CubeText Language Specification

An Open Textual Language for Structural Intent in Visual Models

semture GmbH

Version 1.3, 18 September 2026: Published specification

Publication notice

CubeText is published and maintained by **semture GmbH**. It is developed as part of the Cubetto product family, with **Cubetto 7** serving as the reference implementation for this specification.

Suggested attribution: *CubeText Language Specification, version 1.3, published by semture GmbH.*

Version 1.3 is the first consolidated public edition of the CubeText Language Specification. Earlier versions in the change log record internal language milestones in the Cubetto product line rather than earlier editions of this standalone publication.

Table of Contents

Chapter 1. Status and scope

1.1. Status of this document

This document is the published CubeText 1.3 specification maintained by semture GmbH. It records the portable CubeText language contract independently of any one implementation. Cubetto 7 is the reference implementation, but implementation details are not normative unless this specification explicitly says so.

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHOULD**, **SHOULD NOT**, and **MAY** express requirement levels. A conforming implementation documents every intentional deviation from a **SHOULD** requirement.

1.2. Purpose

CubeText is a compact textual representation of structural modeling intent. It is designed for direct human editing, deterministic serialization, tool integration, and exchange with machine assistants.

CubeText is profile-based. Its common language core defines document shape, aliases, properties, blocks, and relationship operators. A profile gives those constructs meaning for one notation or diagram kind.

1.3. In scope

The language can express, where enabled by a profile:

- visible model objects and their semantic kinds;
- typed relationships and relationship direction;
- containment, attachment, and profile-specific structural membership;
- visible text roles and a limited set of structurally significant properties;
- presentation name and, for orientable profiles, layout orientation.

1.4. Out of scope

CubeText is not a lossless Cubetto project format. The following are outside the common language contract unless a future profile explicitly adds them:

- coordinates, bounds, connector routes, handles, and transient layout state;
- colors, fonts, strokes, icon rendering, and other appearance details;
- persistence identifiers, database entities, and application-internal object identifiers;
- complete interchange semantics of BPMN XML or other external formats;
- arbitrary changes to a notation’s type system.

An implementation MUST NOT infer that omitted, out-of-scope data should be deleted or reset during reconciliation.

1.5. Normative and informative material

The grammar, semantic rules, conformance requirements, and profile constraints are normative. Rationale, worked examples, implementation notes, and the description of Cubetto 7 are informative unless marked otherwise.

1.6. Publisher and stewardship

CubeText was developed by semture GmbH as part of the Cubetto product family. semture GmbH publishes and maintains this specification, the reference profile registry, and the conformance corpus. Independent conforming implementations are permitted under the licenses stated in the publication notice.

Chapter 2. Language concepts

2.1. Document and profile

The first non-empty, non-comment line of a CubeText document is the profile header. It selects exactly one notation profile for the entire document.

```
#bpmn.collaboration [name="Order Approval", orientation=LTR]
```

The profile determines:

- accepted profile tokens and the canonical token;
- valid alias prefixes;
- permitted properties and value ranges;
- relationship operators and their meanings;
- permitted block contexts and structural constraints;
- canonical serialization rules.

2.2. Atoms and aliases

An *atom* is an alias with optional properties. Aliases such as `A1`, `G2`, or `UC3` are local symbolic addresses. They are not persistence identifiers.

The first structural occurrence of an alias declares the corresponding object. Later occurrences refer to that declaration. A processor MUST bind all occurrences of an alias consistently within one document.

```
A1:"Validate request"
A1 -> G1:"Approved?"
```

An alias prefix is interpreted only by the selected profile. `A1` can denote an activity in one profile and an actor or artifact in another.

2.3. Properties and name sugar

Properties appear in square brackets after a profile token, alias, or attributable relationship operator.

```
A1 [name="Validate request", type=manual]
```

For a name without any additional property, the preferred form is:

```
A1:"Validate request"
```

It is semantically equivalent to `A1 [name="Validate request"]`. If another property is present, a

canonical serializer MUST use the property-block form.

2.4. Relationships

A relationship connects a left atom to a right atom through an operator. Chains are shorthand for adjacent relationships.

```
E1:"Request received" -> A1:"Validate" -> G1:"Approved?"
```

The profile decides whether an operator represents a first-class relationship object, a structural relation, an attachment, or another domain concept. Consequently, syntactic acceptance of an operator does not imply that it is valid in every profile.

2.5. Blocks

An object block expresses profile-defined nesting, membership, or ownership.

```
P1:"Provider" {
    L1:"Sales" {
        A1:"Validate request"
    }
}
```

A relationship block applies one relationship to each direct member of the block when the profile permits that form.

```
U1:"Management" -> {
    U2:"Sales"
    U3:"Operations"
}
```

Object and relationship blocks are syntactically similar but semantically distinct. A profile MUST define every permitted block prefix and operator.

2.6. Structural projection

CubeText is a projection of visible structural intent, not a second model. Serialization reads a model into CubeText. Reconciliation compares edited CubeText with a known source model and applies the smallest meaningful change set supported by the host application.

Chapter 3. Lexical structure

3.1. Encoding and line endings

A CubeText document MUST be encoded as UTF-8. Processors MUST accept LF and CRLF line endings. Canonical output uses LF.

3.2. Whitespace

Spaces and tabs separate tokens where necessary. Newlines separate statements. Indentation has no semantic meaning; braces establish structure. Canonical output uses four spaces per block level and does not use tabs.

3.3. Comments

A line comment begins with `//` outside a string literal or property block and continues to the end of the line.

```
// Review path
A1:"Validate" -> G1:"Approved?" // main decision
```

Comments do not participate in the abstract model. A serializer MAY discard comments unless it explicitly claims source-preserving conformance.

3.4. Profile tokens

A profile token begins with `#` followed by an ASCII letter and zero or more ASCII letters, digits, dots, underscores, or hyphens. Profile-token matching is case-sensitive. Profiles SHOULD define lowercase canonical tokens.

3.5. Identifiers and aliases

General identifiers begin with a letter or underscore. Subsequent characters may also contain digits, hyphens, or plus signs. Profiles usually narrow alias syntax to an uppercase prefix followed by a positive decimal number. A profile MAY reserve an unnumbered alias, such as `MT` in the Mind Map profile.

Alias matching is case-sensitive.

3.6. String literals

Strings are enclosed in double quotes. The following escapes are defined:

Escape	Meaning
`\"`	double quote
`\\`	backslash

Escape	Meaning
`\n`	line feed

For forward compatibility, a backslash before any other character yields that character. Canonical output SHOULD avoid unknown escapes.

3.7. Bare values

A bare property value is a non-empty sequence that does not contain whitespace, a comma, `]`, or a line ending. Profiles define its type and allowed values. The common values `true` and `false` are lowercase in canonical output.

