DataMigrate AI

Interview Preparation Guide

Alexander Garcia Angus

OKO Investments

Project: MSSQL to dbt Migration SaaS Platform powered by AI agents

1. PROJECT OVERVIEW

DataMigrate AI is an AI-powered SaaS platform that automates the migration of legacy MSSQL databases to modern dbt (data build tool) projects. The platform uses a multi-agent system built with LangGraph and Claude API to analyze database schemas, generate dbt models, and validate the transformations.

Technology Stack

Component	Technology	Purpose
Frontend	Vue.js 3 + TypeScript	Modern reactive UI with type safety
Backend API	FastAPI + Python 3.12	Async REST API with auto-docs
AI Agents	LangGraph + Claude API	Multi-agent workflow orchestration
Database	PostgreSQL + Redis	Persistent storage + caching
Infrastructure	Kubernetes (EKS)	Container orchestration
IaC	Terraform	Infrastructure as Code
Autoscaling	Karpenter	40-60% cost savings

2. ARCHITECTURE INTERVIEW QUESTIONS

Q1: Explain the overall architecture of DataMigrate AI

Answer: DataMigrate AI uses a 3-tier architecture:

1. Frontend (Vue.js 3 + TypeScript): Single-page application with Pinia state management. Users authenticate, create migrations, and monitor progress in real-time with auto-refresh every 10-30 seconds.

2. Backend (FastAPI): Async REST API that handles authentication, migration CRUD operations, and orchestrates LangGraph agents. Uses PostgreSQL for persistent state and Redis for caching.

3. AI Layer (LangGraph + Claude): Multi-agent system with 6 specialized agents: Assessment, Planner, Executor, Tester, Rebuilder, and Evaluator. Each agent has a specific role in the migration workflow.

The frontend communicates with FastAPI via REST, FastAPI orchestrates LangGraph agents, and agents persist their state to PostgreSQL for checkpoint recovery.

Q2: Why did you choose LangGraph over LangChain?

Answer: I chose LangGraph because migrations can take 30+ minutes and require stateful execution:

1. State Persistence: LangGraph provides built-in checkpointing to PostgreSQL. If a migration fails, it can resume from the last checkpoint instead of starting over.

2. Agent Communication: LangGraph has native support for inter-agent message passing, which LangChain lacks.

3. Complex Workflows: LangGraph's graph-based execution allows conditional routing (e.g., if validation fails, route to rebuilder agent).

Trade-off: LangGraph has a steeper learning curve than LangChain, but the benefits for long-running workflows justify it. For simple Q&A; agents, LangChain would be sufficient.

Q3: Why Kubernetes (EKS) instead of ECS Fargate?

Answer: I chose Kubernetes for three strategic reasons:

1. Cost Optimization with Karpenter: Karpenter provides intelligent node autoscaling that saves 40-60% on compute costs compared to standard Cluster Autoscaler. Analysis showed $960-4,200/year savings.

2. Skill Transferability: Kubernetes skills are cloud-agnostic. If OKO Investments expands to Azure or GCP, the same K8s knowledge applies. ECS is AWS-only.

3. Mature Ecosystem: Kubernetes has better horizontal pod autoscaling (HPA), more third-party tools, and stronger community support.

Trade-off: EKS control plane costs $73/month vs ECS Fargate's $0 control plane. But Karpenter savings offset this within 1 month.

3. TECHNICAL IMPLEMENTATION QUESTIONS

Q4: How do you handle failures in long-running migrations?

Answer: I implemented a checkpoint system with three layers of resilience:

1. Agent-Level Checkpoints: Each LangGraph agent saves its state after completing a step. State is persisted to PostgreSQL with the migration_id as the key.

2. Table-Level Progress: The migration tracks which tables have been processed. If interrupted, it resumes from the next unprocessed table, not from the beginning.

3. Retry Logic: Failed tables are retried up to 3 times with exponential backoff (1s, 2s, 4s). If all retries fail, the table is marked as 'failed' but the migration continues with other tables.

