VERICLAIM

AI-Powered Motor Insurance Fraud Detection System

Complete Project Report

Phases 1 through 6 | Build, Deployment & Interview Defence

Author: Indeebar Ray

GitHub: github.com/Indeebar/VeriClaim_Github

Hugging Face: huggingface.co/Indeebar/VeriClaim_models

1. Executive Summary

VeriClaim is a production-grade, multi-modal AI system designed to detect fraudulent motor insurance claims. The system combines three independent AI models — computer vision, natural language processing, and machine learning — to analyse vehicle damage photographs and claim details simultaneously, returning a fraud probability score with full explainability.

The project was built end-to-end over 6 phases, from data exploration and model training on Google Colab, to a REST API built with FastAPI, a dark professional frontend built with Streamlit, and deployment on Streamlit Cloud. The system is live and publicly accessible.

Metric	Value
XGBoost CV AUC	0.8464
EfficientNet-B0 Val Accuracy	64.6%
NLP Fraud Patterns	15 patterns + 19 keywords
Training Dataset (Fraud)	15,420 rows — Kaggle vehicle claims
Training Dataset (Damage)	~2,300 images — Kaggle car damage
Model Files Size	best_model.pt: 16.3MB | xgb_fraud_model.pkl: 1MB
Frontend	Streamlit Cloud (free, public URL)
Languages / Frameworks	Python 3.11, PyTorch, FastAPI, Streamlit, XGBoost, SHAP

2. System Architecture

2.1 High-Level Architecture

VeriClaim uses a multi-modal fusion architecture. Three independent AI models process different aspects of a claim simultaneously. Their outputs are fused at the application layer to produce a single fraud probability score.

Layer	Component	Technology	Role
Input	Vehicle Image	PIL / torchvision	Crop, resize, normalise for EfficientNet
Input	Claim Form	Pydantic / dict	31 structured features for XGBoost
Input	Incident Description	Plain text string	Semantic embedding via sentence-transformers
Model 1	Damage Classifier	EfficientNet-B0 + timm	Predicts minor / moderate / severe damage
Model 2	NLP Anomaly Scorer	all-MiniLM-L6-v2	Cosine similarity + keyword scoring
Model 3	Fraud Classifier	XGBoost + SMOTE	Fraud probability 0.0 to 1.0
Explainability	SHAP TreeExplainer	SHAP library	Top 3 risk factor attribution
API	FastAPI	Uvicorn ASGI	REST endpoint POST /api/v1/predict/fraud
Frontend	Streamlit	Python	Dark professional UI with results dashboard
Deployment	Streamlit Cloud	GitHub-connected	Live public URL

2.2 Data Flow

1. User uploads vehicle damage photo and fills claim form in Streamlit.
2. Streamlit calls the three model modules directly (no intermediate API in final deployment).
3. EfficientNet-B0 processes the image and returns severity + confidence.
4. sentence-transformers embeds the description and scores it against 15 fraud patterns.
5. XGBoost receives all 31 claim features and returns fraud_probability.
6. SHAP TreeExplainer computes feature attributions and returns the top 3 risk factors.
7. Results are rendered on the right panel of the Streamlit UI.

2.3 Repository Structure

```
VeriClaim_Github/
  api/                      FastAPI app (main.py, schemas.py, routers/)
  models/
    damage_classifier/      model.py, predict.py, best_model.pt
    claim_nlp/              embed.py, anomaly_score.py, fraud_patterns.json
    fraud_classifier/       predict.py, shap_explain.py, xgb_fraud_model.pkl
  notebooks/               train_damage_classifier.ipynb,
train_fraud_classifier.ipynb
  app.py                   Streamlit frontend
  requirements.txt
  Dockerfile
```

3. Phase-by-Phase Implementation

Phase 1 — Project Setup and Planning

Defined the problem statement: insurance fraud costs the Indian insurance industry billions annually. Designed a multi-modal approach combining three signals — visual damage evidence, semantic anomaly detection, and structured claim statistics — to flag suspicious claims before manual review.

