Instructions for an intelligent agent

1 Instructions for ATWL Workflow Review

1.1 Role of the Reviewer

The reviewer evaluates ATWL workflow representations created from research papers. The review covers the following objectives:

1. **Verify correctness** against ATWL syntax and semantics
2. **Check alignment** with the source paper's methodology
3. **Assess abstraction level** appropriateness
4. **Identify issues** and suggest specific corrections

Reviews should provide **constructive, specific feedback** with examples of corrections.

1.2 Review Framework Overview

The review proceeds through five phases:

1. Structural Validation
2. Semantic Correctness
3. Paper Alignment
4. Abstraction Level Assessment
5. Overall Quality Judgment

1.3 Phase 1: Structural Validation

1.3.1 Required Elements

Workflow Header. Verify that the workflow:

- Has a `workflow <ID>` declaration (no trailing colon)
- Has a `template:` with intent sequence
- Has a `description:` summarising purpose
- Template uses only valid intents or `loop(...)`
- Template matches the actual body structure (same sequence of intents, loops correspond to loop constructs)

Common issues include missing templates, templates that do not match the workflow body structure, trailing colons on the workflow declaration, and non-standard syntax in templates.

Artifact Declarations. For each artifact, verify:

- Has a valid type: `entities`, `feature`, `arrangement`, `visualisation`, `pattern`, `knowledge`, `specification`, `model`
- `origin: given` used only for exogenous artifacts not produced by any transform
- Artifacts without `origin: given` are outputs of some transform
- Artifact references in parentheses are valid (e.g., `feature(entity_ID)`)
- Comments use `#` syntax (not `//`)

Common issues include `origin: given` on artifacts that are transform outputs, missing `origin: given` on truly exogenous artifacts, invalid artifact type names, `//` comment syntax, and circular dependencies.

Entities Artifacts. For each `entities` artifact, verify:

- Has `internal structure:` with a standard value: `elementary`, `group/cluster`, `episode`, `region`, `sequence`, or `formation`
- Has `embedment:` when the artifact represents a collection of entities positioned relative to each other. Values (combinable with set notation): `set`, `sequence`, `time`, `space`, `relational structure`
- `embedment` is *omitted* when the artifact represents a single entity (e.g., a single aggregation tree, a single reference frame, a single network)

- Internal features use the unified two-level scheme: `value structure` (`atomic`, `list`, `vector`, `matrix`, `relational configuration`) and optional `value type` (`numeric`, `ordinal`, `categorical`, `temporal`, `spatial`, `text`, `reference`). Mixed types use set notation (e.g., `{numeric, temporal}`)
- Internal features are declared under a `features:` keyword
- Internal features are included only when they are consumed by downstream transforms; omit features that serve no analytical purpose in the workflow

Common issues include non-standard internal structure values (e.g., `network` instead of `formation`, `hierarchical` instead of `formation`, `group` instead of `group/cluster`); embedment on single entities (e.g., a projection reference space); missing embedment on genuine collections; old-style `value type:` without `value structure:` on internal features; missing `features:` keyword; and non-standard value types (e.g., `binary`, `hierarchical`, `set`).

Reference Space Entities. When a `contextualise` transform creates a reference frame (e.g., a 2D projection space), this is modelled as a single `entities` artifact:

- `internal structure: elementary`
- No `embedment` (single entity)
- Features describe the frame itself (e.g., number of dimensions), not positions within it

```
# CORRECT:

artifact projection_space : entities
    internal structure: elementary
    features:
      - id: axes

        value structure: atomic
        value type: numeric
        description: "Number of spatial dimensions (2)"
    description: "Two-dimensional projection space"

# WRONG:

artifact projection_space : entities
    internal structure: elementary
    embedment: space          # single entity, no embedment
    features:
      - id: coordinates       # describes positions, not frame

        value type: spatial
```

Feature Artifacts. For each `feature` artifact, verify:

- References an `entities` artifact: `feature(entities_ID)`
- Has `value structure:` using the same vocabulary as internal features: `atomic`, `list`, `vector`, `matrix`, `relational configuration`
- Optionally has `value type:` using standard types (single or set notation for mixed)
- Optionally has `representation form:` for additional clarification
- Does *not* use `representation form` as a substitute for `value structure`

Common issues include using `representation form` instead of `value structure`; referencing non-entity artifacts (e.g., `feature(visualisation_ID)`); and using the old single-field `value type:` syntax without `value structure:`.

