

Implementing Agents in JavaScript

Timotheus Kampik^[0000-0002-6458-2252]

Abstract This chapter gives an introduction to agent-oriented programming in JavaScript. It provides an example-based walk-through of how to implement abstractions for reasoning loop agents in *vanilla* JavaScript. The initial example is used as a stepping stone for explaining how to implement slightly more advanced agents and multi-agent systems using *JS-son*, a JavaScript library for agent-oriented programming. In this context, the chapter also explains how to integrate reasoning loop agents with generative AI technologies—specifically, large language models. Finally, application scenarios in several technology ecosystems and future research directions are sketched.

1 Introduction

Over the past decades, JavaScript has evolved from a language for simple animations in Web pages to one that is powering not only modern, complex front-ends of many applications in Web browsers, on desktop machines and mobile applications, but also substantial back-end systems, utilizing run-time environments such as Node.js. The language’s importance and, indeed, its outright dominance in some of the most prevalent technology ecosystems, makes using JavaScript—or languages compiling to it—a necessity in many software engineering contexts. Despite its popularity, which is also evidenced by programming language rankings [6], JavaScript still receives relatively little attention in contexts that are heavily academically influenced, possibly due to its perceived ugliness and its weak type system. This may explain why JavaScript is somewhat under-explored as a technology for implementing agents and Multi-Agent Systems (MAS). Still, as we demonstrate in this chapter, it is not only relatively easy to implement reasoning loop agents and MAS using JavaScript, its flexible and functional nature also caters to using the notion of an agent as a fundamental abstraction. We give a brief practical overview of the Agent-Oriented Programming (AOP) paradigm in Sec-

Department of Computing Science
Umeå University, 901 87 Umeå, Sweden, e-mail: tkampik@cs.umu.se

tion 2, followed by a conceptual overview of how AOP can be utilized for implementing agents in JavaScript (Section 3). To emphasize simplicity, we first implement agents from scratch, entirely in vanilla JavaScript, focusing on belief-plan deliberation, before we specify more elaborate agents using the JS-son library [10] (Section 4). We then move from single agents to MAS and agent-based simulations (Section 5). Highlighting broad application potential, we discuss several use cases with different deployment targets (Section 6). Finally, we conclude by sketching usage scenarios and future research directions, and by appealing to students, researchers, and practitioners alike to advance both the foundations and application of JavaScript agents (Section 7). Links to code examples and other resources are provided in the appendix. Note that while we relate to Large Language Models (LLMs) as integration components for reasoning loop agents, the concepts and design patterns we introduce are LLM-agnostic.

2 Agent-Oriented Programming

Agents are a (if not *the*) fundamental abstraction of Artificial Intelligence (AI), which is sometimes described as the “study of agents that receive percepts from the environment and perform actions” [18, p.viii]. The importance of the notion of an *agent* in both academic and practical computer programming contexts motivated researchers to coin the term *Agent-Oriented Programming* (AOP), initially as a specialization of Object-Oriented Programming (OOP) [19]. In contrast to objects in OOP, agents in AOP are proactive (as well as reactive), and run a *reasoning loop* process that revises their internal state—so-called *beliefs*—based on percepts obtained from their environment to ultimately execute plans and decide on actions that are then relayed to the environment. Several conceptualizations of reasoning loop agents have been proposed over the years; particularly prominent are so-called *belief-desire-intention* agents that move from beliefs to actions by first considering what they would like to achieve as potentially mutually inconsistent *desires*, before determining the *intentions* that a given agent actually strives towards realizing [17]. Over the past three decades, a heterogeneous ecosystem of AOP tools and languages has emerged¹. Many languages designed or used for AOP are *not* object-oriented: perhaps most notably, the *AgentSpeak* language and its dialects are Prolog-like, i.e., logic programming is used as a paradigm orthogonally to AOP [16, 3]. In order to facilitate applicability, researchers have integrated *AgentSpeak* into mainstream programming language ecosystems, e.g., for Java [3, 2] and Kotlin [1], and orthogonally into distributed environments such as micro service-based systems [14] and the Web [7]. However, it is still a challenge for practitioners to adopt AOP [11], arguably partially because the combination of AOP and logic programming—two paradigms that are not intuitively understood by typical software engineers—leads to an unnecessarily steep learning curve. Prominently, in the wake of the hype around Large Language Models (LLMs), proposals for designing so-called *AI agents* that integrate LLMs with reasoning loops in the broader sense (cf. [22] for a survey), largely ignore AOP and its rich history.