3.8. Property blocks

Properties are comma-separated key/value pairs enclosed in brackets. Whitespace around commas and equals signs is insignificant. A key MUST occur at most once in a property block.

```
[name="Prepare offer", type=process, collapsed=true]
```

The core grammar accepts string and bare values. A profile validator MUST reject unknown keys, invalid values, and properties used on an unsupported target.

Chapter 4. Core grammar

The following EBNF defines the common syntactic envelope. Profile validation is a separate, mandatory conformance step. In particular, the grammar permits all core operators while a profile normally permits only a subset.

```
(* SPDX-FileCopyrightText: 2026 semture GmbH *)
(* SPDX-License-Identifier: Apache-2.0 *)

document           = spacing, profile-header, line-end,
                     { blank-line | comment-line | statement-line } ;

profile-header     = profile-token, [ spacing, property-block ],
                     [ spacing, comment ] ;

statement-line     = spacing, statement, [ spacing, comment ], line-end ;
blank-line         = spacing, line-end ;
comment-line       = spacing, comment, line-end ;

statement          = relationship-block | object-block | chain ;

chain              = atom, { required-spacing, edge,
                     required-spacing, atom } ;

object-block       = atom, spacing, "{", line-end,
                     { blank-line | comment-line | statement-line },
                     spacing, "}" ;

relationship-block = atom, required-spacing, edge, spacing, "{", line-end,
                     { blank-line | comment-line | atom-line },
                     spacing, "}" ;

atom-line          = spacing, atom, [ spacing, comment ], line-end ;

atom               = alias, [ spacing, annotation ] ;
edge               = operator, [ spacing, annotation ] ;
annotation         = name-sugar | property-block ;
name-sugar         = ":", spacing, string ;

property-block     = "[", spacing,
                     [ property, { spacing, ",", spacing, property } ],
                     spacing, "]" ;
property           = identifier, spacing, "=", spacing, value ;
value              = string | bare-value ;

operator           = "->" | "<-" | "~>" | "<~" | "--" | "*" ;

profile-token      = "#", letter, { letter | digit | "." | "_" | "-" } ;
alias              = identifier ;
identifier         = ( letter | "_" ),
                     { letter | digit | "_" | "-" | "+" } ;

string             = '"', { string-character | escape }, '"' ;
escape             = "\\", ( '"' | "\\" | "n" | any-character ) ;
string-character   = ? any Unicode scalar except '"', "\\", CR, or LF ? ;

bare-value         = bare-character, { bare-character } ;
bare-character     = ? any Unicode scalar except whitespace, comma, or "]" ? ;

comment            = "//", { ? any Unicode scalar except CR or LF ? } ;
spacing            = { " " | tab } ;
```

```
required-spacing   = ( " " | tab ), spacing ;
line-end           = LF | CR, LF ;

letter             = "A" | "B" | "C" | "D" | "E" | "F" | "G" |
                     "H" | "I" | "J" | "K" | "L" | "M" | "N" |
                     "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
                     "V" | "W" | "X" | "Y" | "Z" |
                     "a" | "b" | "c" | "d" | "e" | "f" | "g" |
                     "h" | "i" | "j" | "k" | "l" | "m" | "n" |
                     "o" | "p" | "q" | "r" | "s" | "t" | "u" |
                     "v" | "w" | "x" | "y" | "z" ;
digit              = "0" | "1" | "2" | "3" | "4" |
                     "5" | "6" | "7" | "8" | "9" ;

tab                = ? U+0009 CHARACTER TABULATION ? ;
CR                 = ? U+000D CARRIAGE RETURN ? ;
LF                 = ? U+000A LINE FEED ? ;
any-character      = ? any Unicode scalar ? ;
```

4.1. Grammar interpretation

`line-end` is significant because it terminates a statement. An atom may be a declaration or a reference depending on whether its alias has already been bound. `object-block` and `relationship-block` are distinguished by the token before the opening brace.

The grammar does not encode profile-level rules such as:

- permitted alias prefixes;
- whether an object may own a block;
- whether an operator may carry properties;
- endpoint type constraints;
- uniqueness and acyclicity requirements;
- allowed property keys and ranges.

A processor that only implements this grammar is a Core Parser, not a Profile Processor.

Chapter 5. Common semantics

5.1. Abstract document model

A successfully validated CubeText document denotes:

- one selected profile;
- zero or more object declarations identified by document-local aliases;
- zero or more relationships between declared objects or, where allowed, locally identified relationships;
- zero or more profile-defined containment or attachment facts;
- supported header, object, and relationship properties.

Text order is preserved for deterministic serialization and may provide a stable presentation order. It does not create domain semantics unless a profile explicitly defines an order-sensitive construct.

5.2. Declaration and reference

The first structural occurrence of an alias declares it. A later occurrence refers to the same declaration. A processor MUST NOT derive durable identity from the alias alone and MUST NOT expose host persistence identifiers in place of aliases.

An alias MUST have one profile-defined kind. Reusing an alias with a conflicting kind or conflicting type property is an error.

5.3. Direction

`A1 -> A2` denotes a relationship whose semantic source is `A1` and whose semantic target is `A2`. `A1 <- A2` is inverse spelling: its semantic source is `A2` and target is `A1`.

The same rule applies to `~>` and `<~`. The `--` operator is written left to right but may denote an undirected relationship; profiles may still define source- and target-end properties relative to the written sides.

The `*` operator denotes a profile-specific attachment. It is not a general graph-edge operator and MUST NOT carry relationship properties.

5.4. Chains

A chain is expanded pairwise. The following forms are equivalent:

```
A1 -> A2 -> A3
```

```
A1 -> A2
A2 -> A3
```

Properties attached to an operator apply only to the relationship represented by that operator occurrence.

5.5. Object blocks

An object block assigns each direct structural child to the block owner using the profile-defined membership relation. Nested blocks form a stack; the nearest enclosing valid owner applies.

If the same child is mentioned structurally more than once for a single-valued membership, the first valid structural occurrence wins and a processor SHOULD report a diagnostic for later conflicting occurrences.

5.6. Relationship blocks

If a profile enables relationship blocks, `A -> { B C }` expands to `A -> B` and `A -> C`. The contained atoms do not become object-block children merely because they occur inside a relationship block.

5.7. Property interpretation

Properties are closed by default: a profile processor MUST report an unknown property rather than silently persisting it. Implementations MAY preserve an unknown property as source text only when they clearly report that it has no model effect.

Absence of a supported property has profile-defined meaning. It may select a default, remove a previously represented value, or leave the value unchanged. Every profile MUST specify which behavior applies.

