Below is a **fully atomic, MCP-ready task breakdown** for building the OSKAR MVP.
Each task is:

- Small
- Testable
- Sequential
- Independent when possible
- Clearly scoped
- Includes acceptance criteria

This is designed so an MCP agent like **Kiro** can execute step-by-step without ambiguity.

🚀 OSKAR MVP — Atomic Engineering Task Breakdown

🔵 PHASE 0 — Environment & Foundation Setup

Task 0.1 — Initialize Repository

Objective: Create project skeleton.

Actions:

- Create directory structure
- Add `requirements.txt`
- Add `.env.example`
- Initialize Git repo

Acceptance Criteria:

- Repo builds with no errors
- Folder structure matches specification

Task 0.2 — Python Environment Setup

Objective: Create reproducible environment.

Actions:

- Create virtual environment
- Install dependencies:
  - fastapi
  - uvicorn
  - torch
  - transformers
  - sentence-transformers
  - faiss-cpu
  - numpy
  - pandas
  - scikit-learn
  - pydantic
  - sqlalchemy
  - psycopg2
  - redis
  - prometheus-client

Acceptance Criteria:

- `uvicorn` runs empty FastAPI server
- No import errors

Task 0.3 — FastAPI Base App

Objective: Boot minimal API.

Actions:

- Create `main.py`
- Create `/health` endpoint

Acceptance Criteria:

- GET `/health` returns `{"status": "ok"}`

🔵 PHASE 1 — Hate Classification Module

Task 1.1 — Load Pretrained Transformer

Objective: Load hate detection model.

Actions:

- Use HuggingFace transformer (multilingual)
- Wrap inference in class `HateClassifier`

Acceptance Criteria:

- Given sample toxic text → returns probability score
- Inference < 300ms on GPU

Task 1.2 — Normalize Output

Objective: Standardize model output.

Output Schema:

```
{
  "label": "hate" | "non_hate",
  "score": 0.0-1.0,
  "confidence": float
}
```

Acceptance Criteria:

- Output always in standardized schema
- No raw logits returned

Task 1.3 — Unit Tests for Hate Module

Objective: Validate correctness.

Actions:

- Create 10 test examples
- Assert expected behavior

Acceptance Criteria:

- ≥80% correct on test cases
- No runtime failures

🔵 PHASE 2 — Claim Detection Module

Task 2.1 — Implement Claim Classifier

Objective: Detect if text contains verifiable claim.

Classes:

- Verifiable
- Opinion
- Subjective

Acceptance Criteria:

- Returns class + probability
- Inference < 300ms

Task 2.2 — Claim Type Classification

If Verifiable → classify into:

- Statistical
- Historical
- Policy
- Scientific

Acceptance Criteria:

- Hierarchical classification works
- Confidence score included

Task 2.3 — Unit Tests for Claim Detection

- Create 15 synthetic examples
- Validate correct routing

🔵 PHASE 3 — Evidence Retrieval Engine

Task 3.1 — Build SBERT Embedding Pipeline

Objective: Generate embeddings for text.

Acceptance Criteria:

- 768-dim vector output
- Inference < 150ms

Task 3.2 — Build FAISS Index

Objective: Create local vector search index.

Actions:

- Index sample Wikipedia subset
- Save FAISS index

Acceptance Criteria:

- Retrieval returns top-5 similar docs
- Retrieval < 50ms

Task 3.3 — Verification Logic

Objective: Compare claim vs retrieved evidence.

Simplified MVP:

- Cosine similarity threshold

Return:

```
{
  "verdict": "supported|refuted|uncertain",
  "confidence": float
}
```

🔵 PHASE 4 — Calibration & Uncertainty Layer

Task 4.1 — Implement Temperature Scaling

- Fit temperature on validation set
- Apply to logits

Acceptance Criteria:

- ECE decreases after scaling

Task 4.2 — Entropy Router

Compute entropy:

```
entropy = -sum(p * log(p))
```

Routing logic:

- 0.8 → human_review
- <0.6 → auto_action
- else → soft_warning

Task 4.3 — Unit Test Routing

- High entropy example → correct route
- Low entropy example → auto route

🔵 PHASE 5 — Trust Engine

Task 5.1 — Database Schema

Create `UserTrust` table:

Field	Type
user_id_hash	string
total_claims	int
correct_claims	int
trust_score	float
last_updated	datetime

Task 5.2 — Implement Bayesian Update

```
alpha = 2 + correct_claims
beta = 2 + total_claims - correct_claims
trust_score = alpha / (alpha + beta)
```

Task 5.3 — Integrate Trust into Risk

```
risk_adjusted = base_risk * (1.5 - trust_score)
```

🔵 PHASE 6 — Risk Fusion Engine

Task 6.1 — Combine Module Outputs

Inputs:

- hate_score
- misinfo_score
- bot_score (placeholder for MVP)

Weighted sum.

Task 6.2 — Monte Carlo Simulation

Sample uncertainty distributions 1000 times.

Return:

```
{
  "mean_risk": float,
  "confidence_interval": [low, high]
}
```

Task 6.3 — Unit Tests for Risk Engine

- Ensure CI bounds valid
- Ensure monotonicity with higher scores

🔵 PHASE 7 — API Integration

Task 7.1 — Create `/analyze` Endpoint

Input:

```
{
  "user_id": "...",
  "text": "...",
  "context_thread": []
}
```

Task 7.2 — Full Pipeline Execution

Sequence:

1. Hate module
2. Claim module
3. Verification
4. Calibration
5. Trust
6. Risk fusion
7. Routing

Task 7.3 — Response Schema

```
{
  "risk_score": float,
  "confidence_interval": [],
  "route": "",
  "components": {},
  "trust_score": float
}
```

🔵 PHASE 8 — Dashboard MVP

Task 8.1 — Build Basic Analytics Panel

Display:

- Total posts analyzed
- % routed to human
- Average risk score
- Calibration ECE

Task 8.2 — Risk Distribution Graph

Histogram of risk scores.

🔵 PHASE 9 — Monitoring & MLOps

Task 9.1 — Prometheus Metrics

Track:

- Inference latency
- Routing distribution
- Error rate

Task 9.2 — Drift Detection (Basic)

Monitor:

- Mean embedding shift
- Claim type distribution change

Alert if deviation > threshold.

🔵 PHASE 10 — Dockerization

Task 10.1 — Create Dockerfile

Include:

- GPU support
- Dependency install
- Uvicorn startup

Task 10.2 — Docker Compose

Services:

- API
- PostgreSQL

- Redis

🔵 Execution Order for MCP

Instruct MCP:

1. Complete Phase 0 entirely.
2. Complete Phase 1 with tests.
3. Do not proceed until tests pass.
4. After Phase 3, validate retrieval quality manually.
5. After Phase 7, validate full pipeline latency.
6. Only then implement dashboard.
7. Deploy containerized build last.