Feature representation form. The optional `representation form:` on feature artifacts should be used only when `value structure:` alone is ambiguous about the concrete encoding:

- `relational configuration` — e.g., `"k-NN graph"`, `"directed flows with time series"`
- `vector` — e.g., `"time series per place"`, `"daily measurement profile"`
- `matrix` — e.g., `"pairwise similarity matrix"`, `"term-document matrix"`

Do *not* use `representation form:` as a substitute for `value structure:`, and omit it for `atomic` and `list` structures where `value type:` is sufficient.

Visualisation Artifacts. For each `visualisation` artifact, verify:

- Has `layout:` describing spatial organisation
- Has `form:` describing mark or glyph types
- Has `encoding:` describing mapping of data attributes to visual channels (recommended)
- Does *not* use a single `representation form` field as a substitute for the three visualisation-specific fields
- References in parentheses point to data artifacts, not other visualisations

Common issues include using `representation form` instead of `layout`/`form`/`encoding`; and referencing other visualisation artifacts in parentheses.

Knowledge Artifacts. For each `knowledge` artifact, verify:

- Has `representation form:` describing a *tangible* output form (e.g., `"statements and explanations"`, `"quality judgment"`, `"statements and recommendations"`)
- Does *not* describe a mental state (e.g., `"understanding"`, `"comprehension"`)

ATWL represents only explicit, tangible artifacts. Knowledge artifacts should describe recorded, communicable outputs.

Specification Artifacts. For each `specification` artifact, verify:

- Has `representation form:` describing the structure (e.g., `"parameter settings"`, `"query criteria"`, `"model configuration"`)
- Description clarifies prescriptive nature (controls operations)
- Not marked with `origin: given` unless truly an external default
- Does *not* have an `applies to:` field (not valid ATWL syntax; the relationship to controlled transforms is implicit from the dataflow)

Common issues include missing `representation form`; using `pattern` type for specifications; and the non-standard `applies to:` field.

Model Artifacts. For each `model` artifact, verify:

- Has parameterisation in parentheses referencing artifacts from which the model is built: `model(training_data, spec)` — except for `origin: given` models where training inputs are external to the workflow
- Has a `description:`

Common issue: `model` without parameterisation for derived (non-given) models.

Transform Declarations. For each transform, verify:

- Has a valid `intent:` (`define-unit`, `characterise`, `contextualise`, `visualise`, `abstract`, `assess`, `build-model`, `generate-knowledge`)
- Has `input:` referencing declared artifacts
- Has `output:` declaring produced artifacts
- Has `actor:` (`human`, `machine`, or `hybrid`)
- Has `description:`
- Optional `manner:` is reasonable
- Consistent field ordering across the workflow (recommended: `intent` → `manner` → `input` → `output` → `actor` → `description`)

Common issues include invalid intent names, missing required fields (especially `description`), undeclared input/output artifacts, missing or invalid actor types, and inconsistent field ordering.

Control Structures. For loops, verify:

- Has `loop <ID>:` with unique ID
- Has `purpose:` describing analytical goal
- Has `until:` with qualitative termination condition

- Contains `body:` section
- Ends with `end loop <ID>` matching opening ID
- Assignment statements use `:=` syntax
- Does *not* use `continue loop` (not a valid ATWL construct)

Common issues include missing purpose or until statements, quantitative thresholds in `until` instead of qualitative goals, mismatched loop IDs, use of the non-standard `continue loop` construct, and assignment syntax errors.

