AlphaFlow DSL Specification (Optimized)

This document defines the structure and semantics of the AlphaFlow workflow DSL, which enables the visual or code-driven construction of workflow state machines. The DSL supports various state types, expression mapping, structured error handling, parallel execution, and custom plugins. It uses a declarative JSON or YAML format.

---

1. Top-Level Structure

```
{
  "StartAt": "StateName",
  "Comment": "Optional description",
  "Version": "1.0.0",
  "GlobalConfig": { ... },
  "ErrorHandling": { ... },
  "States": {
    "StateName": { ... }
  }
}
```

Fields

Field	Type	Required	Description
`StartAt`	string	✅	Name of the initial state (must exist in `States`)
`Comment`	string	❌	Human-readable description
`Version`	string	❌	DSL version string (default: 1.0.0)
`GlobalConfig`	object	❌	Global secrets, environment, and shared settings
`ErrorHandling`	object	❌	Global retry and catch behavior
`States`	object	✅	Dictionary of named states

---

2. State Types

Supported State Types:

- `Pass`: Output a constant or forwarded result
- `Task`: Execute a task (local Python, HTTP, Shell, etc.)
- `Wait`: Delay based on time or timestamp
- `Choice`: Conditional branching
- `Parallel`: Run branches concurrently
- `Map`: Iterate over a list and run a sub-workflow

- `Succeed`: Mark the flow as successfully terminated
- `Fail`: Mark the flow as failed with error info
- `Custom`: Plugin-defined extension logic

Required Structure

Each state must have:

- `Type`
- **Exactly one** of `Next` or `End: true`

Common Optional Fields

Field	Description
`Comment`	Optional description of the state
`InputExpr`	Expression to transform incoming input
`OutputExpr`	Expression to transform output
`Retry`	Retry policy array (see below)
`Catch`	Catch handler array (see below)

3. Error Handling

Retry Policy

```
{
  "ErrorEquals": ["TimeoutError", "TaskFailed"],
  "IntervalSeconds": 2,
  "BackoffRate": 2.0,
  "MaxAttempts": 5
}
```

Catch Policy

```
{
  "ErrorEquals": ["AnyError"],
  "Next": "RecoverState",
  "ResultPath": "$.error_info"
}
```

> `ErrorEquals` supports values such as: `TimeoutError`, `TaskFailed`, `HeartbeatTimeout`, `AnyError`.

4. Task State Example

```
{
  "Type": "Task",
  "Resource": "local_python:scripts/do_something.py",
  "Parameters": {
    "name": "Alice",
    "age": 30
  },
  "InputExpr": "$.input",
  "ResultExpr": "$.result",
  "OutputExpr": "$.processed",
  "ExecutionConfig": {
    "timeout": 5
  },
  "Next": "NextState"
}
```

> `Resource` must follow URI-style prefix, e.g. `local_python:`, `http_post:`

---

5. Control Flow States

Choice

```
{
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.value",
      "Operator": "==",
      "Value": 10,
      "Next": "PassA"
    },
    {
      "Variable": "$.value",
      "Operator": ">",
      "Value": 20,
      "Next": "PassB"
    }
  ],
  "Default": "FallbackState"
}
```

Parallel

```
{
  "Type": "Parallel",
  "Branches": [
    { "StartAt": "Step1", "States": { ... } },
    { "StartAt": "StepA", "States": { ... } }
  ],
  "Next": "JoinPoint"
}
```

Map

```
{
  "Type": "Map",
  "ItemsPath": "$.array",
  "Iterator": {
    "StartAt": "ProcessItem",
    "States": {
      "ProcessItem": { "Type": "Task", ... }
    }
  },
  "MaxConcurrency": 4,
  "Next": "AfterMap"
}
```

---

6. Custom State

```
{
  "Type": "Custom",
  "Resource": "plugin:my_custom_plugin",
  "CustomConfig": {
    "some_option": true
  },
  "End": true
}
```

> Custom states must use `plugin:` prefix in `Resource` and may contain `CustomConfig` for plugin parameters.

---

7. Expressions

- Expression fields: `InputExpr`, `OutputExpr`, `ResultExpr`
- Syntax: JSONPath (e.g. `$.key`) or custom plugin-based evaluation
- Used for transforming data between steps

8. Notes

- A valid workflow must have:
  - `StartAt` defined and matching a state name
  - All `Next` references must point to a valid state
  - At least one state marked with `"End": true`
- States must not be isolated (i.e., unreachable)
- The DSL supports both JSON and YAML formats

9. Validation

- Validated against JSON Schema (Draft-07)
- Schema ensures field-level and type constraints
- **Graph-level validation** (e.g., reference resolution, orphan detection) must be performed via semantic validator (code)

10. Future Enhancements

- Add `Metadata` field for UI display: icons, colors, labels
- Add `PluginType` to CustomState for structured plugin registry
- Support DSL imports, reusable flow templates
- Multi-language expression engine backend