5.8. Canonical form

A canonical serializer MUST:

- write the canonical profile token on the first non-empty line;
- use LF line endings and four-space block indentation;
- emit one empty line after the profile header;
- use name sugar only when `name` is the sole property;
- emit profile-defined canonical property values;
- produce deterministic aliases and deterministic statement order;
- omit data outside the selected profile's projection.

Canonicalization is idempotent: serializing a canonical document again without model changes MUST produce identical UTF-8 bytes.

Chapter 6. Reconciliation

6.1. Two processing modes

A Profile Processor MAY support either or both of the following modes:

- **construction** creates a new model projection from a validated document;
- **reconciliation** compares a validated document with the source model from which that CubeText session was serialized.

A processor MUST identify which mode it implements. Parsing alone is not a complete reconciliation contract.

6.2. Session-local binding

During reconciliation, aliases are bound to source objects by a session-local source map. The binding is valid only for that editing session. It MUST NOT be persisted as a domain identifier and MUST NOT cross a model-store boundary as an implementation handle.

An implementation that no longer has the source map MUST treat the input as a construction request or require an explicit, separately specified matching strategy. It MUST NOT guess identity from alias numbering.

6.3. Minimal change set

A reconciler SHOULD derive the smallest meaningful operation set required to make the represented structural projection match the document. Depending on the profile, operations include:

- create and delete;
- rename and retype;
- connect, disconnect, and reconnect;
- attach and detach;
- reparent or change structural membership;
- update supported presentation properties.

6.4. Deletion boundary

A source object or relationship that was part of the serialized profile projection and is absent from the edited document is a deletion candidate. Data that the selected profile cannot express MUST be preserved.

This distinction is fundamental: omission deletes represented structure, but does not erase geometry, appearance, unsupported properties, hidden metadata, or other out-of-scope information.

Profiles MUST describe any exception to this rule, including hidden object kinds and structural relations that are intentionally preserved.

6.5. Validation and atomicity

A reconciler MUST complete syntax and profile validation before applying model changes. If an error prevents unambiguous interpretation, no model mutation may be attributed to the failed operation.

A host application SHOULD apply the resulting operation set atomically and integrate it with its ordinary undo, persistence, and view-update mechanisms. CubeText itself does not define a transaction transport.

6.6. Layout and appearance

Reconciliation may request layout after structural changes, but the language does not prescribe a layout algorithm. A processor MUST NOT treat generated geometry or appearance as additional CubeText semantics.

Chapter 7. Conformance

7.1. Conformance classes

An implementation may claim one or more independently testable classes.

Class	Requirements
Core Parser	Parses the common grammar, reports source ranges, and rejects malformed core syntax.
Profile Validator	Implements the registry and all constraints for each claimed profile.
Canonical Serializer	Produces deterministic canonical CubeText for each claimed profile.
Construction Processor	Creates the abstract profile projection from valid CubeText without relying on a prior source map.
Reconciliation Processor	Applies the reconciliation and preservation rules against a known source model.
Source-Preserving Editor	Preserves comments and non-semantic formatting in addition to meeting another relevant class.

A conformance statement MUST name the specification version, supported profiles, conformance classes, and any optional extensions.

7.2. Diagnostics

Diagnostics MUST distinguish errors from warnings. An error prevents the affected document from being applied. A warning identifies a recoverable, deprecated, or non-canonical construct.

The conformance corpus assigns stable diagnostic categories:

- `CT-SYNTAX` - malformed common syntax;
- `CT-PROFILE` - unknown or misplaced profile token;
- `CT-ALIAS` - invalid, duplicate, or unresolved alias;
- `CT-PROPERTY` - invalid property key, target, or value;
- `CT-RELATION` - invalid operator, endpoints, direction, or edge property;
- `CT-STRUCTURE` - invalid containment, attachment, multiplicity, or cycle;
- `CT-DEPRECATED` - accepted non-canonical syntax.

Implementations MAY add more specific codes below these categories.

7.3. Conformance corpus

Each stable profile requires:

- a machine-readable profile descriptor;
- at least one canonical Golden Sample;
- valid non-canonical inputs and their canonical output;
- invalid inputs with expected diagnostic categories;
- serialization idempotence tests;
- construction tests where construction is claimed;
- reconciliation tests covering creation, deletion, rename, reconnect, containment, preservation, and undo behavior where reconciliation is claimed.

Rendered images are useful review evidence, but visual similarity alone is not a language-conformance test.

7.4. Interoperability rule

A consumer MUST ignore neither an unknown profile nor an unsupported required property. It must reject the document or report that the selected profile is unsupported. Silent fallback to a different profile is non-conforming.

Chapter 8. Versioning and extension policy

8.1. Specification versions

CubeText uses semantic specification versions of the form `major.minor`.

- A minor version may add profiles, optional properties, accepted aliases, or diagnostics without changing the meaning of previously valid canonical documents.
- A major version may remove or reinterpret syntax or profile semantics.
- Editorial corrections that do not change conformance requirements do not change the language version but are recorded in the change log.

The version belongs to the specification and conformance statement. Version 1.3 does not introduce a version token into CubeText documents.

8.2. Profile lifecycle

Profiles have one of four publication states:

- **draft** - under active design and not compatibility-stable;
- **stable** - normative and covered by the conformance corpus;
- **deprecated** - still accepted, with a documented replacement;
- **deferred** - reserved but not supported by a conforming processor.

8.3. Canonical and legacy tokens

A profile has exactly one canonical token. It MAY accept legacy tokens. A validator SHOULD issue `CT-DEPRECATED` for a legacy token, and a canonical serializer MUST write the canonical token.

Canonical tokens use English, notation-neutral names. For the BPMN family, this specification defines namespaced tokens such as `#bpmn.collaboration`. It also replaces the German implementation tokens `#organigramm`, `#epk`, and `#plk` with `#organizational-chart`, `#epc`, and `#process-landscape`. The existing forms remain accepted legacy aliases.

8.4. Extensions

Private extensions MUST use a profile token containing an organization-owned namespace segment and MUST NOT reuse a canonical CubeText token with different semantics. Extension properties SHOULD use a similarly controlled prefix.

An extension document is conforming only to the extension profile it names. A processor MUST NOT claim base-profile conformance for unvalidated extension semantics.

8.5. Governance

semture GmbH maintains the canonical profile registry and release history. Changes intended for a stable release require:

- a written semantic contract;
- grammar or profile-registry updates as applicable;
- positive and negative conformance cases;
- compatibility classification;
- public change-log entry.

Appendix A: Profile registry

This appendix defines the profiles included in version 1.3. Their canonical tokens and the public contracts in this appendix are stable within the 1.x compatibility line. Coverage by the separate positive and negative conformance corpus may continue to grow without changing the profile token.