Loop Termination Patterns. Two termination styles are valid:

Pattern A: Explicit assessment with conditional exit. Used when the loop body contains an assess transform and a refinement step that should execute only when assessment indicates further work is needed:

- An `assess` transform produces evaluative knowledge
- An `if`–`then`–`else` conditional gates the refinement
- One branch contains substantive transforms (refinement, re-specification) typically with `:=` assignment
- The other branch contains `exit loop <ID>` (optionally preceded by one-time terminal transforms)
- `exit loop` may appear in either the `then` or `else` branch, depending on how the condition is phrased
- The branch opposite `exit loop` may be empty or contain only a comment when the loop simply re-enters (e.g., "continue exploring"); however, when a loop's purpose involves parameter refinement, the non-exit branch must contain substantive transforms and an `assign` statement closing the feedback — comment-only branches that hide parameter adjustment are not permitted

```
# exit in else:

if assessment indicates refinement needed:
    then:
        transform T_refine : ...
        assign: spec := updated_spec
    else:
        exit loop L1

# exit in then (equally valid):

if assessment indicates satisfactory:
    then:
```

```
        exit loop L1
    else:
        transform T_refine : ...
        assign: spec := updated_spec
```

Pattern B: Implicit termination. Used when all transforms execute every iteration and the loop continues until the analyst is satisfied, with no conditionally gated transforms:

- No conditional needed; the `until:` clause governs termination
- Appropriate when the stopping criterion reflects gradual, holistic change in the analyst's understanding that cannot be naturally reduced to a single assessable artifact
- An `assess` transform may still appear in the body (its output feeds post-loop synthesis) but does not gate any conditional

Common issues include:

- Conditionals where both branches execute no differential transforms (the loop should use Pattern B instead)
- Comment-only non-exit branches that *hide parameter adjustment* — if the paper describes iterative parameter tuning, the branch must contain substantive transforms with `assign`
- Empty or comment-only non-exit branches in *exploration* loops where iteration is driven by the analyst choosing different facets to examine — such loops should use Pattern B instead, removing the conditional entirely
- Missing conditionals when a refinement step should be gated
- Using the non-standard `continue loop` construct instead of letting the loop body fall through naturally

Assignment Contexts. Assignments appear in two valid contexts:

- **Inside a loop body**, to express iterative update
- **Before a loop**, to initialise an artifact identifier that will be re-assigned within the loop

```
# Pre-loop initialisation:

assign:
    D_current := D_all

loop L1:
```

```
    ...
    assign:
        D_current := D_next
end loop L1
```

Post-Loop Terminal Actions. Transforms that execute once after iteration is complete (e.g., final model selection, knowledge synthesis) may be placed either inside the exit branch (before `exit loop`) or after `end loop`. Placing them after `end loop` is preferred when the same transform would otherwise be duplicated across multiple exit points, or when the transform is conceptually a separate analytical phase rather than the conclusion of the iterative process.

Nested Loops.

- Nested loops in the workflow **body** are permitted when the paper describes genuinely nested iteration
- Nested loops in the **template** are **not** permitted; the template must flatten nested iteration to a single loop level

1.3.2 Artifact Dependency Graph

Verify no cycles exist except in loops with explicit assignment:

- Each artifact's dependencies should form an acyclic graph
- Circular dependencies only through explicit `assign:` statements in loops
- All input artifacts should be declared before use (or produced by earlier transforms)

Artifact Provenance. Every artifact must have exactly one of:

- `origin: given` (exogenous input)
- A producing transform (appears in some transform's `output:`)
- An `assign:` statement binding it to another artifact (for pre-loop initialisations or iterative updates within loops)

Artifacts that have none of these are *orphan artifacts*.

Dead-End Artifacts. Every non-terminal artifact should be consumed as input by at least one downstream transform. Artifacts that are produced but never consumed indicate broken dataflow. This is particularly critical inside loop conditionals, where a refinement artifact produced in the `then` branch must be consumed in the next iteration (via `:=` assignment to an artifact that appears in a loop-body transform's input list).

Exception: Terminal knowledge artifacts (the final output of the workflow) are naturally not consumed by any downstream transform. The dead-end check applies to intermediate artifacts only.

1.4 Phase 2: Semantic Correctness

1.4.1 Intent Appropriateness

For each transform, verify the intent matches its purpose:

define-unit Creating or redefining analysis units.

- Should: Extract, filter, cluster, segment, partition, merge, group, select subsets
- × Should not: Compute features, evaluate quality, discover patterns

characterise Computing features or descriptors of entities.

