Universal Automation Wiki

Introducing WorkSpec

UAW syntax is now formalized into one clear specification language for describing work, validating logic early, and sharing simulations across tools.

AI-native | Human-readable | Validation-first | Platform-agnostic

What is WorkSpec?

A JSON-based language for describing how real-world work is set up, scheduled, and changed, in a way that both people and machines can read.

Objects

The people, equipment, ingredients, products, and services in your simulation. A bakery simulation might include a head baker, an oven, 10kg of flour, and a proofing timer.

Tasks

Each step of work: when it starts, how long it takes, and what must finish first. "Mix dough" might start at 08:00, take 15 minutes, and need "weigh ingredients" to be done.

Interactions

How tasks change things. "Mix dough" uses 2kg of flour (quantity goes down) and changes the mixer from "clean" to "dirty." Every side effect is written out explicitly.

The core idea: WorkSpec lets you catch process problems while you're still writing the spec, not after something breaks in production. Files use the `.workspec.json` extension, are portable across environments, and follow one canonical syntax.

Why formalize now?

Three problems would show up in real usage.

1 Everyone wrote it differently

Different teams could describe the same things using different field names and structures. Reusing someone else's template could be risky because you couldn't be sure it would work the same way. Reviewing specs across teams was slow.

2 Errors showed up too late

Broken references and timing conflicts would survive editing and only surface when someone tried to run the simulation. By then, the fix was never obvious.

3 Validation only worked in one place

Checking a simulation for problems was tied to a single UI. You couldn't run the same checks in a CI pipeline, a code editor, or a terminal, which made it hard for larger teams to adopt.

One contract
same structure for everyone

Earlier feedback
catch errors at authoring time

Wider access
CLI, npm package, and tooling

Lower friction
clear path from legacy syntax

Design goals

Make work models clear, reliable, and easy to move between tools.

📖 Readable

Clear field names, no hidden assumptions. Anyone on a team can open a simulation and understand what it describes without needing tribal knowledge or extra docs.

✅ Predictable

When validation catches a problem, it tells you what's wrong, where it is, and how to fix it. Same rules, same format, every time.

🔌 Portable

Works in the terminal, inside editors, in CI, and in browser UIs. Schema-backed for tooling support. Not locked to a single platform.

🏗️ Deep enough

Handles time, physical space, economics, and production recipes. Supports custom type extensions for specialized domains.

Why WorkSpec is AI-native

Designed so AI can generate, validate, and fix specs with less room for misunderstanding.

WorkSpec uses consistent field names, one canonical syntax (no competing ways to say the same thing), and semantic aliases that help with fuzzy matching in prompts. When something goes wrong, errors come back as structured data, so an AI can read the problem and fix it automatically.

1. Prompt generates a draft spec
2. Validator returns structured errors
3. AI reads errors and applies fixes
4. Clean simulation enters the workflow

> This loop is where "AI-native" becomes practical. The AI doesn't need to guess what went wrong. The validator tells it exactly, in a format it can parse and act on.

Architecture: world + process

Every WorkSpec document splits into two halves: what exists, and what happens.

At the top level, a `.workspec.json` file includes a `$schema` reference for editor tooling, a `schema_version`, and optional `meta` and `config` blocks. The real content lives in two sections:

🌍 world

Everything that exists before work begins. This is where you define your **layout** (the physical or digital space: rooms, stations, areas with coordinates) and your **objects** (actors, equipment, resources, products, and services).

Example: a bakery's world might define a prep area, an oven station, and a storage room, plus a head baker, a stand mixer, 10kg of flour, and a proofing timer service.

⚙️ process

Everything that happens during execution. Contains **tasks** (each step of work, with scheduling, dependencies, and interactions) and optional **recipes** (expected inputs and outputs for each product).

Example: "mix_dough" starts at 08:00, takes 15 minutes, is performed by the head baker, consumes flour, and changes the mixer state to dirty.

Objects, types, and traits

Every object has an ID, a type, and a name. Types come with "traits" that control what you can do with them.

What an object looks like

```
{
  "id": "head_baker",
  "type": "actor",
  "name": "Head Baker",
  "location": "prep_area",
  "properties": {
    "state": "idle",
    "skill_level": "senior"
  }
}
```

Every object gets: id, type, name. Optional: emoji, location, and a properties bag.

Type traits (what objects can do)

- **performer** – can be assigned to do tasks (used in `actor_id`)
- **stateful** – has discrete states with `from/to` transitions
- **quantifiable** – has a numeric quantity, supports `delta` changes

Core types: `actor`, `equipment`, `resource`, `product`, `service`. Digital types like `display` and `screen_element` are also supported.

New in WorkSpec: the `service` type. Services represent automated or background processes, like timers, polling loops, or scheduled checks. They can perform tasks without a human being assigned (e.g. `"actor_id": "proofing_timer"`), and track their own `state`, `interval`, and `last_run`.

Task model

Tasks define each step of work: when it starts, how long it takes, who does it, and what needs to finish first.

Time and duration

Start times can be written as `HH:MM`, `HH:MM:SS`, or full ISO timestamps. For workflows that span more than one day, you can use a day + time object. Durations accept plain numbers (minutes), ISO strings like `"PT30M"`, or shorthand like `"30m"` and `"1h"`.

Task metadata

Each task can carry priority, tags, a location, and interaction definitions. Everything needed to schedule, filter, and trace what happens during execution.

Dependency logic

Tasks can wait for other tasks to finish before they start. There are a few ways to express this:

Array
`["a", "b"]` – wait for all (implicit AND)

`all`
Wait for every listed task to complete

`any`
Wait for the first one to complete

Combined
Mix both: all of `all`, plus any of `any`

Interaction system

Interactions describe exactly what a task changes: stock levels, object states, or the existence of objects themselves.