Profile	Canonical token	Orientation	Accepted legacy token
BPMN Collaboration	`#bpmn.collaboration`	LTR, TTB	`#collaboration`
BPMN Choreography	`#bpmn.choreography`	LTR, TTB	`#choreography`
BPMN Conversation	`#bpmn.conversation`	LTR, TTB	`#conversation`
Organizational Chart	`#organizational-chart`	LTR, TTB	`#organigramm`
Event-driven Process Chain	`#epc`	LTR, TTB	`#epk`
Flowchart	`#flowchart`	LTR, TTB	-
Process Landscape	`#process-landscape`	LTR, TTB	`#plk`
Generic Nodes and Edges	`#generic`	LTR, TTB	-
Swimlanes Business Map	`#swimlanes`	LTR, TTB	-
Mind Map	`#mindmap`	profile-defined	-
UML Use Case Diagram	`#uml.usecase`	profile-defined	`#usecase`
UML Class Diagram	`#uml.class`	profile-defined	`#class`
UML Activity Diagram	`#uml.activity`	LTR, TTB	`#activity`
Architecture View	`#architecture`	LTR, TTB	-

The public registry does not expose implementation-specific model or presentation type identifiers. A profile not listed here has no public CubeText token and MUST NOT be advertised as a supported public profile.

A.1. BPMN Collaboration

Canonical header:

```
#bpmn.collaboration [name="Order Approval", orientation=LTR]
```

`#collaboration` is an accepted legacy token. A canonical serializer writes `#bpmn.collaboration`.

Prefix	Meaning
`P`	Pool

Prefix	Meaning
`L`	Lane
`A`	Activity or task
`E`	Event
`G`	Gateway
`GR`	Group
`M`	Message
`DO`	Data object reference
`DS`	Data storage reference
`N`	Annotation
`PO`	Visible fallback object without a specialized prefix

`->` denotes sequence flow, `~>` denotes message flow, and `--` denotes association. `<-` and `<~` are inverse spellings. `A1 * E1` attaches event `E1` to activity `A1`; `*` does not accept properties.

`P` and `L` support object blocks. Flow nodes inside a non-empty pool MUST be members of a lane. Groups may contain visible members but do not replace pool or lane membership.

The `name` relationship property is permitted on `->`, `~>`, and `--`. Associations involving an annotation express annotation attachment. Metadata that is not visible in the diagram is outside the profile projection.

```
P1:"Provider" {
    L1:"Sales" {
        E1:"Request received" -> A1:"Validate" -> G1:"Approved?"
    }
}
A1 * E2:"Escalation timer"
N1:"Manual review" -- A1
```

A.2. BPMN Choreography

Canonical header:

```
#bpmn.choreography [name="Order Choreography", orientation=LTR]
```

`#choreography` is an accepted legacy token. Aliases use `C` for choreography activities, `E` for events, `G` for gateways, `M` for messages, `N` for annotations, and `PO` for visible fallback objects.

The operators `->`, `~>`, and `--` denote sequence flow, message flow, and association. Inverse forms are accepted for the directed operators.

Choreography activities support participant information through the profile properties `band=top|bottom`, `initiator`, and `multiple`. A processor MUST validate participant references and MUST NOT interpret participant bands as general object containment. Annotation attachment uses `--`; attached events use `*` where supported by the selected activity kind.

A.3. BPMN Conversation

Canonical header:

```
#bpmn.conversation [name="Order Conversation", orientation=LTR]
```

`#conversation` is an accepted legacy token. `C` denotes a conversation, `P` a participant, `N` an annotation, and `PO` a visible fallback object.

`C1 -- P1` denotes a conversation link. `N1 -- C1` or `N1 -- P1` denotes annotation attachment rather than a conversation link. Endpoint kinds therefore participate in relationship resolution.

Conversation objects may use `call=true|false`; participants may use `multiple=true|false`. The `name` property is valid on visible objects and first-class relationships. The profile does not define process flow inside a participant.

A.4. EPC Organizational Chart

The canonical token is `#organizational-chart`; `#organigramm` is accepted as a legacy token. `U` denotes an organizational unit and `P` denotes a position.

`U` supports object blocks for structural assignment. `->` denotes reporting hierarchy and accepts inverse spelling as `<-`. A relationship block expands a single hierarchy source to several direct reports.

```
#organizational-chart [name="Organization", orientation=TTB]

U1:"Management" {
    P1:"Managing Director"
}
U1 -> {
    U2:"Sales"
    U3:"Operations"
}
```

Object-block membership and hierarchy edges are independent facts. A processor MUST NOT infer one merely from the other.

A.5. Event-driven Process Chain

The canonical token is `#epc`; `#epk` is accepted as a legacy token. Aliases use `E` for events, `F` for functions, `C` for connectors, `PI` for process interfaces, and `A` for function artifacts.

Connector `type` values are `xor` (default), `or`, and `and`. Artifact `type` values are `document`, `database`, `itSystem`, `location`, `product`, `risk`, `orgUnit`, `orgPosition`, and `orgRole`.

`->` denotes control flow. `A1 -- F1` places an artifact on the left side of a function; `F1 -- A1` places it on the right. The written sides are therefore semantically significant.

```
#epc [name="Order EPC", orientation=TTB]

E1:"Order received" -> F1:"Check order" -> C1 [name="Available?", type=xor]
F1 -- A1 [name="Order", type=document]
```

A.6. Flowchart

The canonical token is `#flowchart`. `S` denotes a start, `E` an end, `D` a decision, `P` a process, and `R` an on-page or off-page reference.

Process `type` values are `data`, `database`, `delay`, `display`, `document`, and `manualInput`. A reference uses `type=offPage` for an off-page reference; absence of `type` denotes the on-page form.

`->` denotes a connector and `<-` is its inverse spelling. The relationship may carry `name`. No container block is defined by this profile.

```
#flowchart [name="Order Flow", orientation=TTB]

S1:"Start" -> P1:"Validate" -> D1:"Approved?"
D1 ->:"yes" P2 [name="Prepare offer", type=document]
D1 ->:"no" E1:"Rejected"
```

A.7. Process Landscape

The canonical token is `#process-landscape`; `#plk` is accepted as a legacy token. `M` denotes a main process, `T` a target, and `S` a supporting process.

Main-process blocks express their target and support membership. `M1 -> M2` expresses the predecessor relation between main processes; it is a structural relation, not a first-class edge. Consequently, `->` MUST NOT carry name sugar or a property block in this profile.

```
#process-landscape [name="Process Landscape", orientation=LTR]

M1:"Acquire customer" {
    T1:"Growth"
    S1:"Marketing"
}
M1 -> M2:"Deliver service"
```

A.8. Generic Nodes and Edges

The canonical token is `#generic`. `N` denotes a visible node. `->` denotes a directed edge and `<-` is its inverse spelling. Both nodes and edges may carry `name`; no object block is defined.

