# JARVIS PLANNER — ARCHITECTURAL BLUEPRINT

**Task Graph · DSL · Execution Engine · Scoring · Memory Integration**

## 0. Purpose of This Document

This document defines the planner subsystem of the Jarvis backend. It specifies:

- How goals become executable plans
- How plans are represented
- How they are executed
- How decisions are made between alternatives
- How the system improves over time

This is architecture, not implementation.
Any implementation that violates this document is incorrect by design.

## 1. Planner Role in the System

The planner is responsible for converting high-level user intent into a structured, multi-step execution plan.

Key constraints:

- Plans must be explainable
- Plans must be recoverable
- Plans must be auditable
- Plans must be learnable

The planner does not:

- Execute OS actions
- Contain controller logic
- Perform UI automation.

It emits task graphs, nothing else

## 2. Task Graph Model

### 2.1 Core Concept

A task graph is a directed graph where:

- Nodes = tasks
- Edges = execution dependencies
- Entry node = starting point
- Terminal nodes = completion or abort

Graphs are:

- Mostly acyclic
- Allowed limited loops (polling, retries)
- Serializable and replayable

Linear scripts are explicitly forbidden.

### 2.2 Task Node Schema (Canonical)

Every task node MUST conform to the following conceptual schema:

```
{
    "task_id": "string",
     "type": "action | decision | composite | loop",
     "description": "human-readable intent",
     "inputs": {},
     "preconditions": [],
     "postconditions": [],
     "on_failure": "retry | skip | replan | abort",
     "retries": 0,
     "risk": "low | medium | high",
     "controller_action": "string | null"
}
```

If a task cannot be expressed in this form, it does not belong in the planner.

## 3. Task Types

### 3.1 Action Task

Purpose: Perform one atomic controller action.

Constraints:

- Exactly one controller call
- No branching
- No logic
- No side effects beyond declared postconditions

Example:

- open_folder
- shutdown
- open_app

### 3.2 Decision Task

Purpose: Branch execution based on evaluated state.

Properties:

- Contains a boolean or multi-way condition
- Selects next task by edge, not code

Example:
“Is VS Code already running?”
“Is workspace already prepared?”

### 3.3 Composite Task

Purpose: Encapsulate reusable subgraphs.

Properties:

- Expands into a subgraph
- Can be versioned
- Can be learned and reused

Composite tasks are how “skills” emerge.

### 3.4 Loop Task

Purpose: Controlled repetition.

Allowed use cases:

- Polling for system state
- Bounded retries with delay

Unbounded loops are forbidden.

## 4. Planner DSL (Domain-Specific Language)

### 4.1 DSL Design Principles

The DSL must be:

- Declarative
- Deterministic
- Serializable
- Controller-agnostic
- Human-auditable

The DSL must not:

- Embed procedural logic
- Execute code
- Reference runtime objects

### 4.2 DSL Top-Level Structure

```
task_graph:
  name: prepare_work_environment
  version: "1.0"
  entry: check_workspace
```

### 4.3 Task Definition Example

```
tasks:
    check_workspace:
        type: decision
        condition: workspace_ready
        on_true: done
        on_false: open_project_folder

    open_project_folder:
        type: action
        controller: open_folder
        args:  path: "~/projects/jarvis"
        on_success: launch_vscode
        on_failure: abort

    launch_vscode:
        type: action
        controller: launch_app
        args: app: vscode path: "~/projects/jarvis"
        retries: 1
        risk: low
        on_success: done

    done:
        type: action
        controller: notify
        args: message: "Workspace ready"
```

This DSL is intentionally boring.  Boring is stable. Stable is powerful.

# 5. Execution Engine State Machine

## 5.1 Purpose

The execution engine runs task graphs using a strict state machine.

This allows:

- Pause / resume
- Recovery
- Partial completion
- Deterministic failure handling

## 5.2 Task Lifecycle States

Each task moves through:

1. **PENDING :** Preconditions not yet evaluated or dependencies incomplete
2. **READY** : Preconditions satisfied, eligible to run
3. **RUNNING** : Controller action in progress
4. **SUCCEEDED** : Postconditions verified
5. **FAILED** : Action failed or postconditions unmet
6. **ABORTED :** Execution stopped by policy or user

State transitions are logged and immutable.

## 5.3 Failure Handling Rules

On failure, the engine must consult the task's declared policy:

**retry** → retry up to N times
**skip** → mark failed, continue graph
**replan** → return control to planner
**abort** → terminate execution immediately

Implicit behavior is forbidden.

# 6. Planner Scoring & Graph Selection

## 6.1 Why Scoring Exists

Multiple graphs may satisfy the same goal. Random selection is unacceptable. The planner must choose the best plan based on evidence.

## 6.2 Scoring Criteria

Each candidate graph is scored using weighted factors:

- Historical success rate
- Aggregate risk
- User preferences
- Execution efficiency
- Confirmation requirements

## 6.3 Example Scoring Formula

***score = (success_rate × 0.4) + ((1 - risk) × 0.3) + (user_preference × 0.2) + (efficiency × 0.1).***

Weights are tunable but must be explicit.

## 6.4 Selection Rules

- Highest score wins
- Ties broken by lower risk
- Unsafe graphs are discarded before scoring

# 7. Memory Integration

## 7.1 What Gets Stored

After execution, the following is persisted:

```
{
    "graph_name":
    "prepare_work_environment",
    "version": "1.0",
    "success": true,
    "failed_tasks": [],
    "execution_time": 11.2,
    "user_interrupted": false,
    "confidence": 0.92
}
```

## 7.2 How Memory Is Used

Memory biases future planning by:

- Preferring successful graphs
- Penalizing failed paths
- Reducing confirmations for trusted flows
- Adapting to user interruption patterns

This is learning, not guessing.

# 8. Invariants (Non-Negotiable Rules)

- No direct controller calls outside task graphs
- No logic embedded in execution engine
- No planner output that cannot be serialized
- No execution without postcondition verification
- No learning without stored evidence

Violating these creates a fake agent.