

Declarative Formats for DAG/Series-Parallel Workflows

Workflows that are **directed acyclic graphs** (DAGs) – especially those with a **series-parallel (SP)** structure – can be described in various human-readable file formats. In a series-parallel graph, complex flows are composed from simple *sequential* and *parallel* combinations ¹ ². This structured approach maps well to many workflow description languages. Below are some of the most relevant declarative formats for representing such workflows, all designed to be machine-readable while still understandable by humans:

YAML/JSON-Based Workflow Specifications

- **Argo Workflows (YAML)** – Argo is a Kubernetes-native workflow engine that lets you define multi-step processes in a YAML file. It supports both sequential steps and DAG mode with explicit dependencies. For example, one can declare tasks A, B, C, D with dependencies (A has none, B and C depend on A, D depends on B and C) – Argo runs A, then B and C in parallel, then D after both complete ³ ⁴. The YAML syntax uses a list of tasks with fields like `dependencies` to capture the DAG structure ⁴. This approach is fully declarative and allows maximum parallelism in execution ⁵.
- **Common Workflow Language (CWL)** – CWL is an open standard for describing workflows as YAML or JSON documents. It was designed to be declarative and portable across platforms ⁶ ⁷. In CWL, each step (tool invocation) declares its inputs and outputs, and a workflow connects outputs to inputs forming a DAG of task dependencies ⁸. Independent steps run concurrently by default, yielding a parallelizable DAG. **CWL workflows are written in YAML** and explicitly describe how data flows between steps ⁷. This makes CWL both machine-readable and reasonably human-readable (though complex workflows can become verbose).
- **CNCF Serverless Workflow (JSON/YAML)** – The *Serverless Workflow* DSL is a vendor-neutral, community-driven language for defining workflows that can be expressed in JSON or YAML ⁹. It's intended to be intuitive for developers, with constructs for sequences, parallel branches, event triggers, etc. For example, it has a `fork` element to model parallel branches (with a parallel "join" when all branches complete) ¹⁰ ¹¹. Serverless Workflow emphasizes readability and uses a text-based DSL syntax for flow control (under the hood it has a JSON schema for validation) ⁹. Being a modern standard (1.0 in CNCF), it integrates concepts from cloud orchestrators like AWS Step Functions while remaining platform-agnostic.
- **AWS Step Functions – Amazon States Language (JSON)** – Amazon's Step Functions uses a declarative JSON format called the *States Language* to define state machines (which, without cycles, are essentially DAG workflows). The States Language is a structured JSON syntax specifying states (tasks, choices, parallel, etc.) and their transitions ¹². For example, you define a JSON object with a `States` map, `StartAt` entry, and each state can have a `Next` or branching conditions. It supports a **Parallel** state to run branches concurrently and then join ¹³. While JSON is not as easy on the eyes as YAML, it is machine-friendly and the format is

standardized ¹². Many teams use this for serverless orchestration on AWS, writing `.asl.json` files that describe the workflow logic.

- **GitHub Actions and CI Pipelines (YAML)** – Even CI/CD workflow systems use declarative DAG definitions. For instance, **GitHub Actions** requires a YAML file (in the `.github/workflows` folder) to define jobs and their order. Each job can list a `needs: [JobX, JobY]` to declare dependencies on other jobs, effectively forming a DAG of jobs ¹⁴. The workflow is triggered by events and runs jobs in the specified order (jobs without dependencies run in parallel). Similar YAML-based pipeline configs exist for GitLab CI, Jenkins (with declarative Jenkinsfile syntax), Azure Pipelines, etc. These are human-editable and make it easy to visualize and modify the task graph for software build/test/deploy workflows.

