ITEM #182 — Code Part 1

Duality of DBM/ACLM Structural Intelligence and LLM Generative Intelligence

Affine-Operation Language + Runtime Evidence Loop

Conversation：DBM ACLM与LLM AI

20251226

很好。谢谢。请为 ITEM #182 配套生成：📐 Operation DSL 接口骨架（Java 8） 🔁 LLM-ACLM 协作 Pipeline 伪代码 / 时序图 🧪 最小 Runtime Evidence Harness 示例

下面给出 **ITEM #182** 的三套配套工程产物：

1. **Operation DSL（Java 8 接口骨架）**
2. **LLM–ACLM 协作 Pipeline 伪代码 + 时序图（ASCII）**
3. **最小 Runtime Evidence Harness 示例（可跑的骨架形态）**

我按你一贯的“可 drop-in、可扩展、可证据化”的风格写：接口清晰、依赖最少、默认实现可跑起来（MVP），并为后续你在 `com.dbm.ACLM.gap` 或相近包内落地留足接口槽位。

A) Operation DSL 接口骨架（Java 8）

```
package com.dbm.aclm.affine;

import java.util.*;

/**
 * ITEM #182 — Affine-Operation Language (Operation DSL)
 * Core idea:
 *   - OperationPlan is a "sentence" in the affine-operation language.
```

```
 *   - Each Operation has typed pre/post conditions and emits runtime
evidence.
 */
public final class OperationDsl {

    private OperationDsl() {}

    // ------------------------------
    // 1) State Model
    // ------------------------------

    /** Minimal state abstraction for DSL execution. */
    public interface IState {
        /** A stable identifier for debugging and evidence linking. */
        String stateId();

        /** Read-only view for feature extraction or invariants. */
        Map<String, Object> view();

        /** Create a mutable copy for applying operations (copy-on-write
style). */
        IMutableState mutableCopy();
    }

    /** Mutable state used by the executor. */
    public interface IMutableState extends IState {
        void put(String key, Object value);
        void remove(String key);
    }

    // ------------------------------
    // 2) Operation Definition
    // ------------------------------

    /**
     * An operation is a typed transform that may be atomic or macro.
     * It must be checkable and evidence-producing.
     */
    public interface IOperation {

        /** Name used in traces and debugging. */
        String name();

        /** Operation category (atomic/macro/domain) for governance/policies.
*/
        default String category() { return "atomic"; }

        /** Parameter schema (for LLM proposals & validators). */
        IParamSchema paramSchema();

        /** Precondition check (fast, deterministic). */
        ICheckResult checkPre(IState x, IOpArgs args);

        /** Apply operation. Must be deterministic given (x,args) for
verifiability. */
        IApplyResult apply(IState x, IOpArgs args);
```

```
        /** Postcondition check (fast, deterministic). */
        ICheckResult checkPost(IState y, IOpArgs args);

        /** Optional: invariants to enforce globally for this operation. */
        default List<IInvariant> invariants() { return
Collections.emptyList(); }
    }

    /** Operation argument bag; supports typed access. */
    public interface IOpArgs {
        boolean has(String key);
        Object get(String key);
        String getString(String key);
        Integer getInt(String key);
        Double getDouble(String key);

        Map<String, Object> asMap();
    }

    /** A simple default args impl. */
    public static final class OpArgs implements IOpArgs {
        private final Map<String, Object> m;

        public OpArgs(Map<String, Object> m) {
            this.m = (m == null) ? new LinkedHashMap<String, Object>() : new
LinkedHashMap<String, Object>(m);
        }

        public static OpArgs of(String k1, Object v1) {
            Map<String, Object> m = new LinkedHashMap<String, Object>();
            m.put(k1, v1);
            return new OpArgs(m);
        }

        @Override public boolean has(String key) { return
m.containsKey(key); }
        @Override public Object get(String key) { return m.get(key); }
        @Override public String getString(String key) { Object v =
m.get(key); return (v == null) ? null : String.valueOf(v); }
        @Override public Integer getInt(String key) { Object v = m.get(key);
return (v instanceof Number) ? ((Number) v).intValue() : null; }
        @Override public Double getDouble(String key) { Object v =
m.get(key); return (v instanceof Number) ? ((Number) v).doubleValue() :
null; }
        @Override public Map<String, Object> asMap() { return
Collections.unmodifiableMap(m); }
    }

    // ------------------------------
    // 3) Schema / Types
    // ------------------------------

    public interface IParamSchema {
        List<IParam> params();
        default String toCompactString() { return params().toString(); }
    }
```

```
    public interface IParam {
        String name();
        ParamType type();
        boolean required();
        default String doc() { return ""; }
    }

    public enum ParamType {
        STRING, INT, DOUBLE, BOOL, JSON, LIST, MAP
    }

    // ------------------------------
    // 4) Preconditions / Postconditions / Invariants
    // ------------------------------

    public interface IInvariant {
        String name();
        ICheckResult check(IState s);
    }

    public interface ICheckResult {
        boolean ok();
        String reason(); // short, stable string (for governance & LLM
feedback)
        Map<String, Object> details();
    }

    public static final class CheckResult implements ICheckResult {
        private final boolean ok;
        private final String reason;
        private final Map<String, Object> details;

        public CheckResult(boolean ok, String reason, Map<String, Object>
details) {
            this.ok = ok;
            this.reason = reason;
            this.details = (details == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(details);
        }

        public static CheckResult ok() { return new CheckResult(true, "OK",
null); }
        public static CheckResult fail(String reason) { return new
CheckResult(false, reason, null); }

        @Override public boolean ok() { return ok; }
        @Override public String reason() { return reason; }
        @Override public Map<String, Object> details() { return
Collections.unmodifiableMap(details); }
    }

    // ------------------------------
    // 5) Apply Result + Evidence
    // ------------------------------

    public interface IApplyResult {
        boolean success();
```

```
        IState outputState();       // Y
        IEvidence evidence();       // runtime evidence payload
        String failureReason();     // stable reason token
    }

    public interface IEvidence {
        /** Unique id to chain evidence across stages. */
        String evidenceId();

        /** Human-readable trace steps (bounded). */
        List<String> trace();

        /** State diff summary (bounded). */
        Map<String, Object> diff();

        /** Coverage/properties/invariants results (bounded). */
        Map<String, Object> metrics();

        /** Optional: link to contribution accounting (tier weights, costs).
*/
        Map<String, Object> contribution();
    }

    public static final class Evidence implements IEvidence {
        private final String id;
        private final List<String> trace;
        private final Map<String, Object> diff;
        private final Map<String, Object> metrics;
        private final Map<String, Object> contribution;

        public Evidence(String id,
                        List<String> trace,
                        Map<String, Object> diff,
                        Map<String, Object> metrics,
                        Map<String, Object> contribution) {
            this.id = (id == null) ? UUID.randomUUID().toString() : id;
            this.trace = (trace == null) ? new ArrayList<String>() : new
ArrayList<String>(trace);
            this.diff = (diff == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(diff);
            this.metrics = (metrics == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(metrics);
            this.contribution = (contribution == null) ? new
LinkedHashMap<String, Object>() : new LinkedHashMap<String,
Object>(contribution);
        }

        public static Evidence empty() { return new Evidence(null, null,
null, null, null); }

        @Override public String evidenceId() { return id; }
        @Override public List<String> trace() { return
Collections.unmodifiableList(trace); }
        @Override public Map<String, Object> diff() { return
Collections.unmodifiableMap(diff); }
        @Override public Map<String, Object> metrics() { return
Collections.unmodifiableMap(metrics); }
```

```
        @Override public Map<String, Object> contribution() { return
Collections.unmodifiableMap(contribution); }
    }

    public static final class ApplyResult implements IApplyResult {
        private final boolean success;
        private final IState out;
        private final IEvidence evidence;
        private final String failureReason;

        public ApplyResult(boolean success, IState out, IEvidence evidence,
String failureReason) {
            this.success = success;
            this.out = out;
            this.evidence = (evidence == null) ? Evidence.empty() : evidence;
            this.failureReason = failureReason;
        }

        public static ApplyResult ok(IState out, IEvidence evidence) {
            return new ApplyResult(true, out, evidence, null);
        }

        public static ApplyResult fail(String reason, IEvidence evidence) {
            return new ApplyResult(false, null, evidence, reason);
        }

        @Override public boolean success() { return success; }
        @Override public IState outputState() { return out; }
        @Override public IEvidence evidence() { return evidence; }
        @Override public String failureReason() { return failureReason; }
    }

    // ------------------------------
    // 6) Plan / Program
    // ------------------------------

    public interface IOpStep {
        IOperation op();
        IOpArgs args();
    }

    public interface IOperationPlan {
        String planId();
        List<IOpStep> steps();

        /** Optional: LLM-provided assumptions/hypotheses for auditability.
*/
        default List<String> assumptions() { return
Collections.emptyList(); }

        /** Optional: proposal provenance (model name, temperature, etc.). */
        default Map<String, Object> provenance() { return
Collections.emptyMap(); }
    }

    public interface IPlanExecutionResult {
        boolean success();
```

```
        IState finalState();
        List<IEvidence> stepEvidence();
        List<String> failureChain(); // pre/post/invariant reasons in order
        Map<String, Object> summaryMetrics();
    }

    // ----------------------------
    // 7) Executor / Verifier
    // ----------------------------

    public interface IPlanExecutor {
        IPlanExecutionResult execute(IState x, IOperationPlan plan,
IRuntimeHarness harness, IPolicy policy);
    }

    /**
     * Runtime harness: where we attach tests, property checks, trace
collection,
     * coverage collection, and stable evidence formatting.
     */
    public interface IRuntimeHarness {
        /** Called before executing a plan (setup sandbox, init counters). */
        void beforePlan(IState x, IOperationPlan plan);

        /** Called after each step to run verifiable checks. */
        HarnessCheck afterStep(IState current, IOpStep step, IEvidence
stepEvidence);

        /** Called after plan completes (success or fail). */
        HarnessReport afterPlan(IState x, IOperationPlan plan,
IPlanExecutionResult execResult);
    }

    public interface IPolicy {
        boolean allowOperation(IOperation op);
        int maxSteps();
        int maxRuntimeMs();
        default boolean stopOnFirstFailure() { return true; }
    }

    public static final class HarnessCheck {
        public final boolean ok;
        public final String reason;
        public final Map<String, Object> metrics;

        public HarnessCheck(boolean ok, String reason, Map<String, Object>
metrics) {
            this.ok = ok;
            this.reason = reason;
            this.metrics = (metrics == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(metrics);
        }

        public static HarnessCheck ok() { return new HarnessCheck(true, "OK",
null); }
        public static HarnessCheck fail(String reason) { return new
HarnessCheck(false, reason, null); }
```

```
    }

    public static final class HarnessReport {
        public final Map<String, Object> metrics;
        public final List<String> notes;

        public HarnessReport(Map<String, Object> metrics, List<String> notes)
{
            this.metrics = (metrics == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(metrics);
            this.notes = (notes == null) ? new ArrayList<String>() : new
ArrayList<String>(notes);
        }
    }
}
```