- Should: Calculate attributes, aggregate statistics, compute distances, encode values, apply model for prediction, assign labels from existing groupings
- × Should not: Create entities, make decisions, evaluate quality

contextualise Arranging entities in reference structures.

- Should: Position in space/time/projection, create arrangements with a reference context
- Must produce: An `arrangement` artifact (and optionally the reference space entity)
- × Should not: Just transform coordinates without creating an arrangement

visualise Creating visual representations.

- Should: Produce `visualisation` artifacts from data, feature, arrangement, pattern, and specification artifacts
- × Should not: Produce non-visual artifacts as primary output
- × Should not: Take other visualisation artifacts as input — *except* in interactive exploration loops where a `visualise` transform updates a previous view (e.g., expanding clusters, switching overlays). In such cases, the previous visualisation state is a legitimate input, and the updated visualisation is assigned back via `:=`

abstract Discovering or refining patterns in data.

- Should: Identify structures, regularities, motifs; interpret cluster meanings; synthesise observations into named patterns; consolidate or refine existing patterns (e.g., merging semantically similar topics)
- Produces: `pattern` artifacts
- × Should not: Evaluate quality (use `assess`), make strategic decisions, produce concrete groups of identified objects (use `define-unit`)

assess Evaluating quality of results.

- Should: Judge adequacy, evaluate quality, decide if refinement needed
- Produces: `knowledge` artifacts (evaluative judgments); may also produce `specification` artifacts (parameter adjustments triggered by assessment)
- × Should not: Discover patterns in data (use `abstract`)

build-model Constructing predictive or explanatory models.

- Should: Train, calibrate, fit models
- Produces: `model` artifacts
- × Should not: Evaluate model quality (use `assess`)

generate-knowledge Formulating explicit knowledge or methodological decisions.

- Should: Synthesise insights, formulate conclusions, make strategic decisions about analysis direction
- Produces: `knowledge` artifacts; may also produce `specification` artifacts when the analyst's primary act is formulating a methodological decision
- × Should not: Just describe patterns (use `abstract`)

1.4.2 Characterise vs. Define-Unit for Labels

When clustering produces both groups (entities) and per-entity membership labels (feature), both outputs come from the same `define-unit` transform — the labels are an inherent by-product of group creation.

However, when a pre-existing model or grouping is *applied* to assign labels without creating new entities (e.g., assigning documents to their dominant pre-computed topic), the intent is `characterise` — it computes a feature, not new units.

```
# VALID (creates groups + labels):

transform T_cluster :
    intent: define-unit
    output: clusters, cluster_labels

# WRONG (applies existing grouping):

transform T_assign_topics :
    intent: define-unit
    output: topic_assignments  # feature, not entities

# CORRECT:

transform T_assign_topics :
    intent: characterise
    output: topic_assignments
```

1.4.3 Artifact Type Appropriateness

Common Mismatches to Check.

specification vs. pattern:

```
# WRONG:

artifact clustering_params : pattern(clusters)
    description: "Parameters for clustering algorithm"

# CORRECT:

artifact clustering_params : specification
    representation form: "parameter settings"
    description: "Distance thresholds and sensitivity
        parameters"
```

knowledge vs. pattern (from assess):

```
# WRONG:

artifact quality_assessment : pattern(results)

# CORRECT:

artifact quality_assessment : knowledge(results)
    representation form: "quality judgment"
```

pattern vs. knowledge (from abstract):

```
# WRONG:

artifact route_types : knowledge(trajectories)

# CORRECT:

artifact route_types : pattern(trajectories)
    representation form: "categorized route types"
```

feature vs. pattern (from characterise): Symbolic encoding, vectorisation, and other data transformations produce features (descriptors of entities), not patterns (abstractions of regularities).

```
# WRONG:

transform T_encode :
    intent: characterise
    output: symbolic_patterns
artifact symbolic_patterns : pattern(episodes)
```

```
# CORRECT:

artifact symbolic_codes : feature(episodes)
    value structure: vector
    value type: categorical
```

entities vs. pattern (concrete vs. abstract): When an operation produces concrete collections of identified objects (e.g., sets of structurally equivalent components), the result is `entities`, not `pattern`.

```
# WRONG:

artifact structural_templates : pattern(graph)

# CORRECT:

artifact structural_templates : entities
    internal structure: group/cluster
    embedment: set
```

