# Artifact–Transform Workflow Language (ATWL) Clean Specification

## Motivation and Use Cases

Visual analytics workflows are often complex, involving multiple stages of data transformation, feature computation, visualization, and interpretation. ATWL provides a structured way to represent these workflows explicitly, supporting several important use cases:

**Documentation and communication.** Create precise, readable descriptions of analytical processes for research papers, technical reports, and team collaboration. Unlike informal diagrams or prose descriptions, ATWL specifications capture both the structure and the analytical intent of workflows.

**Workflow comparison and analysis.** Systematically compare different analytical approaches by examining their artifact types, transform sequences, and control structures. This supports meta-analyses of visual analytics methods and identification of common patterns.

**Reproducibility and transparency.** Provide a declarative specification that clarifies analytical intent independent of implementation details, facilitating replication and adaptation of published methods.

**Tool development and automation.** Enable automated tools for workflow validation, visualization, recommendation, and comparison. The structured format supports computational analysis of workflows while remaining human-readable.

**Education and training.** Support teaching of visual analytics methodology by providing a common vocabulary and structure for discussing analytical strategies.

**Target users.**

- Visual analytics researchers documenting methods and workflows
- Tool developers building workflow management systems
- Educators teaching visual analytics methodology
- Practitioners seeking to understand and communicate analytical processes
- Researchers conducting meta-analyses of visual analytics approaches

**Relationship to other workflow representations.** Unlike executable workflow languages (e.g., scientific workflow systems like Taverna, Kepler), ATWL focuses on analytical intent rather than operational implementation. Unlike UML activity diagrams, ATWL provides specific constructs for visual analytics. Unlike pure provenance models, ATWL captures forward-looking analytical strategies and explicitly represents human reasoning steps.

# 1 Purpose and Design Principles

ATWL is a declarative language for representing visual analytics workflows as structured transformations of artifacts. The language is intentionally minimal in its core constructs, while allowing extensibility through lightweight ontologies and free-form annotations. This section provides an overview of the main components of the language and how they relate to each other.

# 2 Core Building Blocks

At its core, ATWL consists of the following elements:

- **Artifacts**, which represent identifiable analytical objects created, modified, and consumed during analysis;
- **Transforms**, which describe how artifacts are derived, modified, arranged, represented, or interpreted;
- **Control structures**, which specify sequencing, branching, and iteration within workflows;

## 2.1 Workflows

A workflow in ATWL is a directed structure in which transforms consume one or more artifacts as input and produce one or more artifacts as output. Control structures specify how transforms are composed into workflows, but do not themselves produce or consume artifacts. The meaning of a workflow is determined primarily by the types, roles, and sequencing of the artifacts involved, rather than by specific operational or implementation details.

A workflow representation may optionally include a **template**: a concise sequence of transform intents summarising the main analytic stages.

## 2.2 Artifacts: Overview

ATWL uses a fixed set of artifact types:

- `entities`: a collection of identifiable analytical objects of the same kind. Each entity is a (possibly structured) piece of data treated as a single object for the purposes of analysis.
- `feature`: explicit descriptors of one or more properties of entities (for example, based on their attributes, internal structure, or relationships).
- `arrangement`: the organisation of artifacts within a reference structure (context).
- `visualisation`: an external visual representation for human perception.
- `pattern`: abstracted regularities or structures identified in entities.
- `model`: a formal or computational representation used for prediction, simulation, or explanation.
- `knowledge`: explicitly formulated knowledge (statements, rules, conclusions) produced or injected in the analysis.
- `specification`: an explicit description of parameters, settings, constraints, or method choices.

**Note 1**: Artifact type declarations may include references to other artifacts in parentheses, e.g., feature(D1) indicates a feature describing entities D1.

**Note 2**: A knowledge artifact represents explicitly formulated knowledge (e.g., statements, rules, explanations, or recommendations), as opposed to the analyst's internal, tacit knowledge.

### 2.3 Transforms: Overview

Each **transform**:

- consumes one or more input artifacts;
- produces one or more output artifacts;
- has a single generic *intent* (e.g., `define-unit`, `characterise`);
- may have a more specific `manner` (e.g., `extract`, `cluster-by-similarity`, `time-partitioning`) that refines how the intent is realised;
- specifies an `actor` (`human`, `machine`, or `hybrid`).

## 3 Artifacts

### 3.1 Entities Artifact

**Entities** are sets of items treated as objects of analysis (e.g., events, days, episodes, trajectories, snapshots, places, documents, images). An entity is a (possibly structured) piece of data that is treated as a single object for the purposes of analysis. Entities may correspond directly to original data items or may be derived through transformations such as aggregation, segmentation, grouping, or abstraction. Since entities are pieces of data, the term *data unit* may occasionally be used informally to refer to an entity.

In most cases an entities artifact contains multiple entities, but it may also contain a single entity when the analysis concerns a single structured object (e.g., a network or document). All entities in a given artifact share the same type of internal structure.

### Entities Declaration

```
artifact <ID> : entities
origin: given                          # optional; only for exogenous artifacts
internal structure: <structure-type>
embedment: <embedment-type or {...}>     # omitted when single entity
features:
  - id: <feature_id>
    value structure: <structure>
    value type: <type>                 # optional
    description: "<text>"
  - id: <feature_id2>
    value structure: <structure>
    value type: <type>                 # optional
    description: "<text>"
description: "<text>"
```

**origin.** Specifying the origin as `given` means that the entities artifact is not produced by any transform in this workflow (for example, input data or external context). This field is omitted for entities appearing as transform outputs, as their provenance is specified in the transform declarations.

**internal structure.** Describes how the components *inside* each entity are organised. In the following, the term "component" refers to structural elements that make up an entity (e.g., events in an episode, words in a sentence, nodes in a graph). Common types of internal structure include:

- `elementary`: entities are treated as atomic, not decomposed into smaller components in this workflow (e.g., events, measurements, documents considered as wholes).
- `group/cluster`: each entity consists of multiple components grouped together (e.g., a group of entities, a cluster of days).
- `sequence`: each entity is an ordered list of components (e.g., sequences of events, sentences consisting of words, gene sequences in DNA, time series of measurements).
- `episode`: each entity is defined by a bounded time interval and contains time-referenced components (e.g., daily segments of time series, attack episodes in a football game containing all events and movements that happened during the interval).
- `region`: each entity is defined by a bounded spatial extent and contains space-referenced components. Conceptually, a region is the spatial analogue of an episode.
- `formation`: each entity consists of components organised in a non-linear relational structure, such as a hierarchy, a network, a spatial formation defined by neighbourhood relationships, or a distribution of roles in a team.

Internal structures can be broadly grouped according to the principle by which components are organised: atomic structures (no decomposition), container structures (components grouped within a bounded container), and relational structures (components

organised through explicit relations such as ordering or networks). Table 1 summarises and groups the definitions of the structure types.

**embedment.** Describes the base environment(s) or containers in which the entities themselves reside in the data space, and how entities are related to each other. Values may be single or combined (e.g., `{set, time}`), drawn from:

- `set`: unstructured membership; entities form an unordered collection.
- `sequence`: entities occupy linearly ordered positions (1st, 2nd, 3rd, ...) without a temporal metric; only the order matters (e.g., chapters in a book, ordered lists of tasks, positions of sentences in a document).
- `time`: entities are placed along a temporal axis with a meaningful notion of duration or distance and possibly including cycles; time may be absolute (calendar date, clock time) or relative to a reference event or time point (e.g., beginning of an illness or a process).
- `space`: entities are placed in a spatial reference (coordinates, geometries).
- `relational structure`: entities are located within a relational structure such as a network or hierarchy (e.g., individuals in a social network, employees in a company, rivers and their tributaries in a river system).

