Fortune 500 clients. To demonstrate that I understand the underlying architecture and not just the vendor tools, I built an adaptive authentication system from scratch. It scores login risk in real time across four signals and conditionally enforces MFA. It is deployed live and you can test it yourself."

Then hand them your resume or point them to the live demo URL.

Live Demo: adaptive-auth-production.up.railway.app

GitHub: github.com/ryan-t-ramirez/Adaptive-Auth