Testing: I validated this with chaos engineering - manually killing FastAPI workers and corrupting checkpoint data to ensure graceful degradation.

Q5: Explain your authentication strategy

Answer: DataMigrate AI uses two authentication mechanisms:

1. Session-Based (Frontend): Users log in via Vue.js frontend. FastAPI returns a JWT token stored in httpOnly cookies (prevents XSS). The token expires after 24 hours.

2. API Key-Based (External API): External services use API keys (format: mk_xxxxx). Keys are stored hashed in PostgreSQL using bcrypt. Each key has a rate limit (default: 100 req/hour).

Security Considerations:
- Passwords hashed with bcrypt (cost factor 12)
- JWT signed with HS256 + secret key from environment variables
- API keys never logged (redacted in logs)
- Rate limiting prevents abuse

Q6: How would you optimize a slow SQL query?

Answer: My systematic approach:

1. Identify the Problem: Use EXPLAIN ANALYZE to see the query plan. Look for sequential scans, high cost estimates, or slow actual times.

2. Common Fixes:
- Add indexes on WHERE/JOIN columns
- Fix N+1 queries with eager loading (SQLAlchemy joinedload())
- Avoid SELECT * (specify only needed columns)
- Use query result caching for repeated reads

Real Example from DataMigrate AI: Migration queries were slow because I was loading related model_files in a loop (N+1 problem). I switched to:

```
query = select(Migration).options(joinedload(Migration.model_files))
```

This reduced 100 queries to 1, cutting load time from 5s to 200ms.

4. FRONTEND & USER EXPERIENCE QUESTIONS

Q7: Explain your Vue.js state management strategy

Answer: I use Pinia for centralized state management with two main stores:

1. Auth Store (auth.ts):
- Manages user authentication state (user, token, isAuthenticated)
- Provides login(), logout(), fetchCurrentUser() actions
- Persists token to localStorage for page refresh

2. Migrations Store (migrations.ts):
- Manages migration list and current migration state
- Provides fetchMigrations(), createMigration(), deleteMigration() actions
- Implements polling for real-time updates

Why Pinia over Vuex? Pinia has better TypeScript support, simpler API (no mutations), and is the official recommendation for Vue 3.

Q8: How do you handle real-time updates in the UI?

Answer: I implemented polling with exponential backoff:

Dashboard: Polls every 30 seconds for migration statistics. Longer interval because stats change slowly.

Migrations List: Polls every 10 seconds to show updated status and progress.

Why Polling vs WebSockets? For MVP, polling is simpler and sufficient. WebSockets would add complexity (connection management, reconnection logic) for minimal benefit at current scale. I'd switch to WebSockets at 1000+ concurrent users for lower server load.

Optimization: Polling stops when migration completes (status === 'completed'). This prevents unnecessary API calls.

5. INFRASTRUCTURE & DEVOPS QUESTIONS

Q9: Explain your Terraform infrastructure setup

Answer: I organized Terraform into reusable modules:

Module Structure:
- vpc/ - 3 AZ VPC with public, private, and database subnets
- eks/ - Kubernetes cluster with Karpenter autoscaler
- rds/ - PostgreSQL Multi-AZ with automated backups
- redis/ - ElastiCache for caching and Celery broker
- s3/ - Storage for dbt models and logs

Key Decisions:
1. Multi-AZ deployment for 99.9% uptime SLA
2. Private subnets for EKS (no direct internet access)
3. NAT Gateway for outbound traffic (ECR image pulls)
4. RDS encryption at rest with KMS

Cost Optimization: Dev environment uses t3.small (1 node), Production uses m5.large (3+ nodes with autoscaling).

Q10: What monitoring and observability have you implemented?

Answer: Three-layer observability strategy:

1. Application Metrics (CloudWatch):
- API latency (p50, p95, p99)
- Migration success/failure rates
- Database connection pool usage
- Custom metric: Time per migration phase

2. Infrastructure Metrics (Kubernetes):
- Pod CPU/memory usage
- Node autoscaling events (Karpenter)
- PVC (storage) usage

3. Logging (CloudWatch Logs):
- Structured JSON logs with correlation IDs
- Log levels: ERROR for failures, INFO for events
- Logs retained for 30 days (compliance)

Alerting: SNS alerts for:
- API error rate > 5%
- Database CPU > 80%
- Migration failures

6. PERFORMANCE & SCALABILITY QUESTIONS

Q11: How does your system scale to handle increased load?