Set up the local project structure, virtual environment, FastAPI skeleton, and GitHub repository. Established the three-model architecture and defined the API schema.

Decision	Rationale
XGBoost for fraud classification	Superior on tabular data vs neural networks. Interpretable via SHAP.
EfficientNet-B0 for damage	Lightweight, pretrained, fast inference. No need for large GPU.
sentence-transformers for NLP	Catches semantic fraud signals without fine-tuning. All-MiniLM-L6-v2 is fast.
FastAPI	Async, auto-documented, production-grade Python REST framework.
Streamlit	Rapid ML UI in pure Python. Ideal for internal investigation tools.

Phase 2 — Model Training (Google Colab)

Phase 2A — Damage Classifier (EfficientNet-B0)

Trained on the Kaggle car-damage-detection dataset (~2,300 images). Used EfficientNet-B0 pretrained on ImageNet via the timm library. Applied transfer learning — froze base layers, trained classification head, then fine-tuned with a low learning rate.

- 20 epochs with early stopping (patience 5)
- Data augmentation: random crop, horizontal flip, colour jitter
- Validation accuracy: ~64.6%
- Note: dataset has binary labels (damaged / undamaged). Severity classes (minor / moderate / severe) were approximated by splitting the damaged class. This is a known limitation.
- Output: best_model.pt (16.3 MB)

Phase 2B — NLP Anomaly Scorer

No training required. Uses the pretrained all-MiniLM-L6-v2 model from sentence-transformers. Implements a dual-layer scoring system:

- Layer 1 — Semantic similarity: embeds the incident description and computes cosine similarity against 15 known fraud pattern embeddings (e.g. spontaneous fire, no witnesses, remote location).

- Layer 2 — Keyword matching: scans for 19 high-risk keywords with weighted scores.
- Final anomaly score = weighted average of both layers, normalised to 0.0 to 1.0.
- The model (~80MB) downloads automatically on first run from Hugging Face.

Phase 2C — Fraud Classifier (XGBoost)

Trained on the Kaggle vehicle-claim-fraud-detection dataset (15,420 rows, 31 features). Applied SMOTE to handle class imbalance (only ~6% of claims are fraudulent in the dataset).

- 5-fold cross-validated AUC: 0.8464
- Features: Month, WeekOfMonth, Make, AccidentArea, Age, Fault, PolicyType, VehicleCategory, VehiclePrice, Deductible, DriverRating, PoliceReportFiled, WitnessPresent, BasePolicy and 17 others
- All categorical features label-encoded before training
- Model saved with joblib as xgb_fraud_model.pkl (1MB) along with feature_cols list
- SHAP TreeExplainer fitted on training data for post-hoc explanation

Phase 3 — FastAPI Backend

Built a production REST API with FastAPI. The API loads all three models at startup using the lifespan context manager (avoids cold-start model loading on each request).

- POST /api/v1/predict/fraud — accepts multipart form data (image + JSON claim data)
- GET /health — returns system status
- GET /docs — auto-generated Swagger UI
- Pydantic schemas for request validation with all 31 XGBoost feature columns
- Model loading functions: load_model(), load_nlp_model(), load_fraud_model(), load_explainer()
- SHAP explanation computed per request and returned as top_shap_factors array

Phase 4 — Integration and Debugging

This phase involved significant debugging to resolve mismatches between the trained model and the API.

Key issue: the XGBoost model was trained on Colab with XGBoost 3.2.0 but the local machine had 1.7.6. The pkl file contained a use_label_encoder attribute that did not exist in 1.7.6. Resolution: extracted the booster from the model using get_booster(), saved it to a version-agnostic JSON format, and loaded it back into a fresh XGBClassifier compatible with the local version.

Second issue: the API was sending the wrong feature names to XGBoost. The training dataset had 31 features like Month, WeekOfMonth, AccidentArea — but the API schema was built for a different dataset. Resolution: updated schemas.py, predict.py, and shap_explain.py to match the exact 31 columns from training.

