NuViOLive

INTERVIEW REPORT

Meera Krishnan

Senior Python Backend Engineer

Completed | Rolling 24H | Good



OVERALL SCORE

Good

8.2 / 10 (total 49.0 / 60)

Overview

Overall Summary

Meera demonstrated strong technical expertise in Python backend development, covering FastAPI, PostgreSQL, Redis, Docker, CI/CD, and AWS. Their answers included concrete performance improvements, detailed deployment pipelines, and robust reliability practices. They also showed solid problem-solving skills through a well-documented incident response and systematic query optimization, and communicated clearly throughout. Their experience aligns closely with the job requirements, making them a strong fit for the role.

Recommendation

Good

Proceed to the next interview stage; the candidate's technical depth, problem-solving ability, and alignment with the role make them a strong candidate for hiring.

Recruiter Guidance

Meera Krishnan: **Good.** Proceed to the next interview stage; the candidate's technical depth, problem-solving ability, and alignment with the role make them a strong candidate for hiring.

Score Breakdown

8.0/10
Technical competency

8.0/10
Problem solving

7.0/10
Communication

8.0/10
Role fit

8.0/10
Practical experience

10.0/10
JD match

Score evidence (total 49.0 / 60)

Technical competency — 8.0 / 10

The candidate demonstrated strong technical depth across API design, database optimization, and containerization, providing concrete examples and performance metrics.

Problem solving — 8.0 / 10

The candidate described a clear, systematic incident response process, from containment to root-cause analysis, remediation, communication, and post-mortem improvements.

Communication — 7.0 / 10

The candidate communicated ideas in a structured and articulate manner, using clear language and logical sequencing.

Role fit — 8.0 / 10

The candidate's experience directly matches the role's required technologies and responsibilities, showing capability to deliver on the job's core tasks.

Practical experience — 8.0 / 10

The candidate provided concrete, hands-on examples with measurable outcomes, demonstrating deep practical experience in performance tuning, reliability, and infrastructure.

JD match — 10.0 / 10

The candidate's current role as Senior Software Engineer focusing on backend Python services aligns directly with the Senior Python Backend Engineer position. Job skills: Python: shown in the interview; FastAPI: shown in the interview; PostgreSQL: shown in the interview; REST API design: shown in the interview; Redis caching: shown in the interview; Docker: shown in the interview; CI/CD: shown in the interview; AWS: shown in the interview; Kubernetes: shown in the interview; Kafka: shown in the interview; Observability (Prometheus, Grafana, OpenTelemetry): shown in the interview; System design: shown in the interview. Resume role relevance to this job: same.

Speech & Delivery Metrics (from the transcript)

TRANSCRIPT PACE ESTIMATE	FILLER WORD USAGE	TRANSCRIPT REPLY INTERVAL	SPEECH TONE
112.4 WPM	**0.4% (8 of 1823 words)**	**9.5s**	**Confident & Articulate**

Pace and reply intervals are estimates from transcript timestamps; they include pauses and transcription delay, not audio speech boundaries.

Insights

Strengths

- Extensive hands-on experience with Python, FastAPI, PostgreSQL, Redis, Docker, and AWS.
- Proven ability to design high-performance, scalable APIs and caching strategies.
- Clear, structured communication and systematic problem-solving approach.
- Demonstrated incident response leadership and mentorship of junior engineers.

Areas of Improvement

- Could expand system-design discussions beyond API-level to broader multi-service architecture.
- Would benefit from more examples of leading cross-team initiatives or large-scale project ownership.
- Additional depth on scaling strategies for very large traffic volumes could strengthen the profile.

Risk Flags

- No critical risks identified.

Skill Breakdown

Technical Skills

Python 9.0 / 10

Candidate has six years of Python development experience, indicating expert proficiency.

FastAPI 8.0 / 10

Demonstrated use of FastAPI for async endpoints and performance tuning.

PostgreSQL 8.0 / 10

Showed deep knowledge of PostgreSQL indexing and query optimization.

REST API design 8.0 / 10

Provided a thorough design approach for high-performance REST APIs.