要点：

- **IOperation** 强制定义 `checkPre/apply/checkPost` + `evidence()`，确保“生成不可替代验证”。
- **IRuntimeHarness** 是 ITEM #182 的核心“闸门”：可插入 JUnit/属性测试/coverage/trace。
- **Policy** 负责治理：允许/禁止某些 op、最大步数、最大运行时间。

B) LLM–ACLM 协作 Pipeline：伪代码 + 时序图

B1) Pipeline 伪代码（语言无关）

```
INPUT:
  X: current (misaligned) state
  Y_constraints: target semantic constraints / invariants
  OpLibrary: available operations + DSL schemas
  Policy: maxSteps, allowed categories, runtime budget

PHASE 0: Context Packing (DBM/ACLM)
  ctx := {
    x_summary = summarize(X)             // diffs, missing slots, CCC/SOS
hints
    y_constraints = Y_constraints        // must-pass properties/invariants
    op_schema = OpLibrary.schemas()      // operation names + param schemas +
categories
    evidence_format = stable_schema()    // how evidence should be returned
  }
```

```
PHASE 1: Proposal Generation (LLM)
  proposals := LLM.generate_plans(ctx, topK=K)
  // each proposal: [ (op_name, args), ... ], assumptions, rationale,
confidence

PHASE 2: Static Verification & Pruning (ACLM)
  candidates := []
  for plan in proposals:
    if violates_policy(plan, Policy): continue
    if fails_typecheck(plan, OpLibrary): continue
    if fails_preconditions_fast(plan, X): continue
    score := structural_score(plan, X, Y_constraints)  // ladder distance,
contributionCost priors
    candidates.add(plan, score)

  candidates := take_top_M(candidates, M)

PHASE 3: Runtime Evidence Loop (ACLM + Harness)
  best := null
  for plan in candidates:
    exec_result := executor.execute(X, plan, runtime_harness, Policy)
    if exec_result.success and satisfies(exec_result.finalState,
Y_constraints):
        best := select_best(best, exec_result) // by metrics + minimal edits
+ stability
    feedback := build_feedback(exec_result)     // stable failure reasons +
evidence snippets
    if best is null:
        proposals += LLM.repair_or_regenerate(ctx, feedback, extraK)

OUTPUT:
  best_plan + evidence_chain + explanation(contribution, trace, diffs)
```

B2) 时序图（ASCII）

```
+-------------------+             +-----------------------+             +--
----------------------+
|       LLM         |             |    ACLM Orchestrator  |             |
Runtime Harness      |
+-------------------+             +-----------------------+             +--
----------------------+
          |                                     |
|
          |<----------- ctx (X summary, Y constraints, Op DSL) -------------
----------|
          |                                     |
|
          |---- propose: plan_1..plan_K -------->|
|
          |                                     |
|
          |                                     |-- static check/type/pre
prune ------> (no runtime)
          |                                     |
|
```

```
          |                                          |-- execute(plan_i) --------
----------->|
          |                                          |
|-- afterStep checks
          |                                          |
|-- trace/diff/metrics
          |                                          |<------------- evidence +
pass/fail ---|
          |                                          |
|
          |<----- feedback (failure reason + evidence snippets) -------------
----------|
          |---- repair/regenerate (plan') ------>|
|
          |                                          |-- re-run execute(plan') --
---------->|
          |                                          |<------------- evidence ---
----------|
          |                                          |
|
          |<------------------ selected best plan + evidence chain ----------
----------|
```

C) 最小 Runtime Evidence Harness 示例（MVP）

下面给一个最小可跑的示例：

- 状态是一个 `MapState`
- 操作提供两个：`SetFieldOp` 与 `ConcatFieldOp`
- 目标约束：某字段必须等于期望值
- Harness 做两件事：
    1. 每步收集 trace/diff
    2. 每步执行一个“可验证断言”（属性检查）并返回稳定 reason

