Final Reflection: ML Sentiment Feedback Loop Project

What Worked Well

Architecture and Design Decisions

Microservices Architecture
Breaking the system into independent microservices—such as inference, feedback, evaluation, retraining, model initialization, model registry, notification, and an API gateway—worked well overall. Each service was designed with a single responsibility, which made the system easier to reason about and maintain. This separation also allowed individual services to scale independently when needed. Using an API Gateway as a single entry point simplified client interactions and helped centralize request routing.

AWS Service Integration
Relying on AWS managed services significantly reduced operational complexity. SageMaker handled model training and inference without requiring us to manage ML infrastructure directly. ECS Fargate allowed us to run containers without provisioning or maintaining EC2 instances, which simplified deployment. S3 provided reliable storage for datasets and model artifacts, while ALB handled traffic routing and load balancing. Overall, this combination allowed us to focus more on system logic rather than infrastructure management.

Hugging Face Integration
Switching from a custom TensorFlow model to a pretrained Hugging Face transformer (DistilBERT) turned out to be one of the most important decisions in the project. The custom model suffered from severe overfitting and failed to generalize well, whereas the pretrained model provided much better performance with significantly less training data. Using Hugging Face models also reduced development time and gave us a more production-ready solution.

Data Pipeline and Feedback Loop Design
The feedback loop design—collecting user feedback, evaluating model performance, and triggering retraining—enabled the system to improve over time. Storing feedback data in S3 and reusing it for incremental retraining helped automate the learning process. This made the system closer to a self-improving pipeline rather than a static ML deployment.

Technical Implementation

Containerization
All services were containerized using Docker, which ensured consistent environments across

development and production. This made it easier to test services locally and deploy them reliably to ECS.

Infrastructure as Code
Terraform was used to provision infrastructure in a reproducible way. Defining infrastructure as code allowed changes to be version-controlled and reduced configuration drift. Organizing Terraform into modular components (ECS, S3, IAM, etc.) made the setup easier to manage and extend.

CI/CD Pipeline
GitHub Actions automated the process of building Docker images, pushing them to ECR, and deploying infrastructure changes using Terraform. This streamlined deployments and reduced the risk of manual errors, especially when making frequent changes.

What Didn’t Work Well

Model Training Challenges

Overfitting Issues
The initial custom TensorFlow model consistently overfit the data, predicting almost exclusively positive sentiment. This happened despite trying several mitigation strategies such as dropout, L2 regularization, class weighting, focal loss, and undersampling. After multiple iterations without meaningful improvement, switching to a pretrained model became necessary.

SageMaker Environment Issues
During training, persistent errors related to model and tokenizer saving occurred in the SageMaker environment, particularly `TypeError: 'builtins.safe_open' object is not iterable`. This required workarounds such as saving artifacts to `/tmp` before copying them to the final output directory, which added unnecessary complexity to the training pipeline.

Hyperparameter Tuning
Finding suitable hyperparameters required repeated experimentation. Each training run took approximately 20–30 minutes, making trial-and-error tuning both time-consuming and costly. This slowed iteration speed and made optimization more difficult.

Deployment and Integration Challenges

Service Communication
Integrating microservices required careful configuration of networking and service discovery. The ALB routing rules and ECS service discovery setup needed multiple revisions before communication worked reliably.

Inference Service Refactoring
After switching from custom TensorFlow models to Hugging Face models, the inference service required significant refactoring. It was originally designed around TensorFlow Serving conventions and had to be updated to handle Hugging Face's input and output formats. This caused temporary deployment instability.

Auto-Deployment Reliability
Automatically deploying new models after training completed proved unreliable when using FastAPI's `BackgroundTasks`. This led to race conditions and deployment failures. A threading-based workaround was implemented, but a production-grade solution such as EventBridge with Lambda would be more appropriate.

Data and Testing Limitations

Limited and Imbalanced Data
The dataset was highly imbalanced, with approximately 93.7% positive samples. Addressing this required aggressive undersampling, which significantly reduced the effective dataset size and limited the model's ability to learn nuanced sentiment patterns.

End-to-End Testing Complexity
Testing the full feedback loop—from prediction to feedback, evaluation, retraining, and redeployment—was difficult to automate. Many test cycles required manual orchestration and long wait times for training jobs to complete.

Microservices-Specific Challenges

Communication and Service Discovery

Configuring AWS Service Discovery required careful DNS and registry setup. Initial connectivity issues were traced back to misconfigured security groups and network policies. Debugging these problems took considerable time.

While the API Gateway simplified external access, it also became a single point of failure. Any issues at this layer affected the entire system. Adding health checks, retries, and circuit breakers would improve resilience.

Handling asynchronous operations was also challenging. Model retraining is inherently long-running, but early implementations relied on synchronous calls. This required additional logic for job status tracking and timeout handling.

Data Consistency and Versioning

Because feedback data and model artifacts were stored in separate S3 locations, ensuring consistency between datasets and deployed models required careful naming conventions and versioning. Tracking which training job produced a deployed model often required manual correlation.

A dedicated model registry with metadata would significantly improve traceability. Similarly, aggregating feedback data required handling duplicates and validating data quality before retraining.

Testing Challenges

Testing microservices interactions required either deploying the entire system (which was expensive) or mocking services (which reduced realism). Finding a balance between accuracy and cost was difficult.

Model testing also depended on live SageMaker endpoints, making unit testing impractical. As a result, most validation occurred through integration testing, which increased development time.

What We Would Do Differently with More Time

Architecture Improvements

An event-driven architecture using EventBridge or SNS/SQS would replace synchronous workflows and polling. This would decouple services and improve reliability, especially for long-running training jobs.

Implementing a full model registry would improve version tracking, enable rollbacks, and support A/B testing. Adding a caching layer such as Redis would reduce latency and inference costs.

Improved monitoring and observability using CloudWatch dashboards, AWS X-Ray, and alerting would provide better insight into system health and performance.

Model Development Improvements

Using SageMaker’s automated hyperparameter tuning would reduce manual experimentation. Data augmentation techniques could increase dataset diversity and improve generalization.

Cross-validation would provide more reliable performance estimates, and a structured evaluation framework would offer deeper insight into model behavior across different sentiment classes.

Development and Operations Improvements

A local docker-compose environment that mirrors production would reduce AWS costs and speed up testing. Adding a staging environment would allow safer deployments.

Expanding automated test coverage and improving documentation—such as API specs, architecture diagrams, and operational runbooks—would make the system easier to maintain and extend.

Cost Optimization and Security

Optimizing SageMaker instance selection and implementing scheduled scaling for ECS services would reduce operational costs. S3 lifecycle policies would help manage storage expenses.

Security could be improved by using AWS Secrets Manager for sensitive configuration, implementing VPC endpoints, and enforcing least-privilege IAM policies.

Conclusion

This project successfully demonstrated how to build a production-oriented ML feedback loop using a microservices architecture and AWS managed services. While we encountered challenges related to model training, service integration, and testing, these issues provided valuable learning opportunities. The most important lesson was the benefit of starting with proven, pretrained models rather than building everything from scratch. This approach saved time, reduced risk, and ultimately led to better results, while still allowing room for future improvements and experimentation.