1.4.4 Artifact References

Verify correct references in parentheses:

- **feature artifacts** must reference entities: `feature(entities_ID)`
- **arrangement artifacts** must reference entities being arranged and have a distinct context entity; the entity being arranged must not be its own context
- **visualisation artifacts** should reference the data artifacts they depict, not other visualisations
- **knowledge artifacts** should reference what they are about
- **pattern artifacts** should reference source data artifacts
- **model artifacts** (non-given) should reference artifacts from which they are built

1.4.5 Given Artifact Dependencies

Artifacts marked `origin: given` must not reference derived artifacts in their parameterisation or feature declarations. A given artifact is exogenous to the workflow and cannot depend on artifacts that do not yet exist at workflow start.

```
# WRONG (given artifact references derived entity):

artifact context_info : feature(episodes)
```

```
    origin: given
    # episodes is derived by T_define_episodes

# CORRECT (derive from given + derived inputs):

transform T_characterise_context :
    intent: characterise
    input: episodes, raw_timeseries
    output: context_info

artifact context_info : feature(episodes)
    value structure: vector
    value type: {temporal, spatial, categorical}
```

1.4.6 Loop-Invariant Transforms

Transforms whose inputs never change between loop iterations should be placed **before** the loop, not inside it. This clarifies which computations are iteratively refined and which are performed once.

```
# WRONG (T_arrange inputs are loop-invariant):

loop L1:
    body:
        transform T_cluster : ...
        transform T_arrange :
            input: D_day, D_calendar  # never change
            output: A_calendar
        transform T_visualise :
            input: A_calendar, cluster_labels
            ...
end loop L1

# CORRECT:

transform T_arrange :
    input: D_day, D_calendar
    output: A_calendar

loop L1:
    body:
        transform T_cluster : ...
        transform T_visualise :
            input: A_calendar, cluster_labels
            ...
end loop L1
```

The arrangement `A_calendar` is computed once before the loop and remains accessible inside it. Only transforms that depend on iteratively updated artifacts belong in the loop body.

1.5 Phase 3: Paper Alignment

1.5.1 Workflow Structure Matches Paper

Main Phases. Compare workflow phases to the paper's methodology sections:

- Does the template capture the paper's main analytical progression?
- Are major steps represented as transforms?
- Are iterative procedures shown as loops?
- Are decision points captured in conditionals?

Procedure: list the paper's methodology steps, map to workflow transforms, identify missing or extra steps, and check if the order matches.

Iterative Refinement. Papers often describe iteration with phrases such as "iteratively adjusted parameters until...", "repeated with different settings...", "user guidance to refine...". Verify:

- Loops present where paper describes iteration
- Assessment steps included where paper describes quality evaluation
- Parameter adjustment mechanisms captured as specification artifacts with explicit `assign` statements closing the feedback loop
- No loops with implicit parameter changes (comment-only `then` branches)

1.5.2 Terminology Matches Paper's Domain

Artifact Names. Should reflect the paper's terminology while remaining generic:

```
# GOOD: Uses paper's terminology
artifact trajectories : entities

# TOO GENERIC: Loses domain context
artifact data : entities

# TOO SPECIFIC: Dataset-specific
artifact milan_car_trajectories : entities
```

Transform Descriptions. Should align with the paper's described operations without including implementation specifics:

```
# GOOD:

description: "Cluster trajectories by destination
    similarity"

# WRONG: Too implementation-specific

description: "Apply k-means clustering"

# WRONG: Not in paper

description: "Cluster trajectories by source similarity"
```

1.5.3 Key Contributions Captured

Identify the paper's main methodological contributions and verify:

- Core methodological innovation is represented
- Not over-detailed (algorithm specifics)
- Not under-detailed (missing key stages)

For example, if a paper contributes "two-stage clustering (spatio-temporal then spatial)," the workflow should show both stages explicitly.