Multiple environments may be combined to express that entities are jointly embedded in several bases, for example:

- `{set, time}`: entities are associated with both an unstructured set of objects and a temporal axis (e.g., events identified by *who* and *when*).
- `{space, time}`: entities are associated with both space and time (e.g., spatial regions evolving over time).

When an `entities` artifact represents a single analytical object (e.g., a single aggregation tree or a single network), the `embedment` field is omitted. Embedment characterises how multiple entities are positioned relative to *each other* within a shared environment; it is therefore inapplicable when there is no collection of peer entities to relate.

**Internal structure vs embedment.**

- *Internal structure* describes how components *within each entity* are organised (e.g., the sequence of words in a sentence, or the configuration of players in a football team formation at a given moment).
- *Embedment* describes how the *entities themselves* are positioned and related to each other within one or more base environments (e.g., entities ordered in time, placed in space, arranged in a network, or listed in a sequence).

The same type of relational organisation (e.g., temporal order, spatial arrangement, or network structure) can appear either as internal structure or as embedment, depending on the analytic level. For example:

- If the analyst treats an entire sequence of events as a single entity (e.g., a patient history), then the event order is part of the entity's *internal structure*.
- If the analyst treats each event as an entity on its own, then the temporal order of events becomes part of the artifact's *embedment* (e.g., `embedment: {set, time}`).

Different analytical perspectives may therefore define different entities from the same underlying data. Table 2 summarises the conceptual differences between internal structure and embedment of entities.

**features.** Internal features (attributes) of the entities, such that each entity may have specific values of these features. Each feature declares a `value structure` describing the overall organisation of its values (`atomic`, `list`, `vector`, `matrix`, or `relational configuration`) and, optionally, a `value type` describing the type of atomic components (`numeric`, `ordinal`, `categorical`, `temporal`, `spatial`, `text`, or `reference`). When components are of mixed types, set notation is used (e.g., `{numeric, temporal}`). The `value type` may be omitted when the mixture is complex or the types are evident from context. The same scheme applies to stand-alone `feature` artifacts (see Section 3.2).

```
artifact D1 : entities
    origin: given
    internal structure: elementary
    embedment: time
    features:
      - id: power_consumption
        value structure: atomic
        value type: numeric
        description: "Power consumption volume"
    description: "Numeric measurements referring to hours
        along a year"

artifact D2 : entities
    origin: given
    internal structure: elementary
    embedment: {set, time}
    features:
      - id: event_type
        value structure: atomic
        value type: categorical
        description: "Event type"
      - id: measurements
        value structure: atomic
        value type: numeric
        description: "Measurements"
    description: "Patient health records (separate events
        identified by patient and time)"

artifact D2_seq : entities
    internal structure: sequence
```

```
    embedment: set
    features:
      - id: length
        value structure: atomic
        value type: numeric
        description: "Sequence length"
      - id: duration
        value structure: atomic
        value type: numeric
        description: "Sequence duration"
    description: "Per-patient sequences of health events;
        each entity is the time-ordered sequence of events
        for one patient"

artifact D3 : entities
    origin: given
    internal structure: formation
    embedment: time
    features:
      - id: edge_weight
        value structure: atomic
        value type: numeric
        description: "Edge weight"
      - id: presence_absence
        value structure: atomic
        value type: categorical
        description: "Presence/absence indicator"
    description: "Dynamic network where entities are network
        snapshots organised in a temporal sequence"

artifact D4 : entities
    origin: given
    internal structure: sequence
    embedment: set
    features:
      - id: word_count
        value structure: atomic
        value type: numeric
        description: "Word count"
    description: "Set of text documents treated as sequences
        of words"
```

Examples D2 and D2_seq demonstrate defining entities at different analytical levels from the same base data. In D2, each entity is a single event (internal structure: elementary), embedded in $\{set, time\}$ (patient $\times$ time). In D2_seq, each entity is a sequence of events for one patient (internal structure: sequence); the artifact as a whole is a set of such per-patient sequences.

**Entities used as contexts.** Some entities artifacts may serve as contexts for arranging members of other artifacts. A **context** is not a separate type: it is any `entities` arti-

fact whose members can be treated as "positions" in which other entities can be placed. Example:

```yaml
artifact D_calendar : entities
    origin: given
    internal structure: elementary
    embedment: time
    features:
      - id: month
        value structure: atomic
        value type: categorical
        description: "Month"
      - id: day_of_week
        value structure: atomic
        value type: categorical
        description: "Day of week"
      - id: day_in_month
        value structure: atomic
        value type: numeric
        description: "Day in month"
    description: "Calendar days for one year"
```

## 3.2 Feature Artifact

A **feature** artifact describes properties of entities or relationships between them that are important for subsequent analysis.

Internal features (inside an `entities` artifact) provide background semantics; feature artifacts are used when those features become *first-class operands* of transforms. Feature artifacts are typically produced by transforms with intent `characterise`.

Declaration:

```yaml
artifact <ID> : feature (artifact-ID)
value structure:
value type:                 # optional
representation form: ""     # optional
description: ""
```

Here `artifact-ID` refers to the artifact described by this feature.

**value structure.** Describes the overall organisation of the feature's values. Applies uniformly to both internal features (declared inside `entities` artifacts) and standalone `feature` artifacts.

- `atomic`: single value per entity.
- `list`: ordered or unordered enumeration of values (e.g., member identifiers, ranked selections).
- `vector`: fixed-length array of values (e.g., a daily measurement profile, a composite of speed, direction, and duration).

- `matrix`: two-dimensional array of values, indexed by two dimensions which may represent the same or different entity sets (e.g., pairwise similarity matrix, term-document matrix).
- `relational configuration`: irregular relational structure such as a graph, tree, or network, where relationships may vary in type, directionality, and weight (e.g., k-NN graph, hierarchical grouping, directed flow structure).

**value type (optional).** Describes the type of atomic components within the value structure. When all components share the same type, it is stated as a single value (e.g., `numeric`). When components are of mixed types, set notation is used (e.g., `{numeric, temporal}`). May be omitted when the mixture is complex or the types are evident from context.

- `numeric`: quantitative measurements or counts.
- `ordinal`: values with meaningful order but no metric (e.g., severity levels).
- `categorical`: unordered discrete categories.
- `temporal`: time points, intervals, or durations.
- `spatial`: coordinates, geometries, or spatial references.
- `text`: free-form or structured textual content.
- `reference`: identifier pointing to another entity or artifact.

**representation form (optional).** Specifies the concrete encoding or organisation of the feature values when `value structure` alone is ambiguous. Most useful for `relational configuration` (e.g., `"k-NN graph"`, `"directed flows with time series"`), `vector` (e.g., `"time series per place"`, `"daily measurement profile"`), and `matrix` (e.g., `"pairwise similarity matrix"`, `"term-document matrix"`). Unnecessary for `atomic` and `list` structures where `value type` is sufficient.

## 3.3 Arrangement Artifact

An **arrangement** organises entities in a context, establishing positions or placements to support analysis. It does not create new entities or values; it defines how existing entities (and all their attached features) are positioned within a reference structure.