This profile is intentionally small. It is suitable when structural intent is more important than a specialized notation, but it MUST NOT be used as silent fallback for a document whose profile is unknown.

```
#generic [name="Thinking Processes", orientation=LTR]

N1:"Current reality" ->:"causes" N2:"Undesired effect"
```

A.9. Swimlanes Business Map

The canonical token is `#swimlanes`. `L` denotes a lane, `GR` a group, `O` a business-map object, and `PO` a visible fallback object.

`O` supports the `type` values `process`, `business`, `decision`, `sketch`, `marketing`, `logistics`, `communication`, `realEstate`, `finance`, and `networking`.

`L` and `GR` support object blocks. `->` denotes flow and `--` denotes association. Both relationships may carry `name`.

```
#swimlanes [name="Business Map", orientation=LTR]

L1:"Customer" {
    O1 [name="Request", type=communication]
}
L2:"Provider" {
    O2 [name="Fulfil", type=process]
}
O1 -> O2
```

Lane and group membership are separate structural properties. A group does not replace the lane of an object.

A.10. Mind Map

The canonical token is `#mindmap`. `MT` is the reserved alias for the single main topic; `T1`, `T2`, and subsequent aliases denote topics.

`Parent -> Child` denotes the parent/child relation. It is not a first-class edge, so the operator MUST NOT carry properties or name sugar. A topic has at most one parent and the resulting graph MUST be acyclic.

Topic properties are `name`, `hull=true|false`, and `collapsed=true|false`. Only a direct child of `MT` may carry `direction=left|right`. A deeper topic inherits the effective side of its root branch.

```
#mindmap [name="Social Networks"]

MT:"Social Networks"
MT -> T1 [name="General", hull=true, direction=left]
T1 -> T2:"Communities"
```

Canonical output keeps plain topic relation lines and does not rewrite the structural operator as an attributed edge.

A.11. UML Use Case Diagram

The canonical token is `#uml.usecase`; `#usecase` is accepted as a legacy token. `A` denotes an actor, `UC` a use case, `S` a system boundary, and `N` an annotation. `S` supports object blocks containing use cases.

`--` denotes association. Association edge properties are `name`, `sourceRole`, `sourceMultiplicity`, `sourceCondition`, `sourceEnd`, `targetRole`, `targetMultiplicity`, `targetCondition`, and `targetEnd`. End values are `plain`, `directed`, `aggregation`, or `composition`.

`->` requires `type=include|extend|generalization`. Include and Extend connect use cases. Generalization connects either two actors or two use cases. `N1 -- UC1` denotes annotation attachment, not association.

```
#uml.usecase [name="Online Shop"]

A1:"Customer" -- UC1:"Place order"
UC1 -> [type=include] UC2:"Pay"
S1:"Shop" {
    UC1
    UC2
}
```

A.12. UML Class Diagram

The canonical token is `#uml.class`; `#class` is accepted as a legacy token. `C` denotes a class and supports `type=collapsed`. Classes are blocks containing attributes (`A`), methods (`M`), stereotypes (`ST`), template parameters (`TP`), and template bindings (`TB`). A method block contains parameters (`P`). `N` denotes an annotation.

`--` denotes association and uses the same association-end properties as the UML Use Case profile. `->` requires `type=generalization|realization|dependency|extension`. Annotation attachment uses `--` with an `N` endpoint.

Attributes support `visibility`, `type`, `condition`, `static`, and `abstract`. Methods support `visibility`, `returns`, `static`, `abstract`, and `constructor`.

```
#uml.class [name="Order Domain"]

C1:"Order" {
    A1 [name="number", visibility=private, type="String"]
    M1 [name="submit", visibility=public, returns="Bool"] {
        P1 [name="retry", type="Int"]
    }
}
```

A.13. UML Activity Diagram

The canonical token is `#uml.activity`; `#activity` is accepted as a legacy token. The aliases are `A` (action), `I` (initial node), `AF` (activity final), `FF` (flow final), `D` (decision or merge), `F` (fork or join), `P` (partition), `R` (region), `IO` (region input/output), `ON` (object node), and `N` (annotation). Semantic objects referenced by object

flows use `O`; object-node states use `OS`.

`P`, `R`, and `ON` support blocks. A region accepts `execution=iterative|parallel|stream`. An object node references its semantic object with `object=O1`.

`->` without `type` denotes control flow and may carry `condition`. Object flow uses `type=object`, requires an `object` reference, and is limited by the profile's endpoint rules. Annotation attachment uses `--`.

```
#uml.activity [name="Order Activity", orientation=TTB]

I1:"Start" -> A1:"Validate" -> D1:"Valid?"
D1 -> [condition="yes"] A2:"Accept"
A1 -> [type=object, object=O1, condition="ready"] A2
```

A.14. Architecture View

The canonical token is `#architecture`. The preferred atom form uses the `P` prefix and a required public `type` value. Existing semantic prefixes such as `BA`, `AC`, and `GO` remain accepted as legacy aliases in version 1.3.

The public node type values are:

`ROLE`, `COLLABORATION`, `PATH`, `PROCESS`, `FUNCTION`, `SERVICE`, `EVENT`, `GROUPING`, `LOCATION`, `BUSINESS_ACTOR`, `BUSINESS_INTERFACE`, `BUSINESS_OBJECT`, `PRODUCT`, `APPLICATION_COMPONENT`, `APPLICATION_INTERFACE`, `DATA_OBJECT`, `NODE`, `TECHNOLOGY_INTERFACE`, `DEVICE`, `SYSTEM_SOFTWARE`, `EQUIPMENT`, `FACILITY`, `COMMUNICATION_NETWORK`, `DISTRIBUTION_NETWORK`, `ARTIFACT`, `MATERIAL`, `STAKEHOLDER`, `DRIVER`, `ASSESSMENT`, `GOAL`, `OUTCOME`, `PRINCIPLE`, `REQUIREMENT`, `MEANING`, `VALUE`, `RESOURCE`, `CAPABILITY`, `VALUE_STREAM`, `COURSE_OF_ACTION`, `WORK_PACKAGE`, `DELIVERABLE`, `PLATEAU`, and `JUNCTION`.

All relationships use `->` and require one of these `type` values: `AGGREGATION_EDGE`, `COMPOSITION_EDGE`, `ASSIGNMENT_EDGE`, `REALIZATION_EDGE`, `SERVING_EDGE`, `ACCESS_EDGE`, `INFLUENCE_EDGE`, `ASSOCIATION_EDGE`, `TRIGGERING_EDGE`, `FLOW_EDGE`, or `SPECIALIZATION_EDGE`.

