GUNARATHNE M.D.C.H — POLICY ENGINE & DSL IMPLEMENTATION PLAN

(Fully aligned with the current SmartOps system implemented by Peiris)

This plan integrates smoothly with the **exact orchestrator APIs, closed-loop behaviour, telemetry stack, chaos validation tools, and repository structure** that already exist.

It is clean, professional, and direct—ready for your FYP, GitHub issues, or supervisor submissions.

GUNARATHNE M.D.C.H — COMPREHENSIVE IMPLEMENTATION PLAN

Role: Policy Engine & DSL Architect — AI-Governed Automation Layer

Workshare: ~25% of SmartOps

SECTION 1 — ROLE OVERVIEW & OBJECTIVE

Gunarathne owns the **SmartOps Policy Engine**, the component responsible for:

- Translating **human-authored rules** into actionable decision logic
- Combining **AI signals (Anomaly + RCA)** with **policy rules**
- Producing **Action Plans** executed by Peiris's Orchestrator
- Enforcing **guardrails and safety boundaries**
- Providing policy **versioning, auditability, and governance**
- Supporting adaptive automation through **AI-informed decision weighting**

The Policy Engine acts as the **decision brain** between:

ANOMALY + RCA (Kulathunga) → POLICY ENGINE (Gunarathne) → ACTIONS (Peiris)

SECTION 2 — SYSTEM CONTEXT (ALIGNED WITH CURRENT STATE)

Your orchestrator already includes:

- `/v1/actions/execute` → SCALE, RESTART, PATCH
- `/v1/signals/anomaly`
- `/v1/signals/rca`
- Closed-loop worker
- Verification engine
- Prometheus & OTel instrumentation
- Chaos validation capabilities
- ERP Simulator with well-defined chaos signals

Therefore, the Policy Engine must integrate with:

- Peiris's Orchestrator (REST API over internal service-to-service networking)
- Kulathunga's anomaly & RCA signal schemas
- Telemetry stack via Prometheus
- Git-backed policy storage
- Kubernetes cluster-level guardrails

SECTION 3 — HIGH-LEVEL DELIVERABLES (SUMMARY)

Gunarathne must deliver:

1. Policy DSL Language

Readable rules written by humans:

```
WHEN anomaly.type == "latency" AND score > 0.85 THEN restart(service)
WHEN rca.cause == "cpu_saturation" THEN scale(service, +1)
```

2. Policy Interpreter

Converts DSL → Action Plan JSON → Orchestrator `/v1/actions/execute`.

3. Policy Repository

Versioned, Git-backed storage with rollback support.

4. Policy Evaluation Engine

Combines:

- DSL rules
- AI anomaly + RCA signals
- Telemetry metrics (optional)
- Guardrails (max replicas, cooldowns, safelist dispatch)

5. AI-assisted Optimization

Adaptive policy tuning via feedback from the orchestrator’s action outcomes.

6. APIs for Policy Engine

- `/v1/policy/validate`
- `/v1/policy/apply`
- `/v1/policy/evaluate`
- `/v1/policy/execute`

7. Documentation & examples

Full technical and end-user guides.

SECTION 4 — PHASED IMPLEMENTATION PLAN (BEST VERSION)

Aligned with SmartOps' *real* working state.

PHASE 1 — Repository Bootstrap & Integration Contract (Week 1–2)

Objectives

Create the foundation of the Policy Engine and define clear integration points with Orchestrator.

Tasks

✓ Create directory structure:

```
apps/policy-engine/
    dsl/
    interpreter/
    repository/
    runtime/
    schemas/
    tests/
    requirements.txt
```

✔ Define Orchestrator–Policy API contract

Reference Peiris's **ActionRequest** schema exactly:

```
type: "scale" | "restart" | "patch"
target: { namespace, name, kind }
scale: { replicas }
patch: { patch: { ... } }
dry_run: bool
verify: bool
```

✔ Create schema for incoming signals

Matches orchestrator's `/v1/signals/*` payloads.

✔ Setup local dev environment

Python 3.10
FastAPI
Poetry/pip
pytest

PHASE 2 — DSL DESIGN (Week 2–4)

Objectives

Design a human-readable DSL to express recovery rules.

Deliverables

✔ DSL Grammar

Define EBNF grammar (stored in `dsl/grammar.ebnf`):

- Conditions (anomaly.type, anomaly.score, rca.cause, confidence, time windows)
- Operators (==, <, >, `AND`, `OR`)
- Actions (`scale`, `restart`, `patch`)
- Modifiers (`cooldown`, `priority`, `dry_run`)

✓ Parsing Engine

Using **Lark** or **ANTLR4**:

- Parse DSL into AST
- Validate AST structure
- Reject invalid policies

✓ DSL Examples

```
POLICY "restart_on_memory_leak":
  WHEN rca.cause == "memory_leak" THEN restart(service)
  PRIORITY 10
```

PHASE 3 — Policy Repository & Validation Engine (Week 4–6)

Objectives

Implement version-controlled policy storage and semantic validation.

