I built a credit card fraud scoring application using training data from the Kaggle Credit Card Fraud dataset. The system utilizes a FastAPI service on an EC2 instance that loads a trained XGBoost model and a preprocessing pipeline, accepts transactions as JSON, and returns a probability of fraud. In parallel, the same model is used in a Kafka streaming path. A producer streams the raw data; a consumer reads those events, transforms them, scores them, and writes the results. The service and the consumer exports metrics to Prometheus, which Grafana visualizes to see latency, throughput, errors, and health. The whole codebase ships with a GitHub Actions pipeline that runs basic quality checks on every change, packages the repository, and triggers a deployment onto the EC2 host.

The main advantages are simplicity and control. Because everything runs on a single EC2 machine, time to insight is short. Because the interface is FastAPI and the model is XGBoost with joblib artifacts, the code path between data, features, and prediction is transparent and easy to debug. Because Prometheus is scraping metrics from the API, the consumer, there is meaningful observability. Because the CI/CD path uses GitHub Actions, there is seamless deployment. These choices allow the system to confidently operate.

The trade-offs are the flip side of the simplicity. One EC2 instance or a single node Kafka is a single point of failure and cannot scale automatically when load spikes. Configuration lives in a simple environment file rather than a managed secret store, so governance is manual. The monitoring stack shows what is happening but does not yet page when drift degrades model quality. Finally, infrastructure is installed by scripts, which makes recreating the environment elsewhere manual.

In spite of those limits, the platform is a very good foundation. The model serving path is clean, the streaming path uses the same artifacts as the batch path, the metrics are the right ones

for an inference service, and the deployment pipeline is safe, auditable, and fast. The most important thing it does is make experimentation cheap while already pushing toward production habits: observability and versioned releases.

The first area for improvement is reliability and security. The API and consumer should run on an Auto Scaling Group that spans at least two zones so you can survive an instance failure and add capacity when needed. The secrets and configuration should be stored in a more secured manner. The second area is managing Kafka. Adding a schema registry to the stack means both the producer and the consumer agree on the structure and types of events, and can evolve fields over time without breaking downstream scoring. The third area is observability with actionability. Prometheus metrics already include histograms for latency and counters for throughput and errors, so connect them to alerting with sensible objectives. The fourth area is MLOps discipline. Register each trained model with a small registry such as MLflow or the SageMaker Model Registry, record the training data evaluation metrics, and write the chosen threshold and model version into metadata that both the API and the consumer read on startup. Add a drift job that computes feature and prediction drift daily and flags significant changes in Grafana and, when thresholds are exceeded, in Alertmanager. Simple data validation will catch quality issues early and save resources.

The new technology questions are all about picking the smallest move that unlocks the most value. If you want managed inference specifically, a SageMaker Endpoint will autoscale and give you built-in metrics, at the cost of a little less transparency and some re-packaging of the model.

In closing, this application provides a working, observable fraud-scoring pipeline that is easy to reason about and quick to change, and it now needs a few focused upgrades to be production-ready.