1.6 Phase 4: Abstraction Level Assessment

1.6.1 Too Detailed (Over-Specification)

Warning signs:

Algorithm names in intent or manner:

```
# TOO DETAILED:

manner: "DBSCAN with eps=0.5 and min_samples=5"

# APPROPRIATE:

manner: "density-based clustering"
```

Specific numeric results in descriptions:

```
# TOO DETAILED:

description: "Five clusters representing 23%, 18%..."

# APPROPRIATE:
```

```
description: "Clusters grouping trajectories with
    similar characteristics"
```

Tool/library names:

```
# TOO DETAILED:

manner: "D3.js force-directed layout"

# APPROPRIATE:

manner: "force-directed graph layout"
```

Verbose descriptions: Descriptions and manner fields should be concise. Unnecessary repetition and elaboration reduce readability and hinder workflow comparison.

1.6.2 Too Abstract (Under-Specification)

Warning signs:

Missing essential analytical steps:

```
# TOO ABSTRACT:

template: prepare → analyze → visualize → conclude

# APPROPRIATE:

template: characterise → define-unit (cluster) →
          visualise → assess → generate-knowledge
```

Vague descriptions:

```
# TOO ABSTRACT:

description: "Process the data"

# APPROPRIATE:

description: "Compute movement characteristics: speed,
    direction, acceleration"
```

1.6.3 Appropriate Abstraction Level

Characteristics:

- Captures analytical approach without algorithm details
- Domain context present but not dataset-specific

- Method types specified (density-based, hierarchical) but not implementations
- Artifact roles clear without technical formats
- Iteration patterns evident without exact convergence criteria
- Enables comparison across similar workflows
- Descriptions are concise

1.7 Phase 5: Common Issues Catalogue

This section catalogues issues frequently encountered in LLM-generated ATWL workflows, organised by category.

1.7.1 Feature and Visualisation Syntax

Issue: `representation form` substituted for type-specific fields. The single most common systematic error. LLM agents frequently use a generic `representation form` field on feature and visualisation artifacts instead of the type-specific mandatory fields.

```
# WRONG (feature):

artifact F1 : feature(D1)
    representation form: "per-record similarity scores"

# CORRECT:

artifact F1 : feature(D1)
    value structure: atomic
    value type: numeric
    description: "Per-record similarity scores"

# WRONG (visualisation):

artifact V1 : visualisation(D1)
    representation form: "interactive scatter plot"

# CORRECT:

artifact V1 : visualisation(D1)
    layout: "2D scatterplot"
    form: "colored points"
    encoding: "position from arrangement; color by
        cluster"
```

Issue: Old-style internal features. Internal features declared with only `value type:` instead of the unified two-level scheme.

```
# OLD STYLE:

features:
  - id: speed

    value type: numeric

# CURRENT:

features:
  - id: speed

    value structure: atomic
    value type: numeric
```

1.7.2 Entities and Embedment

Issue: Embedment on single entities. Reference spaces, single aggregation trees, and single networks are single analytical objects. Embedment describes how *multiple* entities relate to each other and should be omitted for single entities.

Issue: Missing embedment on collections. Collections of records, events, or clusters positioned relative to each other should declare embedment. Spatial events need `space` in their embedment even if spatial position is also listed as a feature.

Issue: Non-standard internal structure values.

```
# WRONG:

internal structure: network     # use: formation
internal structure: hierarchical  # use: formation
internal structure: group       # use: group/cluster
```

1.7.3 Intent–Output Mismatches

Issue: `define-unit` producing features. Assigning labels (e.g., topic assignments) is computing a feature value, not creating entities. Use `characterise`.

Exception: `define-unit` producing both entities and label features. When clustering creates new groups *and* assigns membership labels simultaneously, both outputs from a single `define-unit` transform are valid — the labels are an inherent by-product of group creation. The issue arises only when an existing grouping is *applied* to assign labels without creating new entities; in that case, use `characterise`.

```
# VALID (creates groups + labels):

transform T_cluster :
    intent: define-unit
```

```
    output: clusters, cluster_labels

# WRONG (applies existing grouping):

transform T_assign_topics :
    intent: define-unit
    output: topic_assignments  # feature, not entities

# CORRECT:

transform T_assign_topics :
    intent: characterise
    output: topic_assignments
```

Issue: `characterise` producing patterns. Encoding, vectorisation, and other data transformations produce features. Use `abstract` for pattern discovery.