Redis caching 8.0 / 10

Explained cache-aside pattern and invalidation strategy using Redis.

Docker 8.0 / 10

Described multi-stage Docker builds and best practices.

CI/CD 8.0 / 10

Outlined a robust CI/CD pipeline with automated testing and safe deployments.

AWS 7.0 / 10

Works with AWS services such as ECS, ECR, and RDS in daily operations.

Kubernetes 7.0 / 10

Managed Kubernetes workloads on AWS EKS, setting resource requests and limits.

Kafka 7.0 / 10

Implemented reliable event publishing using Kafka and the outbox pattern.

Observability (Prometheus, Grafana, OpenTelemetry) 7.0 / 10

Uses OpenTelemetry for tracing and Prometheus/Grafana for metrics and dashboards.

System design 7.0 / 10

Designed a token-bucket rate limiter, demonstrating system-level thinking.

Behaviour Skills

Industry Awareness 7.0 / 10

Shows awareness of industry-standard tools and user-centric observability practices.

Engineering Mindset 8.0 / 10

Applies data-driven analysis and systematic engineering practices to solve problems.

Attitude 7.0 / 10

Demonstrates a proactive and transparent attitude toward deadlines and quality.

Team Work 8.0 / 10

Actively mentors junior engineers and collaborates through pairing and regular one-on-ones.

Analytical Skill 8.0 / 10

Scored with problem solving (8 of 10). The candidate described a clear, systematic incident response process, from containment to root-cause analysis, remediation, communication, and post-mortem improvements.

Communication and Problem solving are scored once, in the main scores above.

Questions & Answers

Q1. How are you doing today?

ANSWER

I'm doing well, thank you. A little excited, honestly. I've been looking forward to this conversation.

ASSESSMENT

The candidate responded politely and clearly, showing a positive attitude.

Q2. Could you give me a brief self-introduction?

ANSWER

Sure. I'm Meera Krishnan, a backend engineer with about 6 years of experience, almost all of it in Python. For the last 3 years I've been a senior software engineer at Brightline Payments in Kochi, where I own the settlement and reconciliation services that pay out merchants every day. Before that I spent 3 years at TrackFleet, a logistics startup, building the shipment tracking APIs. Day to day I work with FastAPI, PostgreSQL, Redis and Docker on AWS. I spend a good part of my time on performance and reliability work, and I also mentor 2 junior engineers in my team.

ASSESSMENT

Provided a comprehensive self-introduction covering experience, current role, tech stack, and mentorship responsibilities.

Q3. May I begin?

ANSWER

Yes, please go ahead.

ASSESSMENT

Gave a concise affirmative, indicating readiness to proceed.

Q4. How would you design a REST API that has to stay fast under heavy read traffic?

ANSWER

I'd start by looking at the read pattern: which resources are hot, how fresh the data has to be, and what the latency target is, say a p95 under 200 milliseconds. Then a few layers. First, the API itself: small, predictable payloads, cursor-based pagination instead of offsets, and ETag and Cache-Control headers so clients and the CDN can reuse responses. Second, a Redis cache in front of the database for the hottest reads, cache-aside with a short TTL and explicit invalidation when the data changes. Third, the database: the right indexes, and read replicas for heavy reporting queries so they never compete with writes. In FastAPI I keep the endpoints async with a pooled database driver, so a slow call never blocks the workers. And I load test before launch

with Locust. At Brightline the merchant settlement summary went from a p95 of about 900 milliseconds to around 120 after we added a covering index and a Redis cache with roughly an 85% hit rate.

ASSESSMENT
Delivered a thorough REST API design with pagination, caching, indexing, async FastAPI, and performance metrics.

Q5. How do you keep the cache consistent when two writes happen close together?

ANSWER
The main rule I follow is to delete the cache key after the database transaction commits, never before, so a reader can't put the old value back in the gap. The key includes a version, the row's updated-at timestamp, so a stale write can't overwrite a newer one. The short TTL is the safety net if an invalidation is ever lost. For the very hot keys we also use a small lock with Redis SET NX, so when a key expires only one request rebuilds it and we avoid a thundering herd on the database. And we agreed with the product team that a few seconds of staleness was fine for the summary screen. Where it isn't fine, for example a payout status, we simply don't cache and read from the primary.

