Okyline Language Specification

Version: 1.0.0 **Date:** November 2025
Status: Draft

Preamble & License

Reference Implementation

A free version of **Okyline Free Studio**, fully implementing this specification (including live documentation, JSON Schema generation, and real-time validation), is available online:

https://community.studio.okyline.io

This implementation serves as the authoritative behavioral reference in case of ambiguity.

Specification Status

This document is published as a Draft of the Okyline 1.0.0 specification.
“Draft” refers only to the ongoing formalization of the specification text.

Okyline is considered stable for practical use, and the reference implementation —
Okyline Studio Free (https://community.studio.okyline.io) — fully supports the features described in this document.

In case of ambiguity, the behavior of the reference implementation is authoritative.

Table of Contents

1. Introduction

1.1 What is Okyline?

Okyline is a declarative language designed to describe the structure and constraints of JSON documents in a lightweight, readable way. It enriches simple JSON examples with inline constraints, making validation easier while keeping schemas human-friendly.

Purpose: Validate JSON data by example.

Design Philosophy: Start with an example JSON document, then add constraints directly to field names.

1.2 Why Okyline?

Modern data contracts and design-first approaches often struggle under the weight of overly complex schema languages. Okyline takes a lighter path.

The best way to describe data is to show examples.
The best way to validate data is to define rules.

Okyline lets you do both, in one place, with one syntax.

- Easy to write and read
- Progressive: start with examples, add constraints step by step
- Transpilable to JSON Schema when needed

1.3 Key Principles

1. **Example-driven:** Schemas are valid JSON documents with real example values
2. **Type inference:** Field types are automatically inferred from example values
3. **Inline constraints:** Validation rules are expressed as suffixes on field names
4. **Human-readable:** Schemas are self-documenting and easy to understand

1.4 Minimal Example

```
{
  "$oky": {
    "name|@ {2,100}|User name": "Julie",
    "status|@ ('ACTIVE','INACTIVE')|User status":"ACTIVE",
    "$appliedIf status('ACTIVE')": {
      "nbrDaysOfActivities|@ (1..22)|Number of days of activities": 22
    }
  }
}
```

Breakdown:

- `name|@ {2,100}` → required string, length between 2 and 100
- `status|@ ('ACTIVE','INACTIVE')` → required enumeration with two allowed values
- `$appliedIf status('ACTIVE')` → applies the nested block only when status is "ACTIVE"
- `nbrDaysOfActivities|@ (1..22)` → required integer between 1 and 22

Equivalent schema in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "x-oky-generated-from": "okyline",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "title": "User name",
      "examples": ["Julie"],
      "minLength": 2,
      "maxLength": 100
    },
    "status": {
      "type": "string",
      "title": "User status",
      "examples": ["ACTIVE"],
      "enum": [
        "ACTIVE", "INACTIVE"
      ]
    }
  },
  "required": [
    "name", "status"
  ],
  "allOf": [{
    "if": {
      "properties": {
        "status": {
          "enum": ["ACTIVE"]
        }
      },
      "required": ["status"]
    },
    "then": {
      "properties": {
        "nbrDaysOfActivities": {
          "type": "integer",
          "title": "Number of days of activities",
          "examples": [22],
          "minimum": 1.0,
          "maximum": 22.0
        }
      },
      "required": ["nbrDaysOfActivities"]
    }
  }]
}
```

2. Quick Start

2.1 Minimal Example

The fastest way to understand Okyline is by example:

```
  {
    "$oky": {
      "username|@ {3,20}": "alice",
      "email|@ ~$Email~": "alice@example.com",
      "age|@ (18..120)": 25,
      "role|('USER','ADMIN')": "USER",
      "verified|?": true
    }
  }
```

Reading this:

- `username|@ {3,20}` → Required string, 3-20 characters
- `email|@ ~$Email~` → Required, email format
- `age|@ (18..120)` → Required integer, 18-120
- `role|(...)` → Enum: USER or ADMIN
- `verified|?` → nullable

2.2 Core Constraint Symbols

Symbol	Type	Meaning	Example
`@`	All	Required	`"name
`?`	All	nullable	`"field
`{min,max}`	String	Length	`"name
`(min..max)`	Number	Range	`"age
`('a','b')`	All	Enum	`"status
`~format~`	String	Format/regex	`"email
`[min,max]`	Array	Size	`"items
`->`	Array/Map	Element constraint	`"tags
`!`	Array	Uniqueness	`"codes
`#`	Scalar	Key field	`"id

For complete details: See Section 5: Constraint Reference

3. Type System

3.1 Type Inference

Okyline does **not** require explicit type declarations. The field type is automatically inferred from the example value provided.

Principle: The example value determines the type that will be validated.

3.2 Recognized Types

Type	Example Value	Description
String	`"Alice"`	Unicode text
Integer	`42`	Whole numbers
Number	`3.14`	Floating-point numbers
Boolean	`true` or `false`	Boolean values
Array	`["a", "b"]`	Ordered lists
Object	`{"key": "value"}`	Nested structures

3.3 Type Inference Rules

Rule 1: Example value determines type

```
"street": "Yellow tree avenue"  // → String
"age": 42                        // → Integer
"price": 35.5                    // → Number
"active": true                   // → Boolean
"tags": ["eco", "garden"]        // → Array[String]
"address": {"city": "Paris"}     // → Object
```

Rule 2: Arrays infer element type from first item

```
"scores": [10, 20, 30]           // → Array[Integer]
"prices": [10.5, 20.99]          // → Array[Number]
"labels": ["A", "B", "C"]        // → Array[String]
```

Important: All elements in an array must conform to the inferred type.

Rule 3: Empty arrays are prohibited

Arrays in schema examples **MUST** contain at least one element for type inference.

```
// ❌ INVALID - cannot infer type
"tags": []

// ✅ VALID
"tags|[0,10]": ["example"]
```

Rule 4: Null values are prohibited as examples

`null` cannot be used as an example value. To indicate that a field can be null, use the `?` constraint.

```
// ❌ INVALID
"middleName": null

// ✅ VALID - field can be null
"middleName|?": "John"
```

3.4 Type Coercion

Okyline does **not** perform type coercion during validation. A value must exactly match the inferred type.

```
"age": 42  // Integer expected

// Validation results:
42     → ✅ Valid
"42"   → ❌ Invalid (string, not integer)
42.0   → ❌ Invalid (number, not integer)
```

4. Field Syntax

4.1 General Syntax

Every field in an Okyline schema follows this pattern:

```
"fieldName | constraints | label": exampleValue
```

Components:

1. **fieldName** (required): The JSON field name
2. **constraints** (optional): Pipe-separated constraints
3. **label** (optional): The JSON field label

4.2 Field Name

The base identifier for the field in JSON documents.

Rules:

- Must be a valid JSON string
- No special restrictions beyond JSON requirements

4.3 Constraints

Constraints applied to the field, separated from the field name by `|`.

General form: `fieldName|constraint1 constraint2 constraint3`

Example:

```
"email|@ ~$Email~": "user@example.com"
```

- `@` → required field
- `~$Email~` → must match email format

Whitespace policy: Spaces are allowed **everywhere** for readability. All the following forms are equivalent:

```
// Compact form (no spaces)
"email|@ ~$Email~": "user@example.com"

// Spaced form (readable)
"email| @ ~$Email~": "user@example.com"

// Fully spaced (maximum readability)
"email | @ ~$Email~ ": "user@example.com"

// Complex example - all equivalent:
"tags|@ [1,10]->{2,20}!": ["eco", "bio"]
"tags|@ [1,10] -> {2,20} !": ["eco", "bio"]
"tags | @ [1, 10] -> {2, 20} !": ["eco", "bio"]
"tags | @ [ 1 , 10 ] -> { 2 , 20 } ! ": ["eco", "bio"]
```

4.4 Label

An optional human-readable description placed after the constraints.

Syntax: `fieldName|constraints|Description text`

Example:

```
"status|@ ('ACTIVE','INACTIVE')|User account status": "ACTIVE"
```

Normative Rule

- A label **MUST NOT** contain the vertical bar character `|` (U+007C) **after JSON decoding**.

Purpose:

- Self-documenting schemas

- UI generation
- Developer guidance

4.5 Example Value

The value provided determines the expected type and serves as documentation.

Rules:

- MUST be a valid JSON value
- MUST NOT be `null` (use `?` constraint instead)
- For arrays: MUST contain at least one element
- Should represent a realistic example

5. Constraint Reference

5.1 Scalar Field Constraints

These constraints apply to string, integer, number, and boolean fields.

5.1.1 `@` - Required Field

Indicates that the field MUST be present in validated documents.

Applies to: All types

Example:

```
"name|@": "Alice"
```

Validation:

- `{"name": "Bob"}` → ✅ Valid
- `{}` → ❌ Invalid (missing required field)

5.1.2 `?` - Nullable Field

Indicates that the field can contain `null` values.

Applies to: All types

Example:

```
"middleName|?": "Marie"
```

Validation:

- `{"middleName": "John"}` → ✅ Valid
- `{"middleName": null}` → ✅ Valid
- `{}` → ✅ Valid (field is optional)

Note: A field can be both required and nullable using `@?` - the field must be present but can be null.

5.1.3 `{...}` - String Length

Restricts the character length of string values measured in Unicode code points

Applies to: String only

Syntax variants:

- `{max}` - Maximum length
- `{min,max}` - Minimum and maximum length

Examples:

```
"username|{3,10}": "Alice"   // min 3, max 10 characters
"city|{50}": "Paris"          // max 50 characters (no minimum)
"code|{5,5}": "ABC12"         // exactly 5 characters
```

Validation:

- Length is measured in Unicode code points
- Empty string has length 0
- Minimum defaults to 0 if not specified

5.1.4 `(...)` - Value Constraints

Restricts values to specific sets, ranges, or conditions.

Applies to: Strings, integers, numbers

Discrete Values (Enumeration)

List specific allowed values separated by commas.

Syntax: `('value1','value2','value3')`

Example:

```
"status|('ACTIVE','INACTIVE','PENDING')": "ACTIVE"
```

Validation:

- `{"status": "ACTIVE"}` → ✅ Valid
- `{"status": "DELETED"}` → ❌ Invalid

Numeric Ranges

Define inclusive ranges using `..` notation.

Syntax: `(min..max)`

Example:

```
"age|(18..120)": 30
"price|(0..1000)": 49.99
```

Validation:

- Boundaries are **inclusive**: `(1..10)` accepts both 1 and 10
- Works with integers and numbers

Numeric Comparisons

Use comparison operators for one-sided constraints.

Operators: `>` , `<` , `>=` , `<=`

Examples:

```
"quantity|(>0)": 5          // strictly greater than 0
"discount|(<=50)": 20       // less than or equal to 50
"score|(>=10)": 85          // greater than or equal to 10
```

Lexicographic Ranges

String ranges using alphabetical ordering.

Syntax: `('start'..'end')`

Example:

```
"letter|('A'..'Z')": "B"     // A through Z (single uppercase letter)
"grade|('A'..'F')": "C"      // Letter grades
```

Note: Comparisons use Unicode lexicographic ordering.

Combined Constraints

Combine multiple constraints with commas.

Examples:

```
"value|(1,2..5,>10)": 12     // equals 1, OR between 2-5, OR greater than 10
"code|('A','B',100..200)": "A"
```

Validation logic: Value must satisfy **at least one** of the listed constraints (logical OR).

Reference to Nomenclature

Use values from a predefined registry (see §6.1).

Syntax: `($REGISTRY_NAME)`

Example:

```
{
  "$nomenclature": {
    "COLORS": "RED,GREEN,BLUE,YELLOW"
  },
  "$oky": {
    "favoriteColor|($COLORS)": "RED"
  }
}
```

5.1.5 `~...~` - Format Validation

Validates strings against regular expressions or semantic formats.

Applies to: String only

Inline Regex

Embed a regular expression directly in the constraint.

Regex flavor: Okyline uses **ECMA-262** (JavaScript) regular expression syntax.

Syntax: `~pattern~`

Examples:

```
"postalCode|~^[0-9]{5}$~": "75001"
"phoneNumber|~^\\+33[0-9]{9}$~": "+33612345678"
"sku|~^[A-Z]{2}-\\d{4}$~": "AB-1234"
```

Important:

- Backslashes must be escaped in JSON strings (`\d` becomes `\\d`)
- Regex delimiters (`/`) are **not** used; the tilde `~` serves as delimiter
- Flags are not supported in inline patterns (use anchors `^` `$` instead)

ECMA-262 reference: https://262.ecma-international.org/13.0/#sec-regexp-regular-expression-objects

Named Formats

Reference predefined patterns from `$format` block or use built-in formats.

Syntax: `~$FormatName~`

Example:

```
{
  "$format": {
    "PostalCode": "^[0-9]{5}$"
  },
  "$oky": {
    "zipCode|~$PostalCode~": "75001"
  }
}
```

> **Non-normative — Implementation note (regex)**
>
> Okyline specifies ECMA-262 regex syntax and match semantics. Operational concerns such as regex timeouts, backtracking limits, input size limits, or sandboxing are **out of scope** for this specification and are left to implementations. Implementations **MAY** apply safeguards (e.g., timeouts or complexity limits). If a safeguard prevents evaluation, the validator **SHOULD** report a clear execution error rather than altering match semantics.

Built-in Formats

Okyline provides standard formats that can be used without declaration.

Semantic validation (validates logical consistency):

Format	Description	Validation	Example
`$Date`	ISO 8601 date	Validates leap years, month lengths, day ranges	`"2025-05-30"`
`$DateTime`	ISO 8601 datetime	Validates date/time consistency, timezone offsets	`"2025-05-30T14:30:00Z"`

Syntactic validation with structural constraints:

Format	Description	Validation	Example
`$Time`	ISO 8601 time	RFC 3339 syntax (HH:MM:SS + optional fractions/offset)	`"14:30:00"` , `"14:30:00.123Z"`
`$Uri`	URI with scheme	RFC 3986 syntax + port range validation (1-65535)	`"https://example.com:8080/path"`
`$Ipv4`	IPv4 address	Dotted-decimal notation, octets 0-255	`"192.168.1.1"`
`$Ipv6`	IPv6 address	RFC 4291 syntax + single `::` compression	`"2001:db8::1"`
`$Hostname`	Hostname	RFC 1034 labels + total length ≤ 255 chars	`"example.com"`

Syntactic validation (pattern matching):

Format	Description	Example
`$Email`	Email address	`"user@example.com"`
`$Uuid`	UUID (versions 1-5)	`"550e8400-e29b-41d4-a716-446655440000"`

Overriding built-in formats:

All built-in formats, including semantic formats, **can be overridden** in the `$format` block to accommodate different validation rules or regional formats.

Example - Override `$Date` for European format:

```
{
  "$format": {
    "Date": "^(0[1-9]|[12]\\d|3[01])/(0[1-9]|1[0-2])/\\d{2}$"
  },
  "$oky": {
    "birthDate|~$Date~": "15/05/90",
    "eventDate|~$Date~": "31/12/25"
  }
}
```

Warning: When overriding semantic formats with regex, you lose semantic validation. The pattern `31/02/25` would be syntactically valid but semantically incorrect (February 31 doesn't exist).

Example - Override `$Email` for stricter validation:

```
{
  "$format": {
    "Email": "^[a-zA-Z0-9._%+-]+@ [a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"
  },
  "$oky": {
    "email|~$Email~": "user@example.com"
  }
}
```

Validation examples:

```
// Default $Date - Semantic validation (ISO 8601)
"date|~$Date~": "2024-02-29"  // ✅ Valid (2024 is leap year)
"date|~$Date~": "2025-02-29"  // ❌ Invalid (2025 is not leap year)
"date|~$Date~": "2025-13-01"  // ❌ Invalid (month 13 doesn't exist)

// Custom $Date - Regex validation (DD/MM/YY)
{
  "$format": {
    "Date": "^(0[1-9]|[12]\\d|3[01])/(0[1-9]|1[0-2])/\\d{2}$"
  },
  "$oky": {
    "date|~$Date~": "29/02/25"  // ✅ Syntactically valid (but semantically wrong!)
  }
}
```

5.1.6 `#` - Key Field

Marks a field as an identifier within an object. Used for enforcing uniqueness in lists of objects.

Applies to: Scalars inside objects

Example:

```
{
  "users|@ [1,10] -> !": [
    {
      "id|#": "user001",
      "name": "Alice"
    },
    {
      "id|#": "user002",
      "name": "Bob"
    }
  ]
}
```

Behavior:

- Key fields are used to determine object uniqueness in lists
- Multiple fields can be marked as keys (composite key)
- See §5.2.3 for uniqueness validation

5.1.7 `%` - Default Value

Indicates that the example value is also the default value. This is **informational only** and does not affect validation.

Applies to: All types

Example:

```
"country|%": "France",
"theme|%('light','dark')": "light"
```

Use cases:

- UI form generators
- Documentation
- Code generation tools

5.2 List Constraints

These constraints apply to array fields.

5.2.1 `[...]` - List Size

Defines minimum and maximum number of elements in a list.

Syntax variants:

- `[max]` - Maximum size only
- `[min,max]` - Minimum and maximum
- `[min,*]` - Minimum only (no maximum)
- `[*]` - Any size (no constraints)

Examples:

```
"tags|[1,5]": ["eco", "garden"]          // 1 to 5 items
"codes|[10,*]": ["A","B"]                // at least 10 items
"letters|[5]": ["A","B"]                 // max 5 items
"items|[*]": ["x"]                       // any number of items
```

Empty Lists & Type Inference

Array examples **MUST NOT** be empty, regardless of their size constraints.

Validation:

- Empty lists are valid if minimum is 0 or `[*]`
- Boundaries are inclusive

5.2.2 `->` - Element/Value Constraints

Applies constraints to each element in a collection (list or map).

Syntax:

- Arrays: `[size] -> constraints`
- Maps: `[key:size] -> constraints`

Examples:

```
// Each tag must be 2-10 characters and unique
"tags|@ [1,5] -> {2,10}!": ["eco", "garden"]

// Each score must be between 0-100
"scores|[*] -> (0..100)": [85, 92, 78]

