TaskGraph & DAG DSL — Full User Manual

1. Introduction
TaskGraph is a lightweight, asynchronous Directed Acyclic Graph (DAG) execution engine paired with a Gro

This manual explains:
- The architecture of TaskGraph
- The DSL and how to use it
- Execution model
- Forking, joining, conditional routing
- Error handling and events
- Best practices and examples

---

2. System Architecture

2.1 High-Level Layers

```
DSL Layer ■■■ TaskGraph Model ■■■ Scheduler ■■■ Promise Layer
```

Layer	Purpose
DSL Layer	Parses user-provided Groovy DSL into tasks, dependencies, forks, joins.
TaskGraph Model	Stores tasks, their relationships, and global context.
Scheduler	Executes tasks when their prerequisites are complete.
Promise Layer	Provides async semantics and result propagation.

2.2 Components Overview

Task
A unit of work. Each task:
- Runs asynchronously
- Returns a Promise
- Declares predecessor tasks
- Emits lifecycle events

ServiceTask
A task whose execution logic is defined via a user closure:
```groovy
action { ctx, prevOpt -> ... }
```

Join Task
A task that merges multiple predecessor results.

Fork Task
A synthetically generated task that performs conditional or dynamic branching.

TaskGraph
The container that holds all tasks and the execution context.

TaskContext
Shared environment with:
- Global variables
- Config
- Worker pool
- Runtime result map

Promise Layer
Provides the async abstraction. Ensures values or DataflowVariables get wrapped consistently into Promis

---

3. Execution Model

The `TaskGraph.run()` method drives execution.
3.1 States

Each task transitions through these states:
- **PENDING**
- **SCHEDULED**
- **START**
- **SUCCESS**
- **ERROR**
- **SKIPPED**
- **COMPLETED**

3.2 Readiness Check
A task is ready to run when:

```
t.predecessors are all COMPLETED, SKIPPED, or FAILED
```

3.3 Execution Loop
1. Find all ready tasks
2. Execute them
3. Track their Promises
4. When Promises complete, re-evaluate readiness
5. Stop when no tasks remain

The final Promise returned from `TaskGraph.run()` resolves to the result of the last declared task.

---

4. DSL Overview

4.1 Creating a Graph

```groovy
def graph = TaskGraph.build {
    globals {
        apiUrl = "https://example.com"
    }

    serviceTask("loadUser") {
        action { ctx, _ ->
            http.get("${ctx.globals.apiUrl}/user/123")
        }
    }
}
```

---

5. Defining Tasks

5.1 `serviceTask()`

```groovy
serviceTask("getData") {
    dependsOn "init"
    maxRetries = 3

    action { ctx, prevOpt ->
        // Must return a Promise or a value (auto-wrapped)
        remoteService.fetch()
    }
}
```

`prevOpt` is:
- `Optional.empty()` for tasks with no predecessors
- Contains the Promise of the single predecessor for simple tasks

---

6. Forks

6.1 Basic Fork

```groovy
fork("fanOut") {
    from "loadUser"
    to "loadOrders", "loadInvoices"
}
```

6.2 Conditional Fork

```groovy
fork("premiumUserFork") {
    from "loadUser"

    conditionalOn(["loadInvoices"]) { user ->
        user.isPremium
    }
}
```

6.3 Dynamic Routing

```groovy
fork("smartRoute") {
    from "analyze"

    route { result ->
        if (result.type == "A") return ["taskA"]
        if (result.type == "B") return ["taskB", "taskC"]
        return []
    }
}
```

6.4 Implementation Notes

A synthetic routing task is created:

- Depends on the `from` task
- Evaluates routing logic
- Marks unselected tasks as `SKIPPED`
- Ensures selected tasks depend on the router task

---

7. Joins

A join merges multiple predecessor results.

7.1 Example

```groovy
join("mergeResults") {
    from "loadOrders", "loadInvoices"

    action { ctx, promises ->
        def orders = promises[0].get()
        def invoices = promises[1].get()
        return merge(orders, invoices)
    }
}
```

7.2 Implementation

`JoinDsl.build()`:
- Marks the serviceTask as a join
- Wires predecessors
- Installs wrapper action collecting predecessor promises
- Ensures returned value becomes a Promise

---
8. Globals and Context

```groovy
globals {
    apiKey = "123"
    timeout = 5_000
}
```

Accessible in any task:

```groovy
ctx.globals.timeout
```

---

9. Event System

Attach event listeners:

```groovy
onTaskEvent { ev ->
    println "${ev.taskId}: ${ev.type} at ${ev.timestamp}"
}
```

Types:
- START
- SUCCESS
- ERROR
- SKIP

---

10. Error Handling

10.1 Task Errors

If a task throws or returns a failed Promise:
- Task state becomes ERROR
- Downstream tasks still run unless designed otherwise

10.2 Join Task Error Handling

If a predecessor fails:
- The predecessor's Promise contains the failure
- Join closure decides how to react

---

11. Promises & Value Normalization

Task closures may return:
- A Promise ⇒ used as-is
- A DataflowVariable ⇒ auto-wrapped
- A raw value ⇒ wrapped into a Promise
- null ⇒ wrapped as a failed Promise

All normalization goes through:

```
Promises.ensurePromise(x)
```

This prevents leaking internal DFV types or causing Groovy coercion issues.

---

12. Best Practices

Use explicit IDs
IDs define graph topology; name them clearly.
Avoid doing work in DSL mutators

`from()`, `to()`, `action()` should only collect configuration.

Prefer returning values or Promises
The system wraps values automatically.

Use event listeners for monitoring
Helps with debugging and orchestration visibility.

---

13. Example: Full Workflow

```groovy
def g = TaskGraph.build {

    globals {
        baseUrl = "https://api.example.com"
    }

    serviceTask("loadUser") {
        action { ctx, _ ->
            http.get("${ctx.globals.baseUrl}/user/1")
        }
    }

    fork("premiumRouting") {
        from "loadUser"
        conditionalOn(["loadInvoices"]) { user -> user.isPremium }
        to "loadOrders"
    }

    serviceTask("loadOrders") {
        dependsOn "premiumRouting"
        action { ctx, prev ->
            orderService.fetch(prev.get().id)
        }
    }

    serviceTask("loadInvoices") {
        dependsOn "premiumRouting"
        action { ctx, prev ->
            invoiceService.fetch(prev.get().id)
        }
    }

    join("combineData") {
        from "loadOrders", "loadInvoices"
        action { ctx, promises ->
            def orders = promises[0].get()
            def invoices = promises[1].get()
            return summarize(orders, invoices)
        }
    }
}

def finalPromise = g.run()
```

---

14. Conclusion

TaskGraph provides:
- A flexible Groovy DSL
- Dynamic routing and joining
- Promise-based asynchronous execution
- Pluggable concurrency backend
- Lightweight, readable workflow construction

This architecture supports simple tasks up to sophisticated multi-branch DAG workflows.