Our approach leverages a multi-environment strategy, clearly demarcating Production and Development landscapes, along with a robust CI/CD pipeline to ensure a smooth and reliable software delivery lifecycle.

**Production Environment: Engineered for Reliability and Scale**

The Production environment embodies a serverless-first approach, capitalizing on AWS managed services to minimize operational overhead and maximize resilience.

* **Entry Point and Security:** User traffic originates via movie-api.company.com, resolved through **Route 53 (①)** for highly available DNS. The traffic then encounters **AWS WAF & ACM (②)**. AWS WAF provides a crucial security layer, mitigating common web exploits, while ACM handles TLS certificate management, ensuring secure HTTPS communication. This combination aligns with security best practices for public-facing RESTful services.
* **Load Balancing and API Management:** The **Application Load Balancer (ALB) (③)** acts as the intelligent traffic router, distributing incoming HTTPS requests across our API instances. Its layer-7 awareness allows for content-based routing, enhancing flexibility. Behind the ALB sits **API Gateway (Prod) (④)**. This service is pivotal for managing our RESTful API endpoints. It handles request routing, authentication, authorization (integrating with **Cognito (Prod) (⑤)** for secure identity management), rate limiting, and request/response transformation. API Gateway serves as the single point of entry for our backend services, adhering to the API Gateway pattern.
* **Core Application Logic:** The heart of our service, the **Movie API (ECS Fargate - Prod) (⑥)**, leverages a serverless containerization strategy. ECS Fargate allows us to run Docker containers without the need to manage underlying EC2 instances, promoting operational efficiency and scalability. This aligns with the microservices architecture pattern, where the Movie API likely represents a bounded context. The choice between Java Spring Boot and Node.js for the API implementation would depend on factors like team expertise, performance requirements, and the specific needs of the movie domain. Both frameworks are well-suited for building RESTful APIs, adhering to standards like statelessness, resource-based URLs, and standard HTTP methods.
* **Data Persistence:** **Amazon RDS (PostgreSQL - Prod) (⑦)** provides a managed relational database service, offering high availability, durability, and scalability. PostgreSQL is a robust choice for applications requiring transactional integrity and complex querying capabilities. Secure connection strings and database credentials are managed via **Secrets Manager (Prod) (⑧)**, adhering to the principle of least privilege and eliminating hardcoded secrets.
* **Observability:** Comprehensive monitoring is baked into the architecture using **CloudWatch (Prod) (⑨)** for logs, metrics, and alarms. This enables proactive identification and resolution of issues. **X-Ray (⑪)** provides distributed tracing, allowing us to understand the flow of requests across different services and pinpoint performance bottlenecks. Alarms configured in CloudWatch can trigger notifications via **SNS (Prod) (⑩)** for critical events.
* **Container Registry:** Docker images for the Movie API are stored and managed in **Amazon ECR (⑮)**, providing a secure and scalable private Docker registry. The API instances in ECS Fargate pull their images from this registry.

**Development Environment: Fostering Agility and Rapid Iteration**

The Development environment mirrors the Production setup in terms of core components but with a focus on developer productivity and ease of iteration.

* **Virtualized Development Workstations:** Developers access their development environments via **Amazon Workspaces (⑯)**, providing managed virtual desktops. This centralizes development resources, enhances security, and simplifies environment management. The **Dev Workspace (⑰)** on the Workspace instance hosts local development tools.
* **Local Development and Testing:** Developers can leverage **SQLite (Local) (⑱)** for lightweight local database testing during initial development phases. They can also run the **Movie API (Local Docker) (⑰)** using Docker, allowing for isolated testing and rapid iteration on code changes before deploying to the cloud. Local development often employs the "develop on your workstation" pattern for speed and convenience.
* **Cloud-Based Development Resources:** For integration testing and a closer resemblance to the Production environment, a separate set of AWS resources is provisioned, including **API Gateway (Dev) (④)**, **Cognito (Dev) (⑤)**, **ECS Fargate (Dev) (⑥)**, **Amazon RDS (Dev) (⑦)**, **Secrets Manager (Dev) (⑧)**, and **CloudWatch (Dev) (⑨)**. This allows developers to test their code in an environment that closely mirrors production, reducing the risk of environment-specific issues.

**CI/CD Pipeline: Automating Software Delivery**

Our Continuous Integration and Continuous Delivery (CI/CD) pipeline, built on modern DevOps practices, ensures efficient and reliable software releases.

* **Version Control and Workflow:** **GitHub (⑫)** serves as our central code repository, facilitating collaboration and managing code changes through Git-based workflows.
* **Automation Engine:** **GitHub Actions (⑬)** orchestrates the entire CI/CD process. Upon code push to GitHub, Actions triggers automated build, test, and packaging stages.
* **Infrastructure as Code (IaC):** **AWS CDK (⑭)** is employed for defining and provisioning our AWS infrastructure in a declarative manner using familiar programming languages. This Infrastructure as Code approach ensures consistency, repeatability, and version control of our cloud resources. CDK plays a crucial role in deploying both the Development and Production stacks.
* **Containerization and Registry:** Docker images built by GitHub Actions are pushed to **Amazon ECR (⑮)**. This ensures a consistent deployment artifact across all environments.
* **Deployment Strategies:** The pipeline likely employs deployment strategies such as blue/green deployments or canary releases, orchestrated by CDK and ECS, to minimize downtime and risk during production updates.
* **Monitoring and Feedback:** **CloudWatch (CI Logs) (⑨)** provides visibility into the CI/CD pipeline execution. Notifications about pipeline status and deployment outcomes are sent via **Slack / SNS (⑩)**, ensuring timely feedback to the development and operations teams.

**Key Considerations and Best Practices**

* **RESTful API Design:** The Movie API will adhere to RESTful principles, utilizing standard HTTP methods (GET, POST, PUT, DELETE), stateless interactions, and resource-based URLs. Input validation and proper error handling will be implemented to ensure a robust and user-friendly API. API documentation, likely using OpenAPI (Swagger), will be crucial for both internal and external consumers.
* **Security:** Security is paramount. We've incorporated multiple layers of defense, including WAF, TLS encryption, Cognito for authentication and authorization (likely employing OAuth 2.0 flows), and Secrets Manager for secure credential management. Regular security audits and vulnerability scanning will be essential.
* **Scalability and Performance:** The serverless nature of ECS Fargate and API Gateway allows for automatic scaling based on demand. RDS PostgreSQL can also be scaled vertically and horizontally as needed. Caching strategies at the API Gateway and application levels will be considered to optimize performance.
* **Monitoring and Logging:** Comprehensive monitoring and logging are crucial for operational visibility and troubleshooting. CloudWatch provides centralized logging and metrics, while X-Ray helps trace requests across services. Alerting via SNS ensures timely notification of critical issues.
* **Idempotency:** For critical API operations, idempotency will be implemented to prevent unintended side effects from duplicate requests, especially in distributed systems.
* **Error Handling and Resilience:** Robust error handling mechanisms, including appropriate HTTP status codes and informative error messages, will be implemented. Circuit breaker patterns might be considered to prevent cascading failures in the event of service disruptions.

This architecture provides a solid foundation for the Aetna Interview Project, emphasizing scalability, security, and operational efficiency through the strategic use of AWS managed services and modern software development practices. The clear separation of environments and the automated CI/CD pipeline enable rapid development and reliable deployments, ultimately delivering a high-quality and resilient service.