Domain-Specific Workflow Languages (Human-Readable DSLs)

- **Workflow Description Language (WDL)** – WDL is an open standard with a *human-readable and writable syntax* for specifying data processing workflows ¹⁵. Unlike pure JSON/YAML, WDL looks like a simple scripting language, making it easy for humans to author. It allows you to define tasks (with commands, inputs, outputs) and then define a workflow that calls those tasks and links outputs to inputs. WDL natively supports **scatter-gather** (i.e. mapping a task over a collection in parallel) and conditional execution as first-class constructs ¹⁵. This means you can express series-parallel patterns in a structured way (e.g. scatter a task to run in parallel, then join results). WDL is designed for portability across platforms (HPC or cloud runners) and trades off a bit of machine readability for improved human readability and writability ¹⁵.
- **Nextflow DSL** – (Not an official standard like WDL/CWL, but widely used in genomics) Nextflow uses a **Groovy-based DSL** where workflows are written as code defining processes and their input/output channels. It implicitly creates a DAG of tasks via these dataflow channels. While not a simple config file, it is declarative in that the user defines what processes exist and how data flows, and the engine handles execution order. Nextflow's syntax is fairly human-readable (if one is comfortable with code) but it's less of a pure data format. (Nextflow is mentioned here as it's popular for parallel pipelines, though it's more script than config.)
- **BPMN 2.0 (XML)** – *Business Process Model and Notation* is a well-established standard for workflow/process modeling. BPMN primarily provides a graphical diagramming notation, but those diagrams are serialized in an XML format. In BPMN XML, one defines elements like `<parallelGateway>` (AND-split and AND-join) for parallel branches, `<sequenceFlow>` for directed connections, etc. A BPMN process with only parallel splits/joins and no loops essentially forms a series-parallel DAG. However, the XML is verbose and geared toward tools – not very hand-editable. BPMN shines as a visual and *business-friendly* representation, whereas JSON/YAML DSLs are often more developer-friendly ⁹. Still, BPMN is a **declarative graph definition** (the diagram *is* the workflow logic) and there are open-source engines that execute BPMN files.
- **Graphviz DOT Language** – DOT is a plain-text graph description language commonly used to draw graphs. It's not a workflow engine format per se, but it's human-readable and often used to represent DAG structures visually or for interchange. A DOT file can describe directed graphs (using `digraph { A -> B; ... }` syntax). In fact, DOT was historically an acronym for “**DAG of Tomorrow**,” originally designed for representing DAGs ¹⁶. While DOT is great for visualization and is simple to write, it does not have native concepts of “task” or “execution” – it just represents nodes and edges. You could use DOT to specify the graph structure and then have a Rust

program interpret it, but more specialized workflow formats (like the ones above) usually include task definitions, parameters, etc., which DOT does not.

Choosing a Format

Each format has its trade-offs: - **YAML/JSON configs** (Argo, CWL, Serverless Workflow, CI pipelines) are straightforward for machines and familiar to developers. They integrate well with tooling (JSON schema validation, etc.) and are easy to version control. YAML tends to be more readable with support for comments, whereas pure JSON (as in AWS Step Functions) can be harder for humans but is unambiguous for machines ¹⁷ ¹⁸. - **Purpose-built DSLs** (WDL, Nextflow, etc.) aim for a balance favoring human authorship, often at the cost of requiring a custom parser or runtime. These can express common parallel patterns in a concise, structured way (e.g. `scatter { ... }` blocks in WDL correspond to parallel execution). - **Graph description languages** (DOT, BPMN XML) are standardized and tool-supported; they excel at visualization and interchange. However, they may require additional conventions or tooling to actually execute tasks from the graph (BPMN engines exist; DOT would need a custom interpreter in your Rust orchestrator).

In the context of **Rust** orchestration for AI agent tasks, a popular choice is to use a structured data format (JSON, YAML, Toml) and leverage Serde for parsing into Rust types (as you've planned with a `WorkflowConfig`). For example, the **Wanguard** project (which you appear to be working on) currently uses a **TOML** schema for workflows – listing tasks under branches, with concurrency settings ¹⁹ ²⁰. TOML, like YAML, is quite human-readable. Many modern workflow engines, however, gravitate toward YAML, since it handles complex nested structures and is widely recognized for “Infrastructure as Code” use cases ²¹ ¹⁷.