Issue: `abstract` producing entities. When the result is concrete identified objects (sets, groups), use `define-unit`.

1.7.4 Pattern vs. Feature from Characterise

Issue: `characterise` producing patterns. Data transformations such as symbolic encoding (e.g., SAX), vectorisation, normalisation, and feature extraction produce *descriptors* of entities, not abstractions of regularities. Their outputs should be typed as `feature`, not `pattern`, even when the method name includes the word "pattern."

```

# WRONG:

transform T_encode :
    intent: characterise
    output: symbolic_patterns

artifact symbolic_patterns : pattern(episodes)

# CORRECT:

transform T_encode :
    intent: characterise
    output: symbolic_codes

artifact symbolic_codes : feature(episodes)
    value structure: vector
    value type: categorical
```

The downstream `abstract` transform that discovers co-occurrence topics from these codes produces the actual `pattern` artifacts.

1.7.5 Loop Feedback

Issue: Loops with no `assign` closing the feedback. When a loop uses Pattern A (explicit conditional), the `then` branch must contain substantive transforms and an `assign` statement. Comment-only branches indicate that the parameter adjustment mechanism is invisible to ATWL. Add a specification artifact and a `generate-knowledge` (or `assess`) transform producing it.

```
# WRONG (then branch is comment-only):

if assessment indicates refinement needed:
    then:
        # Continue with adjusted parameters
    else:
        exit loop L1

# CORRECT:

if assessment indicates refinement needed:
    then:
        transform T_adjust :
            intent: generate-knowledge
            ...
            output: S_params'
        assign: S_params := S_params'
    else:
        exit loop L1
```

Issue: `continue loop` construct. Not a valid ATWL construct. Restructure the conditional so that the substantive work is in one branch and `exit loop` is in the other, letting the loop naturally re-enter after the substantive branch completes.

1.7.6 Knowledge Tangibility

Issue: Mental states as representation form.

```
# WRONG:

representation form: "architectural understanding"
representation form: "comprehensive model assessment"

# CORRECT:

representation form: "statements and diagrams"
representation form: "statements and recommendations"
```

1.7.7 Model Parameterisation

Issue: Derived models without parameterisation.

```
# WRONG:

artifact M1 : model
    description: "Trained classifier"

# CORRECT:

artifact M1 : model(training_data, model_spec)
    description: "Trained classifier"
```

Given models (`origin: given`) may omit parameterisation since training inputs are external.

1.7.8 Description Issues

Issue: Specific results in descriptions.

```
# WRONG:

description: "Finding that 23% of trips are local"

# CORRECT:

description: "Understanding of trip distance
    distributions and route type organization"
```

Issue: Implementation-focused descriptions.

```
# WRONG:

description: "Apply DBSCAN with eps=0.5"

# CORRECT:

description: "Group events with similar spatial
    positions using density-based clustering"
```

1.8 Review Report Template

The reviewer should produce a structured report:

```
## ATWL Workflow Review Report

Paper Title: [Title]
```

```
Workflow ID: [workflow-id]

---

### Overall Assessment

Summary: [2-3 sentence overall assessment]

Recommendation:
[ ] Approve as-is
[ ] Approve with minor revisions
[ ] Requires revision
[ ] Reject - major rework needed

---

### Strengths

- [Strength 1]
- [Strength 2]
- [Strength 3]

---

### Critical Issues (Must Fix)

#### Issue 1: [Title]

Location: [Transform/Artifact ID]
Problem: [Specific description]
Correction: [Corrected ATWL code]
Why: [Explanation]

[Repeat for each critical issue]

---

### Recommended Improvements (Should Fix)

#### Improvement 1: [Title]

Location: [Transform/Artifact ID]
Suggestion: [Specific recommendation]
Why: [Explanation]

[Repeat for recommendations]

---

### Minor Suggestions (Optional)

- [Suggestion 1]
```

```
- [Suggestion 2]

---

### Alignment with Paper

Captured from paper:
- [pass/fail] Main methodology phases
- [pass/fail] Iterative refinement procedures
- [pass/fail] Key methodological contributions
- [pass/fail] Domain terminology

Missing or incorrect:
- [List any missing or misrepresented elements]

---

### Conclusion

[Final paragraph summarizing assessment and
next steps]
```