ASSESSMENT
Explained a solid cache-invalidation strategy using versioning, TTL, and Redis locks.

Q6. How do you find and fix a slow PostgreSQL query in production?

ANSWER
I start with data rather than guesses. pg_stat_statements shows which queries cost the most total time, not just the slowest single run. For the worst one I run EXPLAIN ANALYZE with BUFFERS on a replica, and I look for sequential scans on big tables, row estimates that are far off, and nested loops over large sets. The fixes are usually an index that matches the filter and the sort, a composite or a partial one, rewriting a condition so the index can actually be used, for example removing a function on an indexed date column, or fixing an N+1 pattern coming from the ORM. One real case: our reconciliation query took about 14 seconds. A partial index on merchant and settlement date for pending rows, plus removing a date function from the where clause, brought it down to about 300 milliseconds. Then I protect the fix: a test that counts queries per request, and a Grafana alert on the p95 latency of that endpoint.

ASSESSMENT
Outlined a systematic PostgreSQL performance troubleshooting process with pg_stat_statements, EXPLAIN ANALYZE, and indexing, achieving a 300 ms query.

Q7. Walk me through how you deploy a Python service with Docker and a CI/CD pipeline.

ANSWER
We use a multi-stage Dockerfile: a build stage that installs dependencies from a lock file, and a slim runtime stage that runs as a non-root user. On every pull request GitHub Actions runs ruff, mypy and the test suite against a real PostgreSQL service container. When we merge to main, the pipeline builds the image, tags it with the commit hash, pushes it to ECR and deploys to ECS with a rolling update and health checks, so a bad image never takes traffic. Database migrations run with Alembic as a separate one-off task before the new code goes out, and every migration has to work with both the old and the new version of the code. Risky changes go behind a feature flag. Rolling back is just redeploying the previous image tag, which takes a couple of minutes. A full deploy takes about 12 minutes and we ship several times a week.

ASSESSMENT

Described a complete Docker CI/CD pipeline, including multi-stage builds, testing, image tagging, ECR push, ECS rollout, and rollback.

Q8. How do you change a column that the running version still uses?

ANSWER

With expand and contract. First I add the new column as nullable, so nothing breaks. The next release writes to both columns. Then a background job backfills the old rows in small batches so we don't lock the table. Once the data matches, a release switches the reads to the new column, and only in a later release do we drop the old one. On large tables I create indexes concurrently, and every migration sets a lock timeout, so if it can't get its lock quickly it fails instead of blocking live traffic. It's slower on paper, but we haven't had a migration cause downtime in 2 years.

ASSESSMENT

Presented a safe schema migration approach using expand-and-contract, backfills, and concurrent indexes, with no downtime in two years.

Q9. Tell me about a production incident you handled from start to finish.

ANSWER

Last year, on the Monday after a long weekend, our settlement batch failed halfway and some merchants received only part of their payout. I was on call. My first step was to contain it: I paused the scheduler so nothing else went out. The logs showed that our partner bank had started returning a new response code that our retry logic treated as a success. I wrote a script that compared our ledger with the bank's statement and found the 312 settlements that were affected. Because our payout calls are idempotent, I could re-run exactly those safely. I sent the support team an update every 30 minutes so they could answer merchants. All merchants were paid in full within 4 hours, and there were no double payouts. Afterwards we added an explicit list of success codes, contract tests against the bank's sandbox, and an alert when the ledger and bank totals don't match. I also wrote a blameless post-mortem and walked the whole team through it.

ASSESSMENT

Narrated a full incident response from containment to root-cause analysis, remediation, communication, and post-mortem actions.

Q10. How do you make sure a payment or an order is never processed twice?

