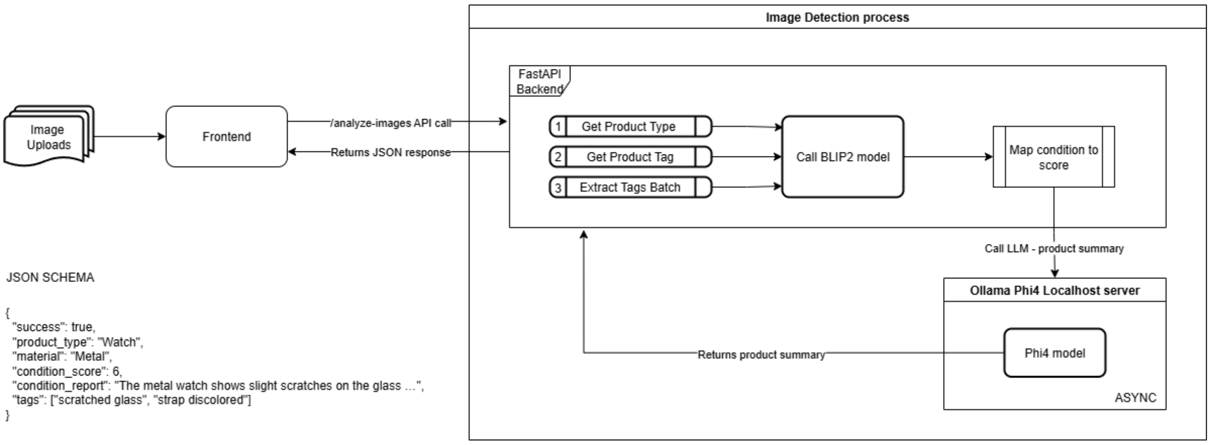
Gluu v0.5 User RWA Image Detection Model – Technical Report

Executive Summary

This report documents the design, implementation, and deployment process for the Gluu user RWA (Real World Assets) - condition assessment service. The system receives one or more product images from a web front-end, detects the product type, material, and visible wear, then returns a condition score and natural-language summary powered by local large-language models (LLMs). The backend is implemented in Python with FastAPI, Hugging Face BLIP-2 for visual reasoning, and Phi-4 (served by Ollama) for text generation; it is hosted on a single AWS EC2 instance for the initial release.

This report and supporting documentation were assisted and partially generated using OpenAI's GPT-4o model to improve clarity, consistency, and architectural planning.

System Overview



Layer	Technology	Purpose
Front-end	Static HTML/JS (S3 + CloudFront in production)	Drag-and-drop image upload UI – calls the `/analyze-images` API
API Layer	**FastAPI** / Uvicorn	Receives images, orchestrates inference pipeline, returns JSON
Vision Model	**`blip2-flan-t5-xl`** (Hugging Face Transformers) - Salesforce	Predict product **type** and **material**; Detect condition **tags**
Text Model	**Phi-4 14B** served locally via **Ollama**	Generates 50-word condition report from model output
Hosting	**AWS EC2** (Ubuntu 22.04)	First-release single-host deployment; supports CPU or GPU

A high-level architecture diagram is included on the project page (see *Architecture Diagram*). The key data flow is:

1. Browser uploads image(s) to FastAPI endpoint `/analyze-images`
2. FastAPI loads the first image into BLIP-2 to classify **product_type** and **material**.
3. FastAPI prompts BLIP-2 once per image (batched) to extract relevant **condition tags**.
4. FastAPI maps tags to a 10-point **condition_score**.
5. FastAPI calls Phi-4 via Ollama at `localhost:11434` to produce a natural-language **condition_report**.
6. JSON response is returned to the front-end and displayed to the user.

Code Base

File: app.py
Frameworks: FastAPI 0.110, Transformers 4.40, Torch 2.2, Pillow, httpx

Key Modules

Function	Responsibility
`analyze_with_blip`	Zero-shot label confirmation for product type/material
`extract_tags_batch`	Single batched BLIP-2 prompt returning all visible condition tags for an image
`map_condition_to_score`	Heuristic mapping of damage tags - 10/8/6/4 score
`call_phi4`	Async call to Ollama / Phi-4, returns streamed text summary
`analyze_images`	FastAPI endpoint; glues the pipeline and returns ReportResponse pydantic model