// Each email must match format
"contacts|[1,10] -> ~$Email~": ["a@example.com", "b@example.com"]
```

Combination with object constraints:

```
"products|[1,50] -> !": [
  {
    "sku|#": "ABC123",
    "name|@ {2,100}": "Product Name",
    "price|@ (0..10000)": 29.99
  }
]
```

5.2.3 `!` - Uniqueness

Enforces that all elements in a list are unique.

Applies to: Arrays

Behavior:

- For scalar lists: uniqueness by value
- For object lists: uniqueness by composite key formed from key fields (`#`)

Examples

Scalar uniqueness:

```
"codes|[1,10] -> !": ["A001", "B002", "C003"]
```

- ✅ Valid: `["A", "B", "C"]`
- ❌ Invalid: `["A", "B", "A"]` (duplicate "A")

Object-based Uniqueness

When applied to a list of objects, the `!` modifier enforces uniqueness based on one or more fields marked with `#`.

Requirements:

- At least one key field (`#`) **MUST** be declared
- If no key fields are declared, validation **MUST fail**
- An object missing **all** declared key fields **MUST** cause a validation error
- An object missing **some** key fields is valid; only present ones contribute to the composite key

Example:

```
"records|[*] -> !": [
  {"type|#": "A", "code|#": "001", "label": "First"},
  {"type|#": "A", "code|#": "002", "label": "Second"},
  {"type|#": "B", "code|#": "001", "label": "Third"}
]
```

- ✅ Valid: all composite keys (`A-001` , `A-002` , `B-001`) are unique
- ❌ Invalid: two objects with the same composite key

Composite Key Construction

When multiple fields are marked as key fields (modifier `#`), Okyline constructs a composite key to verify uniqueness of objects in a list.

Algorithm:

The composite key is formed by concatenating the values of fields marked with `#` , in their **order of declaration** in the schema, separated by a hyphen (`-`).

Rules:

1. **Separator:** `-` (U+002D HYPHEN-MINUS)
2. **Encoding:** Each value is URL-encoded according to RFC 3986 (UTF-8)
3. **Number normalization:** Trailing zeros are removed (`1.0` → `1` , `123.000` → `123`)
4. **Booleans:** Converted to `true` or `false` (lowercase)

5. **Null/absent values:** Ignored (do not contribute to the key)
6. **Non-scalar fields:** Ignored (only scalar values are accepted as keys)

Example 1: Two string fields

Schema:

```
"items|[*] -> !": [
  {
    "country|#": "FR",
    "code|#": "75001"
  }
]
```

Object: `{ "country": "FR", "code": "75001" }`
Composite key: `"FR-75001"`

Example 2: String + number

Schema:

```
"sessions|[*] -> !": [
  {
    "userId|#": 42,
    "sessionId|#": "abc-123"
  }
]
```

Object: `{ "userId": 42, "sessionId": "abc-123" }`
Composite key: `"42-abc%2D123"` (hyphen in "abc-123" is URL-encoded)

Example 3: Missing value

Schema:

```
"addresses|[*] -> !": [
  {
    "country|#": "FR",
    "region|#?": "IDF",
    "code|#": "75001"
  }
]
```

Object: `{ "country": "FR", "code": "75001" }` (region absent)
Composite key: `"FR-75001"` (region ignored because absent)

Example 4: Number normalization

Schema:

```
"products|[*] -> !": [
  {
    "sku|#": "ABC",
    "version|#": 1.0
  }
]
```

Object 1: `{ "sku": "ABC", "version": 1.0 }`

Object 2: `{ "sku": "ABC", "version": 1 }`

Both have composite key: `"ABC-1"` → ❌ Duplicate!

Example 5: Boolean values

Schema:

```
"flags|[*] -> !": [
  {
    "name|#": "feature",
    "enabled|#": true
  }
]
```

Object: `{ "name": "feature", "enabled": true }`

Composite key: `"feature-true"`

Example 6: URL encoding

Schema:

```
"paths|[*] -> !": [
  {
    "path|#": "/api/v1",
    "method|#": "GET"
  }
]
```

Object: `{ "path": "/api/v1", "method": "GET" }`

Composite key: `"%2Fapi%2Fv1-GET"` (slashes are URL-encoded)

Error Handling

Situation	Behavior
No key fields declared	❌ Validation error if uniqueness (`!`) is requested
All key fields absent/null in an object	❌ Validation error (uniqueness not verifiable)
Duplicate composite keys	❌ Validation error: `NOT_UNIQUE`
Non-scalar key field (object/array)	Field is ignored in key construction

Example - Error cases:

```
// ❌ No key fields declared
"items|[*] -> !": [
  {"name": "A"},
  {"name": "B"}
]
// Error: Cannot verify uniqueness without key fields

// ❌ All key fields absent
"items|[*] -> !": [
  {"id|#": 1, "name": "A"},
  {"name": "B"}  // id missing
]
// Error: Object missing all key fields

// ❌ Duplicate keys
"items|[*] -> !": [
  {"id|#": 1, "name": "A"},
  {"id|#": 1, "name": "B"}
]
// Error: Duplicate key "1"
```

Design Rationale

✅ **Explicit semantics** via mandatory key field declaration
✅ **Superior performance** through O(n) hash-based checking
✅ **Business alignment** by validating identity, not structure
✅ **Clear errors** when uniqueness cannot be established
✅ **No collision ambiguity** via RFC 3986 URL encoding

This design makes Okyline schemas self-documenting and production-ready for high-performance validation scenarios.

5.3 Map Constraints

Maps are objects used as dictionaries with dynamic keys.

5.3.1 `[key_constraint:size_constraint]` - Map Constraints

Syntax: `[key_pattern:max_entries]`

Variants:

- `[*:max]` - Any string key, maximum entries
- `[~pattern~:max]` - Keys matching regex, maximum entries
- `[~pattern~:*]` - Keys matching regex, any number of entries

Examples:

Free keys:

```
"translations|[*:5]": {
  "en": "Hello",
  "fr": "Bonjour",
  "es": "Hola"
}
```

- Any string keys allowed
- Maximum 5 entries

Pattern-constrained keys:

```
"products|[~^SKU-\\d{5}$~:*]": {
  "SKU-12345": {
    "name|@": "Product A",
    "price|@ (0..1000)": 29.99
  },
  "SKU-67890": {
    "name|@": "Product B",
    "price|@ (0..1000)": 49.99
  }
}
```

- Keys must match pattern `SKU-` followed by 5 digits
- Unlimited number of entries

Language codes:

```
"labels|[~^[a-z]{2}(-[A-Z]{2})?$~:10] -> {1,100}": {
  "en": "Label",
  "fr": "Étiquette",
  "en-US": "Label (US)"
}
```

- Keys match ISO language codes
- Maximum 10 translations
- Each value 1-100 characters

5.4 Polymorphism Constraints

5.4.1 `$oneOf` - Exclusive Match

The value must match **exactly one** of the provided schema examples.

Applies to: Objects and arrays of objects

Example:

```
"payment|@ $oneOf": [
  {
    "type|@ ('card')": "card",
    "cardNumber|@ {16}": "1234567812345678",
    "cvv|@ {3}": "123"
  },
  {
    "type|@ ('paypal')": "paypal",
    "email|@ ~$Email~": "user@example.com"
  },
  {
    "type|@ ('bank')": "bank",
    "iban|@ {15,34}": "FR7630006000011234567890189"
  }
]
```

Validation:

- Document must match one and only one schema
- Matching is typically determined by a discriminator field (here: `type`)

5.4.2 `$anyOf` - Non-Exclusive Match

The value must match **at least one** of the provided schema examples. This is the **default** behavior when multiple examples are provided.

Applies to: Objects and arrays of objects

Example:

```
"notification|$anyOf": [
  {"email|~$Email~": "user@example.com"},
  {"sms|~^\\+[0-9]{10,15}$~": "+33612345678"}
]
```

Validation:

- Document can match one or more schemas
- Useful for flexible validation

5.5 Combination Rules

Rule 1: One constraint per type

Only **one** constraint of the same type is allowed per field.

Invalid:

```
// ❌ Two length constraints
"name|{10,50}{5,20}": "Alice"

// ❌ Two value constraints
"age|(0..100)(18..65)": 30
```

Valid:

```
// ✅ Multiple constraints in single constraint
"age|(0..18,65..100)": 75

// ✅ Different constraint types
"name|@ {2,50}": "Alice"
```

Rule 2: Different constraint types can be combined

```
// ✅ Required + length + pattern
"email|@ {5,100}~$Email~": "user@example.com"

// ✅ Required + range + label
"age|@ (18..120)|User age in years": 30

// ✅ List size + element constraints + uniqueness
"tags|[1,5] -> {2,20}!": ["eco", "bio"]
```

Rule 3: Order matters for readability (not for parsing)

While order doesn't affect validation, the recommended order is:

1. Existence (`@` , `?`)

2. Value constraints (`(...)` , `{...}` , `~...~`)

Recommended:

```
"email|@ {5,100}~$Email~|User email address": "user@example.com"
```

6. Advanced Features

6.1 Reusable Value Registries - `$nomenclature`

Define centralized lists of allowed values that can be reused across multiple fields.

Purpose:

- Centralize enum definitions
- Improve maintainability
- Ensure consistency

6.1.1 Declaration

Declared at the root level alongside `$oky` .

Syntax:

```
{
  "$nomenclature": {
    "REGISTRY_NAME": "value1,value2,value3"
  }
}
```

Example:

```
{
  "$nomenclature": {
    "STATUS": "DRAFT,VALIDATED,REJECTED,ACTIVE,INACTIVE,ARCHIVED",
    "COUNTRIES": "FRA,DEU,ESP,USA,GBR",
    "UNITS": "kg,m,cm,L,°C"
  }
}
```

Rules:

- Keys are uppercase identifiers
- Values are comma-separated strings

- No quotes needed around individual items

6.1.2 Usage

Reference a nomenclature using `($NAME)` syntax in value constraints.

Example:

```
{
  "$nomenclature": {
    "COLORS": "RED,GREEN,BLUE,YELLOW"
  },
  "$oky": {
    "product": {
      "name|@": "T-Shirt",
      "color|@ ($COLORS)": "RED",
      "secondaryColor|($COLORS)": "BLUE"
    }
  }
}
```

Validation:

- `{"color": "RED"}` → ✅ Valid
- `{"color": "PURPLE"}` → ❌ Invalid

6.2 Reusable Formats - `$format`

Define named regular expressions for reuse across the schema.

6.2.1 Declaration

Declared at root level alongside `$oky` .

Syntax:

```
{
  "$format": {
    "PatternName": "^regular_expression$"
  }
}
```

Example:

```
{
  "$format": {
    "PostalCode": "^[0-9]{5}$",
    "PhoneNumber": "^\\+33[0-9]{9}$",
    "Sku": "^SKU-[0-9]{5}$",
    "IsoDate": "^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\\d|3[01])$"
  }
}
```

Important: Remember to escape backslashes in JSON (`\d` → `\\d`).

6.2.2 Usage

Reference with `~$PatternName~` syntax.

Example:

```
{
  "$format": {
    "Code": "^[A-Z]{2}-\\d{4}$"
  },
  "$oky": {
    "session": {
      "code|@ ~$Code~": "SR-0012",
      "backupCode|~$Code~": "AB-9999"
    }
  }
}
```

6.2.3 Format Resolution

When `~$Name~` is encountered, Okyline resolves in this order:

1. **Custom pattern:** Check `$format[Name]`
2. **Built-in format:** Check if `Name` is a recognized built-in (see §5.1.5)

Override built-ins:

```
{
  "$format": {
    "Date": "^[0-9]{2}/[0-9]{2}/[0-9]{4}$"
  },
  "$oky": {
    "eventDate|~$Date~": "25/12/2025"
  }
}
```

- Custom `$Date` pattern overrides the built-in ISO date format

6.3 Conditional Rules

Apply structural changes based on field values or existence.

6.3.1 `$requiredIf` - Conditional Required Fields

Require fields when a condition is met.

Syntax: `"$requiredIf condition": ["field1", "field2"]`

Example:

```
{
  "$oky": {
    "person": {
      "age": 17,
      "parentConsent": true,
      "$requiredIf age(<18)": ["parentConsent"]
    }
  }
}
```

- If age < 18, then `parentConsent` must be present

6.3.2 `$requiredIfNot` - Required If Condition Not Met

Require fields when a condition is NOT met.

Syntax: `"$requiredIfNot condition": ["field1", "field2"]`

Example:

```
{
  "$oky": {
    "person": {
      "age": 25,
      "idCard": "AB123456",
      "$requiredIfNot age(<18)": ["idCard"]
    }
  }
}
```

- If age is NOT < 18 (i.e., >= 18), then `idCard` must be present

6.3.3 `$forbiddenIf` - Conditional Forbidden Fields

Forbid fields when a condition is met.

Syntax: `"$forbiddenIf condition": ["field1", "field2"]`

Example:

```
{
  "$oky": {
    "account": {
      "status": "CLOSED",
      "lastLogin": "2025-01-15",
      "$forbiddenIf status('CLOSED')": ["lastLogin"]
    }
  }
}
```

- If status is "CLOSED", `lastLogin` must not be present

6.3.4 `$forbiddenIfNot` - Forbidden If Condition Not Met

Forbid fields when a condition is NOT met.

Syntax: `"$forbiddenIfNot condition": ["field1", "field2"]`

Example:

```
{
  "$oky": {
    "account": {
      "status": "ACTIVE",
      "closureReason": "Moved abroad",
      "$forbiddenIfNot status('CLOSED')": ["closureReason"]
    }
  }
}
```

- If status is NOT "CLOSED", then `closureReason` must not be present

6.3.5 `$appliedIf` - Conditional Structure

Add fields dynamically based on a condition.

Syntax variants:

Simple if/else:

```
"$appliedIf condition": {
  "field1|@": value,
  "$else": {
    "field2|@": value
  }
}
```

Example:

```
{
  "$oky": {
    "employee": {
      "status": "ACTIVE",
      "$appliedIf status('ACTIVE')": {
        "workDays|@ (1..22)": 20
      }
    }
  }
}
```

```
{
  "$oky": {
    "employee": {
      "status": "ACTIVE",
      "$appliedIf status('ACTIVE')": {
        "workDays|@ (1..22)": 20
      },
      "$else": {
        "reason|@": "On leave"
      }
    }
  }
}
```

Switch-case:

```
"$appliedIf fieldName": {
  "('value1')": { "field1|@": value },
  "('value2')": { "field2|@": value },
  "$else": { "field3|@": value },
  "$notExist": { "field4|@": value }
}
```

Example:

```
{
  "$oky": {
    "employee": {
      "status": "ACTIVE",
      "$appliedIf status": {
        "('ACTIVE')": {
          "workDays|@ (1..22)": 20
        },
        "('INACTIVE')": {
          "reason|@": "On leave"
        },
        "$else": {
          "note|@": "Status unknown"
        }
      }
    }
  }
}
```

6.3.6 `$requiredIfExist` - Existence-Based Required

Require fields if another field exists.

Example:

```
{
  "$oky": {
    "contact": {
      "firstName": "John",
      "$requiredIfExist firstName": ["lastName"]
    }
  }
}
```

6.3.7 `$requiredIfNotExist` - Require If Field Does Not Exist

Require fields if another field does NOT exist.

Example:

```
{
  "$oky": {
    "contact": {
      "$requiredIfNotExist email": ["phone"]
    }
  }
}
```

- If `email` does not exist, then `phone` must be present

6.3.8 `$forbiddenIfExist` - Existence-Based Forbidden

Forbid fields if another field exists.

Example:

```
{
  "$oky": {
    "product": {
      "archived": true,
      "$forbiddenIfExist archived": ["active"]
    }
  }
}
```

6.3.9 `$forbiddenIfNotExist` - Forbid If Field Does Not Exist

Forbid fields if another field does NOT exist.

Example:

```
{
  "$oky": {
    "product": {
      "$forbiddenIfNotExist sku": ["internalCode"]
    }
  }
}
```

- If `sku` does not exist, then `internalCode` must not be present

6.3.10 `$appliedIfExist` - Existence-Based Structure

Add fields if another field exists.

Example:

```
{
  "$oky": {
    "order": {
      "tracking": "ABC123",
      "$appliedIfExist tracking": {
        "carrier|@": "DHL",
        "estimatedDelivery|@ ~$Date~": "2025-12-25"
      }
    }
  }
}
```

6.3.11 `$appliedIfNotExist` - Add Fields If Field Does Not Exist

Add fields dynamically if another field does NOT exist.

Example:

```
{
  "$oky": {
    "contact": {
      "$appliedIfNotExist email": {
        "phone|@": "+33612345678",
        "phoneVerified|@": true
      }
    }
  }
}
```

- If `email` does not exist, then `phone` and `phoneVerified` fields are required

6.4 Computed Expressions — `$compute`

Define **named expressions** to implement business rules, cross-field validation, and calculated constraints using the
Okyline Expression Language (see *Annex C — Okyline Expression Language*).

Purpose

- Express complex business logic (totals, discounts, validations)
- Reference other fields and computed values
- Chain computations with clear, readable syntax

6.4.1 Declaration

Computed expressions are declared at the **root level** of the schema.

Syntax:

```
{
  "$compute": {
    "ExpressionName": "expression using fields, operators, and functions"
  }
}
```

Context:
Expressions are evaluated in the context of the enclosing object.
All fields in the same object are accessible by name.
Missing fields resolve to `null` .

6.4.2 Usage in Validation

Reference computed expressions using the syntax `(%ExpressionName)` in value constraints.

Example:

```
{
  "$oky": {
    "invoice": {
      "subtotal": 100.0,
      "taxRate": 0.2,
      "total|(%ValidTotal)": 120.0
    }
  },
  "$compute": {
    "ValidTotal": "total == subtotal * (1 + taxRate)"
  }
}
```

6.4.3 Referencing Other Computes

You can reference another computed expression using `%ComputeName` within the same `$compute` block.

Example — Chained Computations:

```
{
  "$compute": {
    "BasePrice": "1000",
    "Tax": "%BasePrice * 0.2",
    "Shipping": "50",
    "Total": "%BasePrice + %Tax + %Shipping"
  }
}
```

- **Circular references** are detected at schema load time and **must raise an error**.

6.4.4 Language Overview

The Okyline Expression Language supports:

Operators

Category	Operators	Description
Arithmetic	`+` , `-` , `*` , `/`	Standard math operations
Comparison	`>` , `<` , `>=` , `<=` , `==` , `!=` , `===` , `!==`	Numeric and lexicographic comparisons
Logical	`&&` , `	
Null coalescing	`??`	Returns right operand if left is `null`

Category	Operators	Description
Ternary	`condition ? trueValue : falseValue`	Conditional expression

Null Handling

- Arithmetic operations **propagate null** (SQL-style three-valued logic).
- String concatenation treats `null` as an empty string `""` .
- Use `??` to provide defaults — e.g., `price ?? 0` .

Function Categories

(For full details, see Annex C.)

Category	Examples
Numeric	`sum()` , `average()` , `min()` , `max()` , `abs()` , `sqrt()` , `round()` , `floor()` , `ceil()`
String	`length()` , `substring()` , `trim()` , `toUpperCase()` , `toLowerCase()` , `contains()` , `replace()`
Date	`date()` , `today()` , `daysBetween()` , `plusDays()` , `formatDate()`
Aggregation	`sum(collection, expr)` , `count(collection, expr)` , `average(collection, expr)`
Utility	`toNum()` , `toStr()`

Note: The `%identifier` syntax is not a function but a **reference operator** for accessing compute expressions.

6.4.5 Complete Example

```
{
  "$oky": {
    "order": {
      "items|@ [1,100]": [
        {"price": 10.0, "quantity": 2},
        {"price": 15.0, "quantity": 3}
      ],
      "discount": 5.0,
      "shippingCost": null,
      "total|(%FinalTotal)": 60.0
    }
  },
  "$compute": {
    "ItemTotal": "price * quantity",
    "SubTotal": "sum(items, %ItemTotal)",
    "Shipping": "shippingCost ?? 10.0",
    "FinalTotal": "total == %SubTotal - discount + %Shipping"
  }
}
```

Explanation

1. `ItemTotal` calculates each item's total (`price × quantity`).
2. `SubTotal` sums all item totals using the `sum()` aggregation function.
3. `Shipping` uses `??` to default to `10.0` when `shippingCost` is `null` .
4. `FinalTotal` validates that `total` equals the computed subtotal minus discount plus shipping.
5. Computed expressions reference each other using `%` for clarity and readability.

6.4.6 Full Language Specification

For the complete specification of the expression language, see **Annex C — Okyline Expression Language**, which defines:

- Operators and precedence rules
- Comprehensive function reference
- Null handling semantics
- String index handling rules
- Type coercion rules
- Error handling requirements

7. Document Structure

7.1 Root Structure

An Okyline document is a JSON object that MUST contain at least the `$oky` key.

Minimal valid document:

```
{
  "$oky": {
    "message": "Hello"
  }
}
```

7.2 Mandatory Key

`$oky`

The central element containing the schema definition. This key is **required**.

Type: Object
Content: Field definitions with optional constraints

7.3 Optional Metadata Keys

`$okylineVersion`

Specifies the version of the Okyline specification used.

Type: String
Format: Semantic versioning (e.g., `"1.0"` , `"1.2.3"`)
Default: `"1.0"`

`$version`

Version of the schema itself (not the Okyline language).

Type: String
Example: `"1.2.3"`

`$title`

Human-readable title for the schema.

Type: String
Example: `"User Profile Schema"`

`$description`

Description of what the schema validates.

Type: String
Example: `"Schema for user profile data including personal information and preferences"`

`$additionalProperties`

Controls whether unknown fields are allowed in validated documents.

Type: Boolean
Default: `false` (unknown attributes are not allowed)

7.4 Complete Example

```
{
  "$okylineVersion": "1.0",
  "$version": "1.2.3",
  "$title": "User Profile",
  "$description": "Schema for user profiles with contact information",
  "$additionalProperties": false,
  "$oky": {
    "user": {
      "id|@": 123,
      "name|@ {2,50}": "Alice",
      "email|~$Email~": "alice@example.com"
    }
  }
}
```

§7.3.5 `$additionalProperties`

Description

Controls whether unknown (undeclared) fields are allowed in validated documents.

Scope and Inheritance

- `$additionalProperties` **MAY** be defined at the **root level** of the Okyline document. When defined at the root, it applies **globally** to the entire JSON structure.
- `$additionalProperties` **MAY** also be defined **inside an object** within `$oky`. When defined inside an object, the rule **applies only to that specific object**, and **does not propagate recursively** to its child objects.
- Child objects **inherit** the global (root) rule **only if** they do not redefine it locally.

Normative Rules

- By default, if not specified, `$additionalProperties` is considered **false** (unknown fields are not allowed).
- A local `$additionalProperties` **overrides** the global rule **for that object only**.
- The setting **is not recursive** — it is **not inherited** by nested objects inside the one where it is defined.

Examples

Global setting only:

```
{
  "$additionalProperties": false,
  "$oky": {
    "user": {
      "name|@": "Alice",
      "age|@": 30
    }
  }
}
```

→ Unknown fields anywhere are rejected.

Local override (non-recursive):

```
{
  "$additionalProperties": false,
  "$oky": {
    "user": {
      "$additionalProperties": true,
      "name|@": "Alice",
      "address": {
        "street|@": "Main St"
      }
    }
  }
}
```

✅ `user` may include extra fields (e.g., `"nickname"`)
❌ `user.address` may **not** include unknown fields (no recursive propagation)

Summary Table

Level	Applies To	Inherited by Children	Default
Root	Entire document	✅ (unless overridden)	`false`
Object (local)	That object only	❌ (not recursive)	—

8. Validation Rules

8.1 Type Validation

Values must match the type inferred from the example.

Valid:

```
// Schema: "age": 42 (Integer)
{"age": 30}  ✅
```

Invalid:

```
{"age": "30"}    ❌ (string, not integer)
{"age": 30.5}    ❌ (number, not integer)
{"age": null}    ❌ (null, not integer - use ? constraint for nullable)
```

8.2 Required Field Validation

Fields marked with `@` must be present.

Valid:

```
// Schema: "name|@": "Alice"
{"name": "Bob"}  ✅
```

Invalid:

```
{}  ❌ (missing required field 'name')
```

8.3 Nullable Field Validation

Fields marked with `?` can be null or absent.

Valid:

```
// Schema: "middleName|?": "John"
{"middleName": "Marie"}  ✅
{"middleName": null}     ✅
{}                       ✅
```

8.4 String Length Validation

Strings must satisfy length constraints.

Valid:

```
// Schema: "username|{3,10}": "alice"
{"username": "bob"}       ✅ (length 3)
{"username": "alexander"} ✅ (length 9)
```

Invalid:

```
{"username": "jo"}          ❌ (length 2, minimum is 3)
{"username": "verylongusername"} ❌ (length > 10)
```

8.5 Value Constraint Validation

Values must satisfy range, enum, or comparison constraints.

Valid:

```
// Schema: "age|(18..65)": 30
{"age": 18}  ✅
{"age": 42}  ✅
{"age": 65}  ✅
```

Invalid:

```
{"age": 17}  ❌ (below minimum)
{"age": 66}  ❌ (above maximum)
```

8.6 List Size Validation

Arrays must satisfy size constraints.

Valid:

```
// Schema: "tags|[1,5]": ["eco"]
{"tags": ["a"]}                ✅
{"tags": ["a","b","c"]}        ✅
{"tags": ["a","b","c","d","e"]} ✅
```

Invalid:

```
{"tags": []}                    ❌ (empty, minimum is 1)
{"tags": ["a","b","c","d","e","f"]} ❌ (6 items, maximum is 5)
```

8.7 Uniqueness Validation

Lists marked with `!` must have unique elements.

Scalar uniqueness:

```
// Schema: "codes|[1,5] -> !": ["A"]

// Valid:
{"codes": ["A", "B", "C"]}  ✅

// Invalid:
{"codes": ["A", "B", "A"]}  ❌ (duplicate "A")
```

Object uniqueness (by key field):

```
// Schema:
"users|[*] -> !": [
  {"id|#": "u1", "name": "Alice"}
]

// Valid:
{"users": [
  {"id": "u1", "name": "Alice"},
  {"id": "u2", "name": "Bob"}
]} ✅

// Invalid:
{"users": [
  {"id": "u1", "name": "Alice"},
  {"id": "u1", "name": "Charlie"}
]} ❌ (duplicate key "u1")
```

8.8 Regex Pattern Validation

Strings must match the specified pattern.

Valid:

```
// Schema: "code|~^[A-Z]{2}-\\d{4}$~": "AB-1234"
{"code": "XY-9999"}  ✅
```

Invalid:

```
{"code": "ab-1234"}  ❌ (lowercase letters)
{"code": "A-1234"}   ❌ (only one letter)
{"code": "AB-123"}   ❌ (only 3 digits)
```

8.9 Additional Properties

By default, unknown fields are **not allowed** unless `$additionalProperties: true` .

Schema:

```
{
  "$additionalProperties": false,
  "$oky": {
    "user": {
      "name|@": "Alice"
    }
  }
}
```

Valid:

```
{"user": {"name": "Bob"}} ✅
```

Invalid:

```
{"user": {"name": "Bob", "age": 30}} ❌ (unknown field 'age')
```

8.10 Validation Error Messages

Implementations should provide clear error messages including:

- Field path (e.g., `user.address.zipCode`)
- Constraint violated (e.g., "string length", "required field")
- Expected vs. actual value
- Constraint details (e.g., "expected length 5-10, got 3")

9. Complete Examples

9.1 Minimal Example

```
{
  "$oky": {
    "message|@ {1,100}": "Hello, Okyline!"
  }
}
```

Validates:

- Required string field named "message"
- Length between 1 and 100 characters

9.2 User Profile

```
{
  "$okylineVersion": "1.0",
  "$version": "1.0.0",
  "$title": "User Profile",
  "$description": "Schema for user account information",
  "$oky": {
    "user": {
      "id|@ (>0)|User identifier": 12345,
      "username|@ {3,20}|Unique username": "alice_dev",
      "email|@ ~$Email~|Email address": "alice@example.com",
      "firstName|@ {1,50}|First name": "Alice",
      "lastName|@ {1,50}|Last name": "Smith",
      "dateOfBirth|~$Date~|Date of birth": "1990-05-15",
      "isActive|@|Account status": true,
      "roles|[1,5] -> ('admin','user','guest')!|User roles": ["user"],
      "preferences|?|User preferences": {
        "theme|('light','dark')": "light",
        "language|('en','fr','es')": "en"
      }
    }
  }
}
```

9.3 E-commerce Order

```
  {
    "$version": "2.1.0",
    "$id": "E-ORDER-001",
    "$title": "Order Schema",
    "$description": "Schema for e-commerce orders",
    "$oky": {
      "order": {
        "orderId|@ #~$OrderId~|Unique order identifier": "ORD-12345678",
        "customerId|@ (>0)|Customer ID": 42,
        "orderDate|@ ~$DateTime~|Order timestamp": "2025-01-15T10:30:00Z",
        "status|@ ($ORDER_STATUS)|Order status": "PENDING",
        "items|@ [1,100] -> !|Order items": [
          {
            "sku|@ #~$Sku~|Product SKU": "SKU-ABC12345",
            "name|@ {2,200}|Product name": "Wireless Mouse",
            "quantity|@ (1..1000)|Quantity": 2,
            "vat|(0.05,0.1,0.15,0.2)":0.2,
            "unitPrice|@ (>0)|Unit price": 50.0,
            "netAmount|(%CheckNetAmount)":100.0,
            "grossAmount|(%CheckGrossAmount)":120.00
          }
        ],
        "shippingAddress|@|Shipping address": {
          "street|@ {5,100}": "123 Main Street",
          "city|@ {2,50}": "Paris",
          "postalCode|@ {5,10}": "75001",
          "country|@ {2}": "FR"
        },
        "paymentMethod|@ ($PAYMENT_METHOD)|Payment method": "CARD",
        "subTotal|(%CheckSubtotal)|Total net amount":100.0,
        "total|@ (>0)|Total gross amount": 120.00,
        "$requiredIf status('SHIPPED','DELIVERED')": ["trackingNumber"],
        "$appliedIf paymentMethod": {
          "('CARD')": {
            "cardLastFour|@ {4}|Last 4 digits": "1234"
          },
          "('PAYPAL')": {
            "paypalEmail|@ ~$Email~|PayPal email": "user@example.com"
          },
          "('BANK_TRANSFER')": {
            "bankReference|@ {10,50}|Bank reference": "REF1234567"
          }
        }
      }
    },
    "$nomenclature": {
      "ORDER_STATUS": "PENDING,CONFIRMED,SHIPPED,DELIVERED,CANCELLED",
      "PAYMENT_METHOD": "CARD,PAYPAL,BANK_TRANSFER"
    },
    "$format": {
      "OrderId": "^ORD-[0-9]{8}$",
      "Sku": "^SKU-[A-Z]{3}[0-9]{5}$"
    },
    "$compute": {
      "CheckNetAmount": "netAmount == round(unitPrice * quantity,2)",
      "CheckGrossAmount": "grossAmount == round(netAmount * (1 + vat),2)",
      "CheckSubtotal": "subTotal == sum(items,netAmount)",
      "CheckTotal": "total == sum(items,grossAmount)"
    }
  }
```

9.4 Garden Management

```
{
  "$okylineVersion": "1.0",
  "$version": "1.0.5",
  "$title": "Shared Vegetable Garden",
  "$description": "Schema for shared vegetable garden crop monitoring",
  "$nomenclature": {
    "VEGGIES": "Carrot,Tomato,Lettuce,Zucchini,Pepper,Cucumber",
    "METHODS": "Permaculture,Direct_sowing,Greenhouse,Container",
    "WEATHER": "SUNNY,CLOUDY,RAINY,WINDY"
  },
  "$format": {
    "SessionCode": "^[A-Z]{2}-\\d{4}$",
    "IsoDate": "^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\\d|3[01])$"
  },
  "$oky": {
    "gardener|@|Gardener information": {
      "id|@ #(>0)|Gardener identifier": 101,
      "name|@ {2,100}|Gardener name": "Julie Martin",
      "email|{6,100}~$Email~|Contact email": "julie@eco-garden.org",
      "status|@ ('ACTIVE','INACTIVE')|Membership status": "ACTIVE"
    },
    "session|@|Gardening session": {
      "code|@ #~$SessionCode~|Unique session code": "SR-0012",
      "date|@ ~$IsoDate~|Session date": "2025-04-10",
      "participantNames|[1,12] -> {2,50}|Participant names": ["Paul", "Léa", "Marc"],
      "surface|@ (1..10,>20)|Planted area in m²": 25,
      "weather|@ ($WEATHER)|Weather conditions": "SUNNY",
      "notes|?{0,500}|Session notes": "Great session with new participants",
      "plants|@ [1,10] -> !|Plants grown": [
        {
          "type|@ #($VEGGIES)|Vegetable type": "Carrot",
          "method|@ [1,2] -> ($METHODS)!|Growing methods": ["Permaculture"],
          "plantedAt|@ ~$IsoDate~|Planting date": "2025-03-20",
          "quantity|(1..1000)|Number of plants": 50
        },
        {
          "type|@ #($VEGGIES)|Vegetable type": "Tomato",
          "method|@ [1,2] -> ($METHODS)!|Growing methods": ["Greenhouse"],
          "plantedAt|@ ~$IsoDate~|Planting date": "2025-04-01",
          "quantity|(1..1000)|Number of plants": 30
        }
      ]
    }
  }
}
```

10. Appendix: Quick Reference

10.1 Constraint Cheat Sheet

Constraint	Applies To	Meaning	Example
`@`	All	Required field	`"name
`?`	All	Nullable field	`"middle
`#`	Scalars	Key field (for uniqueness)	`"id
`{...}`	String	Length constraint	`"name
`(...)`	Scalars	Value constraints	`"age
`~...~`	String	Regex/format	`"email
`[...]`	Array	Size constraint	`"tags
`->`	Array/Map	Element/value constraints	`"tags
`!`	Array	Uniqueness	`"codes
`[...:...]`	Object (map)	Map constraints	`"data
`%`	All	Default value (informational)	`"theme
`$oneOf`	Object/Array	Match exactly one	`"pay

Constraint	Applies To	Meaning	Example
`$anyOf`	Object/Array	Match at least one	`"notif

10.2 Built-in Formats

Format	Description	Example
`$Date`	ISO date (YYYY-MM-DD)	`"2025-05-30"`
`$DateTime`	ISO 8601 datetime	`"2025-05-30T14:30:00Z"`
`$Time`	Time (HH:mm or HH:mm:ss)	`"23:59:00"`
`$Email`	Email address	`"user@example.com"`
`$Uri`	URI with scheme	`"https://example.com"`
`$Ipv4`	IPv4 address	`"192.168.1.1"`
`$Ipv6`	IPv6 address	`"2001:db8::1"`
`$Uuid`	UUID v1-v5	`"f47ac10b-58cc-..."`
`$Hostname`	DNS hostname	`"api.example.com"`

10.3 Special Keys

Key	Location	Purpose
`$oky`	Root	**Required**. Contains the schema definition
`$okylineVersion`	Root	Okyline spec version (e.g., `"1.0"`)
`$version`	Root	Schema version
`$title`	Root	Schema title
`$description`	Root	Schema description
`$additionalProperties`	Root	Allow unknown fields (default: `false`)
`$nomenclature`	Root	Reusable value registries
`$format`	Root	Reusable regex patterns
`$compute`	Root	Computed expressions

10.4 Conditional Directives

Directive	Location	Purpose
`$requiredIf`	Object	Conditional required fields (if condition)
`$requiredIfNot`	Object	Required if condition NOT met
`$requiredIfExist`	Object	Required if another field exists
`$requiredIfNotExist`	Object	Required if field does NOT exist
`$forbiddenIf`	Object	Conditional forbidden fields (if condition)
`$forbiddenIfNot`	Object	Forbidden if condition NOT met
`$forbiddenIfExist`	Object	Forbidden if another field exists
`$forbiddenIfNotExist`	Object	Forbidden if field does NOT exist
`$appliedIf`	Object	Conditional structure (if/switch condition)
`$appliedIfExist`	Object	Conditional structure (if field exists)
`$appliedIfNotExist`	Object	Conditional structure (if field does NOT exists)

10.5 Type Inference Table

Example Value	Inferred Type
`"text"`	String
`42`	Integer
`3.14`	Number
`true` / `false`	Boolean
`["a", "b"]`	Array[String]
`[1, 2, 3]`	Array[Integer]
`{"key": "value"}`	Object

10.6 Common Patterns

Required email:

```
"email|@ ~$Email~": "user@example.com"
```

Optional phone number with pattern:

```
"phone|~^\\+[0-9]{10,15}$~": "+33612345678"
```

List of unique strings, 1-10 items, 2-20 chars each:

```
"tags|@ [1,10] -> {2,20}!": ["eco", "bio"]
```

Enum from nomenclature:

```
{
  "$oky": {
    "status|@ ($STATUS)": "ACTIVE"
  },
  "$nomenclature": {
    "STATUS": "ACTIVE,INACTIVE,PENDING"
  }
}
```

Age between 18 and 120:

```
"age|@ (18..120)": 30
```

Map with pattern-constrained keys:

```
"translations|[~^[a-z]{2}$~:10] -> {1,100}": {
  "en": "Hello",
  "fr": "Bonjour"
}
```

Polymorphic payment with oneOf:

```
"payment|@ $oneOf": [
  {"type|@ ('card')": "card", "number|@ {16}": "1234567812345678"},
  {"type|@ ('paypal')": "paypal", "email|@ ~$Email~": "user@example.com"}
]
```

Document Information

Specification Version: 1.0

Date: November 2025

Status: Draft

Changelog:

- **v1.0 (2025-11):** Initial specification release

Contributing:
Feedback and contributions to this specification are welcome. Please ensure any derivative works are shared under the same CC BY-SA 4.0 license.

Contact:
For questions or suggestions regarding this specification, please contact Akwatype at:
pierre-michel.bret@akwatype.io

End of Okyline Language Specification v1.0.0

Annex A — Conformance

This annex will be fully defined in the Final version of the specification. It will describe the conformance requirements for documents, validators, and producers.

Annex B — Terminology

This annex will be completed in the Final version. It will provide the formal definitions of key terms used throughout the specification.

Annex C — Okyline Expression Language (Normative)

Version: 1.0
Date: November 2025

Status: Draft

Relation to the Core Specification

This annex defines the **Okyline Expression Language**, used inside `$compute` blocks and conditional validation directives (such as `$requiredIf` , `$forbiddenIf` , and `$appliedIf`).

It specifies the **grammar**, **operators**, and **evaluation semantics** of expressions that may appear within an Okyline document.
Implementations claiming full Okyline 1.0.0 conformance **MUST** implement the rules described in this annex.

The Expression Language is an **integral component** of Okyline, not a separate language.
It extends the core validation model with computation and conditional logic while remaining **pure, deterministic, and side-effect-free**.

Although this annex is part of the Okyline 1.0.0 Draft specification,
the expression language defined here is considered stable for practical use.

Non-normative note

The Expression Language can be used independently of validation, for example in documentation tools or template generators,
but only the behaviors defined in this annex are considered part of the Okyline 1.0.0 normative model.

C.1 Overview

The Okyline Expression Language defines the grammar and semantics used in `$compute` blocks and computed constraints.

This annex is **normative**, meaning its contents define the expected behavior of conforming implementations of Okyline 1.0.

Expressions are pure, deterministic, and evaluated within the context of the object where they appear. All functions are **null-safe** and **side-effect free**.

C.2 Grammar (Simplified EBNF)

```
expression       ::= logical_or
logical_or       ::= logical_and ( "||" logical_and )*
logical_and      ::= equality ( "&&" equality )*
equality         ::= comparison ( ("==" | "!=") comparison )*
comparison       ::= addition ( (">" | "<" | ">=" | "<=") addition )*
null_coalescing  ::= addition ( "??" addition )*
addition         ::= multiplication ( ("+" | "-") multiplication )*
multiplication   ::= unary ( ("*" | "/") unary )*
unary            ::= ("!" | "-") unary | primary
primary          ::= literal | identifier | function_call | "(" expression ")"
function_call    ::= identifier "(" [ arguments ] ")"
arguments        ::= expression ( "," expression )*
literal          ::= number | string | boolean | null
```

C.3 Operators

Operator	Type	Description	Example	Null Behavior
??	Null coalescing	Returns left if non-null, otherwise right	price ?? 0 → 0	Short-circuits; high precedence
+	Arithmetic	Addition	2 + 3 → 5	Null propagates (returns null) — Exception: string concatenation treats null as ""
-	Arithmetic	Subtraction	5 - 2 → 3	Null propagates
*	Arithmetic	Multiplication	3 * 2 → 6	Null propagates
/	Arithmetic	Division	6 / 2 → 3.0	Null propagates; division by zero → null
< >= <=	Comparison	Numeric/lexicographic	age > 18	Returns null if either operand is null
== !=	Equality	Value equality	"A" == "A"	null == null → true; null == x → false
=== !==	Strict equality	IEEE-754 bit-exact	1.0 === 1.0	Strict comparison (NaN !== NaN)
&&,			Logical	Boolean conjunction/disjunction
!	Logical	Negation	!true → false	!null → true

Operator	Type	Description	Example	Null Behavior
? :	Ternary	Conditional	x > 10 ? "hi" : "lo"	Condition null → false

C.3.1 Null Coalescing Operator

Operator	Type	Description	Example
??	Null coalescing	Returns the left operand if it is non-null; otherwise returns the right operand (short-circuit evaluation).	`price ?? 0 → 0` (if price is null) `price ?? 100 → 100` (if price is null)

Precedence:
The `??` operator has **high precedence**, binding tighter than comparison operators but looser than arithmetic operators.

Short-circuit evaluation:
The right operand is **not evaluated** if the left operand is non-null.

Examples:

```
// If price is null, use default of 10
finalPrice = price ?? 10

// Chain multiple coalescings
value = a ?? b ?? c ?? 0  // First non-null value or 0

// In arithmetic context
total = (price ?? 0) * (quantity ?? 1)
```

C.4 Date Functions

Function	Description	Example
`date(dateString, pattern?)`	Parses a string into a LocalDate (default pattern `yyyy-MM-dd`).	`date("2024-03-15") → LocalDate(2024-03-15)`
`formatDate(date, pattern?)`	Formats a LocalDate to string (`yyyy-MM-dd` default).	`formatDate(date("2024-03-15"), "dd/MM/yy") → "15/03/24"`

Function	Description	Example
`today()`	Returns the current system date.	`today() → 2025-11-05`
`daysBetween(start, end)`	Number of days difference (`end - start`).	`daysBetween("2024-03-15","2024-03-18") → 3`
`plusDays(date, days)`	Adds days to a date.	`plusDays("2024-02-28",1) → "2024-02-29"`
`minusDays(date, days)`	Subtracts days.	`minusDays("2024-03-01",1) → "2024-02-29"`
`plusMonths(date, months)`	Adds months.	`plusMonths("2024-01-31",1) → "2024-02-29"`
`minusMonths(date, months)`	Subtracts months.	`minusMonths("2024-03-31",1) → "2024-02-29"`
`plusYears(date, years)`	Adds years.	`plusYears("2023-03-15",1) → "2024-03-15"`
`minusYears(date, years)`	Subtracts years.	`minusYears("2024-03-15",1) → "2023-03-15"`
`isWeekend(date)`	True if Saturday or Sunday.	`isWeekend("2024-03-16") → true`
`isLeapYear(date)`	True if leap year.	`isLeapYear("2024-03-15") → true`
`year(date)`	Extracts year.	`year("2024-03-15") → 2024`
`month(date)`	Extracts month.	`month("2024-03-15") → 3`
`day(date)`	Extracts day of month.	`day("2024-03-15") → 15`

C.5 String Functions

Function	Description	Example
`isNullOrEmpty(s)`	True if `s` is `null` or empty.	`isNullOrEmpty("") → true`
`isEmpty(s)`	True if string length is 0.	`isEmpty("") → true`

Function	Description	Example
`substring(s,start,len)`	Returns substring from start.	`substring("Hello",1,3) → "ell"`
`substringBefore(s,delim)`	Part before first occurrence.	`substringBefore("a:b:c",":") → "a"`
`substringAfter(s,delim)`	Part after first occurrence.	`substringAfter("a:b:c",":") → "b:c"`
`replace(s,target,repl)`	Replace all occurrences.	`replace("foo bar foo","foo","baz") → "baz bar baz"`
`trim(s)`	Removes leading/trailing spaces.	`trim(" hi ") → "hi"`
`length(s)`	String length (null → 0).	`length("hey") → 3`
`startsWith(s,prefix)`	Checks prefix.	`startsWith("hello","he") → true`
`endsWith(s,suffix)`	Checks suffix.	`endsWith("hello","lo") → true`
`contains(s,search)`	True if substring found.	`contains("banana","an") → true`
`toUpperCase(s)`	Converts to upper case.	`toUpperCase("Hi") → "HI"`
`toLowerCase(s)`	Converts to lower case.	`toLowerCase("Hi") → "hi"`
`capitalize(s)`	Uppercases first character.	`capitalize("hello") → "Hello"`
`decapitalize(s)`	Lowercases first character.	`decapitalize("Hello") → "hello"`
`padStart(s,len,ch)`	Left pads string.	`padStart("7",3,"0") → "007"`
`padEnd(s,len,ch)`	Right pads string.	`padEnd("7",3,"0") → "700"`
`repeat(times,ch)`	Repeats char.	`repeat(5,"*") → "*****"`
`indexOf(s,sub)`	First index of substring.	`indexOf("abracadabra","bra") → 1`
`indexOfLast(s,sub)`	Last index of	`indexOfLast("abracadabra","bra") → 8`

Function	Description	Example
	substring.	

C.5.1 String Index Handling

String functions use **defensive index handling** to prevent runtime errors and ensure deterministic behavior.

Rules

- Negative start indices are **clamped to 0**.
- Negative lengths are treated as **0** (the result is an empty string).
- A start index greater than the string length returns **an empty string**.
- End indices beyond the string length are **clamped to length()**.
- Functions **never throw exceptions** for out-of-bounds access.
- When applicable, functions return safe default values (e.g., empty string `""`, index `-1`, etc.).

Examples