Verified with a test script hitting the live API endpoint. Result: fraud_probability 0.42, anomaly_score 0.85, SHAP factors working correctly.

Phase 5 — Streamlit Dark UI

Built a professional dark-themed frontend modelled after enterprise fraud investigation tools. Design choices:

- Background #080c14, accent colour #00d4ff (cyan), risk colours red/amber/green
- Fonts: Share Tech Mono (monospace headers), Rajdhani (body text) from Google Fonts
- Two-column layout: left panel for inputs, right panel for results
- Left panel: image uploader, 12 claim form fields, incident description textarea
- Right panel: awaiting state when idle, transforms to results dashboard on analysis
- Results: fraud probability bar with gradient fill, risk level badge, recommendation card, damage severity, NLP anomaly score, flagged keyword tags, SHAP factors with up/down arrows
- Custom CSS injected via st.markdown with unsafe_allow_html=True
- Status footer: pulsing green dot and ANALYSIS COMPLETE indicator

Phase 6 — Deployment

Deployment was the most challenging phase due to environment and version conflicts across local Windows, Render Linux, and Streamlit Cloud environments. The final working architecture:

- Model files (best_model.pt, xgb_fraud_model.pkl) committed directly to the VeriClaim_Github repository
- Streamlit frontend deployed on Streamlit Cloud — connects to GitHub, installs requirements.txt, runs app.py
- Models loaded directly in Streamlit using st.cache_resource — no FastAPI needed for the public deployment
- Absolute paths used for model loading to avoid working directory issues on cloud

4. Issues Faced and Resolutions

Issue	Root Cause	Resolution
XGBoost use_label_encoder error	Model trained on Colab with XGBoost 3.2.0. Local machine had 1.7.6. The pkl stored an attribute that did not exist in older versions.	Extracted booster with get_booster(), saved to JSON format using booster.save_model(), loaded back into fresh XGBClassifier. JSON format is version-agnostic.
Feature column mismatch	API schemas built for wrong dataset. XGBoost expected 31 features like Month, WeekOfMonth but API was sending total_claim_amount, incident_city.	Audited training notebook for exact feature columns. Updated schemas.py, predict.py, and shap_explain.py to match all 31 training columns exactly.
Model file overwritten during setup	A setup script created placeholder model files, overwriting the real 16MB trained model with a 1KB dummy.	User had a backup. Restored real model. Added safeguard: always check file size before running any script that writes to model directories.
Render: Python 3.14 instead of 3.11	Render ignored runtime.txt and used Python 3.14. scikit-learn 1.3.2 has no wheels for 3.14 and tried to compile from source — Cython build failed.	Added PYTHON_VERSION=3.11.0 as environment variable in Render dashboard. Also updated runtime.txt to python-3.11.0 with correct prefix.
Render: Out of memory	Render free tier has 512MB RAM. Loading PyTorch + EfficientNet-B0 + XGBoost + sentence-transformers simultaneously exceeded this limit.	Switched from Render to Hugging Face Spaces (16GB RAM free tier) for backend, then ultimately to direct model loading in Streamlit Cloud which avoids the memory split.
torchvision version not found	torchvision==0.20.0 does not exist on PyPI. Render only had 0.24.0 and above available.	Checked Render build logs for available versions. Updated to torchvision==0.24.0 matching torch==2.9.1.
Streamlit Cloud: FileNotFoundError for model files	Streamlit Cloud working directory is different from the script location. Relative paths like models/damage_classifier/best_model.pt failed.	Used Path(__file__).resolve().parent to get the absolute path of app.py and constructed all model paths relative to it.
huggingface_hub version conflict	transformers required huggingface_hub>=0.34.0 but pip install huggingface_hub==0.20.3 installed to the global user directory instead of venv.	Used python -m pip install from within the activated venv to force installation into the correct site-packages.
Port 8000 already in use	Multiple uvicorn processes from previous sessions held port 8000. New instance could not bind.	Used netstat -ano | findstr :8000 to find the PID, then taskkill /PID <number> /F to release the port.

