WoT-Lab: Executable Service Descriptions for Benchmarking Language Model Agents

Thomas Wehr[1], Sebastian Schmid[1] and Andreas Harth[1,2]

[1]Friedrich-Alexander-Universität Erlangen-Nürnberg, Nuremberg, Germany
[2]Fraunhofer IIS, Division Data Spaces and IoT Solutions, Nuremberg, Germany

Abstract

Today's benchmarks for language model (LM) agents that compose services are seldom built for reuse: the services are code, each task is scored by code of its own and the agent interacts with the services through a custom interface. Adding a service, building an environment or adapting a task therefore means programming, so a benchmark grows only as fast as its developers program, and an agent has to be adapted to every benchmark. We present WoT-Lab, a framework for reusable and interoperable benchmarks: reusable because services, environments and tasks are data, and interoperable because each service is a Thing of the W3C Web of Things (WoT), so an agent needs nothing beyond an HTTP client to discover, observe and invoke it. Every Thing of WoT-Lab is described with a WoT Thing Description that extends each of its Actions with an executable effect specification. WoT-Lab instantiates every Thing from this description and a JSON file declaring its initial state. Each Thing therefore behaves exactly as its effect specifications describe. A task states its goal as predicates over the final state, so every task is scored the same way and no task needs scoring code of its own. One description thus plays three roles: the agent fetches it, the environment executes it and the goals are stated over it. We formalize environments, tasks and their success criterion, and we release a suite of six environments of up to 2279 Things with 76 tasks at seven difficulty levels, each task with the shortest plan we know.

Keywords

benchmark, language model agents, Web of Things

1. Introduction

Describing, discovering and composing services have been studied for more than two decades [1, 2], but for a different kind of agent. Composition was the work of planners. In the Semantic Web Services of the 2000s, each service carried a formal description of its inputs, outputs, preconditions and effects [3, 4, 5]. From these descriptions, a planner derived a sequence of invocations whose combined effects satisfied the preconditions and effects required of the composite service, provided every service behaved as described. A composition could thus be validated against the descriptions alone, without running a service [6]. Today, language model (LM) agents take over the planner's role [7, 8]: they compose services to reach a user's goal, but only with some probability. Additionally, when an LM agent samples each call, the same request may yield different attempts, and an attempt may stop short of the goal or act beyond it. Whether an agent composes services correctly has thus become an empirical question, and answering it requires benchmarks for service-consuming agents.

Benchmarking agents tasked to change the state of the services they call requires stateful services [9, 10, 11]. Existing benchmarks typically express the behaviour of such a service twice, as program code and as a prose description for the agent. They also often add code for scoring attempts and expose the services through a custom interface. Code, prose and interface each carry a cost. Code has to be written for every new service, environment and task, so a benchmark covers only the domains its authors could afford to program. Prose at most summarizes an operation's effect, so an agent cannot plan over it, and where prose and code disagree, an attempt fails due to the benchmark, not the agent. Through the interface, an agent reaches the services as function schemas [10, 11] or, at best, as a listing over the

BenchSOC@ICSOC2026: 1st International Workshop on Benchmarking in Service-Oriented Computing at ICSOC 2026, December 1st, 2026, Łódź, Poland

thomas.wehr@fau.de (T. Wehr); sebastian.schmid@fau.de (S. Schmid); andreas.harth@fau.de (A. Harth)
0000-0002-0678-5019 (T. Wehr); 0000-0002-5836-3029 (S. Schmid); 0000-0002-0702-510X (A. Harth)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Model Context Protocol (MCP) [12, 13], so it has to be adapted to every benchmark. Returning to the formal descriptions of Semantic Web Services would let an agent plan over effects, but such descriptions stay apart from the code that implements the services, and they cost more to author than they returned in value [7]. A benchmark therefore needs a description that is cheap to write, that is the service's behaviour, and that any agent can read.

In this paper, we present WoT-Lab, a framework for authoring benchmarks as data, together with a *suite* of environments and tasks. WoT-Lab replaces interface, prose and code: the interface is a World Wide Web Consortium (W3C) standard, each service is instantiated from the very description the agent reads, and services, environments and tasks are data rather than code. WoT-Lab represents a service as a *Thing* of the Web of Things (WoT) [14] and serves every Thing over HTTP, so an agent needs nothing beyond an HTTP client to discover, observe and invoke the services. Each Thing is described by a machine-readable Thing Description (TD) [15]. From the TD, an agent discovers the Thing's links to other Things, its *Properties* (observable state) and its *Actions* (operations). For a simulation, however, the WoT defines no way to describe how a Thing behaves [16]. WoT-Lab therefore extends each Action in the TD with an executable effect specification [17], an expression that maps the current state and the Action's input to the next state. WoT-Lab executes the effect specification itself, so a service has no code that could diverge from its description. One document thus tells the agent what an Action does and the environment how to execute it.

Environments and tasks are data as well: a JSON file lists an environment's Things and their initial state, and a JSON record states a task's goal as a set of predicates over the final state, so no task needs scoring code of its own. Stating tasks as data requires defining precisely what a goal and a successful attempt are, which we do in a formal model (Section 3).

On this basis, we built a suite of six environments and 76 goal-directed tasks at seven difficulty levels. Four environments model domains in which agents act on a user's behalf: an e-commerce shop, a smart home, a supply chain and a social media network. The other two test an agent at scale: the smart office tests discovery among 2279 Things and the factory tests long plans, with tasks of up to 618 invocations. Some tasks are already satisfied, because the initial state already meets their request, and some are infeasible, because no run can meet their request. In both, an attempt that invokes no Action succeeds. Every task carries an optimal plan, the shortest run known to us that reaches its goal.

Our contributions are thus:

1. A formal model of environments, tasks and a state-based success criterion (Section 3).
2. The WoT-Lab framework, which instantiates each Thing from its Thing Model (TM) and state file and executes the effect specifications of its TD (Section 4).
3. A suite of 76 tasks in six environments instantiated in WoT-Lab, each task with an optimal plan (Section 5).

2. Background

In this section, we describe the background WoT-Lab is built upon: the WoT specifications and the language in which an Action's effects are specified.

Throughout this paper, we use the following running example. Alice wants to read in the evening and instructs her agent to dim her reading lamp to 30 percent. The lamp, however, draws its power from a smart plug that is switched off. Before it can act, the agent must therefore discover the lamp, learn that the lamp is unpowered, follow the lamp's description to the plug and find that toggling the plug powers the lamp. Only then can it switch the plug on and dim the lamp (Figure 1). From the lamp's description, the agent thus needs not only the lamp's operations but also their dependencies and effects.

The environments of this paper follow the WoT Architecture [14], a W3C Recommendation for describing and accessing connected devices and services. Every service, such as Alice's lamp and plug, is a *Thing*. A Thing exposes *Properties* (observable state) and *Actions* (operations that change that state). The smart plug, for example, is a Thing that exposes the Properties `poweredOn` and `device` and provides the Action `toggle()`.

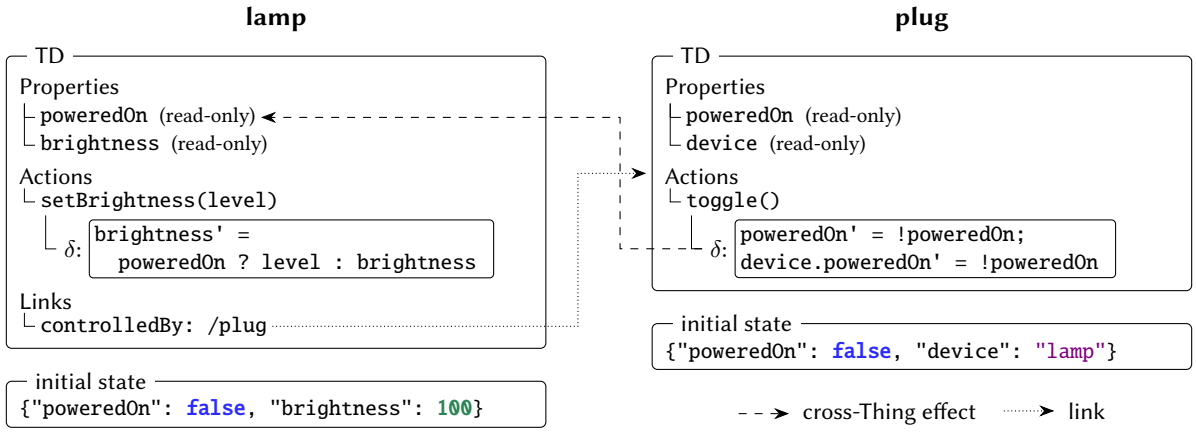


Figure 1: Two Things of Alice's smart home, the lamp and the plug. Each Thing has a TD, which names its Properties, its Actions with their effects (δ) and its links. The initial state gives every Property a value. Action outputs are omitted. The dashed arrow is the effect of the plug's `toggle()` Action on the lamp's `poweredOn` Property, reached through the plug's `device` Property, and the dotted arrow is the lamp's `controlledBy` link to the plug.