这就是 ITEM #182 的“证据闸门”最小形态：你后续可替换为 JUnit、coverage、真实 sandbox 等。

```
package com.dbm.aclm.affine.demo;

import com.dbm.aclm.affine.OperationDsl;
import java.util.*;

/**
 * ITEM #182 — Minimal Runtime Evidence Harness (MVP)
```

```
 */
public final class MinimalEvidenceDemo {

    // ------------------------------
    // 1) Simple State
    // ------------------------------
    public static final class MapState implements OperationDsl.IMutableState
{
        private final String id;
        private final Map<String, Object> m;

        public MapState(String id, Map<String, Object> init) {
            this.id = (id == null) ? UUID.randomUUID().toString() : id;
            this.m = (init == null) ? new LinkedHashMap<String, Object>() :
new LinkedHashMap<String, Object>(init);
        }

        @Override public String stateId() { return id; }
        @Override public Map<String, Object> view() { return
Collections.unmodifiableMap(m); }
        @Override public OperationDsl.IMutableState mutableCopy() { return
new MapState(id + "_copy", m); }
        @Override public void put(String key, Object value) { m.put(key,
value); }
        @Override public void remove(String key) { m.remove(key); }

        @Override public String toString() { return "MapState(" + id + ") " +
m; }
    }

    // ------------------------------
    // 2) Operations
    // ------------------------------
    public static final class SetFieldOp implements OperationDsl.IOperation {
        @Override public String name() { return "setField"; }
        @Override public String category() { return "atomic"; }

        @Override public OperationDsl.IParamSchema paramSchema() {
            return new OperationDsl.IParamSchema() {
                @Override public List<OperationDsl.IParam> params() {
                    return Arrays.asList(
                        param("key", OperationDsl.ParamType.STRING, true,
"field name"),
                        param("value", OperationDsl.ParamType.STRING, true,
"string value")
                    );
                }
            };
        }

        @Override public OperationDsl.ICheckResult
checkPre(OperationDsl.IState x, OperationDsl.IOpArgs args) {
            if (args.getString("key") == null) return
OperationDsl.CheckResult.fail("MISSING_KEY");
            if (args.getString("value") == null) return
OperationDsl.CheckResult.fail("MISSING_VALUE");
            return OperationDsl.CheckResult.ok();
```

```
        }

        @Override public OperationDsl.IApplyResult apply(OperationDsl.IState
x, OperationDsl.IOpArgs args) {
            OperationDsl.IMutableState y = x.mutableCopy();
            String key = args.getString("key");
            String value = args.getString("value");

            Object before = y.view().get(key);
            y.put(key, value);

            List<String> trace = Arrays.asList("SetFieldOp: " + key + " :=
\"" + value + "\"");
            Map<String, Object> diff = new LinkedHashMap<String, Object>();
            diff.put("field", key);
            diff.put("before", before);
            diff.put("after", value);

            return OperationDsl.ApplyResult.ok(
                y,
                new OperationDsl.Evidence(null, trace, diff, metric("op",
"setField"), null)
            );
        }

        @Override public OperationDsl.ICheckResult
checkPost(OperationDsl.IState y, OperationDsl.IOpArgs args) {
            String key = args.getString("key");
            String value = args.getString("value");
            Object got = y.view().get(key);
            if (!String.valueOf(value).equals(String.valueOf(got))) {
                return OperationDsl.CheckResult.fail("POST_MISMATCH");
            }
            return OperationDsl.CheckResult.ok();
        }
    }

    public static final class ConcatFieldOp implements
OperationDsl.IOperation {
        @Override public String name() { return "concatField"; }
        @Override public String category() { return "atomic"; }

        @Override public OperationDsl.IParamSchema paramSchema() {
            return new OperationDsl.IParamSchema() {
                @Override public List<OperationDsl.IParam> params() {
                    return Arrays.asList(
                        param("key", OperationDsl.ParamType.STRING, true,
"field name"),
                        param("suffix", OperationDsl.ParamType.STRING, true,
"suffix")
                    );
                }
            };
        }

        @Override public OperationDsl.ICheckResult
checkPre(OperationDsl.IState x, OperationDsl.IOpArgs args) {
```

```
            String key = args.getString("key");
            String suffix = args.getString("suffix");
            if (key == null) return
OperationDsl.CheckResult.fail("MISSING_KEY");
            if (suffix == null) return
OperationDsl.CheckResult.fail("MISSING_SUFFIX");
            Object v = x.view().get(key);
            if (v == null) return
OperationDsl.CheckResult.fail("MISSING_FIELD_" + key);
            return OperationDsl.CheckResult.ok();
        }

        @Override public OperationDsl.IApplyResult apply(OperationDsl.IState
x, OperationDsl.IOpArgs args) {
            OperationDsl.IMutableState y = x.mutableCopy();
            String key = args.getString("key");
            String suffix = args.getString("suffix");

            String before = String.valueOf(y.view().get(key));
            String after = before + suffix;
            y.put(key, after);

            List<String> trace = Arrays.asList("ConcatFieldOp: " + key + " +=
\"" + suffix + "\"");
            Map<String, Object> diff = new LinkedHashMap<String, Object>();
            diff.put("field", key);
            diff.put("before", before);
            diff.put("after", after);

            return OperationDsl.ApplyResult.ok(
                y,
                new OperationDsl.Evidence(null, trace, diff, metric("op",
"concatField"), null)
            );
        }

        @Override public OperationDsl.ICheckResult
checkPost(OperationDsl.IState y, OperationDsl.IOpArgs args) {
            return OperationDsl.CheckResult.ok();
        }
    }

    // ----------------------------
    // 3) Plan, Executor, Harness
    // ----------------------------
    public static final class Step implements OperationDsl.IOpStep {
        private final OperationDsl.IOperation op;
        private final OperationDsl.IOpArgs args;

        public Step(OperationDsl.IOperation op, OperationDsl.IOpArgs args) {
            this.op = op;
            this.args = args;
        }

        @Override public OperationDsl.IOperation op() { return op; }
        @Override public OperationDsl.IOpArgs args() { return args; }
    }
```

```
    public static final class Plan implements OperationDsl.IOperationPlan {
        private final String id;
        private final List<OperationDsl.IOpStep> steps;

        public Plan(String id, List<OperationDsl.IOpStep> steps) {
            this.id = (id == null) ? UUID.randomUUID().toString() : id;
            this.steps = (steps == null) ? new
ArrayList<OperationDsl.IOpStep>() : steps;
        }

        @Override public String planId() { return id; }
        @Override public List<OperationDsl.IOpStep> steps() { return steps; }
    }

    public static final class SimplePolicy implements OperationDsl.IPolicy {
        private final Set<String> allow;
        private final int maxSteps;
        private final int maxRuntimeMs;

        public SimplePolicy(Set<String> allow, int maxSteps, int
maxRuntimeMs) {
            this.allow = (allow == null) ? new HashSet<String>() : allow;
            this.maxSteps = maxSteps;
            this.maxRuntimeMs = maxRuntimeMs;
        }

        @Override public boolean allowOperation(OperationDsl.IOperation op) {
            return allow.isEmpty() || allow.contains(op.name());
        }

        @Override public int maxSteps() { return maxSteps; }

        @Override public int maxRuntimeMs() { return maxRuntimeMs; }
    }

    public static final class SimpleExecutor implements
OperationDsl.IPlanExecutor {

        @Override
        public OperationDsl.IPlanExecutionResult execute(OperationDsl.IState
x,

OperationDsl.IOperationPlan plan,

OperationDsl.IRuntimeHarness harness,
                                                         OperationDsl.IPolicy
policy) {

            final long t0 = System.currentTimeMillis();
            final List<OperationDsl.IEvidence> evidences = new
ArrayList<OperationDsl.IEvidence>();
            final List<String> failureChain = new ArrayList<String>();

            harness.beforePlan(x, plan);

            OperationDsl.IState cur = x;
```

```
            if (plan.steps().size() > policy.maxSteps()) {
                return result(false, null, evidences,
Arrays.asList("POLICY_MAX_STEPS"), metric("maxSteps", policy.maxSteps()));
            }

            for (OperationDsl.IOpStep step : plan.steps()) {
                if (System.currentTimeMillis() - t0 > policy.maxRuntimeMs())
{
                    failureChain.add("POLICY_TIMEOUT");
                    return result(false, null, evidences, failureChain,
metric("timeoutMs", policy.maxRuntimeMs()));
                }

                OperationDsl.IOperation op = step.op();
                if (!policy.allowOperation(op)) {
                    failureChain.add("POLICY_DENY_" + op.name());
                    return result(false, null, evidences, failureChain,
metric("denyOp", op.name()));
                }

                OperationDsl.ICheckResult pre = op.checkPre(cur,
step.args());
                if (!pre.ok()) {
                    failureChain.add("PRE_" + op.name() + ":" +
pre.reason());
                    return result(false, null, evidences, failureChain,
pre.details());
                }

                OperationDsl.IApplyResult ar = op.apply(cur, step.args());
                evidences.add(ar.evidence());

                if (!ar.success()) {
                    failureChain.add("APPLY_" + op.name() + ":" +
ar.failureReason());
                    return result(false, null, evidences, failureChain,
metric("applyFail", op.name()));
                }

                OperationDsl.ICheckResult post =
op.checkPost(ar.outputState(), step.args());
                if (!post.ok()) {
                    failureChain.add("POST_" + op.name() + ":" +
post.reason());
                    return result(false, null, evidences, failureChain,
post.details());
                }

                OperationDsl.HarnessCheck hc =
harness.afterStep(ar.outputState(), step, ar.evidence());
                if (!hc.ok) {
                    failureChain.add("HARNESS_FAIL:" + hc.reason);
                    return result(false, null, evidences, failureChain,
hc.metrics);
                }
```

```
                cur = ar.outputState();
            }

            return result(true, cur, evidences, failureChain,
metric("status", "OK"));
        }

        private static OperationDsl.IPlanExecutionResult result(final boolean
success,
                                                                final
OperationDsl.IState finalState,
                                                                final
List<OperationDsl.IEvidence> stepEvidence,
                                                                final
List<String> failureChain,
                                                                final
Map<String, Object> summaryMetrics) {
            return new OperationDsl.IPlanExecutionResult() {
                @Override public boolean success() { return success; }
                @Override public OperationDsl.IState finalState() { return
finalState; }
                @Override public List<OperationDsl.IEvidence> stepEvidence()
{ return stepEvidence; }
                @Override public List<String> failureChain() { return
failureChain; }
                @Override public Map<String, Object> summaryMetrics()
{ return summaryMetrics; }
            };
        }
    }

    public static final class PropertyHarness implements
OperationDsl.IRuntimeHarness {

        private final String mustKey;
        private final String mustValue;

        public PropertyHarness(String mustKey, String mustValue) {
            this.mustKey = mustKey;
            this.mustValue = mustValue;
        }

        @Override public void beforePlan(OperationDsl.IState x,
OperationDsl.IOperationPlan plan) {
            // Setup sandbox, reset counters, etc. MVP: no-op.
        }

        @Override
        public OperationDsl.HarnessCheck afterStep(OperationDsl.IState
current, OperationDsl.IOpStep step, OperationDsl.IEvidence stepEvidence) {
            // MVP property check: if key exists, it must not be empty; and
if mustKey is present, check target.
            Object v = current.view().get(mustKey);
            if (v != null && String.valueOf(v).trim().isEmpty()) {
                return OperationDsl.HarnessCheck.fail("EMPTY_FIELD_" +
mustKey);
            }
```

```
            // Soft check each step: if it already matches target, good;
otherwise allow continue.
            return OperationDsl.HarnessCheck.ok();
        }

        @Override
        public OperationDsl.HarnessReport afterPlan(OperationDsl.IState x,

OperationDsl.IOperationPlan plan,

OperationDsl.IPlanExecutionResult execResult) {
            // MVP: summarize
            List<String> notes = new ArrayList<String>();
            notes.add("planId=" + plan.planId());
            notes.add("success=" + execResult.success());
            return new
OperationDsl.HarnessReport(execResult.summaryMetrics(), notes);
        }
    }

    // ------------------------------
    // 4) Demo main
    // ------------------------------
    public static void main(String[] args) {
        // X: missing/partial state
        Map<String, Object> init = new LinkedHashMap<String, Object>();
        init.put("name", "DBM");
        MapState X = new MapState("X", init);

        // Target: name must become "DBM-COT"
        String mustKey = "name";
        String mustValue = "DBM-COT";

        // Define ops
        OperationDsl.IOperation set = new SetFieldOp();
        OperationDsl.IOperation concat = new ConcatFieldOp();

        // A candidate plan (LLM would propose this)
        OperationDsl.IOperationPlan plan = new Plan("plan_1",
Arrays.<OperationDsl.IOpStep>asList(
            new Step(concat, new OperationDsl.OpArgs(mapOf("key", "name",
"suffix", "-COT")))
        ));

        // Policy
        Set<String> allow = new HashSet<String>();
        allow.add("setField");
        allow.add("concatField");
        OperationDsl.IPolicy policy = new SimplePolicy(allow, 8, 2000);

        // Harness
        OperationDsl.IRuntimeHarness harness = new PropertyHarness(mustKey,
mustValue);

        // Execute
        SimpleExecutor executor = new SimpleExecutor();
```

```
        OperationDsl.IPlanExecutionResult r = executor.execute(X, plan,
harness, policy);

        System.out.println("X=" + X);
        System.out.println("success=" + r.success());
        System.out.println("final=" + r.finalState());
        System.out.println("failureChain=" + r.failureChain());

        // Evidence chain
        for (OperationDsl.IEvidence e : r.stepEvidence()) {
            System.out.println("evidenceId=" + e.evidenceId());
            System.out.println("trace=" + e.trace());
            System.out.println("diff=" + e.diff());
            System.out.println("metrics=" + e.metrics());
        }

        // Final constraint check (ACLM side)
        boolean ok = r.success() &&
mustValue.equals(String.valueOf(r.finalState().view().get(mustKey)));
        System.out.println("satisfiesTarget=" + ok);
    }

    // ------------------------------
    // 5) Small helpers
    // ------------------------------
    private static OperationDsl.IParam param(final String name, final
OperationDsl.ParamType type, final boolean req, final String doc) {
        return new OperationDsl.IParam() {
            @Override public String name() { return name; }
            @Override public OperationDsl.ParamType type() { return type; }
            @Override public boolean required() { return req; }
            @Override public String doc() { return doc; }
            @Override public String toString() { return name + ":" + type +
(req ? "!" : ""); }
        };
    }

    private static Map<String, Object> metric(String k, Object v) {
        Map<String, Object> m = new LinkedHashMap<String, Object>();
        m.put(k, v);
        return m;
    }

    private static Map<String, Object> mapOf(String k1, Object v1, String k2,
Object v2) {
        Map<String, Object> m = new LinkedHashMap<String, Object>();
        m.put(k1, v1);
        m.put(k2, v2);
        return m;
    }
}
```