5. Model Details

5.1 Damage Classifier — EfficientNet-B0

Property	Value
Architecture	EfficientNet-B0 pretrained on ImageNet via timm library
Input	224x224 RGB image, normalised with ImageNet mean/std
Output	3-class softmax: minor / moderate / severe
Training hardware	Google Colab T4 GPU
Epochs	20 with early stopping (patience 5)
Optimiser	Adam, lr=1e-4 (fine-tune), lr=1e-3 (head training)
Validation accuracy	~64.6%
Dataset	anujms/car-damage-detection on Kaggle (~2,300 images)
File	best_model.pt (16.3 MB)
Known limitation	Dataset has binary labels. Severity split is approximate.

5.2 NLP Anomaly Scorer

Property	Value
Base model	all-MiniLM-L6-v2 via sentence-transformers
Model size	~80MB, downloads automatically from Hugging Face on first run
Scoring method	Dual layer: semantic cosine similarity + keyword matching
Fraud patterns	15 semantic patterns (e.g. spontaneous fire, no witnesses, remote location)
High-risk keywords	19 keywords with individual weights (e.g. total loss, explosion, collision while fleeing)
Output	anomaly_score 0.0 to 1.0 + list of triggered_keywords
Training required	None — uses pretrained embeddings

5.3 Fraud Classifier — XGBoost

Property	Value
Algorithm	XGBoost (gradient boosted trees)
Imbalance handling	SMOTE oversampling (only ~6% of claims are fraudulent)
Validation	5-fold cross-validation

CV AUC	0.8464
Features	31 columns including Month, Make, AccidentArea, Fault, Age, DriverRating, PoliceReportFiled, WitnessPresent, BasePolicy and others
Explainability	SHAP TreeExplainer — returns top 3 features with impact scores
Dataset	shivamb/vehicle-claim-fraud-detection on Kaggle (15,420 rows)
Training hardware	Google Colab CPU
File	xgb_fraud_model.pkl (1 MB) — contains model + feature_cols list

6. API Reference

POST /api/v1/predict/fraud

Accepts a multipart form submission with a vehicle image and JSON claim data. Returns a complete fraud analysis.

Field	Type	Description
image	file	Vehicle damage photo (.jpg, .jpeg, .png)
claim_data	JSON string	Serialised ClaimInput object with 31 feature fields

Example response:


```
{ "fraud_probability": 0.72,
  "fraud_flag": true,
  "risk_level": "HIGH",
  "recommendation": "Flag for manual investigation immediately.",
  "damage_severity": "severe",
  "damage_confidence": 0.61,
  "anomaly_score": 0.85,
  "triggered_keywords": ["total loss", "fire", "no witnesses"],
  "top_shap_factors": [
    {"feature": "Year", "impact": -1.46},
    {"feature": "BasePolicy", "impact": 0.78},
    {"feature": "AccidentArea", "impact": 0.49}
  ] }
```


Risk Level Thresholds

Risk Level	Fraud Probability	Recommendation
LOW	Below 40%	Approve after standard review.
MEDIUM	40% to 70%	Assign to senior adjuster for review.
HIGH	Above 70%	Flag for manual investigation immediately.

7. Deployment Architecture

7.1 Final Deployment

Component	Platform	URL	Notes
Frontend + Models	Streamlit Cloud	vericlaim.streamlit.app	Free, GitHub-connected, auto-redeploys on push
Model files	GitHub repo	github.com/Indeebar/VeriClaim_Github	best_model.pt and xgb_fraud_model.pkl committed directly
Model training artifacts	Hugging Face Hub	huggingface.co/Indeebar/VeriClaim_models	Backup copy of trained model files

7.2 Deployment History

The deployment went through several iterations before reaching the final architecture:

- Attempt 1 — Render: Build succeeded but ran out of memory (512MB free tier limit). PyTorch + EfficientNet + sentence-transformers together exceed this.
- Attempt 2 — Render with Python 3.11 forced: Build still failed due to scikit-learn Cython compilation on Python 3.14 (runtime.txt was ignored).
- Attempt 3 — Hugging Face Spaces with Docker: Viable but added unnecessary complexity since FastAPI was only needed locally.
- Final solution — Streamlit Cloud with direct model loading: Removed the FastAPI layer from the public deployment. Models are loaded directly in Streamlit using st.cache_resource. Absolute paths used for all model files. This eliminated all memory splitting and environment issues.