Each Thing is described by a *TD* [15], a JSON-LD document that lists the Thing's Properties and Actions. For every Property value and every Action input and output, the TD gives a JSON Schema, and for every Property it states whether a client may read it, write it or both. A TD also contains *links*, which point to related Things. The TD of Alice's lamp links to the plug that powers it under the relation `controlledBy`. The lamp's TD thus gives an agent the lamp's operations, such as the `setBrightness(level)` Action with the schema of its input `level`, and, through the link, the plug they depend on. The TD Recommendation also defines the *Thing Model (TM)*, a template from which TDs are instantiated: a TM describes a class of Things and leaves out instance-specific parts, such as a Thing's identifier and the forms that give the Uniform Resource Identifiers (URIs) of its operations.

What a TD does not give is the effect of these operations on the state. In WoT-Lab's environments, each Action is therefore annotated with its effects in a declarative effect specification language [17]. An effect specification consists of assignments to Properties. A primed name such as `brightness'` denotes the Property's value after the Action, and an unprimed name such as `brightness` denotes its value before. The effect of the `setBrightness(level)` Action in Figure 1 assigns the requested level to the `brightness` Property only if the lamp is powered, and otherwise leaves the brightness unchanged. An effect may also change the Properties of another Thing. The plug's `device` Property holds the identifier of the lamp it powers, so `device.poweredOn` denotes the lamp's `poweredOn` Property. The effect of the plug's `toggle()` Action thus specifies that invoking the `toggle()` Action flips the plug's `poweredOn` Property and sets the lamp's `poweredOn` Property to the plug's new value.

The effects of an Action describe one simultaneous transition, which we formalize in Section 3 as the effect function δ. Every right-hand side is evaluated against the state at the moment of invocation, and all assignments are then applied at once. The order of the effects therefore carries no meaning. An effect may read only the state before the invocation and the input of its Action, so no effect may read a primed Property. The effect specification language expresses arithmetic, comparisons, conditional expressions and operations on lists (appending, removing, counting and testing membership), but neither wall-clock time nor randomness. The state after any sequence of Action invocations is therefore determined by the initial state and the invocations alone, so replaying them from the same initial state always leads to the same state.

3. Model

The formal model underlying WoT-Lab makes precise what an agent must achieve in a task and when an attempt succeeds.

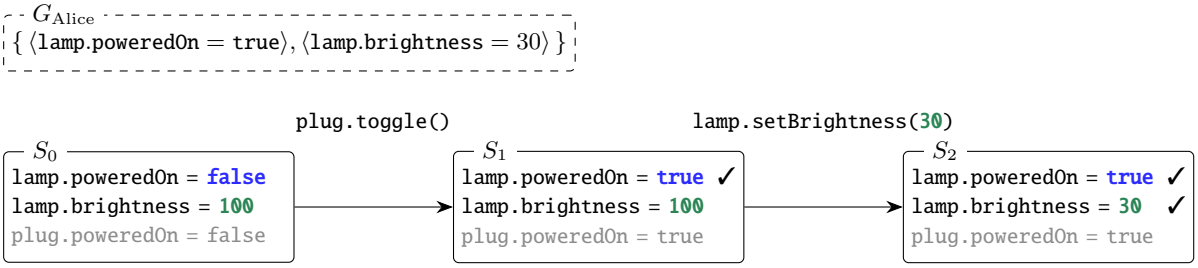


Figure 2: The goal G_{Alice} of Alice's task (dashed box) and the task's optimal plan, a run from the initial state S_0 of Alice's smart home. Box S_i shows the state after the run's first i invocations. A check mark ✓ marks a value that satisfies its goal predicate. The arrows represent the run's two Action invocations.

Things, Properties and Actions. Let $\mathcal{T}$ be a finite set of Things, where a Thing $t \in \mathcal{T}$ is identified by a unique URI uri_t and exposes Properties and Actions, whose names form the finite sets $\mathcal{P}_t$ and $\mathcal{A}_t$, respectively. Within a Thing, Property names are unique, and so are Action names [15]. We write $t.p$ for the URI (uri_t, p) of a Property and $t.\alpha$ for the URI (uri_t, α) of an Action. These URIs form the sets

$$\mathcal{R}_{\text{p}} = \bigcup_{t \in \mathcal{T}} \bigcup_{p \in \mathcal{P}_t} \{(\text{uri}_t, p)\} \ , \qquad \mathcal{R}_{\text{a}} = \bigcup_{t \in \mathcal{T}} \bigcup_{\alpha \in \mathcal{A}_t} \{(\text{uri}_t, \alpha)\} \ .$$

States and environments. Property values and Action inputs are JSON values, which a TD types with JSON Schema. We write $\mathcal{D}$ for the set of all JSON values. A *state* is a function $S \colon \mathcal{R}_{\text{p}} \to \mathcal{D}$ that assigns every Property URI a value, and we write $\mathcal{S}$ for the set of all states. An *environment* is a pair $E = (\mathcal{T}, S_0)$ of a set $\mathcal{T}$ of Things and an *initial state* $S_0 \in \mathcal{S}$. Alice's smart home is an environment whose Things include the lamp and the plug of Figure 1. Its initial state sets $S_0(\texttt{lamp.poweredOn}) = \texttt{false}$, $S_0(\texttt{lamp.brightness}) = 100$ and $S_0(\texttt{plug.poweredOn}) = \texttt{false}$.

Effects. An Action $\alpha \in \mathcal{A}_t$ is invoked at its URI $t.\alpha \in \mathcal{R}_{\text{a}}$ with input $\vec{v} \in \mathcal{D}^n$, a tuple of the n values the Action takes. Each Action is defined by its *effect function* $\delta_{t.\alpha} \colon \mathcal{S} \times \mathcal{D}^n \to \mathcal{S}$, which maps a *pre-state* S and the input to the *post-state* $\delta_{t.\alpha}(S, \vec{v})$. An effect function may change the Properties of several Things at once: $\delta_{\texttt{plug.toggle}}$ flips the plug's `poweredOn` Property and sets the lamp's `poweredOn` Property to the plug's new value. It may also be conditional: $\delta_{\texttt{lamp.setBrightness}}$ sets the lamp's `brightness` Property to its input only if the lamp is powered and otherwise returns the pre-state.

Runs. A *run* is a finite sequence of Action invocations $\rho = t_1.\alpha_1(\vec{v}_1), \ldots, t_k.\alpha_k(\vec{v}_k)$ leading from the initial state S_0 to the run's *final state* $S_\rho = \delta_{t_k.\alpha_k}(\cdots \delta_{t_1.\alpha_1}(S_0, \vec{v}_1) \cdots, \vec{v}_k)$. The empty run leaves the initial state unchanged, $S_\rho = S_0$. The run's *length* $|\rho|$ is its number of invocations k. We write S_i for the state after the first i invocations of a run. In Figure 2, the run `plug.toggle()`, `lamp.setBrightness(30)` leads from S_0 to the state S_2, in which the lamp is powered and dimmed to 30. We assume that the invocations of a run are applied sequentially, that no other party changes the state during the run and that the state changes only through invocations (Section 4).