你将如何把它接到 ACLM Vertical Gap Bridging？

- 你现有的 `UnalignedAndScorer / AssemblyPlanner`（或 gap 包内的对齐/规划器）产出“候选桥接计划”时：
  - 直接把每个候选表达成 `IOperationPlan`
- 你现有的 `contributionCost / ladder distance`：
  - 填到 `Evidence.contribution()` 与 `summaryMetrics()`
- 你已有的“解释链”机制：
  - 把 `trace + diff + harness failureChain` 当作第一层证据，再叠加你更高阶的 CCC/SOS tier 贡献链

你下一步最关键的两处“接口落钉点”

1. **Operation Library 的抽象边界**
   - 哪些是原子 op，哪些是 macro-op
   - macro-op 是否允许展开为多个原子步骤（建议允许，并保留 provenance）
2. **Harness 的“可验证性等级”分层**
   - Level-0：assert on simple properties（如 demo）
   - Level-1：可跑的单元测试集（JUnit）
   - Level-2：沙箱执行 + coverage + 差分回归
   - Level-3：跨版本治理（白名单/黑名单 op，风险分级）

如果你愿意，我可以在你既有 gap 包命名风格下，把上述三块拆成多文件结构（`IOperation`, `IPlanExecutor`, `IRuntimeHarness` 等各自一文件），并补一组 **JUnit4 玩具测试**（成功、失败、policy deny、timeout、precondition fail、postcondition fail、harness fail），让这个 harness 立即变成你 repo 里可跑的“证据闸门”。

ME：

很好。请继续在既有 gap 包命名风格下，把上述三块拆成多文件结构（IOperation, IPlanExecutor, IRuntimeHarness 等各自一文件），并补一组 JUnit4 玩具测试（成功、失败、policy deny、timeout、precondition fail、postcondition fail、harness fail），让这个 harness 立即变成 repo 里可跑的“证据闸门”。

下面给你一套 **“可直接 drop-in 编译 + 可跑 JUnit4”** 的最小证据闸门（Runtime Evidence Harness）实现，并且严格按**多文件拆分**组织，便于你在 `com.dbm.ACLM.gap` 下直接落地、后续扩展到你现有的 `UnalignedAndScorer / AssemblyPlanner` 流程。

我按你要求把核心拆成：`IOperation / IPlanExecutor / IRuntimeHarness` 等各自一文件，并补齐一组 **JUnit4 玩具测试**覆盖：

- 成功（success）
- 失败（apply fail）
- policy deny
- timeout
- precondition fail
- postcondition fail
- harness fail

---

1) 建议目录结构（drop-in）

把这些文件放到你的 repo 里：

```
src/main/java/com/dbm/ACLM/gap/affine/...
src/test/java/com/dbm/ACLM/gap/affine/...
```

包名全部是：

```
package com.dbm.ACLM.gap.affine;
```

---

2) 生产代码（src/main/java）

2.1 IState.java

```
package com.dbm.ACLM.gap.affine;

import java.util.Map;

public interface IState {
    String stateId();
    Map<String, Object> view();
    IMutableState mutableCopy();
}
```

2.2 IMutableState.java

```
package com.dbm.ACLM.gap.affine;

public interface IMutableState extends IState {
    void put(String key, Object value);
    void remove(String key);
}
```

2.3 MapState.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class MapState implements IMutableState {
    private final String id;
    private final Map<String, Object> m;

    public MapState(String id, Map<String, Object> init) {
        this.id = (id == null) ? UUID.randomUUID().toString() : id;
        this.m = (init == null) ? new LinkedHashMap<String, Object>() : new
LinkedHashMap<String, Object>(init);
    }

    @Override public String stateId() { return id; }

    @Override public Map<String, Object> view() { return
Collections.unmodifiableMap(m); }

    @Override public IMutableState mutableCopy() { return new MapState(id +
"_copy", m); }

    @Override public void put(String key, Object value) { m.put(key,
value); }

    @Override public void remove(String key) { m.remove(key); }

    @Override public String toString() { return "MapState(" + id + ") " +
m; }
}
```

2.4 ParamType.java

```
package com.dbm.ACLM.gap.affine;

public enum ParamType {
    STRING, INT, DOUBLE, BOOL, JSON, LIST, MAP
}
```

2.5 IParam.java

```
package com.dbm.ACLM.gap.affine;

public interface IParam {
    String name();
    ParamType type();
    boolean required();
    String doc();
}
```

2.6 IParamSchema.java