Declaration:

```
artifact <ID> : arrangement(entities-ID)
context: entities-ID
principle: "<text>"
description: "<text>"
```

- `entities-ID`: reference to the entities artifact whose members are arranged (e.g., daily episodes, events or event sequences, spatial objects).
- `context`: an entities artifact acting as a reference structure (e.g., calendar days, map regions, a grid or projection layout). The context provides the positions that can be occupied by the arranged entities.

- `principle`: how positions are determined (e.g., `"calendar(year, month, weekday)"`, `"geographic coordinates"`, `"2D projection based on similarity"`).

Conceptually, an arrangement defines a mapping from the entities in `entities-ID` to positions (entities) in the `context` entities artifact. All features attached to the arranged entities remain available and can be used later in visualisations, but the arrangement itself does not specify which features will be shown.

Example:

```
artifact A_calendar : arrangement(D_daily_episodes)
    context: D_calendar
    principle: "calendar(year, month, weekday)"
    description: "Calendar-based arrangement of daily episodes;
        each episode is positioned in its corresponding calendar cell"
```

## 3.4 Visualisation Artifact

A **visualisation** presents artifacts visually for human perception.

Declaration:

```
artifact <ID> : visualisation(artifact-IDs)
layout: "<text>"
form: "<text>"
encoding: <optional (but recommended) description of which
           features/components are mapped to layout and visual marks>
description: "<text>"
```

- `artifact-IDs` may refer to one or more artifacts of various types, such as entities, features, arrangements, patterns, models, or contexts.
- `layout`: structure of the visual space (e.g., `"calendar grid"`, `"2D projection"`, `"aligned timelines"`).
- `form`: visual mark type (e.g., `"coloured marks"`, `"line graph"`, `"node-link diagram"`).
- `encoding` (optional): free-text description of how positions and visual attributes are derived from the source artifacts (e.g., layout represents an arrangement, mark colours or sizes represent specific features).

Example:

```
artifact V_calendar : visualisation(A_calendar, F_cluster_IDs)
    layout: "calendar grid"
    form: "coloured marks"
    encoding: "layout: from A_cluster_calendar (day → calendar cell);
               colour: from F_cluster_IDs (cluster label per day)"
    description: "Calendar view showing cluster membership of each day"
```

**Relationship Between Embedment, Arrangement, and Visualisation**

- **Embedment** (data-level) describes where entities live in the data world (time, space, sequence, relational structure). It is a property of an `entities` artifact and its members, independent of any particular workflow: the same entities artifact has the same embedment in any workflow that uses it.
- **Arrangement** (analytic-level) describes how artifacts (typically entities) are organised into a context structure for analysis (e.g., calendar, map, projection, grid). Arrangements are produced by `contextualise` transforms and define a mapping between entities and positions in a context entities artifact. An arrangement can be used both for visualisation and for further machine processing, such as computing relative times (e.g., time since a key event), grouping entities by positions in a cycle (e.g., day-of-week, phase in a periodic process), or aggregating values over context regions.
- **Visualisation** (perceptual-level) describes how artifacts are mapped into visual space for human perception. Visualisations are produced by `visualise` transforms and specify the visual layout and visual form (e.g., calendar grid with coloured marks, projected points, aligned timelines). When an `arrangement` artifact is available, it is often used to define the visual layout (e.g., positions in a calendar or on a map). However, visualisations may also derive layouts directly from entities or features without an explicit arrangement (e.g., a simple time series plot or histogram).

## 3.5 Pattern Artifact

A **pattern** is an abstract regularity or structure identified in data, features, arrangements, visualisations, or models, e.g., a trend, a motif, or an ordering. A pattern is not itself a piece of data or a composition of concrete entities; it is an abstraction that may be instantiated by specific entities or their compositions. A similar difference exists, for example, between the abstract concept "triangle" and various concrete objects having triangular shape, which can be seen as instantiating the concept "triangle".

Entities may be labelled (annotated) in terms of patterns that are instantiated in these entities.

Declaration:

```
artifact <ID> : pattern(artifact-IDs)
representation form: "<text>"
description: "<text>"
```

- `artifact-IDs`: references to artifacts (entities, features, arrangements, models, etc.) from which the pattern is abstracted or which it summarises.
- `representation form`: the form in which the pattern is represented (e.g., "textual descriptions", "ranked list of motifs", "rules", "templates").

Example:

```
artifact P_daily_profiles : pattern(F_cluster_profile)
    representation form: "textual descriptions of profile shapes"
    description: "Descriptions of recurring daily profiles"
```

## 3.6 Model Artifact

A **model** is a formal (mathematical, logical) or computational representation built from data and/or features. A model generalises data in a way that supports explanation, prediction, simulation, or reasoning about the underlying phenomenon.

Declaration:

```
artifact <ID> : model(artifact-IDs)
model type: "<text>"
representation form: "<text>"
description: "<text>"
```

- `artifact-IDs`: references to artifacts used to define, train, or calibrate the model (typically entities and features, and possibly patterns or prior knowledge).
- `model type`: a high-level category describing the role of the model (e.g., "classifier", "regression model", "topic model", "simulation model", "clustering model", "rule-based model").
- `representation form`: an informal description of how the model is represented internally, i.e., its structural or parametric form (e.g., "decision tree", "parameter vector", "matrix factorisation (W,H)", "system of differential equations", "neural network weights"). This field is optional but useful for understanding and comparing models.
- `description`: a brief textual explanation of what the model captures and how it is intended to be used in the analysis.

Example:

```
artifact M_classifier : model(F_training, K_labels)
    model type: "classifier"
    representation form: "parameter set of a logistic regression model"
    description: "Supervised model for recognising behavioural patterns"
```

**Distinction between models and patterns:** Patterns describe *observed* regularities within the available data (descriptive and bounded by observations), while models provide mechanisms that generalize beyond the available data and can be used for prediction, simulation, or explanation. Models support interpolation between available data points and extrapolation to new contexts (future time points, unobserved spatial locations, new populations or products, etc.).

## 3.7 Knowledge Artifact

A **knowledge** artifact is an explicit representation of knowledge that is either derived in the analysis (insights, explanations, rules, recommendations) or injected during the

analysis (constraints, rankings, labels, feedback, domain assumptions).

**Derived vs. injected knowledge.**

- **Derived knowledge** is produced by transforms (typically with intent `generate-knowledge`) based on patterns, models, visualisations, and other artifacts in the workflow. Its provenance is specified by the transform that produces it.
- **Injected knowledge** enters the workflow from external sources (domain expertise, regulations, prior studies, user feedback). Such artifacts are marked with `origin: given`.

Declaration:

```
artifact <ID> : knowledge(artifact-IDs)
origin: given                  # required for injected knowledge; omitted for
    derived
representation form: "<text>"
description: "<text>"
```

- `origin`: `given` for injected knowledge (external domain knowledge, regulations, feedback); omitted for knowledge derived by transforms in this workflow.
- `artifact-IDs`: references to artifacts (entities, features, patterns, models, visualisations, etc.) that this knowledge is based on or that it refers to. These are the supporting or related artifacts. For injected knowledge, these are the artifacts to which the knowledge applies.
- `representation form`: the form in which the knowledge is represented (e.g., "statements", "if-then rules", "constraints", "labels", "ranking", "taxonomy", "annotations", "configuration settings").
- `description`: a brief textual explanation of what the knowledge expresses and how it is intended to be used in the analysis.