¹ For a recent survey of tools and platforms for programming agents and multi-agent systems, see [5].

The aforementioned challenges motivate the approach to AOP that we take in this chapter: the implementation of AOP design patterns as re-usable abstractions in and for a mainstream programming language that is of crucial importance for a broad range of application scenarios. We argue that the focus on JavaScript is particularly interesting, not only because of its tremendous real-world prevalence, but also because it is exactly *not* one of the eloquent and theoretically well-understood languages that academics prefer to study. In other words, we turn the approach that typical AOP approaches take on its head, starting with a mainstream programming language and adopting AOP-like abstractions to the extent we find practically useful. While this means that we sacrifice some of the formal and philosophical elegance of traditional AOP, it allows us to position AOP in a way that may be more intuitive for practitioners, as well as for students and researchers that primarily work with mainstream programming languages. To this aim, we also present our AOP approach to JavaScript both generically and as part of a somewhat *lean* library, with little code overhead and no external dependencies: while the library makes it easier to get started with implementing agents in JavaScript, serious applications may require custom implementations of agent-oriented abstractions; for the latter scenario, this chapter provides a conceptual overview as well as technical descriptions relating AOP directly to JavaScript.

3 Conceptual Overview

Before moving to hands-on implementation tutorials, we provide an overview of relevant abstractions for designing reasoning loop agents, already hinting at how these can potentially be implemented in JavaScript, and giving basic application intuitions. As a starting point, let us consider the reasoning loop architecture sketched in Figure 1. As we can see in the figure, our reasoning loop agent perceives its environment in that it receives *percepts*. The percepts are then, in conjunction with its existing *beliefs*, applied to *revise* the belief base. Based on the revised beliefs, the agent *deliberates* to determine which of its *plans* are activated (based on a given plan’s *head*, see below) to then execute the *bodies* of activated plans, yielding *actions*. Subsequently, the environment processes the actions, returning another set of percepts, and the cycle continues.

The selection of the abstraction is, in parts, opinionated and based on an engineering intuition of what is and is not useful for practical applications. The two overarching notions of *agents* and *environments* are obvious and uncontroversial choices.

Agents. As described above, agents process percepts, maintain beliefs and plans, activate plans based on current beliefs, and determine actions that are to be registered with the environment (by executing plans).

Example. An agent tasked with handling a person’s travel expenses and claims.

Environments. The environment processes actions of agents and generates agents’ percepts, thus also orchestrating interactions between agents.

Example. A middle-ware integrating our travel expense agent with different systems.

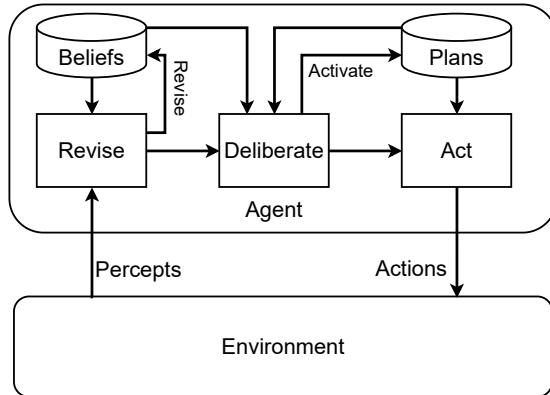


Fig. 1: A reasoning loop agent, interacting with a black-box environment (unlabeled arrows represent information transfer).