ANSWER

Several layers, because money is involved. The client sends an idempotency key with every request, and we store it with a unique constraint together with a hash of the request. If the same key comes again we return the stored response instead of processing again, and if the same key comes with a different body we reject it. Inside the service, the payment row itself has a unique constraint on the merchant and the external reference, so even a bug in the application layer can't create a duplicate. When we publish events to Kafka we use the outbox pattern: the event is written in the same transaction as the payment and sent afterwards, and every consumer keeps a table of processed message ids, so redelivery is harmless. And the daily reconciliation against the bank is the final safety net.

ASSESSMENT

Outlined a multi-layer idempotency strategy with client keys, DB constraints, outbox pattern, and reconciliation, ensuring no double payments.

Q11. Have you run Python services on Kubernetes, and how did you set resource requests and limits?

ANSWER

Yes, at TrackFleet we ran everything on EKS. I set requests from what we actually observed, roughly the median usage under normal load, and I set the memory limit equal to the request, because a Python process that gets killed for memory is better found early than at random. CPU limits I kept looser to avoid throttling. The horizontal pod autoscaler scaled on CPU for the APIs and on queue depth for the workers. We kept readiness and liveness probes separate, so a slow dependency took a pod out of the load balancer without restarting it, handled SIGTERM in the app so in-flight requests finished cleanly, and used pod disruption budgets so node upgrades never took the service down.

ASSESSMENT

Explained Kubernetes resource requests, limits, autoscaling, probes, and disruption budgets, showing operational competence.

Q12. How do you approach observability for a backend service?

ANSWER

I start from what users feel. For each service we track request rate, errors and duration, and we have service level objectives, for example 99.9% of settlement API calls succeeding, with an error budget the team actually looks at. Logs are structured JSON with a request id that flows through every service. Traces come from OpenTelemetry, and metrics go to Prometheus with Grafana dashboards. Alerts fire on symptoms, like the error rate or latency going over the objective, not on every CPU spike. Tracing once showed us that 40% of a request's time went into JSON serialisation; switching to a faster serializer cut that endpoint's latency by about a third.

ASSESSMENT

Provided an observability plan covering SLOs, structured logs, OpenTelemetry tracing, Prometheus metrics, and actionable alerts.

Q13. How do you help them grow?

ANSWER

We have a short one-to-one every week, and I pair with them on the tickets that are tricky rather than the easy ones. In code reviews I explain the why, not just the what, and I link to examples in our own code. The thing that worked best was giving each of them ownership of something real, end to end, with me as a safety net. One of them now leads our refunds module and runs its on-call, which I'm quite proud of.

ASSESSMENT

Described mentorship practices including weekly one-on-ones, pairing, code review guidance, and ownership opportunities.

Q14. What made you choose FastAPI, and how did the migration go?

ANSWER

The settlement service spends most of its time waiting on bank APIs, so async I/O was the main reason. FastAPI also gave us request validation with Pydantic and OpenAPI documentation for free, which our partner team really wanted. We migrated route by route behind the same API gateway, the strangler approach, with contract tests to prove each new route behaved like the old one, and 2 weeks of shadow traffic before switching. Throughput went up about 3 times on the same instances and p95 latency dropped by around 45%. The lesson was a synchronous database driver we had kept at first: it blocked the event loop under load, and moving to an async connection pool fixed it. Since then I measure before and after every change like that.

ASSESSMENT

Justified FastAPI adoption for async I/O and detailed a staged migration with contract tests and performance gains.

Q15. How do you handle it when a deadline conflicts with code quality?

ANSWER

I make the trade-off explicit instead of quietly cutting corners. I go to the product manager with options: for example, we can ship on the date with a smaller scope, or ship the full scope a week later. For our merchant dashboard we launched on time without the export feature, but with full tests, and added export 2 weeks later. I keep a short written list of the risks we accepted. The one thing I don't negotiate is tests and reviews for code that moves money.

ASSESSMENT

Explained a balanced approach to deadline-quality trade-offs, documenting risks and maintaining test coverage for critical code.

Q16. How would you design a rate limiter for a public API?