Example of derived knowledge:

```
artifact K1 : knowledge(P_daily_profiles, V_calendar)
    representation form: "statements"
    description: "Statements about seasonal and weekly variation
        in daily profiles"
```

Examples of injected knowledge:

```
artifact K_domain : knowledge(D2)
    origin: given
    representation form: "labels"
    description: "Domain expert labels marking normal vs. anomalous events"
```

```
artifact K_taxonomy : knowledge(D_documents)
    origin: given
    representation form: "hierarchical taxonomy"
    description: "Domain-specific taxonomy of document categories,
    used to guide topic modelling"
```

## 3.8 Specification Artifact

A **specification** artifact is an explicit description of parameters, settings, constraints, or method choices that determine how transforms are executed. Specifications represent *control knowledge* in the workflow: they do not describe data or analytical results, but influence how these are produced.

Specifications may be:

- **given** (injected), representing analyst-defined choices, domain constraints, or prior assumptions;
- **derived**, resulting from evaluation, optimisation, or iterative refinement during the workflow.

### Specification Declaration

```
artifact <ID> : specification
origin: given                  # required for injected specifications
representation form: "<text>"
description: "<text>"
```

**representation form.** Describes how the specification is expressed, for example:

- `"parameter settings"` (e.g., number of clusters, thresholds);
- `"method choice"` (e.g., distance metric, algorithm variant);
- `"constraints"` (e.g., must-link / cannot-link, domain rules);
- `"configuration schema"` (structured or hierarchical settings).

**Role in workflows.** Specification artifacts represent *control knowledge* in the workflow: they encode the analyst's or system's decisions about *how* subsequent analysis should be carried out. They are typically consumed by transforms whose behaviour depends on configurable choices (e.g., clustering, filtering, projection, or model training). Specifications may be *given* (exogenous), or they may be *derived* by transforms, most commonly by `generate-knowledge` transforms in which the analyst formulates methodological decisions, or as a result of `assess` transforms that trigger parameter adjustments. They may be updated iteratively within loops as the analyst refines the analytic strategy.

```
artifact S_grouping : specification
    origin: given
    representation form: "criteria"
    description: "Criteria defining when entities should be considered similar
    enough to belong to the same group"

artifact S_partitioning : specification
    origin: given
    representation form: "rules"
```

```
    description: "Rules for partitioning entities into units based on
    temporal or spatial boundaries"

artifact S_representation : specification
    representation form: "method choice"
    description: "Choice of how entities are represented for further analysis
    (e.g., selection of attributes or encoding strategy)"

artifact S_evaluation : specification
    representation form: "quality criteria"
    description: "Criteria used to assess adequacy or usefulness of results"

artifact S_analysis : specification
    origin: given
    representation form: "configuration schema"
    description: "Integrated set of choices defining how entities are
    partitioned, characterised, and evaluated during analysis"
```

# 4 Transforms

## 4.1 Transform Declaration

General form:

```
transform <ID> :
    intent: <generic-intent>
    manner: "<text>"          # optional; specialisation
    input: <artifact-IDs>
    output: <artifact-IDs>
    actor: <human|machine|hybrid>
    description: "<text>"
```

- `intent` – one of a small set of generic intents (see below), describing the main analytic purpose of the transform.
- `manner` – an optional specialisation of the intent (e.g., `extract`, `cluster-by-similarity`, `time-partitioning`, `projection`, `perception-and-interpretation`) that refines *how* the intent is realised.
- `input` – a comma-separated list of artifact IDs consumed by the transform.
- `output` – a comma-separated list of artifact IDs produced by the transform.
- `actor` – who or what executes the transform (`human`, `machine`, or `hybrid`).
  - `human`: Transform executed entirely by human analyst (e.g., visual interpretation, manual labelling)
  - `machine`: Transform executed by computational methods (e.g., clustering algorithm, aggregation)

– `hybrid`: Transform requires human-machine collaboration (e.g., semi-supervised clustering, interactive parameter tuning)

Each transform has one main `intent`, even if it produces multiple artifact types; the intent reflects its primary analytic purpose.

## 4.2 Generic Transform Intents

ATWL uses a single level of generic intents, which may be specialised via `manner`.

**1.** `define-unit` Create or redefine entities to serve as units of analysis from existing entities or, possibly, other artifacts.

**Note:** In ATWL, **entities** is the artifact type, while **analysis units** refers to the analytical role that entities play in a workflow. A `define-unit` transform specifies what will be treated as the fundamental objects of analysis (the analysis units) and produces an `entities` artifact containing them.

Typical uses:

- select subsets of entities;
- extract specific parts of internally structured entities, e.g., peaks from time series, stops from trajectories;
- partition time into episodes, sequences into segments, space or images into regions;
- group entities into clusters;
- re-aggregate entities (group days into weeks, merge groups, etc.).

**Note:** The realisation of a `define-unit` transform typically depends on the structural properties of the input entities, including both their *embedment* and their *internal structure*. The embedment determines how entities are situated and related in the data space, enabling operations such as temporal aggregation or segmentation (`time`), spatial grouping into regions (`space`), sequence partitioning (`sequence`), or grouping based on connectivity in a `relational structure`. In contrast, the internal structure of entities enables defining units by extracting or isolating components within each entity, such as identifying peaks in time series, stops in trajectories, or substructures in networks.

Thus, the `define-unit` intent captures a general analytical operation whose concrete realisation may rely on relationships *between* entities (via embedment) or on decomposition *within* entities (via internal structure), or both.

Typical outputs: `entities` artifacts comprising the new units of analysis. The transform may also produce auxiliary `feature` artifacts such as labels that assign each unit to a group or state.

**2.** `characterise` Compute or transform `feature`s that describe analysis units.

**Note:** The realisation of a `characterise` transform depends on the structural properties of the input entities, including both their *internal structure* and their *embedment*.

Features may be derived from properties *within* individual entities (e.g., statistics of feature values or representations computed from their internal structure) or from relationships *between* entities induced by their embedment (e.g., temporal proximity, spatial neighbourhood, or network connectivity). In many cases, both sources are combined, for example when computing context-aware descriptors.

Typical uses:

- compute synoptic statistics or profiles for analysis units based on their internal structure (e.g., daily averages, episode summaries);
- generate machine-processable representations of complex or unstructured analysis units, such as networks, flows in space, texts, or images;
- compute relational features (similarity, co-occurrence, adjacency, neighbourhood) between analysis units based on their embedment or derived relations;
- project high-dimensional features into new feature spaces (e.g., dimensionality reduction, learned embeddings).

Typical outputs: `feature` artifacts, including features with `value structure: relational configuration` when relations between entities are characterised.

**3.** `contextualise` Place analysis units into a *context*, creating an `arrangement` that specifies their positions or placements.

**Note:** The realisation of a `contextualise` transform depends primarily on the *embedment* of the input entities, which determines the natural reference systems in which entities can be positioned (e.g., temporal axes, spatial frames, sequences, or relational structures). In addition, properties derived from the *internal structure* of entities (often via `features`) may be used to construct or parametrise contexts, for example when arranging entities in a feature-based projection or embedding space.

Typical uses:

- arrange daily episodes into a calendar (time-based context);
- position spatial entities on a map (space-based context);
- arrange snapshots in a 2D projection derived from features;
- align event sequences or episodes on a relative time axis centred at a key event.