A relationship may declare a document-local `id=REL1`. A local relationship ID is required when parallel relationships must be distinguished or when a relationship is itself used as an endpoint. It is not persisted as a domain ID.

Relationship properties are `mode=unspecified|read|write|read-write` for Access, free-text `strength` for Influence, `direction=undirected|directed` for Association, and `sourceMultiplicity` or `targetMultiplicity` for any relationship. Nodes support `representation=BOX|ICON`; junctions use `junction=AND|OR`.

```
#architecture [name="Application Cooperation", orientation=LTR]

P1 [name="Customer", type=BUSINESS_ACTOR, representation=ICON]
P2 [name="Order Service", type=APPLICATION_COMPONENT, representation=BOX]
P1 -> [id=REL1, type=SERVING_EDGE] P2
```

Appendix B: Sample gallery

This appendix publishes a curated selection of existing Cubetto 7 samples. They are deliberately drawn from the regular sample set, not from the Golden Samples used for automated conformance evidence. The examples are informative: they illustrate how CubeText source becomes a visual model but do not add requirements to the profile contracts.

Each diagram below was rendered by the Cubetto 7 reference implementation from the adjacent CubeText source. The selection spans five profiles and shows containment, typed objects, directional and attachment relationships, profile properties, and layout orientation.

Profile	Sample	Primary concepts
BPMN Collaboration	Order processing	Participants, lanes, events, activities, gateways, messages, and artifacts
BPMN Choreography	Gateways and messages	Participant bands, initiating parties, multiplicity, and message exchange
Organizational Chart	Units and positions	Containment and hierarchical relationships
Swimlanes Business Map	Business map	Groups, lanes, typed objects, directed flows, and attachments
Mind Map	Product ideas	Branch direction, collapsed subtrees, and hulls

B.1. BPMN collaboration: order processing

This collaboration combines participants, lanes, events, activities, gateways, message flows, attachments, and a data store. It demonstrates that the compact source preserves structural intent while leaving coordinates and routing to the renderer.

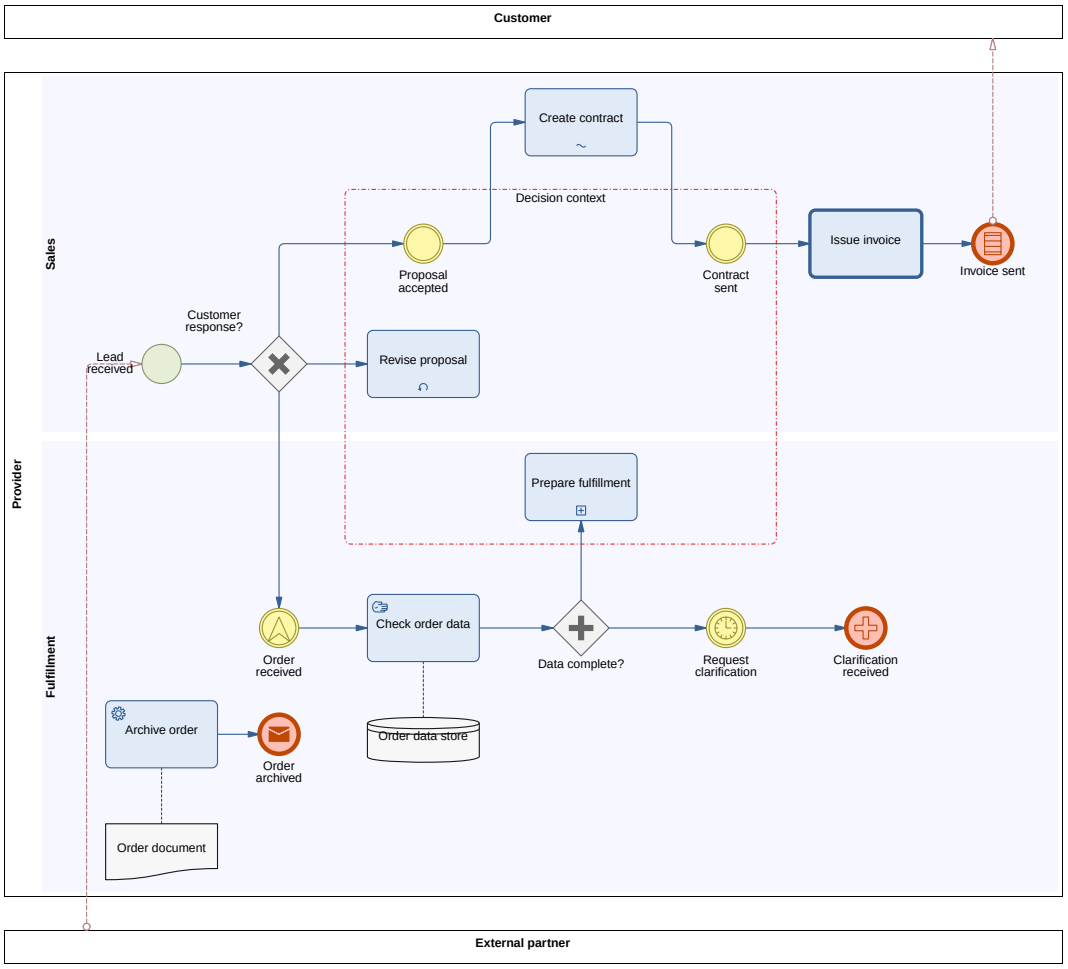

Figure 1. Order-processing collaboration rendered by Cubetto 7

B.1.1. CubeText source

```
#bpmn.collaboration [name="Default BPMN Sample", orientation="LTR"]

P1:"Customer" {}
P2:"Provider" {
    L2:"Fulfillment" {
        DS1:"Order data store"
        E5:"Order received" -> A4:"Check order data" -> G2:"Data complete?"
        A6:"Archive order" -- DO1:"Order document"
        A6 -> E6:"Order archived"
        G2 -> A5:"Prepare fulfillment"
        G2 -> E7:"Request clarification" -> E8:"Clarification received"
    }
    L1:"Sales" {
        G1:"Customer response?" -> A3:"Revise proposal"
        G1 -> E4:"Proposal accepted" -> A1:"Create contract" -> E2:"Contract sent" ->
A2:"Issue invoice" -> E3:"Invoice sent"
        G1 -> A3
        E1:"Lead received" GR1:"Decision context" { E4 A3 }
        E1 -> G1
    }
}
P3:"External partner" {}

P3 ~> E1
A4 -- DS1
G1 -> E5
E3 ~> P1
```

Source: `docs/cubetext/examples/BPMN_COLLABORATION/default_named.cubetext`.

B.2. BPMN choreography: gateways and messages

This example uses choreography tasks with participant bands, an exclusive gateway, and initiating and response messages. Profile properties distinguish initiators, repeated participants, and band placement.