Tasks

✓ Git-backed repository

Features:

- Add policy
- Validate policy
- Version history
- Rollback
- Approval workflow

✓ Semantic validation

Ensure:

- Actions map to real orchestrator actions
- Conditions reference valid AI fields
- Thresholds are safe
- No forbidden actions (deletes, exec, etc.)

✓ REST Endpoints

- `POST /v1/policy/add`
- `GET /v1/policy/version/{id}`
- `POST /v1/policy/validate`

PHASE 4 — Interpreter & Runtime Execution Engine (Week 6–9)

Objectives

Convert AST + signals + telemetry → executable **ActionPlan**.

Key responsibilities

✓ Condition evaluator

Matches:

- anomaly.type
- anomaly.score
- rca.cause
- rca.probability
- service name
- orchestrator namespace

✓ Rule resolution

Precedence:

1. Higher priority policies
2. More specific conditions
3. Recent (non-expired) rules
4. Guardrails (override everything)

✓ ActionPlan generation

Exact JSON schema compatible with orchestrator:

```
{
  "type": "scale",
  "dry_run": false,
  "verify": true,
  "target": {...},
  "scale": { "replicas": 6 }
}
```

✓ API Endpoint

`POST /v1/policy/evaluate` returns an ActionPlan.

PHASE 5 — Guardrails & Safety Enforcement (Week 9–11)

Objectives

Prevent unsafe automation actions before they reach the orchestrator.

Guardrails to implement

✓ Maximum replica limit

From project security requirements:

```
minReplicas = 1
maxReplicas = 10
```

✓ Allowed patch paths only

Whitelist:

- `spec.template.metadata.annotations`
- `spec.replicas`

Reject everything else.

✓ Restart cooldowns

Prevent oscillation:

```
restartCooldownSeconds = 120
```

✓ Policy conflict resolution

If two policies conflict, pick:

- higher priority
- newer version
- more specific condition

✓ Verification alignment

Ensure all actions sent to orchestrator include:

- `verify = true`
- `dry_run = false` (unless uncertain)

PHASE 6 — AI-Assisted Policy Optimization (Week 11–13)

Objectives

Adaptively improve recovery decisions based on outcomes.

Tasks

✓ Collect orchestrator outcomes

Peiris's metrics:

- `smartops_orchestrator_actions_total`
- `smartops_k8s_scale_total`
- `closed_loop_actions_total`

✓ Reward model

Success → positive reward
Failure / timeout → negative reward

✓ Update policy weights

Example:

```
IF restart fails repeatedly for memory_leak:
  THEN escalate to scale + restart
```

✓ Produce updated policy files

PHASE 7 — Full System Integration (Week 13–14)

Objectives

Connect all modules into live SmartOps workflow.

Steps

1. AI (Kulathunga) produces anomaly + RCA signals
2. Policy Engine evaluates DSL rules
3. Produces ActionPlan JSON

4. Sends to Orchestrator `/v1/actions/execute`
5. Peiris's Orchestrator completes action
6. Policy Engine logs result for learning

Integration Tests

- Memory leak chaos → correct policy chosen
- CPU saturation → scale action triggered
- Latency jitter → restart or scale
- Invalid patch → safely rejected

PHASE 8 — Evaluation & Performance Validation (Week 14–15)

Objectives

Validate correctness, speed, and guardrail safety.

Metrics

- Policy evaluation latency (< 3s)
- Policy accuracy (> 90%)
- Guardrail violation count (< 1%)
- MTTR improvement (> 35%)

Artifacts

- `/docs/reports/policy_eval.pdf`
- Prometheus dashboards
- Test results for chaos scenarios

PHASE 9 — Security & Audit Logging (Week 15–16)

Objectives

Ensure safe, traceable policy-driven automation.

Deliverables

✓ Audit logs

For each policy evaluation:

- policy name
- action taken
- decision reason
- timestamp
- verification

✓ RBAC

Minimal verbs for Policy Engine:

- `get`, `list` (pods, deployments)
- no direct scale/restart (only via Orchestrator)

✓ Immutable logs (optional)

Store evaluation logs in a DB (Mongo/Postgres).

PHASE 10 — Documentation & Knowledge Transfer

Deliverables

- **policy_dsl_reference.md**
- **interpreter_design.md**
- **ai_policy_optimizer.md**
- **policy_authoring_guide.md**
- Example policies for:
  - CPU saturation
  - Memory leak
  - Latency jitter
  - Error bursting

FINAL SUMMARY OF GUNARATHNE'S RESPONSIBILITIES

✓ Fully designs DSL

✓ Implements interpreter

✓ Manages policy repository

✓ Evaluates anomaly + RCA signals

✓ Produces orchestrator-ready ActionPlans

✓ Implements guardrails & safety logic

✓ Integrates AI to optimize policy decisions

✓ Works with Peiris (Orchestrator) & Kulathunga (AI)

✓ Ensures end-to-end governed automation

✓ Documents and validates everything end-to-end