7.3 Local Development Setup

To run locally:

```
git clone https://github.com/Indeebar/VeriClaim_Github.git
cd VeriClaim_Github
python -m venv venv
venv\Scripts\activate   (Windows)
pip install -r requirements.txt
streamlit run app.py
```

8. Interview Defence Guide

This section covers the most likely interview questions and how to answer them confidently.

Q: Why did you choose XGBoost over a neural network for fraud detection?

XGBoost outperforms neural networks on tabular data in most benchmarks. The fraud classification problem has structured features like age, policy type, deductible — exactly the domain where gradient boosting excels. More importantly, XGBoost is interpretable via SHAP. In insurance fraud detection, you cannot just say a claim is suspicious — you must explain which factors drove the decision. A neural network would give you a black-box probability. SHAP on XGBoost gives you specific attributions like BasePolicy had the highest positive impact on fraud score.

Q: Your damage classifier accuracy is only 64.6%. Is that acceptable?

The low accuracy is a known limitation, and it is important to be transparent about it. The training dataset only has binary labels — damaged versus undamaged. There are no ground truth severity labels. The three classes (minor, moderate, severe) were approximated. Given that constraint, 64.6% is reasonable. In the system, the damage severity is one signal among three. The fraud probability is primarily driven by XGBoost on the claim features. The damage model contributes context — a claim for total loss with a photo showing minor scratches is itself a suspicious signal, even if the severity classification is imprecise.

Q: How does the NLP module work without any training?

The all-MiniLM-L6-v2 model was pretrained on hundreds of millions of sentence pairs. It already understands semantic meaning — it knows that total loss and vehicle became inoperable are similar concepts. I leverage this by embedding 15 known fraud patterns and then computing cosine similarity between any new incident description and those patterns. A claim saying the vehicle caught fire spontaneously at 3am in a remote location will score high similarity to multiple fraud patterns without any fine-tuning. The keyword layer adds a second signal for explicit red flag terms.

Q: Why did you use SMOTE?

The fraud dataset is severely imbalanced — only about 6% of claims are fraudulent. If you train directly on this, the model will learn to predict not fraud for everything and achieve 94% accuracy while being useless. SMOTE generates synthetic minority class samples by interpolating between existing fraud cases in feature space. This gives the model balanced exposure to both classes during training and improves recall on the fraud class, which is the more important metric in this domain.

Q: What is SHAP and why did you use it?

SHAP stands for SHapley Additive exPlanations. It is a game theory-based method for explaining the output of any machine learning model. For each prediction, SHAP computes how much each feature contributed to pushing the score above or below the base rate. A positive SHAP value means that feature increased the fraud probability. A negative value means it decreased it. I use SHAP TreeExplainer specifically, which is optimised for tree-based models like XGBoost and runs in polynomial time rather than exponential. This is essential for production use — you cannot have explanation generation taking seconds per request.

Q: Why did you use Streamlit instead of React or Next.js?

VeriClaim is an internal fraud investigation tool, not a consumer-facing product. Streamlit allows building data-heavy ML interfaces in pure Python without context switching between frontend and backend languages. The entire stack — models, API, and UI — is Python. For a portfolio project demonstrating ML engineering skills, this is the right choice. If this were a customer-facing product with thousands of users, React would be more appropriate.

Q: Walk me through what happens when I click Analyse Claim.

When the button is clicked, three things happen in sequence. First, the uploaded image is passed to the EfficientNet-B0 model which runs a forward pass and returns a softmax probability over three damage severity classes. Second, the incident description text is embedded using the sentence-transformers model and compared against 15 fraud pattern embeddings using cosine similarity, combined with a keyword scan. Third, the 31 claim features are passed to the XGBoost model which returns a fraud probability. The SHAP explainer then computes feature attributions for that specific prediction. All results are assembled and rendered in the right panel of the UI.