Typical outputs: `arrangement` artifacts.

**External vs. computed contexts.** Contexts used in `contextualise` transforms may be either **external** (pre-existing, marked `origin: given`) or **computed** (created within the workflow). External contexts (e.g., calendars, maps) are inputs to `contextualise` transforms, which produce only arrangements. Computed contexts (e.g., dimensionality reduction projections, data-driven grids) are outputs of `contextualise` transforms along with their arrangements: the transform simultaneously creates the reference system and positions entities within it. Examples:

```
# External context:
transform: contextualise
```

```
input: D_entities, D_calendar
output: A_calendar

# Computed context:
transform: contextualise
input: D_entities, F_features
output: D_proj_space, A_projection
```

**4.** `visualise` Create `visualisation` artifacts for human perception and interaction.
Typical uses:

- visualise arrangements (maps, calendars, projections);
- visualise profiles, distributions, flows;
- visualise model outputs or errors.

Typical outputs: `visualisation` artifacts.

**5.** `abstract` Derive `pattern`s and conceptual structures from entities, features, arrangements, visualisations, or models. Abstraction can be performed by computational methods (e.g., pattern mining algorithms) and/or by human perception and interpretation, often in combination. Computational methods may propose candidate patterns, cluster summaries, rankings, or saliency scores that the analyst then inspects, filters, refines, or renames.
Typical uses:

- identify salient shapes, cohesive groups, or individual items with interesting characteristics (e.g., typical temporal profiles, spatial clusters, anomalous outliers);
- recognise instantiations of known concepts or structures, e.g., "peak", "dense cluster", "alignment", "cycle", "trend", "transition pattern";
- extract re-occurring motifs or subsequences (e.g., frequent event patterns, repeated movement segments);
- assign semantic labels or short names to behavioural types or topics (e.g., "commuting pattern", "weekend profile", "sports-related topic");
- define structural templates that capture the form of a pattern (e.g., "rise–plateau–drop" profile, "burst–pause–burst" event sequence).

Typical outputs: `pattern` artifacts (possibly in combination with `feature` artifacts that label analysis units with the patterns they instantiate).

**6.** `build-model` Construct or refine `model`s based on existing artifacts.
Typical uses:

- train classifiers or regressions;
- build topic models;
- calibrate simulations.

Typical outputs: `model` artifacts.

**7.** `generate-knowledge` Formulate `knowledge` artifacts from patterns, models, and visual evidence. Because `specification` artifacts represent control knowledge (i.e., explicitly formulated decisions about how analysis should proceed), `generate-knowledge` transforms may also produce `specification` artifacts when the analyst's primary cognitive act is to decide on a course of action, for example, selecting a modelling strategy, choosing features to include, or setting parameters for the next iteration.

Typical uses:

- write analytic conclusions, statements, or explanations;
- derive rules or guidelines (e.g., if–then rules, decision criteria, recommendations);
- summarise findings for communication or decision support;
- formulate methodological decisions as specifications that control downstream transforms (e.g., choosing a clustering method, selecting a subset of features, setting thresholds).

Conceptually, `abstract` and `generate-knowledge` are closely related but play different roles. The intent `abstract` focuses on identifying, defining, and naming patterns or conceptual structures in the data (and representing them as `pattern` artifacts). The intent `generate-knowledge` then uses these patterns, together with models, visual evidence, and the analyst's domain understanding, to formulate explicit knowledge (e.g., statements, rules, recommendations) as `knowledge` artifacts, or explicit methodological decisions as `specification` artifacts. In many workflows, `abstract` transforms produce pattern artifacts that are subsequently consumed by `generate-knowledge` transforms.

**8.** `assess` Evaluate the quality, adequacy, or appropriateness of artifacts with respect to analytic goals, criteria, or constraints, in order to guide further analysis or refinement.

**Note:** The `assess` intent operates at a meta-analytic level. It does not derive new structures from data (as in `abstract`) nor formulate domain-level conclusions (as in `generate-knowledge`), but instead evaluates intermediate or final results to determine whether they are satisfactory or require adjustment. Assessment may be performed by computational methods (e.g., quality metrics, validation scores), by human judgment (e.g., interpretability, usefulness), or by a combination of both.

Typical uses:

- evaluate clustering quality (e.g., cohesion, separation, interpretability);
- assess model performance (e.g., accuracy, error, generalisation);
- judge the adequacy or readability of visualisations;
- evaluate whether extracted patterns are meaningful or relevant;
- compare alternative results produced under different specifications.

Typical outputs:

- `knowledge` artifacts representing evaluation results (e.g., scores, judgments, qualitative assessments);
- optionally, updated `specification` artifacts suggesting revised parameter settings, constraints, or method choices.

**Note on intent types.** The intent types capture the typical purposes of operations occurring in analytical workflows. They do not form a single homogeneous category of computational transformations; instead, they correspond to different **phases of the analytical process**. Some intents describe transformations that structure or derive analytical artifacts (e.g., defining entities or deriving features). Others organise artifacts within reference contexts or represent them for human interpretation (e.g., arranging entities or producing visualisations). Further intents correspond to analytical inference and interpretation, such as discovering patterns, building models, or formulating explicit knowledge.

This diversity reflects the nature of visual analytics workflows, which combine data transformation, analytical reasoning, and human interpretation. Accordingly, the intent classification should be understood as a high-level description of the analytical purpose of a transform within the workflow, rather than a strict taxonomy of algorithms or computational procedures.

## 4.3 Specialisations via `manner`

The `manner` field specifies how a generic intent is realised, at a conceptual level. Possible values (illustrative, extensible) include:

- For `define-unit`: `"extract"`, `"filter"`, `"group-by-entity"`, `"time-partitioning"`, `"space-partitioning"`, `"cluster-by-similarity"`, `"merge-groups"`, `"derive-places"`, ...
- For `characterise`: `"summarise"`, `"aggregate"`, `"profile"`, `"project"`, `"encode"`, `"define-relations (similarity)"`, `"define-relations (co-occurrence)"`, `"compute-statistics"`, ...
- For `contextualise`: `"calendar-based"`, `"map-based"`, `"projection-based"`, `"grid-based"`, `"relative-time-alignment"`, ...
- For `visualise`: `"line-graph"`, `"coloured-marks"`, `"node-link-diagram"`, `"matrix-plot"`, ...
- For `abstract`: `"pattern-mining"`, `"find-salient-groups"`, `"rank-patterns"`, `"name-patterns"`, `"perception-and-interpretation"`, ...
- For `build-model`: `"train-classifier"`, `"fit-topic-model"`, `"calibrate-simulation"`, ...
- For `generate-knowledge`: `"formulate-statements"`, `"derive-rules"`, `"write-summary"`, ...

These values are not part of the core type system but are recommended for consistency and to make it easier for software tools to automatically identify and compare similar transforms (for example, all transforms with manner: "cluster-by-similarity").

## 4.4 Transform Intents and Specifications

The way of achieving transform intents may be regulated by `specification` artifacts, which can represent parameters, settings, constraints, or desired characteristics of trans-

form results. `specification` artifacts, when not given externally (marked `origin: given`), are typically created or updated as part of other transform intents rather than by a dedicated intent. In particular:

- `assess` transforms may produce updated specifications as a result of evaluating the adequacy of analytical results (e.g., suggesting revised parameters);
- `generate-knowledge` transforms may produce specifications when analysis choices are explicitly formulated or justified;

Thus, specifications represent control knowledge that is injected, derived, and refined within the analytical process, rather than a separate category of analytical operation.