```
substring("Hello", -10, 3)   → "Hel"    // negative start clamped to 0
substring("Hello", 1, -5)    → ""       // negative length treated as 0
substring("Hello", 100, 5)   → ""       // start beyond length → empty
substring("Hello", 1, 999)   → "ello"   // end clamped to string length
substring("Hello", 2, 0)     → ""       // zero length → empty
indexOf("abc", "z")          → -1       // not found
padStart("Hi", 1, "x")       → "Hi"     // already long enough (no truncation)
```

Notes

- The third parameter of `substring(s, start, length)` is interpreted as a **length**, not an end index.
- Negative lengths are **not** treated as “counting from the end” — they are simply clamped to zero.
- This ensures that all string operations remain **null-safe**, **exception-free**, and **consistent** across implementations.

C.6 Numeric Functions

Function	Description	Example
`abs(x)`	Absolute value.	`abs(-5) → 5`
`sqrt(x)`	Square root.	`sqrt(9) → 3.0`
`floor(x,scale?)`	Rounds down (optional decimals).	`floor(3.1415,2) → 3.14`

Function	Description	Example
`ceil(x,scale?)`	Rounds up.	`ceil(3.1415,2) → 3.15`
`round(x,scale?,mode?)`	Round with mode (`HALF_UP` default).	`round(3.5,0,"HALF_EVEN") → 4.0`
`mod(a,b)`	Remainder of division.	`mod(10,3) → 1`
`pow(base,exp)`	Power function.	`pow(2,3) → 8.0`
`log(x)`	Natural logarithm.	`log(2.71828) → 1.0`
`log10(x)`	Base-10 logarithm.	`log10(1000) → 3.0`
`random(min?,max?)`	Random integer in range.	`random(5,10) → 7`
`toInt(v)`	Converts to integer.	`toInt(3.7) → 4`
`toNum(v)`	Converts to number.	`toNum("42") → 42.0`

C.7 Aggregation & Utility Functions

Function	Description	Example
`sum(collection, expr)`	Sums values of expression evaluated for each element in collection.	`sum(lines, price) → 120.0` `sum(lines, %LineTotal) → 120.0`
`average(collection, expr)`	Average of expression evaluated for each element in collection.	`average(lines, quantity) → 5.4`
`min(collection, expr)`	Minimum value of expression in collection.	`min(scores, score) → 12`
`max(collection, expr)`	Maximum value of expression in collection.	`max(scores, score) → 99`
`count(collection)`	Count of non-null elements in collection.	`count(items) → 5`
`countAll(collection)`	Count of all elements in collection (including null).	`countAll(items) → 7`
`countIf(collection, expr)`	Count of elements where expression evaluates to true.	`countIf(users, active) → 3`

Note: The `%identifier` syntax is not a function but a **reference operator** for accessing compute expressions. See C.7.1 Compute Reference Syntax for details.

C.7.1 Compute Reference Syntax

Referencing Compute Expressions: `%identifier`

The `%identifier` syntax allows expressions to reference other computed expressions defined in the same `$compute` block.

Syntax:

```
%<identifier>
```

Where `<identifier>` is the name of a compute expression defined in the same `$compute` block.

Semantics:

- `%ComputeName` evaluates the expression named `ComputeName` from the current `$compute` block
- The `%` prefix distinguishes compute references from field accesses
- Circular dependencies are detected at schema load time and result in a validation error
- References are resolved in the context where the compute expression is evaluated

Examples:

```
{
  "$compute": {
    "BasePrice": "1000",
    "Tax": "%BasePrice * 0.2",
    "Total": "%BasePrice + %Tax"
  }
}
```

In this example:

- `Tax` references `BasePrice` using `%BasePrice`
- `Total` references both `BasePrice` and `Tax` using `%BasePrice` and `%Tax`
- All references are resolved when the expressions are evaluated

Aggregation Functions:

When using aggregation functions (such as `sum` , `map` , `filter`), the second parameter can be:

- A **field reference** (direct access to a field in each collection element)
- An **inline expression** (evaluated for each element)

- A **compute reference** (using `%identifier` to reference a compute expression)

```
{
  "$compute": {
    "LineTotal": "netAmount * (1 + vat)",
    "SubTotal": "sum(items, netAmount)",
    "TotalWithFees": "sum(items, netAmount * 1.2)",
    "InvoiceTotal": "sum(lines, %LineTotal)"
  }
}
```

Benefits:

- Clear visual distinction between field access (`fieldName`) and compute references (`%ComputeName`)
- No ambiguity in aggregation functions
- Consistent with validation constraint syntax (`field|(%ComputeName)`)
- Readable and concise expressions
- Reduces visual noise in complex compute chains

Constraints:

- `%` must be followed immediately by a valid identifier
- The referenced identifier must exist in the current `$compute` block
- Circular references result in a validation error at schema load time
- Compute references are resolved before evaluation begins

Implementation Note:

The parser transforms `%identifier` into an internal representation at parse time. The exact internal mechanism is implementation-defined, but the observable behavior must match the semantics described above.

C.8 Evaluation Rules

- Expressions are evaluated in the context of the enclosing object (fields accessible by name).
- Missing fields resolve to `null` .
- Type coercion **MUST NOT** be implicit except where explicitly defined by helper functions (`toNum` , `toStr` , etc.).
- Division by zero returns `null` .
- Comparisons with `null` evaluate to `false` unless explicitly tested.
- Implementations **MUST** guarantee deterministic results for the same input.
- Functions **MUST NOT** have side effects.
- Execution errors **SHOULD** return a structured error (`COMPUTE_ERROR` , `INVALID_ARGUMENT` , etc.).

C.9 Null Handling Rules

C.9.1 Arithmetic Operations (Null Propagation)

Arithmetic operations (`+` , `-` , `*` , `/`) follow **three-valued logic** (similar to SQL semantics):

- If any operand is null, the result is null (null propagates).
- **Exception:** String concatenation with `+` treats null as the empty string `""` .

Examples:

```
10 + null       → null
null * 5        → null
null / 2        → null
100 - null      → null

// String concatenation exception
"Hello" + null       → "Hello"
null + " World"      → " World"
```

C.9.2 Comparison Operations

Comparison operators (`<` , `<=` , `>` , `>=`) return **null** if either operand is null.

```
10 > null       → null
null <= 5       → null
null > null     → null
```

C.9.3 Equality Operations

Equality operators have specific null handling:

- `null == null` → `true`
- `null != null` → `false`
- `null == <any-value>` → `false`
- `null != <any-value>` → `true`

Note:
For numeric equality (`==`), values are compared with **implicit rounding to 6 decimal places** using **HALF_UP** rounding mode.

C.9.4 Logical Operations

Boolean operations require boolean operands:

- `null` is treated as `false` in boolean contexts.
- Non-boolean values are also treated as `false` .

```
null && true    → false
null || true    → true
!null           → true
```

C.9.5 Function Arguments

Functions handle null arguments according to their specific semantics:

- **String functions:** Most treat null as empty string `""` .
- **Numeric functions:** Most propagate null (return null if input is null).
- **Date functions:** Null dates typically return null.
- See each function's documentation for specific behavior.

C.9.6 Field Resolution

- Missing fields resolve to null.
- Accessing a field on a null value returns null (null-safe navigation).

```
user.address.city  → null  // if user or address is null
```

Rationale for Null Propagation

The null propagation semantics for arithmetic operations follow **SQL/database conventions**, where operations involving unknown values (null) produce unknown results (null).
This prevents silent calculation errors and makes null handling **explicit** via the `??` operator.

End of Annex C — Okyline Expression Language (Normative)