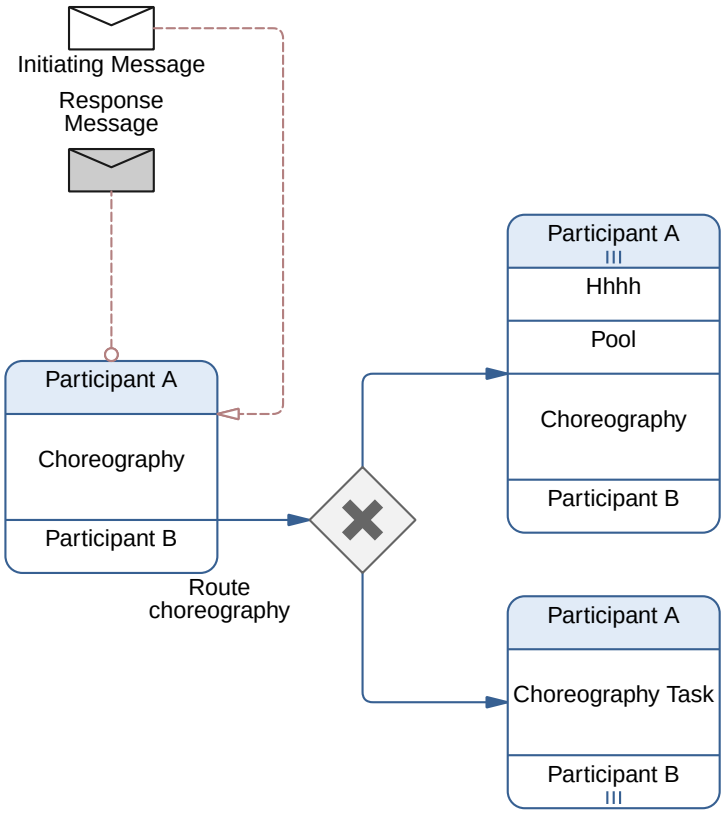

Figure 2. Choreography with participant bands and messages rendered by Cubetto 7

B.2.1. CubeText source

```
#bpmn.choreography [name="Choreography", orientation="LTR"]

C1:"Choreography" {
    Participant A [initiator=true, band=top]
    Participant B [band=bottom]
}

C2:"Choreography" {
    Pool [band=top]
    Participant A [initiator=true, multiple=true, band=top]
    Hhhh [band=top]
    Participant B [band=bottom]
}

C3:"Choreography Task" {
    Participant A [initiator=true, band=top]
    Participant B [multiple=true, band=bottom]
}

C1 -> G1:"Route choreography" -> C3
G1 -> C2
C1 ~> M2:"Response Message"
M1:"Initiating Message" ~> C1
```

Source: `docs/cubetext/examples/BPMN_CHOREOGRAPHY/gateway_messages.cubetext`.

B.3. Organizational chart: units and positions

Nested blocks place positions inside organizational units; relationships between units and positions describe the hierarchy independently of its final left-to-right layout.

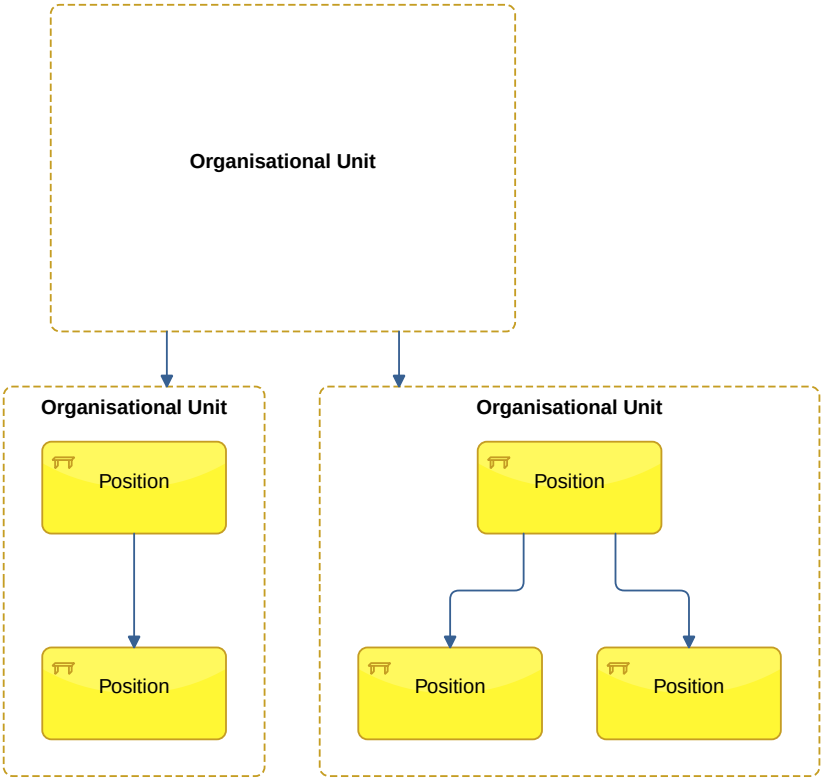

Figure 3. Organizational chart rendered by Cubetto 7

B.3.1. CubeText source

```
#organizational-chart [name="Organisational Chart", orientation="LTR"]

U1:"Organisational Unit" {
    P4:"Position" -> P1:"Position"
    P4 -> P5:"Position"
}
U2:"Organisational Unit" {}
U3:"Organisational Unit" {
    P2:"Position" -> P3:"Position"
}

U2 -> U3
U2 -> U1
```

Source: `docs/cubetext/examples/EPK_ORG_CHART/default_ltr.cubetext`.

B.4. Swimlanes business map

The swimlanes profile combines a group, a lane, named and unnamed objects, typed icons, directed flows, and cross-structure attachments. The result shows how profile properties can select meaningful visual variants without encoding their drawing primitives.