ANSWER

I'd use a token bucket per API key, kept in Redis, with the check and the decrement done in one Lua script so it stays atomic across many app servers. Each response carries headers for the limit and the remaining requests, and when a client is over the limit it gets a 429 with a Retry-After header. For simple per-minute limits a sliding window counter is enough. I'd also keep a coarse limit at the gateway to absorb abuse before it reaches the application, and decide up front whether the limiter fails open or closed if Redis is unavailable; for a public read API I'd fail open with an alert.

ASSESSMENT

Designed a Redis-backed token bucket rate limiter with atomic Lua script and fallback strategies.

Q17. Do you have any questions for me about the role or the team?

ANSWER

Yes, two questions. What does the on-call rotation look like for this team? And what would success look like for this role in the first 6 months?

ASSESSMENT

Asked relevant questions about on-call rotation and success metrics, showing genuine interest in the role.

Q18. Is there anything else you would like to ask?

ANSWER

No, that's all from my side. Thank you, I really enjoyed this conversation.

ASSESSMENT

Politely concluded the interview, expressing appreciation for the conversation.

Recording

Recording was not part of this interview

The interview type booked did not include a recording of the screen, camera and audio.

Security Detections

No cheating events detected

The browser security monitor did not report suspicious activity for this interview.

Transcript

00:00	**Interviewer**	Hi Meera, this is the AI interviewer for the Senior Python Backend Engineer role at IoTides. Don't worry, this is a relaxed conversation and you can take your time. How are you doing today?
00:10	**Candidate**	I'm doing well, thank you. A little excited, honestly. I've been looking forward to this conversation.
00:16	**Interviewer**	Glad to hear that, Meera. Could you give me a brief self-introduction?
01:09	**Candidate**	Sure. I'm Meera Krishnan, a backend engineer with about 6 years of experience, almost all of it in Python. For the last 3 years I've been a senior software engineer at Brightline Payments in Kochi, where I own the settlement and reconciliation services that pay out merchants every day. Before that I spent 3 years at TrackFleet, a logistics startup, building the shipment tracking APIs. Day to day I work with FastAPI, PostgreSQL, Redis and Docker on AWS. I spend a good part of my time on performance and reliability work, and I also mentor 2 junior engineers in my team.
01:17	**Interviewer**	Thanks, that's a clear picture. I will now move to the main round. May I begin?
01:21	**Candidate**	Yes, please go ahead.
01:32	**Interviewer**	Great. Let's start with API design. How would you design a REST API that has to stay fast under heavy read traffic?
03:01	**Candidate**	I'd start by looking at the read pattern: which resources are hot, how fresh the data has to be, and what the latency target is, say a p95 under 200 milliseconds. Then a few layers. First, the API itself: small, predictable payloads, cursor-based pagination instead of offsets, and ETag and Cache-Control headers so clients and the CDN can reuse responses. Second, a Redis cache in front of the database for the hottest reads, cache-aside with a short TTL and explicit invalidation when the data changes. Third, the database: the right indexes, and read replicas for heavy reporting queries so they never compete with writes. In FastAPI I keep the endpoints async with a pooled database driver, so a slow call never blocks the workers. And I load test before launch with Locust. At Brightline the merchant settlement summary went from a p95 of about 900 milliseconds to around 120 after we added a covering index and a Redis cache with roughly an 85% hit rate.
03:14	**Interviewer**	That's a clear approach. You mentioned invalidating the cache when data changes. How do you keep the cache consistent when two writes happen close together?
04:23	**Candidate**	The main rule I follow is to delete the cache key after the database transaction commits, never before, so a reader can't put the old value back in the gap. The key includes a version, the row's updated-at timestamp, so a stale write can't overwrite a newer one. The short TTL is the safety net if an invalidation is ever lost. For the very hot keys we also use a small lock with Redis SET NX, so when a key expires only one request rebuilds it and we avoid a thundering herd on the database. And we agreed with the product team that a few seconds of staleness was fine for the summary screen. Where it isn't fine, for example a payout status, we simply don't cache and read from the primary.
04:31	**Interviewer**	Thank you. Moving on. How do you find and fix a slow PostgreSQL query in production?