### Types of Knowledge Artifacts and Their Origins

ATWL distinguishes between different kinds of `knowledge` artifacts based on the transform intent that produces them. In particular, it is important to differentiate between *evaluative knowledge* produced by `assess` and *substantive knowledge* produced by `generate-knowledge`.

**Two types of knowledge artifacts.**

| Source intent | Knowledge type | Purpose | Example |
|---|---|---|---|
| `assess` | evaluative knowledge | quality judgments, adequacy decisions | "Clusters are not coherent", "Refinement needed" |
| `generate-knowledge` | substantive knowledge | domain insights, conclusions | "Peak usage occurs at 8am", "Three behavioural patterns exist" |

**`assess` produces evaluative knowledge.** Outputs of `assess` transforms are *judgments about artifacts* with respect to quality criteria, analytical goals, or constraints. These judgments may be quantitative (e.g., scores) or qualitative (e.g., interpretability), but in all cases they express an evaluation rather than a structure inherent in the data.

Example:

```
artifact K_cluster_assessment : knowledge(D_clusters)
    representation form: "quality judgment"
    description: "Clusters are spatially coherent and interpretable;
        no further refinement needed"
```

**Important distinction:** such artifacts are treated as `knowledge`, not as `pattern`. Statements such as "clusters are coherent" or "refinement is needed" are judgments based on evaluation criteria and analytical goals; they do not describe regularities or structures present in the data itself.

`generate-knowledge` **produces substantive knowledge.** `generate-knowledge` transforms produce knowledge that captures *insights about the domain or the analysed phenomenon*. This includes statements about observed patterns, relationships, or behaviours.

Example:

```
artifact K_movement_patterns : knowledge(D_trajectories)
    representation form: "domain insights"
    description: "Peak usage occurs between 8 and 9am;
        three behavioural patterns are identified"
```

Such knowledge is typically based on `pattern`, `model`, or `visualisation` artifacts, but goes beyond them by formulating explicit, interpretable conclusions.

**Relationship to specifications.** Some outcomes of analysis involve decisions about how the workflow should proceed (e.g., choice of methods, parameter settings, or strategies). These should be represented as `specification` artifacts rather than `knowledge`, as they define how subsequent transforms are configured rather than expressing insights about the data.

Example:

```
artifact S_analysis_strategy : specification
    representation form: "methodological choice"
    description: "Use similarity-based grouping with domain-specific
        distance function"
```

**Summary.**

- `pattern` artifacts describe structures or regularities in data;
- `knowledge` artifacts express interpreted statements about data or results;
- `assess` produces *evaluative knowledge* about the adequacy of artifacts;
- `generate-knowledge` produces *substantive knowledge* about the domain;
- `specification` artifacts capture decisions that control how analysis is performed.

# 5 Control Structures

Control structures in ATWL describe analytic logic, not executable control.

## 5.1 Loops

A **loop** indicates iterative refinement or repetition until a qualitative condition holds.

Syntax:

```
loop <ID>:
    purpose: "<text>"
    until: "<qualitative stopping condition>"
    body:
```

```
        <transforms and artifact declarations>
end loop <ID>
```

The `until` field states a qualitative stopping condition describing when iteration terminates. Two styles of loop termination are common in practice:

- **Explicit assessment.** The loop body contains an `assess` transform that produces an evaluative `knowledge` artifact, followed by an `if–then–else` conditional with `exit loop <ID>` in one of the two branches. This style is appropriate when the stopping criterion can be naturally expressed as a judgment about a specific artifact (e.g., "model accuracy is sufficient").
- **Implicit termination.** No explicit assessment or conditional exit appears in the loop body; the `until` condition is understood to be monitored informally by the analyst throughout the iteration. This style is appropriate when the stopping criterion reflects a gradual, holistic change in the analyst's understanding that cannot be naturally reduced to a single assessable artifact (e.g., "the analyst has developed a sufficient understanding of the data patterns").

Both styles are valid. The choice depends on whether the workflow benefits from modelling the stopping decision as an explicit transform–artifact step.

Example (implicit termination):

```
loop L1:
    purpose: "Refine clustering until groups are few
        and interpretable"
    until: "Cluster separation and interpretability
        are satisfactory"
    body:
      transform T7 :
          intent: define-unit
          manner: "cluster-by-similarity"
          ...
end loop L1
```

Example (explicit assessment):

```
loop L2:
    purpose: "Iteratively simplify display until
        compact and adequate"
    until: "Display is interpretable and sufficient
        for analytical questions"
    body:
      ...
      transform T_assess :
          intent: assess
          ...
          output: K_assessment
      ...
      if K_assessment indicates further refinement needed:
```

```
            then:
                ...
                assign:
                    D_records := D_records'
            else:
                exit loop L2
end loop L2
```

## 5.2 Conditionals

A **conditional** describes alternative paths depending on artifact properties or human judgments.

Syntax:

```
if <condition>:
    then:
        <transforms and artifacts>
    else:
        <transforms and artifacts>
```

Conditions refer conceptually to artifacts (e.g., "number of clusters too high", "model accuracy insufficient").

## 5.3 Assignment

Assignments indicate that an artifact identifier is bound to a new artifact version, replacing the previous one. Assignments appear in two contexts:

- **Inside a loop body**, to express iterative update: each iteration replaces the previous version of an artifact with a refined one, enabling convergence towards the `until` condition.
- **Before a loop**, to initialise an artifact identifier that will be re-assigned within the loop. This introduces the identifier as a "variable" whose initial binding is set before iteration begins.

Syntax:

```
assign:
    <artifact-ID>  :=  <artifact-ID'>
    <artifact-ID2> :=  <artifact-ID2'>
```

Example (initialisation before a loop and update inside):

```
artifact D_all : entities
    ...

assign:
    D_current := D_all
```

```
loop L1:
    purpose: "Progressively analyse subsets"
    until: "All relevant subsets have been examined"
    body:
        ...
        transform T_select :
            intent: define-unit
            input: D_current, ...
            output: D_next
            ...
        assign:
            D_current := D_next
end loop L1
```

# 6 Workflow-Level Constructs

## 6.1 Workflow Header and Template

A workflow starts with:

```
workflow <workflow-ID>
template: <intent> → <intent> → ...     # optional
description: "<text>"                     # optional but recommended
```

- `template` uses the same intent vocabulary as transforms (`define-unit`, `characterise`, ...).
- It summarises the main analytic stages; one template step may correspond to several transforms.

Example:

```
workflow cluster-calendar
template: define-unit → characterise → define-unit (similarity-based) →
          characterise (groups) → contextualise → abstract → generate-knowledge
description: "Cluster daily temporal profiles and analyse their distribution over
    the year"
```

Parenthetical qualifiers in the template (e.g., `(similarity-based)`) are informal annotations, not new intents.

### Loop Notation in Templates

Templates may include `loop(...)` constructs to indicate that a sequence of intents is executed iteratively in the workflow body:

```
template: intent1 → loop(intent2 → intent3) → intent4
```

This notation indicates that the sequence of intents inside `loop(...)` is repeated until a qualitative condition is satisfied. Iterations typically involve assessment of intermediate results and, optionally, adjustment of specifications (e.g., parameters, constraints, or method choices) before re-executing the looped transforms.

