OriginHub — Multi-Agent LLM + RAG Innovation Pipeline

Automated Startup Idea Analysis • Retrieval-Augmented Generation • Vertex AI CI/CD • MLOps-Aligned

---

Overview

OriginHub is an AI-powered open innovation platform that transforms raw user problems → validated startup ideas, using:

- A multi-agent LLM architecture
- A Retrieval-Augmented Generation (RAG) system
- Automated evaluation + bias checks
- A full Vertex AI CI/CD pipeline
- Scrapy-based crawling for external idea/problem ingestion

This project follows the instructor's model-pipeline flow:

```
Preprocess → Train → Evaluate → Bias Check → Register / Stop
```

System Architecture

Multi-Agent Pipeline

Each user submission flows through the following agents:

- **InterpreterAgent** — Converts unstructured input to structured JSON
- **ClarifierAgent** — Asks clarifying questions and merges answers
- **RAGAgent** — Retrieves similar ideas via Weaviate
- **ReviewerAgent** — Compares idea to competitors
- **MiniReviewAgent** — Condensed critique
- **StrategistAgent** — MVP, strategy, monetization
- **EvaluatorAgent** — Scores correctness, coherence, and hallucination risk
- **SummarizerAgent** — Produces final deliverable

LLM Stack

We use two optimized models:

- **Qwen 7B** — high reasoning quality
- **Qwen 1.5B** — fast/lightweight

Optimizations include:

- llama.cpp backend
- Q4_K_M quantization
- GPU offloading
- Thread tuning
- Temperature / top-p / token controls
- Structured output mode

ModelManager + InferenceEngine dynamically switch between models.

RAG System (Weaviate)

- Sentence embeddings
- Vector search
- Competitor retrieval
- Stored idea database
- Context injection into prompts

Acts as the preprocessing + data loading stage.

Data Pipeline — Web Crawlers

Scrapy spiders collect real startup/problem data from:

- Hacker News
- Reddit (r/startups)
- ProductHunt (Atom Feed)
- IndieHackers
- TechCrunch

Pipeline:

```
HTML → Clean Text → Vector Embeddings → Weaviate
```

Vertex AI CI/CD Pipeline

A full CI/CD workflow automates model pipeline compilation, validation, and deployment.

Trigger Conditions

Triggered on:

- Push/PR to `main` or `nishitha/ci-cd`

Changes to:

- `Pipeline/pipeline.py`

- `Pipeline/run_pipeline.py`
- `configs/*.yaml`
- The workflow file

Manual runs with:

- `config_file`
- `force_version`
- `skip_deployment`

Job 1: compile-pipeline

Purpose: compile and validate KFP pipeline.

Steps:

- Checkout repo
- Install Python + KFP + aiplatform
- Detect which files changed
- Validate Python syntax
- Compile `pipeline.py` → `slm_vertex_pipeline.json`
- Validate KFP v2 structure
- Upload artifact

Job 2: submit-pipeline

Runs per config file (matrix job).

Steps:

- Authenticate to GCP
- Download compiled pipeline artifact
- Validate YAML config
- Submit Vertex AI pipeline
- Add PR comment with job link

This triggers:

```
Preprocess
→ Train
→ Evaluate (with threshold)
→ Bias Check
→ Model Registry
```

Job 3: notify

Runs always:

- Email on failure

- Email on success
- Includes run metadata

Vertex AI Pipeline Flow (Inside Vertex)

Matches instructor's flow exactly:

1. Preprocess

- Load data
- Build prompts
- Retrieve vectors

2. Train

- Configure Qwen model
- Quantization
- Agent pipeline initialization

3. Evaluate

- EvaluatorAgent
- Threshold check (e.g., F1-like metric ≥ 0.75)

4. Bias Check

- Slice-based diff evaluation (≤ 0.1)

5. Register or STOP

- If both gates pass → register in Vertex Model Registry
- Else → block deployment

Testing Layer

We use full unit + integration testing.

Unit Tests

- ModelManager
- InferenceEngine
- Agents
- PromptBuilder
- Pipeline state transitions

Integration Tests

- Full pipeline runner
- Interactive chat pipeline

CI Enforcement

All tests run on every PR.

Alignment With Instructor's Model Pipeline

Instructor Requirement	Our Implementation
Preprocess	RAG, PromptBuilder, embeddings loading
Train	Qwen config, quantization, agent pipeline
Evaluate	EvaluatorAgent + metric threshold
Bias Check	diff ≤ 0.1, automatic STOP
Register	Vertex Model Registry
CI/CD	GitHub Actions → Vertex AI
Experiment Tracking	Tests + logs (MLflow planned)

Bias Detection Implementation

Overview

The **bias_detection.py** module evaluates model fairness across demographic and contextual slices. It loads a fine-tuned transformer, runs inference, and computes per-slice metrics to identify disparities.