05:58
Candidate

I start with data rather than guesses. pg_stat_statements shows which queries cost the most total time, not just the slowest single run. For the worst one I run EXPLAIN ANALYZE with BUFFERS on a replica, and I look for sequential scans on big tables, row estimates that are far off, and nested loops over large sets. The fixes are usually an index that matches the filter and the sort, a composite or a partial one, rewriting a condition so the index can actually be used, for example removing a function on an indexed date column, or fixing an N+1 pattern coming from the ORM. One real case: our reconciliation query took about 14 seconds. A partial index on merchant and settlement date for pending rows, plus removing a date function from the where clause, brought it down to about 300 milliseconds. Then I protect the fix: a test that counts queries per request, and a Grafana alert on the p95 latency of that endpoint.

06:07
Interviewer

Got it. Walk me through how you deploy a Python service with Docker and a CI/CD pipeline.

07:26
Candidate

We use a multi-stage Dockerfile: a build stage that installs dependencies from a lock file, and a slim runtime stage that runs as a non-root user. On every pull request GitHub Actions runs ruff, mypy and the test suite against a real PostgreSQL service container. When we merge to main, the pipeline builds the image, tags it with the commit hash, pushes it to ECR and deploys to ECS with a rolling update and health checks, so a bad image never takes traffic. Database migrations run with Alembic as a separate one-off task before the new code goes out, and every migration has to work with both the old and the new version of the code. Risky changes go behind a feature flag. Rolling back is just redeploying the previous image tag, which takes a couple of minutes. A full deploy takes about 12 minutes and we ship several times a week.

07:40
Interviewer

That's helpful. You said every migration has to work with the old and the new code. How do you change a column that the running version still uses?

08:37
Candidate

With expand and contract. First I add the new column as nullable, so nothing breaks. The next release writes to both columns. Then a background job backfills the old rows in small batches so we don't lock the table. Once the data matches, a release switches the reads to the new column, and only in a later release do we drop the old one. On large tables I create indexes concurrently, and every migration sets a lock timeout, so if it can't get its lock quickly it fails instead of blocking live traffic. It's slower on paper, but we haven't had a migration cause downtime in 2 years.

08:45
Interviewer

Thank you. Next question. Tell me about a production incident you handled from start to finish.

10:13
Candidate

Last year, on the Monday after a long weekend, our settlement batch failed halfway and some merchants received only part of their payout. I was on call. My first step was to contain it: I paused the scheduler so nothing else went out. The logs showed that our partner bank had started returning a new response code that our retry logic treated as a success. I wrote a script that compared our ledger with the bank's statement and found the 312 settlements that were affected. Because our payout calls are idempotent, I could re-run exactly those safely. I sent the support team an update every 30 minutes so they could answer merchants. All merchants were paid in full within 4 hours, and there were no double payouts. Afterwards we added an explicit list of success codes, contract tests against the bank's sandbox, and an alert when the ledger and bank totals don't match. I also wrote a blameless post-mortem and walked the whole team through it.

10:24
Interviewer

Thank you for walking me through that. How do you make sure a payment or an order is never processed twice?