### Guidelines

Loop notation is used when iteration is analytically significant (e.g., progressive refinement, interactive parameter tuning, or iterative model improvement), but not for representing low-level or implementation-driven repetition (e.g., processing each item in a collection).

The use of `loop(...)` abstracts away from the exact control logic (e.g., stopping criteria or branching decisions), which are specified in the workflow body using control structures. Templates capture only the presence and role of iteration at a high level.

Nested loops should generally be avoided in templates to maintain clarity and abstraction. Loop notation is optional and should be used only when the iterative structure is a key characteristic of the analytical method.

### Examples

```
# Interactive refinement:
template: define-unit →
          loop(visualise → abstract → assess → define-unit) →
          generate-knowledge

# Progressive multi-phase analysis:
template: loop(define-unit (spatial clustering) → assess) →
          loop(define-unit (route clustering) → assess) →
          abstract → generate-knowledge
```

## 6.2 Artifact and Transform Ordering

Artifacts and transforms can be declared in any order that aids readability. Common practice:

- exogenous artifacts (`origin: given`) near the top, or near their first use;
- context entities declared just before the `contextualise` transform that uses them;
- loop bodies and conditionals grouped logically.

`origin: given` is used only for artifacts that are not outputs of any transform in the workflow.

## 6.3 Workflow Validity

A well-formed ATWL workflow should satisfy:

- All artifact IDs are unique within the workflow
- Transform inputs reference declared artifacts
- Artifacts with `origin: given` are not transform outputs
- The artifact dependency graph is acyclic (no circular dependencies)

These are recommended constraints rather than strict requirements; some workflows may intentionally violate them for specific purposes (e.g., cycles in iterative workflows with explicit assignment statements).

# A ATWL Syntax Reference

This section provides a concise syntax reference for the **Artifact–Transform Workflow Language (ATWL)**, a declarative language for representing visual analytics workflows as structured transformations of artifacts. Angle brackets (< >) denote placeholders; the # character introduces comments.

## A.1 Workflow Declaration

```
workflow <workflow-ID>
  template: <intent> → <intent> → ...        # optional
  description: "<text>"                        # optional
```

- `template` summarises the main analytic stages using the same intent vocabulary as transforms (Section A.3.1). Parenthetical qualifiers (e.g. `(similarity-based)`) are informal annotations, not new intents.
- Iteration may be indicated with `loop(<intent> -> <intent> -> ...)`. Loop notation in templates is optional and used only when the iterative structure is a key characteristic of the analytical method.
- Artifacts and transforms may be declared in any order that aids readability.

## A.2 Artifacts

ATWL defines eight artifact types. Every artifact is declared with a unique identifier and its type. The field `origin: given` marks *exogenous* artifacts not produced by any transform in the workflow; it is omitted for artifacts that appear as transform outputs. Artifact type declarations may include references to other artifacts in parentheses, e.g. `feature(D1)`.

### A.2.1 Entities

A collection of identifiable analytical objects of the same kind, each treated as a single (possibly structured) piece of data for the purposes of analysis.

```
artifact <ID> : entities
  origin: given                          # only for exogenous artifacts
  internal structure: <structure-type>
  embedment: <embedment-type or {...}>   # omitted for single entity
  features:
    - id: <feature_id>

      value structure: <structure>
      value type: <type>                 # optional
      description: "<text>"
  description: "<text>"
```

**Internal structure.** Describes how components inside each entity are organised.

**Embedment.** Describes the shared environment(s) in which the entities reside and how they are related to each other. Omitted when the artifact represents a single entity. Values may be single or combined in set notation (e.g. `{set, time}`).

**Features (inside entities).** Each feature declares a **value structure** and, optionally, a **value type**. The same scheme applies to standalone feature artifacts (Section A.2.2).

When atomic components are of mixed types, set notation is used (e.g. `{numeric, temporal}`). The value type may be omitted when the mixture is complex or the types are evident from context.

### A.2.2 Feature

Explicit descriptors of properties of entities or relationships between them. Typically produced by `characterise` transforms.

```
artifact <ID> : feature(<artifact-ID>)
  value structure: <structure>
  value type: <type>                    # optional
  representation form: "<text>"         # optional
  description: "<text>"
```

`<artifact-ID>` refers to the artifact described by the feature. `representation form` clarifies encoding for complex features (e.g. `"similarity matrix"`, `"k-NN graph"`).

### A.2.3 Arrangement

An arrangement organises entities in a context, defining how they are positioned within a reference structure. It does not create new entities or values.

```
artifact <ID> : arrangement(<entities-ID>)
  context: <entities-ID>
  principle: "<text>"
  description: "<text>"
```

- `<entities-ID>`: the entities whose members are arranged.
- `context`: an entities artifact acting as the reference structure (e.g. calendar days, map regions, a projection layout).
- `principle`: how positions are determined (e.g. `"calendar(year, month, weekday)"`, `"2D projection based on similarity"`).

### A.2.4 Visualisation

An external visual representation for human perception.

```
artifact <ID> : visualisation(<artifact-IDs>)
  layout: "<text>"
  form: "<text>"
  encoding: "<text>"                    # optional but recommended
  description: "<text>"
```

- `<artifact-IDs>`: one or more artifacts of any type.
- `layout`: structure of the visual space (e.g. `"calendar grid"`, `"2D projection"`).
- `form`: visual mark type (e.g. `"coloured marks"`, `"node-link diagram"`).
- `encoding`: how features and arrangements map to visual attributes.

### A.2.5 Pattern

An abstract regularity or structure identified in data, features, arrangements, visualisations, or models.

```
artifact <ID> : pattern(<artifact-IDs>)
  representation form: "<text>"
  description: "<text>"
```

`representation form` describes how the pattern is represented (e.g. `"textual descriptions"`, `"ranked list of motifs"`, `"rules"`).

### A.2.6 Model

A formal or computational representation used for prediction, simulation, or explanation.

```
artifact <ID> : model(<artifact-IDs>)
  model type: "<text>"
  representation form: "<text>"         # optional
  description: "<text>"
```

- `model type`: high-level category (e.g. `"classifier"`, `"topic model"`, `"simulation model"`).
- `representation form`: internal parametric or structural form (e.g. `"decision tree"`, `"neural network weights"`).

### A.2.7 Knowledge

Explicitly formulated knowledge, either *derived* in the analysis (insights, explanations, rules) or *injected* during the analysis (domain expertise, constraints, feedback).

```
artifact <ID> : knowledge(<artifact-IDs>)
  origin: given                         # for injected; omitted for derived
  representation form: "<text>"
  description: "<text>"
```

`<artifact-IDs>` reference the artifacts this knowledge is based on or applies to. `representation form` examples: `"statements"`, `"if-then rules"`, `"labels"`, `"quality judgment"`.

Note: `assess` transforms produce *evaluative* knowledge (quality judgments, adequacy decisions), while `generate-knowledge` transforms produce *substantive* knowledge (domain insights, conclusions).

### A.2.8 Specification

An explicit description of parameters, settings, constraints, or method choices that determine how transforms are executed.

```
artifact <ID> : specification
  origin: given                          # for injected; omitted for derived
  representation form: "<text>"
  description: "<text>"
```

`representation form` examples: `"parameter settings"`, `"method choice"`, `"constraints"`, `"configuration schema"`.