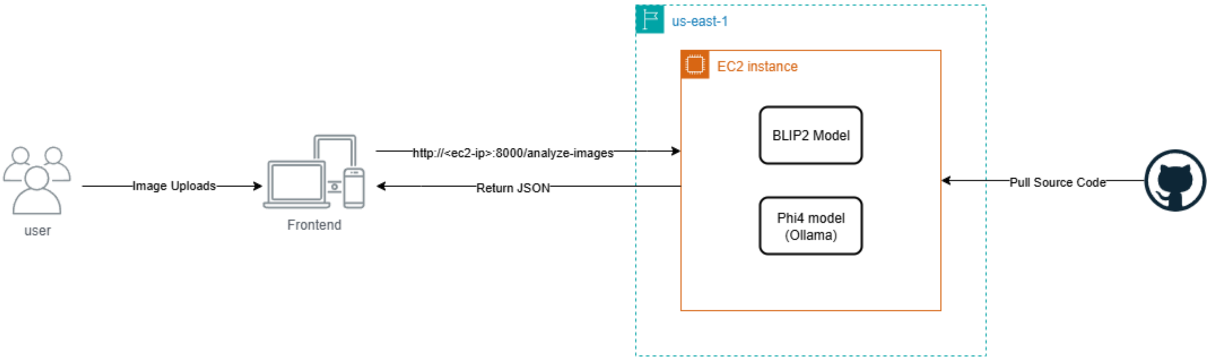
The current codebase contains optimisations over the original proof-of-concept:

- **Batched tag detection** – one BLIP-2 pass per image instead of per tag.
- **Async httpx** for non-blocking LLM calls.
- **Model warm-up** event to reduce first-hit latency.
- Validation on image size and MIME type.

Response Schema

```
{
  "success": true,
  "product_type": "Watch",
  "material": "Metal",
  "condition_score": 6,
  "condition_report": "The metal watch shows slight scratches on the glass …",
  "tags": ["scratched glass", "strap discolored"]
}
```

Deployment Process (AWS)



EC2 Provisioning

Parameter	Recommendation
AMI	Ubuntu 22.04 LTS
Instance Type	*Dev/Test:* `t3.xlarge` (16 GiB RAM, CPU only) *GPU:* `g4dn.xlarge` for real-time
Storage	30 GB gp3 EBS (expandable)
Key Pair	RSA 2048 `.pem` downloaded at launch
Security Group	22/TCP (SSH) from admin IP 8000/TCP (FastAPI) from web **No** ingress to 11434 (Ollama)

Bootstrap Commands

```
# Login
ssh -i key.pem ubuntu@<public-ip>

# System packages
sudo apt update && sudo apt install -y git curl python3-venv

# Ollama install & model preload
curl -fsSL https://ollama.com/install.sh | sh
ollama run phi4:14b-q4_K_M &

# Backend from private GitHub repo (HTTPS)
git clone https://<your-username>:<your-personal-access-
token>@github.com/your-username/your-private-repo.git gluu
cd gluu/backend

# Python venv setup
python3 -m venv venv && source venv/bin/activate
pip install -r requirements.txt

# Run API (development)
uvicorn app:app --host 0.0.0.0 --port 8000
```

For production, wrap Uvicorn in **systemd** or a **process manager** (e.g. `gunicorn -k uvicorn.workers.UvicornWorker`) and optionally front with **NGINX** + **Certbot** for HTTPS.

Testing & Validation

Test	Expected Result
POSTMAN UI request	API reachable on: EC2 public IP on port 8000 with /analyze-images endpoint
Upload sample image	JSON includes correct product/material
Latency - single CPU image	≥ 350 sec (t3.xlarge)
Latency - GPU instance	Not Tested (g4dn.xlarge)

Performance & Scaling

- **CPU-only** OK for low-volume QA; GPU strongly recommended for production.
- Transition path: containerise backend, deploy via **ECS Fargate** with **ECR** images; isolate Ollama on its own GPU node or use alternate LLM such as GPT.
- Future: offload BLIP-2 to Amazon SageMaker or use a lighter version of BLIP for lighter inference.

Security Considerations

- Ollama bound to 127.0.0.1, no external ingress.
- CORS middleware is permissive (*) – tighten in production.
- Implement request size limits (e.g. 5 MB per image) and rate-limiting.
- Store images in /tmp only transiently; consider S3 pre-signed URLs for large uploads.

Future Improvements

1. **Docker-compose** stack (FastAPI + Ollama) for reproducibility.
2. Add **CloudWatch** metrics (latency, error rate).
3. Automatic model warm pool / auto-scaling groups.
4. Fine-tune BLIP-2 on product catalogue for greater accuracy.