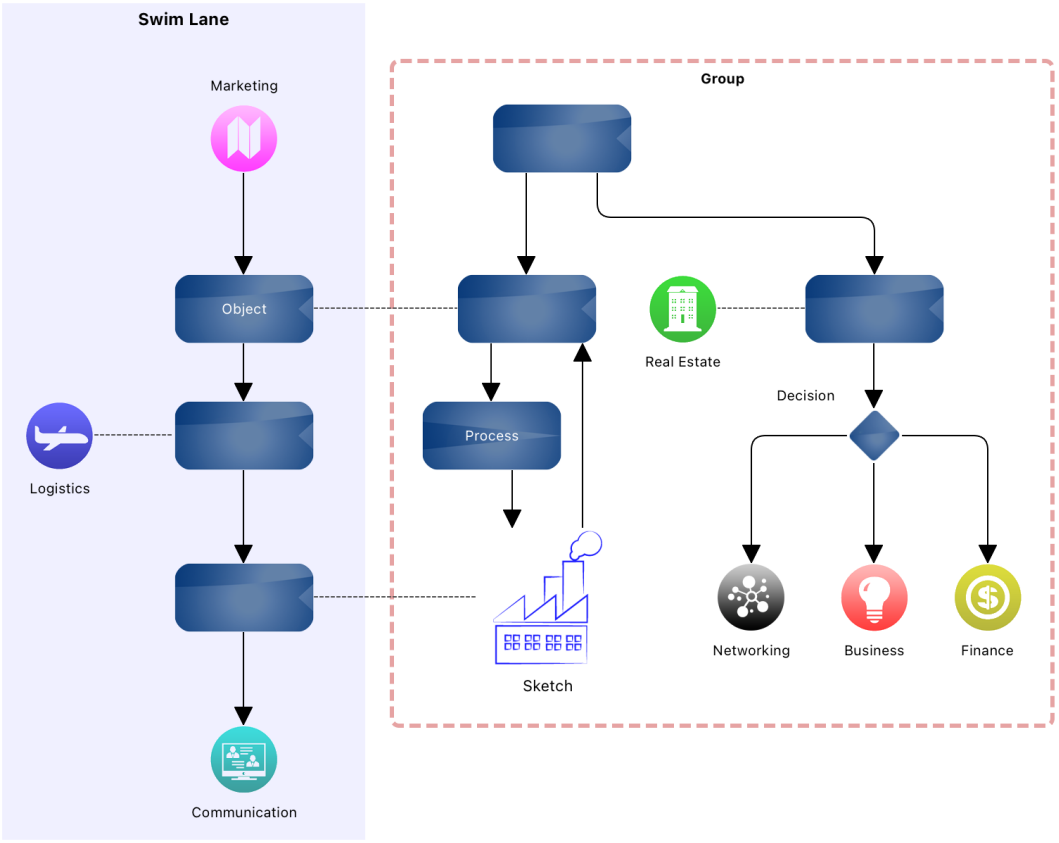

Figure 4. Business map rendered by Cubetto 7

B.4.1. CubeText source

```
#swimlanes [name="Business Map LTR", orientation="LTR"]

GR1:"Group" {
    O15 [name="Real Estate", type=realEstate]
    O2 -> O14 [name="Process", type=process] -> O16 [name="Sketch", type=sketch] -> O2
    O1 -> O2
    O1 -> O4 -> O8 [name="Decision", type=decision]
    O8 -> O9 [name="Finance", type=finance]
    O8 -> O6 [name="Business", type=business]
    O8 -> O12 [name="Networking", type=networking]
}
L1:"Swim Lane" {
    O10 [name="Logistics", type=logistics]
    O11 [name="Marketing", type=marketing] -> O13:"Object" -> O3 -> O5 -> O7
[name="Communication ", type=communication]
}

O4 -- O15
O13 -- O2
O10 -- O3
O5 -- O16
```

Source: `docs/cubetext/examples/SL_PT/default_ltr.cubetext`.

B.5. Mind map: collapsed branch and hull

This mind map illustrates a central topic, directed branch placement, a collapsed subtree, and a visual hull. These properties affect presentation while the topic relationships remain explicit in the text.

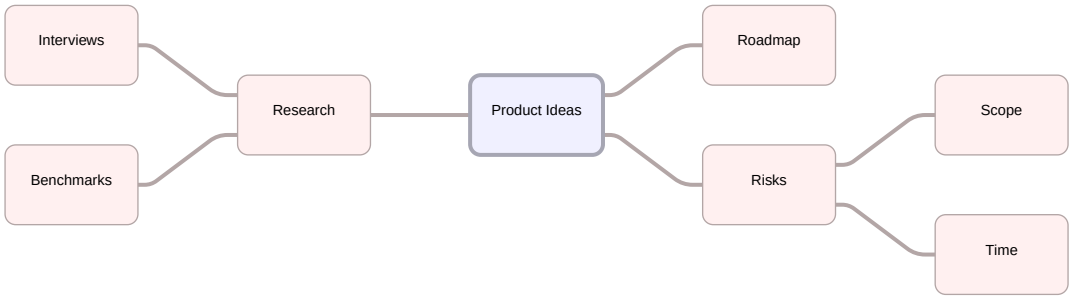

Figure 5. Product-ideas mind map rendered by Cubetto 7

B.5.1. CubeText source

```
#mindmap [name="Product Ideas"]

MT:"Product Ideas"

MT -> T1 [name="Roadmap", collapsed=true, direction=right]
T1 -> T2:"MVP"
T1 -> T3:"Launch"

MT -> T4 [name="Risks", direction=right]
T4 -> T5:"Scope"
T4 -> T6:"Time"

MT -> T7 [name="Research", hull=true, direction=left]
T7 -> T8:"Interviews"
T7 -> T9:"Benchmarks"
```

Source: `docs/cubetext/examples/MM2_PT/collapsed.cubetext`.

B.6. Provenance and reuse

The source files above predate this standalone publication and remain executable C7 samples. Their inclusion in version 1.3 and the corresponding rendered figures are licensed under CC BY 4.0 as described in `LICENSE.md`. The renderer adds no third-party artwork to these figures.

Appendix C: Normative references

- ISO/IEC 14977, *Information technology - Syntactic metalanguage - Extended BNF*. CubeText uses an ISO/IEC 14977-inspired notation; the grammar file is authoritative for the notation used by this specification.
- Creative Commons Attribution 4.0 International.
- Apache License, Version 2.0.

Appendix D: Change log

Release-history scope. Versions 0.1 through 1.2 are a reconstructed release history of CubeText as it evolved inside the Cubetto product line. They identify language milestones, not previously published editions of this standalone specification. Version 1.3 is the first consolidated public specification release.

Version	Date	Change
0.1	2024-02	First internal compact notation for objects, nesting, and directed relationships in Cubetto 7 modeling experiments.
0.4	2024-07	Introduced profile headers, document-local aliases, properties, and blocks, separating the common text shape from notation-specific vocabulary.
0.8	2024-11	Established deterministic formatting, attachment relationships, canonical serialization, and profile-specific validation.
1.0	2025-03	Stabilized the core document shape, relationship operators, and canonical output contract for productive use.
1.1	2025-09	Defined construction and reconciliation semantics, session-local aliases, and the preservation boundary for omitted model state.
1.2	2026-03	Expanded the profile family, diagnostics, and conformance model for automated tool integration.
1.3	2026-09-18	First consolidated public specification: English canonical profile tokens, legacy-token compatibility, EBNF grammar, profile registry, licensing, reproducible publication pipeline, and curated sample gallery.