Specifications are typically consumed by transforms whose behaviour depends on configurable choices. They may be given (exogenous) or derived—most commonly by `generate-knowledge` or `assess` transforms.

## A.3 Transforms

Each transform consumes one or more input artifacts and produces one or more output artifacts.

```
transform <ID> :
  intent: <generic-intent>
  manner: "<text>"                       # optional specialisation
  input: <artifact-IDs>
  output: <artifact-IDs>
  actor: human | machine | hybrid
  description: "<text>"
```

- `intent`: one of the generic intents in Table 6.
- `manner`: optional free-text specialisation of intent (see Table 7).
- `input`/`output`: comma-separated artifact IDs.
- `actor`: `human` (analyst only), `machine` (computation only), or `hybrid` (human–machine collaboration).

### A.3.1 Generic Intents

### A.3.2 Specialisations via Manner

The `manner` field is not part of the core type system but is recommended for consistency and tool interoperability. Table 7 gives illustrative (extensible) values.

## A.4 Control Structures

Control structures describe analytic logic, not executable control.

### A.4.1 Loop

```
loop <ID>:
  purpose: "<text>"
  until: "<qualitative stopping condition>"
  body:
    <transforms, artifact declarations, conditionals, assignments>
end loop <ID>
```

The `until` field states a qualitative stopping condition. Two styles of termination are supported:

- **Explicit assessment.** The loop body contains an `assess` transform followed by a conditional with `exit loop <ID>` in one branch.
- **Implicit termination.** No explicit assessment or conditional exit appears; the `until` condition is monitored informally by the analyst.

### A.4.2 Conditional

```
if <condition>:
  then:
    <transforms and artifacts>
  else:
    <transforms and artifacts>
```

Conditions refer to artifact properties or human judgements. Use `exit loop <ID>` in a branch to leave an enclosing loop.

### A.4.3 Assignment

```
assign:
  <artifact-ID>  := <artifact-ID'>
  <artifact-ID2> := <artifact-ID2'>
```

Assignments bind an artifact identifier to a new version. They appear in two contexts:

- **Before a loop**, to initialise an identifier that will be reassigned during iteration.

- **Inside a loop body**, to express iterative update of an artifact.

## A.5 Workflow Validity and Conventions

A well-formed ATWL workflow should satisfy:

1. All artifact identifiers are unique within the workflow.
2. Every transform input references a declared artifact.
3. Artifacts with `origin: given` are never transform outputs.
4. The artifact dependency graph is acyclic (except for explicit `assign` statements in loops).

Additional conventions:

- Exogenous artifacts (`origin: given`) are typically declared near the top or near their first use.
- Lists of artifact IDs are comma-separated.
- Artifact references appear in parentheses after the type keyword, e.g. `feature(D1)`, `arrangement(D_events)`.
- Each transform carries exactly one `intent`, even when it produces multiple artifact types; the intent reflects the primary analytic purpose.

Table 1: Types of internal structure for entities

| Category | Structure type | Description | Examples |
|---|---|---|---|
| Atomic | `elementary` | The entity is treated as an indivisible object in the workflow; its internal composition is not analysed. | Individual events, measurements, images considered as wholes |
| Container | `group/cluster` | An unordered collection of components grouped together without internal ordering or relational structure. | Cluster of days with similar behaviour; group of related objects |
| | `episode` | A container defined by a bounded time interval that includes time-referenced components. | Daily segment of a time series; attack episode in a sports match |
| | `region` | A container defined by a bounded spatial extent that includes spatially referenced components. | City district containing events; spatial cell containing measurements |
| Relational | `sequence` | Components are organised in a linear order where the ordering relation is essential. | Sequence of events; words in a sentence; ordered list of tasks |
| | `formation` | Components are organised through general relational structures such as networks, hierarchies, or spatial neighbourhood relations. | Network snapshot; organisational hierarchy; team formation |

Table 2: Conceptual differences between internal structure and embedment

| **Aspect** | **Internal Structure** | **Embedment** |
|---|---|---|
| What it describes | Organization *within* each entity | Relationships *between* entities |
| Examples | Sequence of events in an episode | Episodes ordered in time |
| Same data, different views | Episode with time-ordered events | Each event as separate entity, time-embedded |

| **Structure type** | **Description** | **Category** |
|---|---|---|
| `elementary` | Entity is indivisible in this workflow | Atomic |
| `group/cluster` | Unordered collection of components | Container |
| `episode` | Bounded time interval with time-referenced components | Container |
| `region` | Bounded spatial extent with space-referenced components | Container |
| `sequence` | Components in a linear order | Relational |
| `formation` | Components in a general relational structure (network, hierarchy, etc.) | Relational |

Table 3: Types of internal structure for entities.

| **Embedment type** | **Description** |
|---|---|
| `set` | Unordered collection |
| `sequence` | Linearly ordered positions (no temporal metric) |
| `time` | Temporal axis (absolute or relative) |
| `space` | Spatial reference (coordinates, geometries) |
| `relational structure` | Network, hierarchy, or similar relational structure |

Table 4: Embedment types for entities.

| **Value structure** — overall organisation of feature values | |
|---|---|
| `atomic` | Single value per entity |
| `list` | Ordered or unordered enumeration of values |
| `vector` | Fixed-length array of values |
| `matrix` | Two-dimensional array of values |
| `relational configuration` | Irregular relational structure (graph, tree, etc.) |
| **Value type** (optional) — type of atomic components | |
| `numeric` | Quantitative measurements or counts |
| `ordinal` | Values with meaningful order but no metric |
| `categorical` | Unordered discrete categories |
| `temporal` | Time points, intervals, or durations |
| `spatial` | Coordinates, geometries, or spatial references |
| `text` | Free-form or structured textual content |
| `reference` | Identifier pointing to another entity or artifact |

Table 5: Value structure and value type for features.

| **Intent** | **Purpose** | **Typical outputs** |
|---|---|---|
| `define-unit` | Create or redefine entities as units of analysis (select, extract, partition, group, aggregate) | entities (+ feature) |
| `characterise` | Compute or transform features describing entities | feature |
| `contextualise` | Place entities into a context, producing an arrangement | arrangement (+ entities for computed contexts) |
| `visualise` | Create visual representations for human perception | visualisation |
| `abstract` | Derive patterns or conceptual structures | pattern (+ feature) |
| `build-model` | Construct or refine formal/computational models | model |
| `generate-knowledge` | Formulate explicit knowledge or specifications | knowledge and/or specification |
| `assess` | Evaluate quality, adequacy, or appropriateness of artifacts | knowledge (+ specification) |

Table 6: Generic transform intents.

| Intent | Example manner values |
|---|---|
| define-unit | extract, filter, time-partitioning, space-partitioning, cluster-by-similarity, group-by-entity, merge-groups |
| characterise | summarise, aggregate, profile, project, encode, define-relations (similarity) |
| contextualise | calendar-based, map-based, projection-based, relative-time-alignment |
| visualise | line-graph, coloured-marks, node-link-diagram, matrix-plot |
| abstract | pattern-mining, find-salient-groups, perception-and-interpretation, name-patterns |
| build-model | train-classifier, fit-topic-model, calibrate-simulation |
| generate-knowledge | formulate-statements, derive-rules, write-summary |
| assess | evaluate-clustering-quality, assess-model-performance, judge-visualisation-adequacy |

Table 7: Illustrative manner values by intent.