Goals. A goal states a condition on the final state of a run. A *goal predicate* is a function $g \colon \mathcal{S} \to \mathbb{B}$, where $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$. A state S *satisfies* g, written $S \models g$, iff $g(S) = \texttt{true}$. In our tasks, every goal predicate compares one Property with a constant $c \in \mathcal{D}$. We write $\langle t.p = c \rangle$ for the predicate that holds in S iff $S(t.p) = c$, and likewise for the other comparison operators (Section 5.2). A *goal* is a finite non-empty set G of goal predicates, read as a conjunction: a state S satisfies G, written $S \models G$, iff $S \models g$ for every $g \in G$. A run ρ *reaches* G iff $S_\rho \models G$. Alice's request corresponds to the goal G_{Alice} (Figure 2), which asks for a powered lamp at brightness 30. Goals specify the final state only in part: they constrain only the Properties named by their predicates, and other Properties may change. No predicate of G_{Alice}, for example, names `plug.poweredOn`, although reaching the goal requires switching the plug on.

Tasks. A *task* is a triple (E, N, G) of an environment $E = (\mathcal{T}, S_0)$, a natural-language request N and a goal G such that some run reaches G. In Alice's task, E is Alice's smart home (Figure 1), N is the sentence "Dim my reading lamp to 30 percent" and G is the goal G_{Alice}. An *optimal plan* for a task is a shortest run that reaches G. The two-step run in Figure 2 is the optimal plan of Alice's task. We denote the length of a task's optimal plan by ℓ, which is zero when $S_0 \models G$. The optimal plan is empty for two kinds of tasks (examples in Table 2). An *already-satisfied* task's request N asks for a state that already holds in S_0, and an *infeasible* task's request N asks for a state that no run reaches, so its goal states that the initial state stays unchanged.

Attempts, success and efficiency. An agent attempting a task receives the request and the environment in its initial state. Its *attempt* is the run ρ of the Action invocations it issues, so reading Properties and fetching TDs are not part of the attempt. The attempt leads to its final state S_ρ and *succeeds* iff it meets the task's *success criterion* $S_\rho \models G$. In an already-satisfied or infeasible task, an attempt that invokes no Action therefore succeeds, and any other attempt fails only if its invocations leave a predicate of the goal violated. An agent may additionally report that a request is unreachable, and the harness in our repository records such a report. Success, however, stays a property of the final state alone. If the task's optimal plan is not empty ($\ell > 0$), a successful attempt has *efficiency* $\ell/|\rho|$. Suppose an agent attempting Alice's task first invokes `lamp.setBrightness(30)`, which changes nothing on the unpowered lamp, and then `plug.toggle()` and `lamp.setBrightness(30)` again. The agent's attempt succeeds with three invocations where the optimal plan needs two, so its efficiency is $2/3$. A failed attempt can be graded further by the fraction of the predicates of G that S_ρ satisfies.

4. Instantiating Environments

WoT-Lab serves an environment $E = (\mathcal{T}, S_0)$ in which every Thing executes the effect specifications of the very TD that an agent fetches from it.

A Thing is authored as two JSON files: a TM, which annotates each of the Thing's Actions with its effect specification, and a *state file*, which declares the Thing's initial state. An environment is authored as an *environment file*, a JSON file whose entries each name a TM to instantiate and fix the identifier of the resulting Thing. Under this identifier, WoT-Lab serves each Thing over the WoT HTTP binding [18]. WoT-Lab instantiates one Thing per entry in the environment file, so the same TM may be instantiated many times: for the smart office, WoT-Lab instantiates 2279 Things from ten TMs (Section 5.1). An entry may also override the Thing's title, initial state and links, and it may annotate the Thing's TD. In the environment file of Alice's smart home, the plug's entry sets the plug's `device` Property to the lamp's identifier, and the lamp's entry links the lamp to the plug (Figure 1). The state files and these overrides together declare an initial state, which we call the environment's *declared initial state*.

```
"setBrightness": {
  "input": { "type": "object", "properties": {
               "level": { "type": "integer", "minimum": 0, "maximum": 100 } },
             "required": ["level"] },
  "output": { "type": "object", "properties": { "success": { "type": "boolean" },
              "brightness": { "type": "integer" } } },
  "lab:effects": "brightness' = poweredOn ? level : brightness;
                  output.success = poweredOn; output.brightness = brightness';" }
```

Listing 1: The `setBrightness(level)` Action in the lamp's TD, with its title and description omitted. The first clause of the effect specification `lab:effects` is the effect that Figure 1 shows.

An agent learns from an Action's response whether its invocation took effect. WoT-Lab computes this response, the Action's *output*, from *output clauses* that the effect specification declares next to the effects. Outputs do not change the state, so the effect function $\delta_{t.\alpha}$ leaves them out. Listing 1 shows

Table 1
The six environments of the suite and their tasks. Things, Properties and Actions are counted over the instantiated Things, and *Cross* counts the Actions whose effect writes a Property of another Thing. DL0–DL5 and S count the tasks per difficulty level (Table 2). $\tilde{\ell}$ and $\ell_{\max}$ are the median and the maximum of the optimal plan's length ℓ over the environment's tasks, counting the empty plans of DL0 and DL5, and $\overline{|G|}$ is the mean number of goal predicates per task.

	Environment				Tasks per level								Per task		
	Things	Prop.	Act.	Cross	DL0	DL1	DL2	DL3	DL4	DL5	S	All	$\tilde{\ell}$	$\ell_{\max}$	$\overline{\lvert G\rvert}$
E-commerce	4	9	12	4	1	2	2	6	1	2	–	14	1	7	3.5
Smart home	7	22	11	7	2	3	2	4	2	2	–	15	1	5	3.1
Supply chain	6	18	6	6	2	1	2	3	6	2	–	16	1	4	3.3
Social media	3	27	12	9	2	2	2	6	1	1	–	14	1	4	2.7
Smart office	2279	3684	2113	843	–	–	–	–	–	–	11	11	2	10	2.4
Factory	37	115	27	25	0	0	1	4	0	1	–	6	78	618	2.3
Total	2336	3875	2181	894	7	8	9	23	10	8	11	76	–	–	–

the `setBrightness(level)` Action in the lamp's TD. Its `output` schema types the response, and its effect specification defines each field of the response by one clause `output.<field> = <expression>`. WoT-Lab evaluates the output clauses after applying the effects. An output clause may therefore read a primed Property. For example, `output.brightness` reports the brightness after the invocation. In S_0, where the lamp is unpowered, the invocation `lamp.setBrightness(30)` answers `{"success": false, "brightness": 100}`, which tells the agent that the invocation took no effect.

An *evaluator*, the program or person that runs a task and scores its attempt, controls an environment through routes under the reserved path `/_lab`. The route `POST /_lab/reset` restores the declared initial state, and `POST /_lab/state` assigns Property values, even of read-only Properties, to set a task's initial state (Section 5.2). A third route lists an environment's tasks with their goals. By default, WoT-Lab answers these routes only from the local machine, so an agent on another machine can neither change the state through them nor read the goals. An agent on the local machine could do both, so an evaluator who runs an agent there confines it, for example, to a container. Besides these routes, WoT-Lab serves an HTML dashboard on which a person can invoke Actions, inspect the state and replay a recorded run against a task (Section 5.4).

An evaluator runs a task in three steps. 1) The evaluator resets the environment and assigns the task's initial state, if the task sets one. 2) The agent receives the request and the root URI of the environment and attempts the task. 3) The evaluator reads every Property that the goal names. Together, these steps and WoT-Lab meet the assumptions of the model. Step 1 starts the attempt in the task's initial state S_0. WoT-Lab applies each invocation atomically, so even concurrent invocations are applied sequentially. No other party changes the state during the attempt as long as the evaluator runs one agent at a time and no other party invokes Actions or uses the `/_lab` routes or the dashboard. As every Property of the six environments is read-only and no effect reads wall-clock time or randomness, the state changes only through invocations.

5. Suite: Environments and Tasks

In this section, we present the suite: the six environments we instantiated in WoT-Lab and the 76 tasks they host. The suite consists only of data: a TM and a state file per authored Thing, one environment file per environment and one record per task. We describe the environments and what couples their Things (Section 5.1), the record in which a task is shipped (Section 5.2), the difficulty levels of the tasks (Section 5.3) and how we authored and validated the tasks (Section 5.4).