```
package com.dbm.ACLM.gap.affine;

import java.util.List;

public interface IParamSchema {
    List<IParam> params();
}
```

2.7 IOpArgs.java

```
package com.dbm.ACLM.gap.affine;

import java.util.Map;

public interface IOpArgs {
    boolean has(String key);
    Object get(String key);
    String getString(String key);
    Integer getInt(String key);
    Double getDouble(String key);
    Map<String, Object> asMap();
}
```

2.8 OpArgs.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class OpArgs implements IOpArgs {
    private final Map<String, Object> m;

    public OpArgs(Map<String, Object> m) {
```

```
        this.m = (m == null) ? new LinkedHashMap<String, Object>() : new
LinkedHashMap<String, Object>(m);
    }

    public static OpArgs of(String k1, Object v1) {
        Map<String, Object> m = new LinkedHashMap<String, Object>();
        m.put(k1, v1);
        return new OpArgs(m);
    }

    @Override public boolean has(String key) { return m.containsKey(key); }
    @Override public Object get(String key) { return m.get(key); }
    @Override public String getString(String key) { Object v = m.get(key);
return (v == null) ? null : String.valueOf(v); }
    @Override public Integer getInt(String key) { Object v = m.get(key);
return (v instanceof Number) ? ((Number) v).intValue() : null; }
    @Override public Double getDouble(String key) { Object v = m.get(key);
return (v instanceof Number) ? ((Number) v).doubleValue() : null; }
    @Override public Map<String, Object> asMap() { return
Collections.unmodifiableMap(m); }
}
```

2.9 ICheckResult.java

```
package com.dbm.ACLM.gap.affine;

import java.util.Map;

public interface ICheckResult {
    boolean ok();
    String reason();               // stable token for governance & LLM
feedback
    Map<String, Object> details(); // bounded
}
```

2.10 CheckResult.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class CheckResult implements ICheckResult {
    private final boolean ok;
    private final String reason;
    private final Map<String, Object> details;

    public CheckResult(boolean ok, String reason, Map<String, Object>
details) {
        this.ok = ok;
        this.reason = reason;
        this.details = (details == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(details);
    }
```

```
    public static CheckResult ok() { return new CheckResult(true, "OK",
null); }
    public static CheckResult fail(String reason) { return new
CheckResult(false, reason, null); }

    @Override public boolean ok() { return ok; }
    @Override public String reason() { return reason; }
    @Override public Map<String, Object> details() { return
Collections.unmodifiableMap(details); }
}
```

2.11 IEvidence.java

```
package com.dbm.ACLM.gap.affine;

import java.util.List;
import java.util.Map;

public interface IEvidence {
    String evidenceId();
    List<String> trace();
    Map<String, Object> diff();
    Map<String, Object> metrics();
    Map<String, Object> contribution();
}
```

2.12 Evidence.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class Evidence implements IEvidence {
    private final String id;
    private final List<String> trace;
    private final Map<String, Object> diff;
    private final Map<String, Object> metrics;
    private final Map<String, Object> contribution;

    public Evidence(String id,
                    List<String> trace,
                    Map<String, Object> diff,
                    Map<String, Object> metrics,
                    Map<String, Object> contribution) {
        this.id = (id == null) ? UUID.randomUUID().toString() : id;
        this.trace = (trace == null) ? new ArrayList<String>() : new
ArrayList<String>(trace);
        this.diff = (diff == null) ? new LinkedHashMap<String, Object>() :
new LinkedHashMap<String, Object>(diff);
        this.metrics = (metrics == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(metrics);
        this.contribution = (contribution == null) ? new
LinkedHashMap<String, Object>() : new LinkedHashMap<String,
Object>(contribution);
```

```
    }

    public static Evidence empty() { return new Evidence(null, null, null,
null, null); }

    @Override public String evidenceId() { return id; }
    @Override public List<String> trace() { return
Collections.unmodifiableList(trace); }
    @Override public Map<String, Object> diff() { return
Collections.unmodifiableMap(diff); }
    @Override public Map<String, Object> metrics() { return
Collections.unmodifiableMap(metrics); }
    @Override public Map<String, Object> contribution() { return
Collections.unmodifiableMap(contribution); }
}
```

2.13 IApplyResult.java

```
package com.dbm.ACLM.gap.affine;

public interface IApplyResult {
    boolean success();
    IState outputState();      // Y
    IEvidence evidence();      // evidence payload
    String failureReason();    // stable token
}
```

2.14 ApplyResult.java

```
package com.dbm.ACLM.gap.affine;

public final class ApplyResult implements IApplyResult {
    private final boolean success;
    private final IState out;
    private final IEvidence evidence;
    private final String failureReason;

    public ApplyResult(boolean success, IState out, IEvidence evidence,
String failureReason) {
        this.success = success;
        this.out = out;
        this.evidence = (evidence == null) ? Evidence.empty() : evidence;
        this.failureReason = failureReason;
    }

    public static ApplyResult ok(IState out, IEvidence evidence) {
        return new ApplyResult(true, out, evidence, null);
    }

    public static ApplyResult fail(String reason, IEvidence evidence) {
        return new ApplyResult(false, null, evidence, reason);
    }

    @Override public boolean success() { return success; }
```

```
    @Override public IState outputState() { return out; }
    @Override public IEvidence evidence() { return evidence; }
    @Override public String failureReason() { return failureReason; }
}
```

2.15 IOperation.java

```
package com.dbm.ACLM.gap.affine;

import java.util.Collections;
import java.util.List;

public interface IOperation {
    String name();

    default String category() { return "atomic"; }

    IParamSchema paramSchema();

    ICheckResult checkPre(IState x, IOpArgs args);

    IApplyResult apply(IState x, IOpArgs args);

    ICheckResult checkPost(IState y, IOpArgs args);

    default List<IInvariant> invariants() { return Collections.emptyList(); }
}
```

2.16 IInvariant.java

```
package com.dbm.ACLM.gap.affine;

public interface IInvariant {
    String name();
    ICheckResult check(IState s);
}
```

2.17 IOpStep.java

```
package com.dbm.ACLM.gap.affine;

public interface IOpStep {
    IOperation op();
    IOpArgs args();
}
```

2.18 OpStep.java

```
package com.dbm.ACLM.gap.affine;

public final class OpStep implements IOpStep {
    private final IOperation op;
```

```
    private final IOpArgs args;

    public OpStep(IOperation op, IOpArgs args) {
        this.op = op;
        this.args = args;
    }

    @Override public IOperation op() { return op; }
    @Override public IOpArgs args() { return args; }
}
```

2.19 IOperationPlan.java

```
package com.dbm.ACLM.gap.affine;

import java.util.Collections;
import java.util.List;
import java.util.Map;

public interface IOperationPlan {
    String planId();
    List<IOpStep> steps();

    default List<String> assumptions() { return Collections.emptyList(); }
    default Map<String, Object> provenance() { return
Collections.emptyMap(); }
}
```

2.20 OperationPlan.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class OperationPlan implements IOperationPlan {
    private final String id;
    private final List<IOpStep> steps;
    private final List<String> assumptions;
    private final Map<String, Object> provenance;

    public OperationPlan(String id, List<IOpStep> steps) {
        this(id, steps, null, null);
    }

    public OperationPlan(String id, List<IOpStep> steps, List<String>
assumptions, Map<String, Object> provenance) {
        this.id = (id == null) ? UUID.randomUUID().toString() : id;
        this.steps = (steps == null) ? new ArrayList<IOpStep>() : new
ArrayList<IOpStep>(steps);
        this.assumptions = (assumptions == null) ? new ArrayList<String>() :
new ArrayList<String>(assumptions);
        this.provenance = (provenance == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(provenance);
    }
```

```
    @Override public String planId() { return id; }
    @Override public List<IOpStep> steps() { return
Collections.unmodifiableList(steps); }
    @Override public List<String> assumptions() { return
Collections.unmodifiableList(assumptions); }
    @Override public Map<String, Object> provenance() { return
Collections.unmodifiableMap(provenance); }
}
```

2.21 IPolicy.java

```
package com.dbm.ACLM.gap.affine;

public interface IPolicy {
    boolean allowOperation(IOperation op);
    int maxSteps();
    int maxRuntimeMs();
    boolean stopOnFirstFailure();
}
```

2.22 SimplePolicy.java

```
package com.dbm.ACLM.gap.affine;

import java.util.Set;

public final class SimplePolicy implements IPolicy {
    private final Set<String> allowOpNames;
    private final int maxSteps;
    private final int maxRuntimeMs;
    private final boolean stopOnFirstFailure;

    public SimplePolicy(Set<String> allowOpNames, int maxSteps, int
maxRuntimeMs, boolean stopOnFirstFailure) {
        this.allowOpNames = allowOpNames;
        this.maxSteps = maxSteps;
        this.maxRuntimeMs = maxRuntimeMs;
        this.stopOnFirstFailure = stopOnFirstFailure;
    }