1.9 Quick Reference Checklist

1.9.1 Red Flags (Critical Issues)

- `representation form` used instead of `value structure` on feature artifacts
- `representation form` used instead of `layout`/`form`/`encoding` on visualisation artifacts
- `assess` producing `pattern` artifacts
- Parameters/settings as `pattern` artifacts
- `origin: given` on transform outputs
- Missing `loop(...)` when paper describes iterative refinement
- Intent types clearly mismatched to purposes
- Comment-only conditional branches that hide parameter adjustment (should contain substantive transforms with `assign`)
- `continue loop` construct
- `applies to:` field on specifications
- Orphan artifacts (no provenance)
- Dead-end artifacts in loop conditionals

- Non-standard internal structure values
- Embedment on single entities
- Knowledge representation form describing mental states
- Trailing colon on workflow declaration
- Given artifacts referencing derived artifacts in parameterisation
- `characterise` intent producing `pattern` artifacts
- Loop-invariant transforms inside loop bodies
- `//` comment syntax (use `#`)

1.9.2 Yellow Flags (Likely Issues)

- `abstract` used for quality evaluation
- `characterise` used for strategic decisions
- `define-unit` producing features instead of entities
- `abstract` intent producing `entities` instead of `pattern` (verify whether the result is concrete objects or abstracted regularities)
- Missing embedment on entity collections
- Missing `value type` on feature artifacts (optional but improves clarity)
- Missing `description` on transforms
- Template not matching body structure
- Model artifacts without parameterisation
- Specification artifacts missing `representation form:`
- Arrangement context referencing the same entity being arranged
- Post-loop terminal action duplicated across multiple exit branches (consider moving after `end loop`)
- Comment-only non-exit branch in a refinement loop (verify that no parameter adjustment is being hidden)
- Verbose descriptions without added information
- Internal features with no downstream consumers

- Old-style `value type:` without `value structure:` on internal features
- Transforms with inputs that never change between loop iterations (candidates for moving before the loop)

1.9.3 Green Flags (Quality Indicators)

- Clear template matching body structure
- `assess` produces `knowledge`
- `abstract` produces `pattern`
- Feature artifacts have `value structure` (and optionally `value type`)
- Visualisation artifacts have `layout`/`form`/`encoding`
- Embedment present on collections, absent on single entities
- Generic but domain-appropriate terminology
- Loop conditionals gate substantive transforms with `assign` closing feedback
- All non-given artifacts traceable to a producing transform
- Post-loop terminal actions clearly placed (consistently after `end loop` or within exit branches)
- Concise, informative descriptions
- Knowledge representation forms describe tangible outputs
- Consistent transform field ordering

1.10 Final Guidance

Reviews should be:

Specific Point to exact locations with artifact/transform IDs

Constructive Provide corrected code examples

Educational Explain why changes are needed

Prioritised Critical issues first, then recommendations, then suggestions

Balanced Acknowledge strengths as well as issues

Key principles to remember:

- ATWL represents analytical logic, not implementation details

- Features use `value structure` + `value type`; visualisations use `layout`/`form`/`encoding`
- `assess` → `knowledge` (and optionally `specification`); `abstract` → `pattern`
- `generate-knowledge` may produce both `knowledge` and `specification`
- Specifications control operations; patterns describe data regularities
- Embedment characterises collections, not single entities
- Knowledge artifacts represent tangible, recorded outputs
- Every non-given artifact needs a producer; every non-terminal artifact needs a consumer
- Loop feedback must be closed with explicit `assign` statements
- Given artifacts must not reference derived artifacts
- Loop-invariant transforms belong before the loop
- Abstraction level should enable workflow comparison

The goal is helping create high-quality ATWL representations that accurately capture research workflows at appropriate abstraction for comparison and reuse.