5.1. Environments

Table 1 lists the six environments of the suite and the tasks they host. Four of them model service domains in which agents act on a user's behalf: e-commerce, smart home, supply chain and social media. The remaining two test an agent at scale: the smart office tests discovery among 2279 Things, and the factory's tasks include plans of up to 618 invocations. All six couple their Things: an Action on one Thing changes or depends on the state of another. Because the difficulty levels DL3 and DL4 build on this coupling (Section 5.3), we describe for each environment the Actions that couple its Things.

E-commerce. Alice shops from a cart and pays from one of three bank accounts. The cart's `checkout(...)` Action takes the paying account as an input and subtracts the cart's total from that account's `balance` Property, and an account's `transfer(...)` Action writes the receiving account's `balance` Property. No Action checks an account's funds, so a purchase can overdraw the paying account, and a task goal on an account's `balance` Property decides whether money must be transferred before the `checkout(...)` Action.

Smart home. Alice's home holds a thermostat, a washer, a dryer, a car charger, the lamp and the plug of Figure 1, and a Thing for the home itself that aggregates power draw and monthly energy use. Starting or stopping an appliance writes the home's `currentPowerUsageW` Property, and the plug's `toggle()` Action writes the lamp's `poweredOn` Property. A goal that bounds the power draw constrains which appliances may run together.

Supply chain. A company runs three warehouses and buys stock from a supplier against a monthly budget. The warehouses, the supplier and the budget are Things of their own, and a warehouse-system Thing aggregates the three warehouses. A warehouse's `orderStock(...)` Action takes a quantity of units, adds the quantity times the supplier's unit price to the budget's `monthlySpent` Property, and writes the system's `totalCurrentStock` Property, whereas the `transferStock(...)` Action moves stock to another warehouse and spends nothing. Stock can therefore be raised by ordering or by transferring, and a goal on the budget decides between the two.

Social media. Alice, Bob and Charlie each host their own social media profile page as a Thing, as in a decentralized social network. A page's `followUser(...)`, `unfollowUser(...)` and `likePost(...)` Actions take another page as an input and, besides the page's own Properties, write the input page's `followers` or `likesReceived` Property. A goal on one page's followers is thus reached only through Actions of other pages.

Smart office. We generated the smart office's environment file from a public model of an office building of the IBM campus in Dublin.[1] The model is written in Brick, a metadata schema for buildings [19]. Each Thing carries an annotation that locates it in one of the model's rooms, floors or zones. Of the file's 2279 Things, 312 are points of the model: 146 luminance command points (which switch groups of lights), 63 luminance sensors and 103 occupancy sensors. To each of the model's 281 rooms, we added a lamp, a radiator, a temperature sensor, and a contact sensor and an actuator for the door and for the window. An actuator's `setOpen(...)` Action writes the `isOpen` Property of the contact sensor it references, and a radiator's `increaseTemperature(...)` Action writes the `temperature` Property of the temperature sensor in the radiator's room. The goals resemble those of the smart home, but an agent must first find the Things a request concerns among thousands. To find them, the agent matches the room, floor or zone that the request names against each Thing's annotation.

[1] https://github.com/BuildSysUniformMetadata/GroundTruth/blob/2e48662/building_instances/IBM_B3.ttl

```
1 { "id": "sh-8", "environment": "smart-home", "level": "DL3",
2   "initialState": {},
3   "request": "Dim my reading lamp to 30 percent.",
4   "goal": [
5     { "thing": "lamp", "property": "brightness", "op": "=", "value": 30 },
6     { "thing": "lamp", "property": "poweredOn",  "op": "=", "value": true } ],
7   "optimalPlan": [
8     { "thing": "plug", "action": "toggle",         "input": {} },
9     { "thing": "lamp", "action": "setBrightness", "input": { "level": 30 } } ] }
```

Listing 2: The task record of Alice's task. Its `goal` is G_{Alice} and its `optimalPlan` the run of Figure 2.

Factory. We based the factory environment on MOSAIK [20], a formal model of self-organizing smartphone manufacturing systems. The factory models a shopfloor. Its Things are 25 products, ten workstations and one transporter on a grid of 37×22 cells, and a shopfloor Thing that holds the 12 blueprints, each naming the inputs from which a workstation produces one product. The transporter's move Actions write the position of the product the transporter carries, the `pickup(...)` Action takes effect only where the requested product lies, and a workstation's produce Action takes effect only once every input of its blueprint lies at the workstation. The factory's optimal plans can therefore be long: in one task, carrying a single product to a workstation takes 75 invocations, and producing the smartphone takes 618.

5.2. Task records

A task is shipped as a JSON record that holds the task's environment, request and goal, the triple (E, N, G), together with three labels: an identifier, a difficulty level (Section 5.3) and an optimal plan. At DL4 and DL5, a record also holds fields that only validation uses (Section 5.4). To run the task and score an attempt at it, an evaluator needs nothing beyond the record and WoT-Lab. Listing 2 shows the record of Alice's task. The `environment` field names the environment file WoT-Lab instantiates, and `request` holds the request N. The `initialState` field lists the Property values in which the task's initial state S_0 differs from the environment's declared initial state. For Alice's task in Listing 2, the field is empty. Another smart-home record poses Alice's request in a state where the lamp is already powered: its `initialState` sets the `poweredOn` Properties of the plug and the lamp to `true`, so the `setBrightness(level)` Action alone reaches the goal. The `optimalPlan` field lists the Action invocations of the task's optimal plan, the shortest run known to us (Section 5.4). From this list, an evaluator derives the plan's length ℓ.

The `goal` field lists the goal predicates of G, each as an object with the fields `thing`, `property`, `op` and `value`. For the Thing t that `thing` names and its Property p that `property` names, the object denotes the goal predicate that holds in a state S iff $S(t.p)$ compares with `value` as the operator `op` demands. The object in line 5 of Listing 2, for example, denotes the goal predicate $\langle \texttt{lamp.brightness} = 30 \rangle$ of G_{Alice}. Our task records restrict a goal predicate to such a test of one Property against a constant, whereas the goals of our model admit any function $g \colon \mathcal{S} \to \mathbb{B}$ as a goal predicate. The operator `op` is one of eight: the six comparisons $=, \neq, <, \leq, >$ and $\geq$, and two operators for array-valued Properties. The operator `length` compares the array's length with the value, and `contains` tests whether the value is a member of the array. The goal of the e-commerce request "Put the laptop in Alice's cart", for example, uses both: the cart's `items` array must have length 1 and contain `laptop`.

Besides the requested change, a goal may list predicates that the initial state already satisfies, such as one that keeps a company's spending unchanged. An attempt whose final state violates such a predicate fails, even when it reaches the requested state.

Table 2
The difficulty levels, each with what it is designed to isolate and one request of the suite. A parenthesis states the fact about the environment that decides the level.

Level	Isolates	Example request
DL0	already satisfied: the goal holds initially	"Make sure the dryer is not running." (it is not)
DL1	one Action on one Thing	"Set the thermostat to 68 degrees."
DL2	several independent Actions	"Set the thermostat to 68 degrees in heat mode."
DL3	a chain of Actions across coupled Things	"Dim my reading lamp to 30 percent."
DL4	a choice among alternatives under coupled state	"Warehouse C needs 2000 units, but no money may be spent this month."
DL5	infeasibility: no run reaches the requested state	"Bring Warehouse C to 4000 units." (its capacity is 3000)
S	DL1–DL3 goals over thousands of Things	"Switch on all lamps in rooms SOR42_G_01 to SOR42_G_10."

5.3. Difficulty levels

Every task carries one of the seven difficulty levels of Table 2, each designed to isolate one ability. The tasks of DL1 to DL4 and S require Actions, and those of DL0 and DL5 require none.

At DL1, a single Action on a single Thing reaches the goal, so the level tests whether an agent discovers the Thing and invokes the Action with a valid input. At DL2, the goal needs several Actions that do not depend on one another, so the level tests whether an agent carries out every part of a request, in any order.