Answer: Multi-layer scaling strategy:

1. Application Layer (Kubernetes HPA):
- FastAPI pods scale 2-20 based on CPU > 70%
- Celery worker pods scale 1-10 based on queue depth
- Average scale-up time: 30 seconds

2. Infrastructure Layer (Karpenter):
- Adds EC2 nodes when pod scheduling fails
- Consolidates nodes when underutilized (cost savings)
- Uses spot instances for 70% cost reduction

3. Database Layer:
- RDS read replicas for read-heavy queries
- Connection pooling (max 100 connections)
- Redis caching for frequently accessed data (TTL: 5 minutes)

Bottleneck Monitoring: CloudWatch alarms trigger when scaling is needed. Manual intervention for database vertical scaling (larger instance).

Q12: When would you add Rust microservices?

Answer: I'd add Rust when specific bottlenecks emerge:

Trigger Conditions:
1. API latency p95 > 1 second consistently
2. CPU usage > 80% with optimized Python code
3. Processing > 10,000 migrations/month

Candidate Services for Rust:
1. SQL Parser (5s → 500ms, 10x faster)
2. dbt Compiler (10s → 1s, 10x faster)
3. Schema Validator (2s → 200ms, 10x faster)

Hybrid Strategy: Keep FastAPI for 80% (CRUD, auth, orchestration), add Rust for 20% bottlenecks. FastAPI calls Rust microservices via HTTP.

Expected Benefits: 50-70% cost reduction, 10x performance improvement, better user experience (faster migrations).

7. TESTING & QUALITY ASSURANCE QUESTIONS

Q13: What testing strategy did you implement?

Answer: Multi-layer testing pyramid:

1. Unit Tests (pytest):
- Test individual functions (SQL parsing, dbt generation)
- Mock external dependencies (Claude API, database)
- Coverage target: 80%

2. Integration Tests:
- Test LangGraph agent workflow end-to-end
- Use test database (SQLite in CI, PostgreSQL locally)
- Verify checkpoint recovery

3. E2E Tests (Playwright):
- Test critical user flows (login, create migration, view results)
- Run against staging environment before production deploy

Chaos Testing: Manually kill workers, corrupt checkpoints, simulate network failures to validate resilience.

Q14: How do you ensure code quality?

Answer: Automated quality gates in CI/CD:

1. Linting (Pre-commit hooks):
- Python: black (formatting), flake8 (linting), mypy (type checking)
- TypeScript: ESLint + Prettier
- Pre-commit hooks prevent committing bad code

2. Security Scanning:
- Dependabot for dependency vulnerabilities
- Bandit for Python security issues
- Trivy for Docker image scanning

3. Code Reviews:
- All changes require review before merge
- Checklist: Tests pass, documentation updated, no secrets committed

4. CI Pipeline:
- Lint → Test → Build → Deploy
- Blocks merge if any step fails

8. BEHAVIORAL & PROJECT MANAGEMENT QUESTIONS

Q15: Describe a technical challenge you faced and how you solved it

Challenge: LangGraph agents were failing randomly with 'state not found' errors.

Investigation:
1. Added detailed logging to trace execution flow
2. Discovered race condition: Multiple agents writing to same state key
3. PostgreSQL row locking wasn't sufficient due to async execution

Solution:
1. Implemented Redis distributed lock (SETNX command)
2. Each agent acquires lock before updating state
3. Lock expires after 30 seconds (prevents deadlock)

Result: Zero 'state not found' errors after fix. Migration success rate improved from 85% to 100%.

Learning: Async systems need explicit coordination. Row-level locks aren't enough for distributed systems.

Q16: How do you prioritize features when building a product?