    @Override
    public boolean allowOperation(IOperation op) {
        return (allowOpNames == null) || allowOpNames.isEmpty() ||
allowOpNames.contains(op.name());
    }

    @Override public int maxSteps() { return maxSteps; }
    @Override public int maxRuntimeMs() { return maxRuntimeMs; }
    @Override public boolean stopOnFirstFailure() { return
stopOnFirstFailure; }
}
```

2.23 HarnessCheck.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class HarnessCheck {
    public final boolean ok;
    public final String reason;
    public final Map<String, Object> metrics;

    public HarnessCheck(boolean ok, String reason, Map<String, Object>
metrics) {
        this.ok = ok;
        this.reason = reason;
        this.metrics = (metrics == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(metrics);
    }

    public static HarnessCheck ok() { return new HarnessCheck(true, "OK",
null); }
    public static HarnessCheck fail(String reason) { return new
HarnessCheck(false, reason, null); }
}
```

2.24 HarnessReport.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class HarnessReport {
    public final Map<String, Object> metrics;
    public final List<String> notes;

    public HarnessReport(Map<String, Object> metrics, List<String> notes) {
        this.metrics = (metrics == null) ? new LinkedHashMap<String,
Object>() : new LinkedHashMap<String, Object>(metrics);
        this.notes = (notes == null) ? new ArrayList<String>() : new
ArrayList<String>(notes);
    }
}
```

2.25 IRuntimeHarness.java

```
package com.dbm.ACLM.gap.affine;

public interface IRuntimeHarness {
    void beforePlan(IState x, IOperationPlan plan);
    HarnessCheck afterStep(IState current, IOpStep step, IEvidence
stepEvidence);
    HarnessReport afterPlan(IState x, IOperationPlan plan,
IPlanExecutionResult execResult);
```

```
}
```

2.26 IPlanExecutionResult.java

```
package com.dbm.ACLM.gap.affine;

import java.util.List;
import java.util.Map;

public interface IPlanExecutionResult {
    boolean success();
    IState finalState();
    List<IEvidence> stepEvidence();
    List<String> failureChain();
    Map<String, Object> summaryMetrics();
}
```

2.27 PlanExecutionResult.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class PlanExecutionResult implements IPlanExecutionResult {
    private final boolean success;
    private final IState finalState;
    private final List<IEvidence> stepEvidence;
    private final List<String> failureChain;
    private final Map<String, Object> summaryMetrics;

    public PlanExecutionResult(boolean success,
                               IState finalState,
                               List<IEvidence> stepEvidence,
                               List<String> failureChain,
                               Map<String, Object> summaryMetrics) {
        this.success = success;
        this.finalState = finalState;
        this.stepEvidence = (stepEvidence == null) ? new
ArrayList<IEvidence>() : new ArrayList<IEvidence>(stepEvidence);
        this.failureChain = (failureChain == null) ? new
ArrayList<String>() : new ArrayList<String>(failureChain);
        this.summaryMetrics = (summaryMetrics == null) ? new
LinkedHashMap<String, Object>() : new LinkedHashMap<String,
Object>(summaryMetrics);
    }

    @Override public boolean success() { return success; }
    @Override public IState finalState() { return finalState; }
    @Override public List<IEvidence> stepEvidence() { return
Collections.unmodifiableList(stepEvidence); }
    @Override public List<String> failureChain() { return
Collections.unmodifiableList(failureChain); }
    @Override public Map<String, Object> summaryMetrics() { return
Collections.unmodifiableMap(summaryMetrics); }
```

```
}
```

2.28 IPlanExecutor.java

```
package com.dbm.ACLM.gap.affine;

public interface IPlanExecutor {
    IPlanExecutionResult execute(IState x, IOperationPlan plan,
IRuntimeHarness harness, IPolicy policy);
}
```

2.29 SimplePlanExecutor.java

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class SimplePlanExecutor implements IPlanExecutor {

    @Override
    public IPlanExecutionResult execute(IState x, IOperationPlan plan,
IRuntimeHarness harness, IPolicy policy) {

        final long t0 = System.currentTimeMillis();
        final List<IEvidence> evidences = new ArrayList<IEvidence>();
        final List<String> failureChain = new ArrayList<String>();
        final Map<String, Object> summary = new LinkedHashMap<String,
Object>();

        harness.beforePlan(x, plan);

        if (plan.steps().size() > policy.maxSteps()) {
            failureChain.add("POLICY_MAX_STEPS");
            summary.put("maxSteps", policy.maxSteps());
            return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
        }

        IState cur = x;

        for (IOpStep step : plan.steps()) {
            if (System.currentTimeMillis() - t0 > policy.maxRuntimeMs()) {
                failureChain.add("POLICY_TIMEOUT");
                summary.put("timeoutMs", policy.maxRuntimeMs());
                return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
            }

            IOperation op = step.op();

            if (!policy.allowOperation(op)) {
                failureChain.add("POLICY_DENY_" + op.name());
                summary.put("denyOp", op.name());
```

```
                return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
            }

            ICheckResult pre = op.checkPre(cur, step.args());
            if (!pre.ok()) {
                failureChain.add("PRE_" + op.name() + ":" + pre.reason());
                summary.put("preFail", pre.reason());
                return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
            }

            // Invariants on current state before apply (optional but
deterministic)
            for (IInvariant inv : op.invariants()) {
                ICheckResult invr = inv.check(cur);
                if (!invr.ok()) {
                    failureChain.add("INV_PRE_" + op.name() + ":" +
invr.reason());
                    summary.put("invPreFail", invr.reason());
                    return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
                }
            }

            IApplyResult ar = op.apply(cur, step.args());
            evidences.add(ar.evidence());

            if (!ar.success()) {
                failureChain.add("APPLY_" + op.name() + ":" +
ar.failureReason());
                summary.put("applyFail", ar.failureReason());
                return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
            }

            ICheckResult post = op.checkPost(ar.outputState(), step.args());
            if (!post.ok()) {
                failureChain.add("POST_" + op.name() + ":" + post.reason());
                summary.put("postFail", post.reason());
                return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
            }

            HarnessCheck hc = harness.afterStep(ar.outputState(), step,
ar.evidence());
            if (!hc.ok) {
                failureChain.add("HARNESS_FAIL:" + hc.reason);
                summary.put("harnessFail", hc.reason);
                return new PlanExecutionResult(false, null, evidences,
failureChain, summary);
            }

            cur = ar.outputState();

            if (!policy.stopOnFirstFailure()) {
```

```
                // MVP: we stop anyway because failures return early; hook
reserved for future "continue & collect".
            }
        }

        summary.put("status", "OK");
        PlanExecutionResult ok = new PlanExecutionResult(true, cur,
evidences, failureChain, summary);

        harness.afterPlan(x, plan, ok);

        return ok;
    }
}
```

2.30 AffineParamUtil.java

```
package com.dbm.ACLM.gap.affine;