The levels DL3 and DL4 build on the coupling of Things. At DL3, the goal spans coupled Things: an Action of the task's optimal plan writes a Property of a different Thing which is named by the goal. Alice's task is a DL3 task, because the plug's `toggle()` Action writes the lamp's `poweredOn` Property, which her goal names. With the lamp already powered, the same request yields a DL1 task. In general, a task whose optimal plan contains no such Action is a DL1 task if the plan has one invocation and a DL2 task if it has several. At DL4, several runs reach the requested state, but the coupling makes some of them violate another predicate of the goal. We call such a run a *distractor run*. The DL4 request in Table 2, for example, is met by ordering stock or by transferring stock from another warehouse, and only the transfer leaves the company's spending unchanged. The scale level S applies only to the smart office and builds on the number of Things instead: its goals resemble those of DL1 to DL3, but the agent must find the relevant Things among thousands.

DL0 and DL5 share the empty run as their optimal plan but differ in why invoking no Action is correct: DL0 holds the already-satisfied tasks and DL5 the infeasible tasks. In a DL0 task, the request describes a state that already holds, so an agent that reads the state finds nothing to do. In a DL5 task, the request describes a state that no run from the initial state reaches. The state is unreachable for one of four reasons: a Property's bound, such as a warehouse's capacity or a thermostat's maximum, makes an Action keep the pre-state; no Action's effect changes the requested Property in the way asked; the requested conditions cannot hold together under the effects; or the requested Thing is absent from the environment. The goal of a DL5 task therefore requires every Property it names to keep its initial value. The goal of the DL5 request in Table 2, for example, requires Warehouse C to keep its 1200 units.

5.4. Authoring and validation

Every request and every goal was written by hand, against the environment instantiated on the dashboard. The Property values of each task's initial state were set through the route `POST /_lab/state`. Two parts of the records were too large to write by hand, so a script generated them: the Thing identifiers in the smart office's goals and the factory's optimal plans, which it derived from the blueprints. The remaining optimal plans we wrote by hand and replayed on the dashboard, which shows whether the goal holds in every state of the run. Three further fields serve only validation. Every DL4 record holds a distractor run, such as ordering the missing stock for the DL4 request of Table 2. Every DL5 record states in prose what makes the requested state unreachable. Two DL5 records, the thermostat request of the smart home and the warehouse request of Table 2, also hold a naive attempt: the invocation an

agent would most plausibly try, such as an order that would bring Warehouse C to 4000 units.

A replay script validates all 76 records against the environments. For each record, it first resolves every Thing, Property and Action the record names against the served TDs and tests that every initial value fits its Property's schema. It then resets the environment, assigns the task's initial state and asserts that every invocation of the optimal plan is answered successfully, that the plan reaches the goal and that no proper prefix of it does. Three further assertions test the level labels. For every DL4 task, the final state of the distractor run violates at least one goal predicate. For every DL0 and DL5 task, the initial state satisfies the goal. For both DL5 tasks with a naive attempt, invoking it changes no Property value, so the Action an agent would most plausibly invoke cannot reach the requested state. All 76 records pass the replay. No assertion tests the labels DL1 to DL3 and S, and none shows that a plan is the shortest run. We therefore describe an optimal plan as the shortest run known to us.

Availability. WoT-Lab, the six environments, the 76 task records and the replay script are available in one public repository[2] under the AGPL-3.0 licence. The replay script's goal check reads the final state of any run and evaluates the task record's goal on it, so an evaluator scores an attempt without writing checker code. The release we describe here, version 0.4.1, is archived on Zenodo.[3]

6. Related Work

In this section, we relate WoT-Lab to four groups of resources: stateful environments for tool-using LM agents, environments defined by descriptions, benchmarks and datasets in service-oriented computing, and service descriptions that state behaviour. We compare each group with WoT-Lab by the three roles that one TD plays at once: the agent fetches it, the environment executes it and the goals are stated over it.

Stateful environments for tool-using agents. AppWorld [9], τ-bench [10] and τ^2-bench [21] let an agent operate on simulated apps and databases and score an attempt by its final state, the two τ-benches also by whether the agent's replies to the user contain the information the task requires. AgentDojo [11] adds attacks to such tasks, and Toolathlon [12] and MCP-Bench [13] reach their tools over MCP. WebArena [22] and OSWorld [23] pose such tasks on websites and desktop applications. ToolSandbox [24] scores an attempt by the milestones it passes, and TheAgentCompany [25] by the checkpoints it passes on the self-hosted websites of a simulated software company. Like AppWorld and the two τ-benches, WoT-Lab scores an attempt by its final state and resets every attempt to a fixed initial state. The agents of all these environments, however, reach the operations as function schemas, as documented APIs that they call from code, as an MCP listing or through a web or desktop interface. An attempt is scored by code written per task, by a judging model, by comparison with the state that a reference sequence of calls leads to or by a task record that names scoring functions from a vocabulary written per domain. A τ^2-bench task, for example, names assertion functions of its domain [21], and a task of the MCP-Universe benchmark [26] chains scoring functions over the agent's answer or the live service. In WoT-Lab, the TD an agent fetches carries the effects, and an attempt succeeds if its final state satisfies the goal's predicates over the Properties that the TD declares. No task needs scoring code of its own, no model judges an attempt and no domain needs a vocabulary of scoring functions.

Environments defined by descriptions. Like WoT-Lab, other resources add a service by writing its description, but they differ from WoT-Lab in who executes the description. ToolEmu [27], ToolAlpaca [28] and StableToolBench [29] prompt an LM with a tool specification, an OpenAPI document or the documentation of a public API and let the model write the responses, and StableToolBench does so only when its cache of recorded responses misses. A service thus costs one document, but the responses are sampled, a write need not change a later read and a judging model scores the attempt. Asked to

[2] https://github.com/wintechis/wot-lab
[3] https://doi.org/10.5281/zenodo.23949010

simulate a state machine from its rules or even from its code, LMs mispredict many of the transitions that change the state, and most of those that change several values at once [30, 31]. AgentScaler [32], EnvScaler [33] and Agent World Model [34] instead derive code from a description: a schema of the state and a function per tool, which then run deterministically. The running artefact, however, is code apart from the description, and its faithfulness to the description is tested only afterwards, if at all. PDDLGym [35] executes a domain in the Planning Domain Definition Language (PDDL) with a generic interpreter, so a domain costs PDDL files rather than code, but the domain serves a planning agent in-process, and no service is fetched. In the smart home domain, HomeEnv of SMH-Bench [36] comes closest to WoT-Lab: it keeps each device's effects as code strings inside a device record of HomeEnv's own schema. However, no running device serves the record, and the code and data of SMH-Bench are not yet released. WoT-Lab, by contrast, states the effects in the standard TD itself, and the agent fetches the very TD the environment executes. For a task that changes the home, SMH-Bench, like WoT-Lab, scores an attempt by its final state, but it requires every device that the request does not concern to keep its state. A goal of WoT-Lab, by contrast, constrains only the Properties it names, and it states what must stay true as predicates that the initial state already satisfies.

Benchmarks and datasets in service-oriented computing. The service-oriented computing community shares datasets on the quality of service of public Web services, such as WS-DREAM [37] and QWS [38], and microservice systems for studying performance and faults, such as DeathStarBench [39] and TrainTicket [40]. HSC [41] pairs compositions of machine-learning services with requirements in natural language and with the optimal composition for each. For LMs, SOCBench-D [42] scores service discovery by the endpoints a retriever returns from OpenAPI documents, and BIS [43] evaluates one kind of service, the translation of questions into queries. Compositio Prompto [44] lets an LM compose OpenAPI services and evaluates the architecture in a case study. These resources measure services, state the expected composition for a request or score what an LM produces: an endpoint, a query or a composition. None of them runs stateful services against which a composition is executed and scored by its final state, and such services are what WoT-Lab adds.

Service descriptions that state behaviour. OWL-S [3] describes a Web service by its inputs, outputs, preconditions and effects. Planners composed services from such descriptions, for instance by adapting the logic programming language Golog [4], by hierarchical task network planning with SHOP2 [5, 45] and by translating the descriptions into PDDL [46, 47]. The Semantic Web Services Challenge [48] offered these approaches common scenarios, but it certified participants on mediation and discovery problems and did not score an attempt by the final state of the services.