Key Components

GCSArtifactFetcher - Downloads model artifacts and datasets from Google Cloud Storage - Supports both files (.csv) and directories (model weights) - Handles service account credentials automatically - Provides fallback for local paths

BiasEvaluator - Loads pre-trained transformer (distilbert-base-uncased by default) - Runs batched inference with configurable batch size - Computes classification metrics (accuracy, precision, recall, F1) - Evaluates model across metadata slices (ai_group, source_type, region, article_type, org_type, length_bucket)

Slice-Based Fairness Analysis - Breaks dataset into slices by demographic/contextual columns - Computes per-slice metrics independently - Identifies F1 disparities (max F1 - min F1) for each column - Flags bias when disparity exceeds threshold (≤ 0.1 target)

Workflow

1. **Load Configuration**: Parse CLI args or env vars for model/data URIs
2. **Download Artifacts**: Fetch model and dataset from GCS if needed
3. **Run Inference**: Predict on full dataset with confidence scores
4. **Compute Metrics**: Calculate overall accuracy, precision, recall, F1
5. **Evaluate Slices**: Compute metrics for each slice column

6. **Summarize Disparities**: Calculate max F1 range per slice
7. **Generate Report**: Write bias_metrics.json with all results

Output Format


```json
{
  "overall": {
    "accuracy": 0.92,
    "precision": 0.88,
    "recall": 0.91,
    "f1": 0.895
  },
  "slices": {
    "ai_group": [
      {
        "slice_name": "ai_group",
        "slice_value": "high_ai_adoption",
        "count": 245,
        "accuracy": 0.93,
        "precision": 0.90,
        "recall": 0.92,
        "f1": 0.91
      }
    ]
  },
  "disparities": {
    "ai_group": 0.08,
    "source_type": 0.12,
    "region": 0.05
  }
}
```


CLI Usage

```bash
python bias_detection.py \
  --model-uri gs://my-bucket/model/v1/ \
  --data-uri gs://my-bucket/data/bias_dataset.csv \
  --workdir ./artifacts \
  --slice-cols ai_group source_type region \
  --batch-size 32 \
  --verbose
```

Environment Variables

- `MODEL_BUCKET`: GCS bucket for model artifacts
- `MODEL_NAME`: Model identifier
- `MODEL_VERSION`: Model version tag
- `DATA_BUCKET`: GCS bucket for datasets
- `DATA_FOLDER`: Subfolder in DATA_BUCKET
- `DATA_FILE`: CSV filename
- `GOOGLE_APPLICATION_CREDENTIALS`: Path to service account JSON

- `MODEL_TMP_PATH`: Optional cache directory for downloaded weights

Integration with Pipeline

The bias detection module runs as **Job 4** in the Vertex AI pipeline:

- **Triggered after**: EvaluatorAgent produces predictions
- **Input**: Model + dataset with ground truth labels
- **Gate**: If max disparity > 0.1 across any slice → **STOP** (block deployment)
- **Output**: bias_metrics.json logged to GCP
- **Next**: If disparities ≤ 0.1 → proceed to Model Registry

GitHub Actions CI/CD Pipeline

Workflow: Vertex AI Pipeline CI/CD

Triggers on code changes and manual dispatch:

- **Push/PR** to `main` or `nishitha/ci-cd` on:
  - `Pipeline/pipeline.py`
  - `Pipeline/run_pipeline.py`
  - `configs/*.yaml`
  - Workflow file itself
- **Manual trigger** (workflow_dispatch) with inputs:
  - `config_file`: Which config to use
  - `force_version`: Optional version override
  - `skip_deployment`: Boolean to skip submission

Job 1: compile-pipeline

Purpose: Validate, compile, and upload the KFP pipeline JSON.

Steps: 1. Checkout code → fetch repo for diff checks 2. Set up Python → install Python 3.10 3. Install dependencies → kfp, google-cloud-aiplatform, pyyaml 4. Detect changes → identify if pipeline or configs changed (outputs: pipeline-changed, config-changed) 5. Detect config files → outputs config-files (for matrix jobs) 6. Validate syntax → run py_compile on Python files 7. Compile pipeline → execute pipeline.py → slm_vertex_pipeline.json 8. Validate compiled pipeline → check KFP v2 structure 9. Upload artifact → store JSON in GitHub for downstream jobs 10. Create summary → writes status, branch, commit, size to summary

Job 2: submit-pipeline

Purpose: Submit compiled pipeline to Vertex AI.

Depends on: compile-pipeline

Condition: Only if push or workflow_dispatch and not skipping deployment

Matrix strategy: Run for each config file (parallel, max 2)

Steps: 1. Checkout code 2. Set up Python 3. Install dependencies → google-cloud-aiplatform, pyyaml 4. Download compiled pipeline → from previous job artifact 5. Authenticate to GCP → service account key 6. Set up Cloud SDK → gcloud commands 7. Validate config → check required keys (model_name, base_model, data_path, gcs_model_bucket) 8. Submit pipeline job → run run_pipeline.py with GCP parameters 9. Create submission summary → status, project, region, job URL 10. Comment on PR (if PR event) → post status with pipeline link

Job 3: notify

Purpose: Send email notifications.

Depends on: compile-pipeline & submit-pipeline

Condition: Always runs (if: always())

Steps: 1. Send email on failure → uses dawidd6/action-send-mail@v3 with failure details 2. Send email on success → same action with success details

---