Property operators

Each interaction targets an object and applies one or more changes:

`from/to`	state transition	`delta`	numeric change
`set`	direct assignment	`multiply`	proportional update
`append`	add to array	`remove`	remove from array
`increment`	+1 shorthand	`decrement`	-1 shorthand

Lifecycle controls

Beyond changing properties, interactions can **create** new objects during execution (a task that produces a batch of bread) or **delete** objects from active state (a task that disposes of waste).

If a change should only last for the duration of a task, like reserving a piece of equipment, mark it as `temporary: true` and it will revert automatically when the task ends.

Every side effect is written out explicitly. No hidden mutations, no guessing what changed after a task runs.

Before and after

The same task, mixing dough, expressed in legacy syntax and in WorkSpec.

LEGACY

```
{
  "id": "mix_dough 🔶 🔪",
  "actor_id": "baker",
  "consumes": {
    "flour": 2
  },
  "equipment_interactions": [
    {
      "id": "mixer",
      "from_state": "clean",
      "to_state": "dirty"
    }
  ]
}
```

WORKSPEC

```
{
  "id": "mix_dough",
  "emoji": "🔪",
  "actor_id": "baker",
  "interactions": [
    {
      "target_id": "flour",
      "property_changes": {
        "quantity": {
          "delta": -2
        }
      }
    },
    {
      "target_id": "mixer",
      "property_changes": {
        "state": {
          "from": "clean",
          "to": "dirty"
        }
      }
    }
  ]
}
```

Validation model

Errors are structured so both people and tools can act on them quickly.

Problem output

```
{
  "type": ".../errors/invalid-time-format",
  "title": "Invalid Time Format",
  "severity": "error",
  "detail": "Task has invalid start time.",
  "instance": "/simulation/process/tasks/2/start",
  "metric_id": "temporal.scheduling.invalid_start_time",
  "suggestions": ["Use 08:00"]
}
```

Why this works

Each problem points to the exact failing path in the document. Severity is consistent across all rules: error, warning, or info. Metric IDs make filtering and reporting trivial.

Suggestions are included inline so authors can fix issues without leaving their editor. Structured context fields give debugging tools the data they need to surface meaningful explanations.

Validation taxonomy

Every rule follows a naming pattern, which makes filtering, reporting, and automation easy.

How rules are named

Each rule ID follows the pattern `{domain}.{category}.{specific}`.

- **Domains:** schema, object, task, resource, temporal, economic, spatial, recipe, custom
- **Categories:** integrity, reference, scheduling, flow, state, optimization

How teams use this

Set CI gates by severity level: block on errors, allow warnings. Track which rules fail most often. Spot quality trends as your simulation library grows, and feed the results back into writing guidance to prevent repeat mistakes.

Example rule IDs

- `schema.integrity.missing_root`
- `task.reference.invalid_actor`
- `temporal.scheduling.actor_overlap`
- `resource.flow.negative_stock`
- `recipe.compliance.missing_inputs`

Rule IDs are stable, so you can use them in configs, dashboards, and ignore lists without worrying about them changing between versions.

Built-in modeling

WorkSpec handles time, space, costs, and production recipes natively, not as afterthoughts.

⏱️ Temporal

Strict time parsing with dependency and overlap checks. Multi-day support via day offsets. When a time format is wrong, the validator suggests the right one.

📐 Spatial

Define layouts with units, origins, and coordinate systems. Locations use `rect`, `circle`, or `polygon` shapes. Object locations are validated against your layout. Works for physical floors and digital environments alike.

💰 Economic

Labor and equipment carry cost-per-hour. Resources and products have per-unit cost and revenue. The validator can check profitability at authoring time.

Gross Profit = Revenue - Labor Cost - Material Cost

🧪 Recipes

Define expected inputs for each product, with optional equipment and timing metadata. The validator checks outputs against consumed inputs. Missing inputs produce warnings (not hard errors), so partial specs stay useful.

Migration and tooling

Moving from legacy syntax is straightforward, and the CLI handles validation, migration, and formatting.

Legacy to WorkSpec mapping

`consumes` / `produces`	→	`interactions` + `delta`
`equipment_interactions`	→	**state** `from/to`
`object_id`	→	`target_id`
`revert_after`	→	`temporary`
`article_title`	→	`title`
flat document	→	`world` + `process`
emoji in task IDs	→	**separate** `id` + `emoji`
type aliases	→	**canonical types**

CLI usage

```
npm install -g workspec
workspec validate bakery.workspec.json
workspec migrate legacy.json -o modern.workspec.json
workspec format modern.workspec.json
```

```
✓ schema.integrity.missing_root
✓ object.reference.valid_locations
✗ resource.flow.negative_stock
  └─ 'flour' goes negative after 'big_batch'
  └─ Suggestion: Increase initial quantity

2 passed, 1 error
```

Getting started

Four steps, under a minute.

1. Install: `npm i -g workspec`
2. Create a `.workspec.json` with `$schema`
3. Run: `workspec validate`
4. Read the docs, iterate, ship

What WorkSpec is

A specification and validation language. A shared contract so different tools and teams can all work with the same format. A portable JSON schema with a built-in validator.

What WorkSpec is not

Not a runtime engine; it doesn't execute your processes. Not a process-mining tool. Not locked to one UI. It's the authoring layer that sits before execution, making sure everything is correct before it runs.

Universal Automation Wiki

Define work once. Validate everywhere.

Read the specification, try the tooling, and migrate existing simulations at your own pace.

Spec npm package UAW