In the WoT, several approaches likewise describe the behaviour of Things. Korkan et al. [49] annotate a TD with the sequence of interactions that leads to a state, Salama et al. [50] annotate which Actions affect which Properties, and Freund et al. [51] annotate preconditions and effects from which a planner derives the interactions. AEPIS [52] plans over such annotations at run time, repairs its plan when the environment deviates from it, and compares its planner with a rule-based and an LM agent in an office building and on the MOSAIK shopfloor. Noura and Gaedke [53] describe devices instead in WoTDL, an extension of OWL-S of their own, translate its preconditions and effects into PDDL and re-plan when monitoring detects a failed device or an unmet goal. Like an OWL-S description, each of these descriptions states the behaviour of a service that is implemented apart from it, whereas in WoT-Lab the effect specification is the behaviour that the Thing executes. Two approaches derive a running Thing from a description. Virtual-Thing [54] serves a copy of a TD and answers every Property read and Action with fresh random values that conform to the TD's data schemas. Stateful-WoT [55] keeps state, but it generates a digital twin and its extended TD from a statechart in SCXML. The WoT Use Cases and Requirements note lists as a gap that "WoT does not define a way to describe the behavior of a thing to use for a simulation" [16]. WoT-Lab addresses it by executing the effects that the TD itself states as a context extension, and it adds tasks whose goals are stated over the final state.

Most recently, LMs plan over TDs: LLM-MaGe [56] lets an LM translate a request and the TDs into

PDDL for a classical planner, and Kinder and Käfer [57] let an LM translate a device's TD into a Python class and write code that calls it. Executing this code yields the plan, and in an adaptive variant they copy the device's Property values into the class after each Action, so the code repairs the plan without a further LM call. The IoT Whisperer [58] lets an LM turn a natural-language system description into a Node-RED workflow over WoT devices and compares the generated workflows with manually written ones. Each of the WoT approaches above was evaluated on its authors' own setup. Kinder and Käfer release their setup as a benchmark of 123 planning tasks over the TDs of 29 devices [59, 57], but each of its tasks concerns a single device. A simulator per device scores a plan by the state it leads to, and by design such a simulator may hold state that the TD does not describe. In our suite, by contrast, the tasks at DL3 and DL4 span coupled Things, and the effect specifications are executed by the Things the agent calls, so the effects an agent plans over cannot diverge from the behaviour it meets.

Across these four groups, WoT-Lab is, to our knowledge, the only resource in which one standard service description, fetched from the running service, carries the state, the links and the effects, is executed without code written per service, by hand or by an LM, and declares the Properties over which the goals are stated.

7. Conclusion

In this paper, we presented WoT-Lab, a framework for reusable and interoperable benchmarks for service-consuming agents, together with a formal model of its tasks and a suite of environments and tasks built with it. In WoT-Lab, a service is a Thing whose TD carries an effect specification for each Action, and the environment executes these specifications directly, so the effects an agent reads cannot diverge from the behaviour it meets. WoT-Lab serves every Thing over the WoT HTTP binding, so an agent needs nothing beyond an HTTP client to discover, observe and invoke the services. A task is authored as a JSON record whose goal is a set of predicates over the final state, so no task needs scoring code of its own.

The suite comprises 76 tasks in six environments of up to 2279 Things. Its seven difficulty levels range from a request that the initial state already meets, through chains of Actions across coupled Things, to a request that no run can meet. Every task carries an optimal plan, and replaying each plan against its environment shows that it reaches the task's goal.

The suite has three limitations. First, this paper does not report how agents perform on the suite, so whether its tasks and levels separate agents is untested. The repository includes a harness with which an evaluator can run an LM agent on the suite and score its attempts, and our next step is a baseline in which agents built on several LMs attempt the suite. Second, we compared none of our environments with a deployed service. In the e-commerce environment, for example, no Action checks an account's funds, so a purchase may overdraw the account where a deployed service would typically refuse it. Third, our task records restrict a goal predicate to testing one Property against a constant. We plan to admit predicates that relate two Properties and goals that quantify over Things, such as all lamps on one floor.

Any domain whose Actions the effect specification language can express needs only the files our six environments needed: a TM and a state file per authored Thing, one environment file and a record per task. WoT-Lab and the suite are publicly available under the AGPL-3.0 licence, and we invite others to contribute environments and tasks.

Declaration on Generative AI

During the preparation of this work, the authors used Claude and Grammarly in order to conduct grammar and spelling checks and to paraphrase and reword. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. P. Papazoglou, D. Georgakopoulos, Service-oriented computing, Commun. ACM 46 (2003) 24–28. doi:10.1145/944217.944233.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: State of the art and research challenges, Computer 40 (2007) 38–45. doi:10.1109/MC.2007.400.

[3] D. L. Martin, M. H. Burstein, D. V. McDermott, S. A. McIlraith, M. Paolucci, K. P. Sycara, D. L. McGuinness, E. Sirin, N. Srinivasan, Bringing semantics to web services with OWL-S, World Wide Web (2007) 243–277. doi:10.1007/S11280-007-0033-X.

[4] S. A. McIlraith, T. C. Son, H. Zeng, Semantic web services, IEEE Intell. Syst. 16 (2001) 46–53. doi:10.1109/5254.920599.

[5] D. Wu, B. Parsia, E. Sirin, J. A. Hendler, D. S. Nau, Automating DAML-S web services composition using SHOP2, in: D. Fensel, K. P. Sycara, J. Mylopoulos (Eds.), The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003, Proceedings, volume 2870 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 195–210. doi:10.1007/978-3-540-39718-2_13.

[6] R. Howey, D. Long, M. Fox, VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL, in: 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA, IEEE Computer Society, 2004, pp. 294–301. doi:10.1109/ICTAI.2004.120.

[7] A. Polleres, F. Bauer, D. Dobriy, T. Käfer, L. Kubelka, A. Harth, T. Wehr, On the historic roots of agentic AI in semantic web services, in: Companion Proceedings of the ACM Web Conference 2026, WWW Companion 2026, Dubai, United Arab Emirates, 29 June 2026 - 3 July 2026, ACM, 2026, pp. 10–14. doi:10.1145/3774905.3795076.

[8] M. Aiello, A paradigm shift in service research: The case of service composition, IEEE Trans. Serv. Comput. 18 (2025) 1213–1215. doi:10.1109/TSC.2025.3552345.

[9] H. Trivedi, T. Khot, M. Hartmann, R. Manku, V. Dong, E. Li, S. Gupta, A. Sabharwal, N. Balasubramanian, Appworld: A controllable world of apps and people for benchmarking interactive coding agents, in: L. Ku, A. Martins, V. Srikumar (Eds.), Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024, Association for Computational Linguistics, 2024, pp. 16022–16076. doi:10.18653/V1/2024.ACL-LONG.850.

[10] S. Yao, N. Shinn, P. Razavi, K. Narasimhan, τ-bench: A benchmark for tool-agent-user interaction in real-world domains, CoRR abs/2406.12045 (2024). arXiv:2406.12045.

[11] E. Debenedetti, J. Zhang, M. Balunovic, L. Beurer-Kellner, M. Fischer, F. Tramèr, Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents, in: A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, C. Zhang (Eds.), Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024, 2024. doi:10.52202/079017-2636.

[12] J. Li, W. Zhao, J. Zhao, W. Zeng, H. Wu, X. Wang, R. Ge, Y. Cao, Y. Huang, W. Liu, J. Liu, Z. Su, Y. Guo, F. Zhou, L. Zhang, J. Michelini, X. Wang, X. Yue, S. Zhou, G. Neubig, J. He, The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution, CoRR abs/2510.25726 (2025). arXiv:2510.25726.

[13] Z. Wang, Q. Chang, H. Patel, S. Biju, C. Wu, Q. Liu, A. Ding, A. Rezazadeh, A. Shah, Y. Bao, E. Siow, Mcp-bench: Benchmarking tool-using LLM agents with complex real-world tasks via MCP servers, CoRR abs/2508.20453 (2025). arXiv:2508.20453.

[14] K. Toumura, M. Lagally, M. McCool, R. Matsukura, Web of Things (WoT) Architecture 1.1, W3C Recommendation, W3C, 2023. URL: https://www.w3.org/TR/wot-architecture11/.