Answer: I use the RICE framework (Reach, Impact, Confidence, Effort):

Example from DataMigrate AI:

High Priority (Built for MVP):
- User authentication (High reach, high impact, high confidence, low effort)
- Basic migration workflow (High reach, critical impact)
- Real-time progress updates (High impact on UX)

Medium Priority (Post-MVP):
- Email notifications (Medium reach, medium impact)
- Advanced filtering (Medium impact, low effort)

Low Priority (Future):
- Rust microservices (Low reach initially, high effort)
- Multi-region deployment (Low reach for beta)

Principle: Deliver core value first, then optimize.

9. COST OPTIMIZATION & BUSINESS QUESTIONS

Q17: How much does it cost to run DataMigrate AI monthly?

Answer: Cost breakdown by environment:

Service	Dev (Monthly)	Production (Monthly)
EKS Control Plane	$73	$73
EC2 Nodes (Karpenter)	$45 (t3.small)	$180-360 (m5.large spot)
RDS PostgreSQL	$25 (t4g.small)	$120 (r6g.large Multi-AZ)
ElastiCache Redis	$15 (t3.micro)	$60 (r6g.large)
S3 Storage	$5	$20
CloudWatch/Logs	$10	$40
Data Transfer	$5	$30
TOTAL	**$178/month**	**$523-703/month**

Cost Optimizations Implemented:
1. Karpenter autoscaling saves 40-60% vs standard EC2
2. Spot instances for non-critical workloads (70% savings)
3. Redis caching reduces database queries by 60%
4. S3 Intelligent-Tiering for automatic archival

Revenue Model: $49/month per user (breakeven at 15 users in production).

10. YOUR 60-SECOND ELEVATOR PITCH

Practice this opening statement for interviews:

"I'm Alexander Garcia Angus, and I built DataMigrate AI - an AI-powered SaaS platform that automates MSSQL to dbt migrations. The platform uses a multi-agent system with LangGraph and Claude API, where 6 specialized agents coordinate to analyze schemas, generate dbt models, and validate transformations.

The frontend is built with Vue.js 3 and TypeScript, providing real-time progress updates. The backend uses FastAPI with PostgreSQL for state management. Infrastructure runs on Kubernetes (EKS) with Karpenter autoscaling, which saves 40-60% on compute costs compared to standard autoscaling.

I implemented a checkpoint system that persists agent state every 30 seconds, allowing migrations to resume after interruptions - critical for using spot instances. The system has achieved 100% success rate on test migrations and is ready for production deployment.

I can walk you through the architecture, explain any technical decisions, or demonstrate the system live. What aspect would you like to dive deeper into?"

11. QUESTIONS TO ASK YOUR INTERVIEWER

Always prepare thoughtful questions. Here are strong technical questions:

• What's your current deployment strategy? (Shows interest in their infrastructure)

• How do you handle database migrations in production? (Technical depth)

• What observability tools do you use for distributed systems? (Shows ops awareness)

• What's your approach to incident response and on-call? (Work-life balance concern)

• How do you balance technical debt with new features? (Product thinking)

• What's your code review process? (Team collaboration)

• How do you make build vs buy decisions? (Business acumen)

• What's the most interesting technical challenge your team is facing? (Shows engagement)

12. QUICK REFERENCE CARD

Memorize these key facts:

Metric	Value
Migration Success Rate	100% (7/7 test models)
Agent Count	6 specialized agents
Average Migration Time	5-30 minutes (depends on size)
Checkpoint Frequency	Every 30 seconds
Cost Savings (Karpenter)	40-60% vs standard autoscaling
Database	PostgreSQL (Multi-AZ)
Cache	Redis (ElastiCache)
Frontend	Vue.js 3 + TypeScript
Backend	FastAPI + Python 3.12
Infrastructure	Kubernetes (EKS) + Terraform
Autoscaling	Karpenter (2.5min scale-up)
API Latency Target	p95 < 500ms
Uptime SLA	99.9% (Multi-AZ)

Remember: Confidence comes from genuine understanding. You built this system, you understand the trade-offs, and you can explain it clearly. Good luck!