11:36 **Candidate**	Several layers, because money is involved. The client sends an idempotency key with every request, and we store it with a unique constraint together with a hash of the request. If the same key comes again we return the stored response instead of processing again, and if the same key comes with a different body we reject it. Inside the service, the payment row itself has a unique constraint on the merchant and the external reference, so even a bug in the application layer can't create a duplicate. When we publish events to Kafka we use the outbox pattern: the event is written in the same transaction as the payment and sent afterwards, and every consumer keeps a table of processed message ids, so redelivery is harmless. And the daily reconciliation against the bank is the final safety net.
11:48 **Interviewer**	Understood. Let's move on to the job's skills. Have you run Python services on Kubernetes, and how did you set resource requests and limits?
12:51 **Candidate**	Yes, at TrackFleet we ran everything on EKS. I set requests from what we actually observed, roughly the median usage under normal load, and I set the memory limit equal to the request, because a Python process that gets killed for memory is better found early than at random. CPU limits I kept looser to avoid throttling. The horizontal pod autoscaler scaled on CPU for the APIs and on queue depth for the workers. We kept readiness and liveness probes separate, so a slow dependency took a pod out of the load balancer without restarting it, handled SIGTERM in the app so in-flight requests finished cleanly, and used pod disruption budgets so node upgrades never took the service down.
12:56 **Interviewer**	Thanks. How do you approach observability for a backend service?
13:56 **Candidate**	I start from what users feel. For each service we track request rate, errors and duration, and we have service level objectives, for example 99.9% of settlement API calls succeeding, with an error budget the team actually looks at. Logs are structured JSON with a request id that flows through every service. Traces come from OpenTelemetry, and metrics go to Prometheus with Grafana dashboards. Alerts fire on symptoms, like the error rate or latency going over the objective, not on every CPU spike. Tracing once showed us that 40% of a request's time went into JSON serialisation; switching to a faster serializer cut that endpoint's latency by about a third.
14:06 **Interviewer**	That's clear, thank you. Your resume mentions that you mentor two junior engineers. How do you help them grow?
14:50 **Candidate**	We have a short one-to-one every week, and I pair with them on the tickets that are tricky rather than the easy ones. In code reviews I explain the why, not just the what, and I link to examples in our own code. The thing that worked best was giving each of them ownership of something real, end to end, with me as a safety net. One of them now leads our refunds module and runs its on-call, which I'm quite proud of.
15:02 **Interviewer**	Your resume also mentions that you moved a Flask service to FastAPI at Brightline. What made you choose FastAPI, and how did the migration go?
16:10 **Candidate**	The settlement service spends most of its time waiting on bank APIs, so async I/O was the main reason. FastAPI also gave us request validation with Pydantic and OpenAPI documentation for free, which our partner team really wanted. We migrated route by route behind the same API gateway, the strangler approach, with contract tests to prove each new route behaved like the old one, and 2 weeks of shadow traffic before switching. Throughput went up about 3 times on the same instances and p95 latency dropped by around 45%. The lesson was a synchronous database driver we had kept at first: it blocked the event loop under load, and moving to an async connection pool fixed it. Since then I measure before and after every change like that.
16:17 **Interviewer**	Thank you. How do you handle it when a deadline conflicts with code quality?

17:03 **Candidate**	I make the trade-off explicit instead of quietly cutting corners. I go to the product manager with options: for example, we can ship on the date with a smaller scope, or ship the full scope a week later. For our merchant dashboard we launched on time without the export feature, but with full tests, and added export 2 weeks later. I keep a short written list of the risks we accepted. The one thing I don't negotiate is tests and reviews for code that moves money.
17:11 **Interviewer**	Good. One last technical question. How would you design a rate limiter for a public API?
18:09 **Candidate**	I'd use a token bucket per API key, kept in Redis, with the check and the decrement done in one Lua script so it stays atomic across many app servers. Each response carries headers for the limit and the remaining requests, and when a client is over the limit it gets a 429 with a Retry-After header. For simple per-minute limits a sliding window counter is enough. I'd also keep a coarse limit at the gateway to absorb abuse before it reaches the application, and decide up front whether the limiter fails open or closed if Redis is unavailable; for a public read API I'd fail open with an alert.
18:20 **Interviewer**	Thank you, Meera. You have some time left. Do you have any questions for me about the role or the team?
18:35 **Candidate**	Yes, two questions. What does the on-call rotation look like for this team? And what would success look like for this role in the first 6 months?
18:54 **Interviewer**	Those are good questions. I don't have the details of the on-call rotation or the six-month goals; the hiring team will be able to answer both in the next conversation. Is there anything else you would like to ask?
19:02 **Candidate**	No, that's all from my side. Thank you, I really enjoyed this conversation.
19:13 **Interviewer**	Thank you, Meera. It was great talking with you. That is all for this interview. Please click Leave, then End interview. Goodbye.