[15] M. McCool, E. Korkan, S. Käbisch, Web of Things (WoT) Thing Description 1.1, W3C Recommendation, W3C, 2023. URL: https://www.w3.org/TR/wot-thing-description11/.

[16] M. McCool, T. Mizushima, Web of Things (WoT): Use Cases and Requirements, W3C Group Note,

W3C, 2026. URL: https://www.w3.org/TR/2026/NOTE-wot-usecases-20260205/.

[17] Anonymous Authors, V-Realm: Verifiable runtime enforcement for agentic language models, 2026. Manuscript under review. Artifact: https://anonymous.4open.science/r/V-Realm-9C42/.

[18] M. Koster, E. Korkan, Web of Things (WoT) HTTP Binding, W3C Editor's Draft, W3C, 2026. URL: https://w3c.github.io/wot-binding-templates/bindings/protocols/http/.

[19] B. Balaji, A. Bhattacharya, G. Fierro, J. Gao, J. Gluck, D. Hong, A. Johansen, J. Koh, J. Ploennigs, Y. Agarwal, M. Berges, D. Culler, R. Gupta, M. B. Kjærgaard, M. Srivastava, K. Whitehouse, Brick: Towards a unified metadata schema for buildings, in: Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments, BuildSys 2016, Palo Alto, CA, USA, November 16-17, 2016, ACM, 2016, pp. 41–50. doi:10.1145/2993422.2993577.

[20] V. Charpenay, D. Schraudner, T. Seidelmann, T. Spieldenner, J. Weise, R. Schubotz, S. Mostaghim, A. Harth, MOSAIK: A formal model for self-organizing manufacturing systems, IEEE Pervasive Comput. 20 (2021) 9–18. doi:10.1109/MPRV.2020.3035837.

[21] V. Barrès, H. Dong, S. Ray, X. Si, K. Narasimhan, τ^2-bench: Evaluating conversational agents in a dual-control environment, CoRR abs/2506.07982 (2025). arXiv:2506.07982.

[22] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, Y. Bisk, D. Fried, U. Alon, G. Neubig, Webarena: A realistic web environment for building autonomous agents, CoRR abs/2307.13854 (2023). arXiv:2307.13854.

[23] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, T. Yu, Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, in: A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, C. Zhang (Eds.), Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024, 2024. doi:10.52202/079017-1650.

[24] J. Lu, T. Holleis, Y. Zhang, B. Aumayer, F. Nan, H. Bai, S. Ma, S. Ma, M. Li, G. Yin, Z. Wang, R. Pang, Toolsandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities, in: L. Chiruzzo, A. Ritter, L. Wang (Eds.), Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025, volume NAACL 2025 of *Findings of ACL*, Association for Computational Linguistics, 2025, pp. 1160–1183. doi:10.18653/V1/2025.FINDINGS-NAACL.65.

[25] F. F. Xu, Y. Song, B. Li, Y. Tang, K. Jain, M. Bao, Z. Z. Wang, X. Zhou, Z. Guo, M. Cao, M. Yang, H. Y. Lu, A. Martin, Z. Su, L. Maben, R. Mehta, W. Chi, L. Jang, Y. Xie, S. Zhou, G. Neubig, Theagentcompany: Benchmarking LLM agents on consequential real world tasks, in: D. Belgrave, C. Zhang, L. N. Montoya, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, N. Chen, I. V. M. Ruíz, A. Loaiza-Bonilla (Eds.), Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025, 2025. doi:10.52202/085713-0315.

[26] Z. Luo, Z. Shen, W. Yang, Z. Zhao, P. Jwalapuram, A. Saha, D. Sahoo, S. Savarese, C. Xiong, J. Li, Mcp-universe: Benchmarking large language models with real-world model context protocol servers, CoRR abs/2508.14704 (2025). doi:10.48550/ARXIV.2508.14704.

[27] Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, T. Hashimoto, Identifying the risks of LM agents with an lm-emulated sandbox, in: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024, OpenReview.net, 2024. URL: https://openreview.net/forum?id=GEcwtMk1uA.

[28] Q. Tang, Z. Deng, H. Lin, X. Han, Q. Liang, B. Cao, L. Sun, Toolalpaca: Generalized tool learning for language models with 3000 simulated cases, CoRR abs/2306.05301 (2023). doi:10.48550/ARXIV.2306.05301.

[29] Z. Guo, S. Cheng, H. Wang, S. Liang, Y. Qin, P. Li, Z. Liu, M. Sun, Y. Liu, Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models, in: Findings of the Association for Computational Linguistics: ACL 2024, Bangkok, Thailand, August 11-16, 2024, Association for Computational Linguistics, 2024, pp. 11143–11156. doi:10.18653/V1/2024.FINDINGS-ACL.664.

[30] R. Wang, G. Todd, Z. Xiao, X. Yuan, M. Côté, P. Clark, P. Jansen, Can language models serve as text-based world simulators?, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024, Association for Computational Linguistics, 2024, pp. 1–17. doi:10.18653/V1/2024.ACL-SHORT.1.

[31] Y. Liu, T. Hui, W. Zhang, L. Sun, N. Su, J. Wang, S. Su, Envsimbench: A benchmark for evaluating and improving llm-based environment simulation, CoRR abs/2605.07247 (2026). doi:10.48550/ARXIV.2605.07247.

[32] R. Fang, S. Cai, B. Li, J. Wu, G. Li, W. Yin, X. Wang, X. Wang, L. Su, Z. Zhang, S. Wu, Z. Tao, Y. Jiang, P. Xie, N. Zhang, F. Huang, W. Zhang, J. Zhou, Towards general agentic intelligence via environment scaling, in: Findings of the Association for Computational Linguistics: ACL 2026, San Diego, CA, USA, July 2026, Association for Computational Linguistics, 2026, pp. 17610–17621. doi:10.18653/V1/2026.FINDINGS-ACL.872.

[33] X. Song, H. Chang, G. Dong, Y. Zhu, J. Wen, Z. Dou, Envscaler: Scaling tool-interactive environments for LLM agent via programmatic synthesis, in: Findings of the Association for Computational Linguistics: ACL 2026, San Diego, CA, USA, July 2026, Association for Computational Linguistics, 2026, pp. 8326–8357. doi:10.18653/V1/2026.FINDINGS-ACL.407.

[34] Z. Wang, C. Xu, B. Liu, Y. Wang, S. Han, Z. Yao, H. Yao, Y. He, Agent world model: Infinity synthetic environments for agentic reinforcement learning, CoRR abs/2602.10090 (2026). doi:10.48550/ARXIV.2602.10090.

[35] T. Silver, R. Chitnis, PDDLGym: Gym environments from PDDL problems, in: ICAPS 2020 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL), 2020. doi:10.48550/ARXIV.2002.06432.

[36] K. Li, S. Zhang, H. Wang, F. Yu, Z. Sheng, Y. Gu, W. Ming, L. Xue, C. Liu, S. Hu, R. Chen, S. Lin, Y. Hou, X. Mou, Y. Xu, Smh-bench: Benchmarking LLM agents for environment-grounded reasoning and action in smart homes, CoRR abs/2606.01912 (2026). arXiv:2606.01912.

[37] Z. Zheng, Y. Zhang, M. R. Lyu, Investigating qos of real-world web services, IEEE Trans. Serv. Comput. 7 (2014) 32–39. doi:10.1109/TSC.2012.34.

[38] E. Al-Masri, Q. H. Mahmoud, Investigating web services on the world wide web, in: J. Huai, R. Chen, H. Hon, Y. Liu, W. Ma, A. Tomkins, X. Zhang (Eds.), Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008, ACM, 2008, pp. 795–804. doi:10.1145/1367497.1367605.

[39] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: I. Bahar, M. Herlihy, E. Witchel, A. R. Lebeck (Eds.), Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019, ACM, 2019, pp. 3–18. doi:10.1145/3297858.3304013.

[40] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, D. Ding, Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study, IEEE Trans. Software Eng. 47 (2021) 243–260. doi:10.1109/TSE.2018.2887384.