Q: What would you improve if you had more time?

Four things. First, get a properly labelled severity dataset with ground truth minor/moderate/severe labels to improve the damage classifier accuracy. Second, use proprietary Indian insurance claims data with geographic and temporal fraud patterns specific to the Indian market. Third, add an authentication layer to the API with JWT tokens. Fourth, implement a feedback loop where investigator decisions (confirmed fraud or not) are fed back into the model to continuously improve it.

9. Limitations and Future Work

Limitation	Impact	Future Solution
Damage classifier accuracy 64.6%	Severity prediction unreliable for edge cases	Collect properly labelled severity dataset
XGBoost trained on non-Indian data	Fraud patterns may not match Indian market	Retrain on Indian insurance company data
No authentication on API	Any request can hit the endpoint	Add JWT authentication layer
Render free tier memory limit	Backend cannot be hosted on Render free tier	Use paid tier or containerise on AWS EC2
Single image input	Multi-angle damage harder to assess	Accept multiple images, ensemble damage predictions
Static fraud patterns	New fraud types not detected	Periodic retraining with updated patterns
Streamlit cold start	First load can take 30-60 seconds	Pre-warm instance or upgrade to paid tier

10. Complete Technology Stack

Category	Technology	Version	Purpose
Deep Learning	PyTorch	2.2.2	EfficientNet-B0 training and inference
Deep Learning	torchvision	0.17.2	Image transforms and preprocessing
Deep Learning	timm	0.9.12	EfficientNet-B0 pretrained model
NLP	sentence-transformers	2.7.0	Sentence embedding for anomaly scoring
NLP	transformers	4.40.0	HuggingFace model hub access
ML	XGBoost	1.7.6 (local) / 2.0.3 (cloud)	Gradient boosted fraud classifier
ML	scikit-learn	1.4.2	Label encoding, train/test split
ML	imbalanced-learn	0.12.3	SMOTE oversampling
Explainability	SHAP	0.48.0	TreeExplainer for feature attribution
Data	pandas	2.1.4	Feature engineering and data manipulation
Data	numpy	1.26.4	Numerical operations
API	FastAPI	0.109.0	REST API framework
API	uvicorn	0.27.0	ASGI server for FastAPI
API	pydantic	2.5.3	Request/response schema validation
Frontend	Streamlit	1.32.0	Web UI framework
Image	Pillow	10.4.0	Image loading and preprocessing
Serialisation	joblib	1.5.1	Model file serialisation
Training	Google Colab	T4 GPU	EfficientNet training environment
Deployment	Streamlit Cloud	Free tier	Frontend hosting
Version control	GitHub	-	Source code and model files
Model storage	Hugging Face Hub	-	Backup model file hosting

11. Quick Reference for Interview

Numbers to remember

- XGBoost CV AUC: 0.8464
- EfficientNet-B0 validation accuracy: 64.6%
- Training dataset size: 15,420 rows (fraud), ~2,300 images (damage)
- Number of XGBoost features: 31
- NLP fraud patterns: 15 semantic patterns + 19 keywords
- Model sizes: 16.3MB (PyTorch), 1MB (XGBoost)
- SMOTE reason: only 6% of dataset is fraudulent
- Training platform: Google Colab T4 GPU (damage) + CPU (XGBoost)

Key phrases to use

- Multi-modal fusion architecture
- Transfer learning with frozen base layers
- SHAP TreeExplainer for post-hoc explainability
- SMOTE to handle class imbalance
- st.cache_resource to load models once across Streamlit sessions
- Absolute path resolution using Path(__file__).resolve().parent
- Version-agnostic model serialisation via XGBoost JSON booster format

Repositories

- Main code: github.com/Indeebar/VeriClaim_Github
- Model files on Hugging Face: huggingface.co/Indeebar/VeriClaim_models

---

End of VeriClaim Project Report