Given that your target graphs are series-parallel, you might also consider a format that naturally represents nested parallel and sequential blocks. For instance, you could design a YAML/TOML structure where **sequence** and **parallel** are explicit keys that contain lists of steps (similar to how some CI systems allow grouping jobs). This would enforce an SP topology by construction. If you prefer an existing standard, **CWL** or **WDL** are instructive – they show how to represent DAG edges via naming and references (CWL links outputs to inputs, WDL uses variables), rather than an explicit “depends on” list, which can make the file easier to read. On the other hand, an explicit **dependencies list** (as in Argo or GitHub Actions) is straightforward and flexible (any DAG can be expressed, though you'd have to ensure it stays series-parallel if that's a requirement).

In summary, widely-used declarative formats for DAG/SP workflows include **YAML-based configs** (Argo Workflows, CI pipelines, Kestra, etc.), **JSON DSLs** (AWS Step Functions, CNCF Serverless Workflow), and **specialized workflow languages** like CWL and WDL which emphasize human readability. Each provides a way to describe tasks and their relationships (sequential or parallel) in a text file. By studying these, you can determine the syntax that best fits your use-case – whether it's a simple YAML with task IDs and dependency lists ⁴, a more semantic DSL with block structures, or even an adaptation of an existing standard for your Rust engine. All are machine-readable and human-comprehensible, so it comes down to the level of complexity vs. clarity you need for representing your AI-agent task graph.

Sources:

- Argo Workflows documentation – DAG mode example (YAML) ³ ⁴
- Common Workflow Language v1.0 – declarative workflow spec (YAML/JSON) ⁸ ⁷
- OpenWDL (Workflow Description Language) – human-readable workflow DSL (WDL) ¹⁵
- CNCF Serverless Workflow vs BPMN – comparison of JSON/YAML DSL vs BPMN XML ⁹

- AWS Step Functions – States Language (JSON) description ¹²
- GitHub Actions Workflow Syntax – YAML with job dependencies (`needs`) ¹⁴
- Graphviz DOT language – plain text graph description (originated for DAGs) ¹⁶
- Series-Parallel graph – defined via recursive series/parallel composition ¹ ²
- Kestra Orchestration – YAML workflow config benefits (human & machine readable) ¹⁷ ¹⁸
- Vanguard project docs – example `.wgr/workflow.toml` (parallel tasks list) ¹⁹ ²⁰

¹ ² **Series-parallel graph - Wikipedia**

https://en.wikipedia.org/wiki/Series%E2%80%93parallel_graph

³ ⁴ ⁵ **DAG - Argo Workflows - The workflow engine for Kubernetes**

<https://argo-workflows.readthedocs.io/en/latest/walk-through/dag/>

⁶ ⁷ **Introduction to Workflows with Common Workflow Language**

<https://carpentries-incubator.github.io/cwl-novice-tutorial/aio/index.html>

⁸ **Common Workflow Language (CWL) Workflow Description, v1.0.2**

<https://www.commonwl.org/v1.0/Workflow.html>

⁹ **Serverless Workflow vs BPMN - Automatiko Blog**

<https://blog.automatiko.io/2022/05/15/serverless-vs-bpmn.html>

¹⁰ ¹¹ **Serverless Workflow**

<https://serverlessworkflow.io/>

¹² **Using Amazon States Language to define Step Functions workflows - AWS Step Functions**

<https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html>

¹³ **Parallel workflow state - AWS Step Functions**

<https://docs.aws.amazon.com/step-functions/latest/dg/state-parallel.html>

¹⁴ **Workflow syntax for GitHub Actions - GitHub Docs**

<https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax>

¹⁵ **OpenWDL**

<https://openwdl.org/>

¹⁶ **DOT (graph description language) - Wikipedia**

[https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language))

¹⁷ ¹⁸ ²¹ **Declarative Orchestration with Kestra**

<https://kestra.io/features/declarative-data-orchestration>

¹⁹ ²⁰ **sprint-1.md**

<https://github.com/scalar0/wanguard/blob/1177e18a287780cc3def2c68f22bbbeb66312008/.github/sprints/sprint-1.md>