[41] X. Wang, D. Rong, H. Xu, X. He, Z. Wang, HSC: an artificial intelligence service composition dataset from hugging face, in: W. Gaaloul, M. Sheng, Q. Yu, S. Yangui (Eds.), Service-Oriented Computing - 22nd International Conference, ICSOC 2024, Tunis, Tunisia, December 3-6, 2024, Proceedings, Part II, volume 15405 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 225–239. doi:10.1007/978-981-96-0808-9_17.

[42] R. D. Pesl, J. G. Mathew, M. Mecella, M. Aiello, Retrieval-augmented generation for service discovery: Chunking strategies and benchmarking, IEEE Trans. Serv. Comput. 19 (2026) 1520–1534. doi:10.1109/TSC.2026.3665441.

[43] B. Caglayan, M. Wang, J. D. Kelleher, S. Fei, G. Tong, J. Ding, P. Zhang, BIS: NL2SQL service evaluation benchmark for business intelligence scenarios, in: W. Gaaloul, M. Sheng, Q. Yu,

S. Yangui (Eds.), Service-Oriented Computing - 22nd International Conference, ICSOC 2024, Tunis, Tunisia, December 3-6, 2024, Proceedings, Part II, volume 15405 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 357–372. doi:10.1007/978-981-96-0808-9_27.

[44] R. D. Pesl, C. Mombrey, K. Klein, D. Zyberaj, I. Georgievski, S. Becker, G. Herzwurm, M. Aiello, Compositio prompto: An architecture to employ large language models in automated service computing, in: W. Gaaloul, M. Sheng, Q. Yu, S. Yangui (Eds.), Service-Oriented Computing - 22nd International Conference, ICSOC 2024, Tunis, Tunisia, December 3-6, 2024, Proceedings, Part II, volume 15405 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 276–286. doi:10.1007/978-981-96-0808-9_20.

[45] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, D. S. Nau, HTN planning for web service composition using SHOP2, J. Web Semant. 1 (2004) 377–396. doi:10.1016/J.WEBSEM.2004.06.005.

[46] M. Klusch, A. Gerber, M. Schmidt, Semantic web service composition planning with owls-xplan, in: T. R. Payne, V. Tamma (Eds.), Agents and the Semantic Web, Papers from the 2005 AAAI Fall Symposium, Arlington, Virginia, USA, November 4-6, 2005, volume FS-05-01 of *AAAI Technical Report*, AAAI Press, 2005, pp. 55–62. URL: https://www.aaai.org/Library/Symposia/Fall/2005/fs05-01-008.php.

[47] M. Klusch, K. Renner, Fast dynamic re-planning of composite OWL-S services, in: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology - Workshops, Hong Kong, China, 18-22 December 2006, IEEE Computer Society, 2006, pp. 134–137. doi:10.1109/WI-IATW.2006.72.

[48] C. Petrie, T. Margaria, H. Lausen, M. Zaremba (Eds.), Semantic Web Services Challenge, Results from the First Year, volume 8 of *Semantic Web And Beyond*, Springer, 2009. doi:10.1007/978-0-387-72496-6.

[49] E. Korkan, S. Käbisch, M. Kovatsch, S. Steinhorst, Sequential behavioral modeling for scalable iot devices and systems, in: H. D. Patel, T. J. Kazmierski, S. Steinhorst (Eds.), 2018 Forum on Specification & Design Languages, FDL 2018, Garching, Germany, September 10-12, 2018, IEEE, 2018, pp. 5–16. doi:10.1109/FDL.2018.8524065.

[50] F. Salama, E. Korkan, S. Käbisch, S. Steinhorst, Towards a behavioral description of cyber-physical systems using the thing description, in: DAI-SNAC@CoNEXT 2021: Proceedings of the 2021 Workshop on Descriptive Approaches to IoT Security, Network, and Application Configuration, Virtual Event, Munich, Germany, 7 December 2021, ACM, 2021, pp. 6–9. doi:10.1145/3488661.3494030.

[51] M. Freund, D. Schraudner, S. Schmid, C. Stade, T. Wehr, A. Harth, The SPA ontology: Towards a web of things ready for robotic agents, in: B. Sartini, J. Raad, P. Lisena, A. Meroño-Peñuela, M. Beetz, I. Blin, P. Cimiano, J. de Berardinis, S. Gottschalk, F. Ilievski, N. Jain, J. Kim, M. Kümpel, E. Motta, I. Tiddi, J. Töberg (Eds.), Joint Proceedings of the ESWC 2024 Workshops and Tutorials co-located with 21th European Semantic Web Conference (ESWC 2024), Hersonissos, Greece, May 26-27, 2024, volume 3749 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3749/akr3-02.pdf.

[52] S. Schmid, Z. Gui, M. Freund, T. Wehr, A. Harth, AEPIS: Agent-enabled planning at scale, in: Knowledge Capture Conference 2025, K-CAP 2025, Dayton, OH, USA, December 10–12, 2025, ACM, 2025, pp. 5–8. doi:10.1145/3731443.3771341.

[53] M. Noura, M. Gaedke, An automated cyclic planning framework based on plan-do-check-act for web of things composition, in: P. Boldi, B. F. Welles, K. Kinder-Kurlanda, C. Wilson, I. Peters, W. M. Jr. (Eds.), Proceedings of the 11th ACM Conference on Web Science, WebSci 2019, Boston, MA, USA, June 30 - July 03, 2019, ACM, 2019, pp. 205–214. doi:10.1145/3292522.3326044.

[54] H. B. Hassine, E. Korkan, S. Steinhorst, Virtual-thing: Thing description based virtualization, 2019. URL: https://arxiv.org/abs/1909.03297. arXiv:1909.03297.

[55] F. Salama, E. Korkan, S. Käbisch, S. Steinhorst, Stateful-wot: Capturing the behavior of highly dynamic cyber-physical systems, in: IEEE International Conference on Omni-layer Intelligent Systems, COINS 2024, London, United Kingdom, July 29-31, 2024, IEEE, 2024, pp. 1–8. doi:10.1109/COINS61597.2024.10622288.

[56] F. Salama, F. J. Ennemoser, R. Binkert, E. Korkan, S. Käbisch, S. Steinhorst, Llm-mage: A generative mashup planner for the web of things, in: H. Verma, A. Bozzon, A. Mauri, J. Yang (Eds.), Web Engineering - 25th International Conference, ICWE 2025, Delft, The Netherlands, June 30 - July 3, 2025, Proceedings, volume 15749 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 342–357. doi:10.1007/978-3-031-97207-2_26.

[57] L. Kinder, T. Käfer, Static and adaptive planning with wot TD by generating python objects as intermediary representations using large language models, in: A. Meroño-Peñuela, Ó. Corcho, P. Groth, E. Simperl, V. Tamma, A. G. Nuzzolese, M. Poveda-Villalón, M. Sabou, V. Presutti, I. Celino, A. Revenko, J. Raad, B. Sartini, P. Lisena (Eds.), The Semantic Web: ESWC 2024 Satellite Events - Hersonissos, Crete, Greece, May 26-30, 2024, Proceedings, Part II, volume 15345 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 256–273. doi:10.1007/978-3-031-78955-7_21.

[58] E. Warburton, A. Elhabbash, S. Ezzini, Y. Elkhatib, The iot whisperer: A framework for intelligent iot service composition through llms, in: 18th IEEE International Conference on Cloud Computing, CLOUD 2025, IEEE, 2025, pp. 54–64. doi:10.1109/CLOUD67622.2025.00016.

[59] L. Kinder, T. Käfer, Towards improving large language models' planning capabilities on wot thing descriptions by generating python objects as intermediary representations, in: B. Sartini, J. Raad, P. Lisena, A. Meroño-Peñuela, M. Beetz, I. Blin, P. Cimiano, J. de Berardinis, S. Gottschalk, F. Ilievski, N. Jain, J. Kim, M. Kümpel, E. Motta, I. Tiddi, J. Töberg (Eds.), Joint Proceedings of the ESWC 2024 Workshops and Tutorials co-located with 21th European Semantic Web Conference (ESWC 2024), Hersonissos, Greece, May 26-27, 2024, volume 3749 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3749/akr3-06.pdf.