public final class AffineParamUtil {
    private AffineParamUtil() {}

    public static IParam param(final String name, final ParamType type, final
boolean req, final String doc) {
        return new IParam() {
            @Override public String name() { return name; }
            @Override public ParamType type() { return type; }
            @Override public boolean required() { return req; }
            @Override public String doc() { return doc; }
            @Override public String toString() { return name + ":" + type +
(req ? "!" : ""); }
        };
    }
}
```

2.31 DemoOps.java（两三个玩具 op，覆盖 pre/post/apply fail）

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class DemoOps {

    private DemoOps() {}

    public static final class SetFieldOp implements IOperation {
        @Override public String name() { return "setField"; }

        @Override public IParamSchema paramSchema() {
            return new IParamSchema() {
```

```
                @Override public List<IParam> params() {
                    return Arrays.asList(
                        AffineParamUtil.param("key", ParamType.STRING, true,
"field name"),
                        AffineParamUtil.param("value", ParamType.STRING,
true, "string value")
                    );
                }
            };
        }

        @Override public ICheckResult checkPre(IState x, IOpArgs args) {
            if (args.getString("key") == null) return
CheckResult.fail("MISSING_KEY");
            if (args.getString("value") == null) return
CheckResult.fail("MISSING_VALUE");
            return CheckResult.ok();
        }

        @Override public IApplyResult apply(IState x, IOpArgs args) {
            IMutableState y = x.mutableCopy();
            String key = args.getString("key");
            String value = args.getString("value");

            Object before = y.view().get(key);
            y.put(key, value);

            List<String> trace = Arrays.asList("SetFieldOp: " + key + " :=
\"" + value + "\"");
            Map<String, Object> diff = new LinkedHashMap<String, Object>();
            diff.put("field", key);
            diff.put("before", before);
            diff.put("after", value);

            Map<String, Object> metrics = new LinkedHashMap<String,
Object>();
            metrics.put("op", "setField");

            return ApplyResult.ok(y, new Evidence(null, trace, diff, metrics,
null));
        }

        @Override public ICheckResult checkPost(IState y, IOpArgs args) {
            String key = args.getString("key");
            String value = args.getString("value");
            Object got = y.view().get(key);
            if (!String.valueOf(value).equals(String.valueOf(got))) return
CheckResult.fail("POST_MISMATCH");
            return CheckResult.ok();
        }
    }

    public static final class ConcatFieldOp implements IOperation {
        @Override public String name() { return "concatField"; }

        @Override public IParamSchema paramSchema() {
            return new IParamSchema() {
```

```
                @Override public List<IParam> params() {
                    return Arrays.asList(
                        AffineParamUtil.param("key", ParamType.STRING, true,
"field name"),
                        AffineParamUtil.param("suffix", ParamType.STRING,
true, "suffix")
                    );
                }
            };
        }

        @Override public ICheckResult checkPre(IState x, IOpArgs args) {
            String key = args.getString("key");
            String suffix = args.getString("suffix");
            if (key == null) return CheckResult.fail("MISSING_KEY");
            if (suffix == null) return CheckResult.fail("MISSING_SUFFIX");
            Object v = x.view().get(key);
            if (v == null) return CheckResult.fail("MISSING_FIELD_" + key);
            return CheckResult.ok();
        }

        @Override public IApplyResult apply(IState x, IOpArgs args) {
            IMutableState y = x.mutableCopy();
            String key = args.getString("key");
            String suffix = args.getString("suffix");

            String before = String.valueOf(y.view().get(key));
            String after = before + suffix;
            y.put(key, after);

            List<String> trace = Arrays.asList("ConcatFieldOp: " + key + " +=
\"" + suffix + "\"");
            Map<String, Object> diff = new LinkedHashMap<String, Object>();
            diff.put("field", key);
            diff.put("before", before);
            diff.put("after", after);

            Map<String, Object> metrics = new LinkedHashMap<String,
Object>();
            metrics.put("op", "concatField");

            return ApplyResult.ok(y, new Evidence(null, trace, diff, metrics,
null));
        }

        @Override public ICheckResult checkPost(IState y, IOpArgs args) {
            return CheckResult.ok();
        }
    }

    /** Always fails during apply to test APPLY_* path. */
    public static final class AlwaysFailApplyOp implements IOperation {
        @Override public String name() { return "alwaysFailApply"; }

        @Override public IParamSchema paramSchema() {
            return new IParamSchema() {
```

```
                @Override public List<IParam> params() { return
Collections.<IParam>emptyList(); }
            };
        }

        @Override public ICheckResult checkPre(IState x, IOpArgs args)
{ return CheckResult.ok(); }

        @Override public IApplyResult apply(IState x, IOpArgs args) {
            List<String> trace = Arrays.asList("AlwaysFailApplyOp: fail");
            Map<String, Object> metrics = new LinkedHashMap<String,
Object>();
            metrics.put("op", "alwaysFailApply");
            return ApplyResult.fail("APPLY_FAIL", new Evidence(null, trace,
null, metrics, null));
        }

        @Override public ICheckResult checkPost(IState y, IOpArgs args)
{ return CheckResult.ok(); }
    }

    /** Postcondition intentionally fails to test POST_* path. */
    public static final class PostFailOp implements IOperation {
        @Override public String name() { return "postFailOp"; }

        @Override public IParamSchema paramSchema() {
            return new IParamSchema() {
                @Override public List<IParam> params() { return
Collections.<IParam>emptyList(); }
            };
        }

        @Override public ICheckResult checkPre(IState x, IOpArgs args)
{ return CheckResult.ok(); }

        @Override public IApplyResult apply(IState x, IOpArgs args) {
            IMutableState y = x.mutableCopy();
            List<String> trace = Arrays.asList("PostFailOp: applied");
            return ApplyResult.ok(y, new Evidence(null, trace, null,
metric("op","postFailOp"), null));
        }

        @Override public ICheckResult checkPost(IState y, IOpArgs args)
{ return CheckResult.fail("FORCED_POST_FAIL"); }
    }

    private static Map<String, Object> metric(String k, Object v) {
        Map<String, Object> m = new LinkedHashMap<String, Object>();
        m.put(k, v);
        return m;
    }
}
```

2.32 SimpleRuntimeHarness.java（证据闸门：可插入你的 JUnit/coverage/trace）

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public class SimpleRuntimeHarness implements IRuntimeHarness {

    private final String mustKey;
    private final String mustValue;
    private final boolean failOnAnyStep; // for harness-fail test

    public SimpleRuntimeHarness(String mustKey, String mustValue) {
        this(mustKey, mustValue, false);
    }

    public SimpleRuntimeHarness(String mustKey, String mustValue, boolean
failOnAnyStep) {
        this.mustKey = mustKey;
        this.mustValue = mustValue;
        this.failOnAnyStep = failOnAnyStep;
    }

    @Override
    public void beforePlan(IState x, IOperationPlan plan) {
        // MVP: hook for sandbox setup / counters reset
    }

    @Override
    public HarnessCheck afterStep(IState current, IOpStep step, IEvidence
stepEvidence) {
        if (failOnAnyStep) return HarnessCheck.fail("FORCED_HARNESS_FAIL");

        // Minimal property: if mustKey exists, it must not be empty.
        Object v = current.view().get(mustKey);
        if (v != null && String.valueOf(v).trim().isEmpty()) {
            return HarnessCheck.fail("EMPTY_FIELD_" + mustKey);
        }
        return HarnessCheck.ok();
    }

    @Override
    public HarnessReport afterPlan(IState x, IOperationPlan plan,
IPlanExecutionResult execResult) {
        Map<String, Object> metrics = new LinkedHashMap<String, Object>();
        metrics.put("planId", plan.planId());
        metrics.put("success", execResult.success());

        // Optional final check summary (not a gate; caller can enforce hard
target)
        Object v = (execResult.finalState() == null) ? null :
execResult.finalState().view().get(mustKey);
```

```
        metrics.put("final_" + mustKey, v);
        metrics.put("target_" + mustKey, mustValue);

        List<String> notes = new ArrayList<String>();
        notes.add("steps=" + plan.steps().size());
        return new HarnessReport(metrics, notes);
    }
}
```

2.33 SleepOp.java（用于 timeout 测试）

```
package com.dbm.ACLM.gap.affine;

import java.util.*;

public final class SleepOp implements IOperation {

    @Override public String name() { return "sleepMs"; }

    @Override
    public IParamSchema paramSchema() {
        return new IParamSchema() {
            @Override public List<IParam> params() {
                return Arrays.asList(AffineParamUtil.param("ms",
ParamType.INT, true, "sleep millis"));
            }
        };
    }

    @Override
    public ICheckResult checkPre(IState x, IOpArgs args) {
        Integer ms = args.getInt("ms");
        if (ms == null) return CheckResult.fail("MISSING_MS");
        if (ms < 0) return CheckResult.fail("NEGATIVE_MS");
        return CheckResult.ok();
    }

    @Override
    public IApplyResult apply(IState x, IOpArgs args) {
        Integer ms = args.getInt("ms");
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return ApplyResult.fail("INTERRUPTED", new Evidence(null,
Arrays.asList("SleepOp interrupted"), null, metric("op","sleepMs"), null));
        }
        // No state change; still produce evidence
        List<String> trace = Arrays.asList("SleepOp: " + ms + "ms");
        return ApplyResult.ok(x, new Evidence(null, trace, null,
metric("op","sleepMs"), null));
    }

    @Override
```

```
    public ICheckResult checkPost(IState y, IOpArgs args) {
        return CheckResult.ok();
    }

    private static Map<String, Object> metric(String k, Object v) {
        Map<String, Object> m = new LinkedHashMap<String, Object>();
        m.put(k, v);
        return m;
    }
}
```

---

3) 测试代码（src/test/java）— JUnit4 全覆盖

3.1 RuntimeEvidenceHarnessTest.java

```
package com.dbm.ACLM.gap.affine;

import org.junit.Test;

import java.util.*;

import static org.junit.Assert.*;

public class RuntimeEvidenceHarnessTest {

    @Test
    public void test_success() {
        // X
        Map<String, Object> init = new LinkedHashMap<String, Object>();
        init.put("name", "DBM");
        IState X = new MapState("X", init);

        // Plan: concatField(name, "-COT") => "DBM-COT"
        IOperation concat = new DemoOps.ConcatFieldOp();
        IOperationPlan plan = new OperationPlan("p_success",
Arrays.<IOpStep>asList(
            new OpStep(concat, new OpArgs(mapOf("key","name","suffix","-
COT")))
        ));

        // Policy allow
        IPolicy policy = new SimplePolicy(setOf("concatField"), 8, 2000,
true);

        // Harness
        IRuntimeHarness harness = new SimpleRuntimeHarness("name", "DBM-
COT");

        IPlanExecutor ex = new SimplePlanExecutor();
        IPlanExecutionResult r = ex.execute(X, plan, harness, policy);

        assertTrue(r.success());
        assertNotNull(r.finalState());
```

```
        assertEquals("DBM-COT",
String.valueOf(r.finalState().view().get("name")));
        assertTrue(r.failureChain().isEmpty());
        assertEquals(1, r.stepEvidence().size());
    }

    @Test
    public void test_apply_fail() {
        IState X = new MapState("X", mapOf1("x", "1"));

        IOperation fail = new DemoOps.AlwaysFailApplyOp();
        IOperationPlan plan = new OperationPlan("p_apply_fail",
Arrays.<IOpStep>asList(
            new OpStep(fail, new OpArgs(null))
        ));

        IPolicy policy = new SimplePolicy(setOf("alwaysFailApply"), 8, 2000,
true);
        IRuntimeHarness harness = new SimpleRuntimeHarness("x", "1");

        IPlanExecutionResult r = new SimplePlanExecutor().execute(X, plan,
harness, policy);

        assertFalse(r.success());
        assertNull(r.finalState());
        assertTrue(containsPrefix(r.failureChain(),
"APPLY_alwaysFailApply:"));
    }

    @Test
    public void test_policy_deny() {
        IState X = new MapState("X", mapOf1("name", "DBM"));

        IOperation concat = new DemoOps.ConcatFieldOp();
        IOperationPlan plan = new OperationPlan("p_policy_deny",
Arrays.<IOpStep>asList(
            new OpStep(concat, new OpArgs(mapOf("key","name","suffix","-
COT")))
        ));

        // deny by not allowing concatField
        IPolicy policy = new SimplePolicy(setOf("setField"), 8, 2000, true);
        IRuntimeHarness harness = new SimpleRuntimeHarness("name", "DBM-
COT");

        IPlanExecutionResult r = new SimplePlanExecutor().execute(X, plan,
harness, policy);

        assertFalse(r.success());
        assertTrue(r.failureChain().contains("POLICY_DENY_concatField"));
    }

    @Test
    public void test_timeout() {
        IState X = new MapState("X", mapOf1("k", "v"));

        IOperation sleep = new SleepOp();
```

```
        IOperationPlan plan = new OperationPlan("p_timeout",
Arrays.<IOpStep>asList(
            new OpStep(sleep, new OpArgs(mapOf1("ms", 50)))
        ));

        // very small runtime budget: expect timeout in executor loop
(checked before each step)
        IPolicy policy = new SimplePolicy(setOf("sleepMs"), 8, 1, true);
        IRuntimeHarness harness = new SimpleRuntimeHarness("k", "v");

        IPlanExecutionResult r = new SimplePlanExecutor().execute(X, plan,
harness, policy);

        assertFalse(r.success());
        assertTrue(r.failureChain().contains("POLICY_TIMEOUT"));
    }

    @Test
    public void test_precondition_fail() {
        // ConcatFieldOp requires existing field
        IState X = new MapState("X", mapOf1("other", "x"));

        IOperation concat = new DemoOps.ConcatFieldOp();
        IOperationPlan plan = new OperationPlan("p_pre_fail",
Arrays.<IOpStep>asList(
            new OpStep(concat, new OpArgs(mapOf("key","name","suffix","-
COT")))
        ));

        IPolicy policy = new SimplePolicy(setOf("concatField"), 8, 2000,
true);
        IRuntimeHarness harness = new SimpleRuntimeHarness("name", "DBM-
COT");

        IPlanExecutionResult r = new SimplePlanExecutor().execute(X, plan,
harness, policy);

        assertFalse(r.success());
        assertTrue(containsPrefix(r.failureChain(), "PRE_concatField:"));
    }

    @Test
    public void test_postcondition_fail() {
        IState X = new MapState("X", mapOf1("x","y"));

        IOperation postFail = new DemoOps.PostFailOp();
        IOperationPlan plan = new OperationPlan("p_post_fail",
Arrays.<IOpStep>asList(
            new OpStep(postFail, new OpArgs(null))
        ));

        IPolicy policy = new SimplePolicy(setOf("postFailOp"), 8, 2000,
true);
        IRuntimeHarness harness = new SimpleRuntimeHarness("x", "y");

        IPlanExecutionResult r = new SimplePlanExecutor().execute(X, plan,
harness, policy);
```

```
        assertFalse(r.success());
        assertTrue(containsPrefix(r.failureChain(), "POST_postFailOp:"));
    }

    @Test
    public void test_harness_fail() {
        IState X = new MapState("X", mapOf1("name", "DBM"));

        IOperation concat = new DemoOps.ConcatFieldOp();
        IOperationPlan plan = new OperationPlan("p_harness_fail",
Arrays.<IOpStep>asList(
            new OpStep(concat, new OpArgs(mapOf("key","name","suffix","-
COT")))
        ));

        IPolicy policy = new SimplePolicy(setOf("concatField"), 8, 2000,
true);

        // Force harness to fail at afterStep
        IRuntimeHarness harness = new SimpleRuntimeHarness("name", "DBM-COT",
true);

        IPlanExecutionResult r = new SimplePlanExecutor().execute(X, plan,
harness, policy);

        assertFalse(r.success());
        assertTrue(containsPrefix(r.failureChain(), "HARNESS_FAIL:"));
    }

    // ---------------- helpers ----------------

    private static boolean containsPrefix(List<String> xs, String prefix) {
        for (String x : xs) if (x != null && x.startsWith(prefix)) return
true;
        return false;
    }

    private static Map<String, Object> mapOf(String k1, Object v1, String k2,
Object v2) {
        Map<String, Object> m = new LinkedHashMap<String, Object>();
        m.put(k1, v1);
        m.put(k2, v2);
        return m;
    }

    private static Map<String, Object> mapOf1(String k1, Object v1) {
        Map<String, Object> m = new LinkedHashMap<String, Object>();
        m.put(k1, v1);
        return m;
    }

    private static Set<String> setOf(String... xs) {
        Set<String> s = new HashSet<String>();
        if (xs != null) Collections.addAll(s, xs);
        return s;
    }
```

```
}
```

---

4) 这套“证据闸门”如何对接你现有 gap 流程

你现在的 `Vertical Gap Bridging` 若已有：

- `AssemblyPlanner` 产出候选路径/桥接步骤
- `UnalignedAndScorer` 负责打分与解释链

那么对接点非常直接：

1. **把候选路径统一表达成** `IOperationPlan`（步骤就是 `IOpStep(op,args)`）
2. **在计划执行阶段**调用 `SimplePlanExecutor.execute(X, plan, harness, policy)`
3. **把证据链回灌**：
   - `stepEvidence().trace/diff/metrics` → 解释链第一层（runtime）
   - `failureChain()` → LLM 修复 prompt 的稳定 token（PRE/POST/HARNESS/POLICY/APPLY）
4. 你已有 `contributionCost/ladder distance`：
   - 建议写到 `Evidence.contribution()` 或 `PlanExecutionResult.summaryMetrics()`，和 runtime evidence 合并为一条“对齐解释链”。

---

5) 你落地后立刻会得到的收益

- 你现在可以把 LLM 输出的候选计划当“提案”，但最终**只能通过 Evidence Harness 证据闸门**进入下一步固化（ITEM #181 的精神落到代码里了）。
- 失败原因是**稳定 token**（`PRE_xxx`, `POST_xxx`, `HARNESS_FAIL:xxx`, `POLICY_*`），非常适合反向喂给 LLM 做 targeted regeneration，而不是盲改。

如果你下一步希望我把这套再“完全贴合你 gap 包既有命名风格”（例如：你已有 `*Kit / *Planner / *Scorer / *Policy` 的命名习惯、或你已有 `ITrace / IDiff / IEvidenceChain` 类型），我可以在不改算法的前提下，把这些类名与接口槽位再做一次“贴合式重命名与适配层”，让你几乎零摩擦接入现有代码流。