Both *agents* and *environments* are proper JavaScript objects, i.e., neither merely simple JSON objects nor functions: because agents and environments carry state, they cannot (at least not conveniently) be functions; because they must actively process inputs and generate outputs, they cannot be plain JSON objects (in the sense of the Internet Engineering Taskforce spec [4]), which only allow for *representation* but not for *reasoning*.

As the chapter focuses on the implementation of *agents* in JavaScript, we are mainly interested in *agent internal* abstractions and relegate—colloquially speaking—the role of the environment to plumbing and piping. Accordingly, we focus on agent internals in this initial conceptual overview². Let us first introduce the key notions of agent-internal state (as well as the state update that an agent receives from the environment), primarily dividing between *percepts* and *beliefs*, i.e., the agent’s world model, and *plans* and related notions that are the foundation of the agent’s specific reasoning behavior.

Percepts and Beliefs. Percepts and beliefs determine an agent’s model of the world: the agent receives percepts from the environment and updates its beliefs considering these percepts, to an extent the agent decides autonomously. In the JS-son library, percepts are, intuitively, *incoming* beliefs that agents merge with the already existing internal beliefs when applying the belief revision functions: i.e., percepts are not modeled using a separate abstraction, but instead are belief objects that an agent perceives and then revises before adding them to its belief base. In its most basic form, percepts and beliefs are static and can be modeled as (key, value)-tuples in a JSON object. However, in advanced scenarios beliefs may, in addition, have numeric priority values helping decide which of two conflicting beliefs should be adopted, or are functions themselves that dynamically infer values given other beliefs.

² Still, some important environment characteristics are discussed in Section 5.

Example. Expense claims that need to be handled and their meta-data, as well as documents that serve as proofs of actual spending.

Plans (with Head and Body), Goals, and Actions. Plans are (*activation condition*, *action inference function*)-tuples, allowing the agent to dynamically and autonomously decide what actions to take. Here, the activation condition is the *head* of the plan in the form of a Boolean function that takes as input an agent's beliefs. One may say that an activated plan models a *goal*. The body of a plan determines the plan's action(s), again potentially depending on the beliefs of the agents.

Example. Whether to gather more information about a claim and whether to submit one (heads), as well as where and how to gather information/submit claims (bodies).

Given its internal state and perception of the environment, an agent's reasoning loop consists of the following key steps.

Belief Revision (*revise*). The agent's belief revision function takes a set of incoming beliefs as perceived by the agent (hence: *percepts*), as well as the beliefs the agent currently holds, and returns a set of beliefs that forms the agent's new belief base. The key idea is that the agent has *autonomy* over its beliefs, using the belief revision function to decide how to process what it perceives in its environment. In Figure 1, belief revision is the *revise* step of the reasoning loop.

Example. Augmenting and revising the meta-data of a previously rejected claim.

Plan Deliberation (*deliberate* and *act*). The agent's plan deliberation function first checks which plans in an agent's plan library are active and then executes active plans, yielding *actions*. In Figure 1, plan deliberation is covered by the *deliberate* and *act* steps: *deliberate* activates plans based on their heads; *act* determines actions that are then registered with the environment, based on the bodies of activated plans.

Example. Deciding what to request and submit, and where.

Table 1 provides an overview of the AOP abstractions and how they can be modeled in JavaScript. Also, the table indicates whether the corresponding abstraction is an explicit, first-class abstraction in our AOP approach (and thus in the JS-son library) or rather modeled implicitly, as part of another abstraction. Let us highlight that the table does not reflect the details of the function signatures and object definitions of the JS-son library, as it is aimed at a higher level of conceptual abstraction. Technical details are available in the JS-son documentation³. Notably, we skip the additional capabilities for goal and desire deliberation that JS-son supports, in order to increase ease-of-understanding and, ultimately, applicability.

4 Implementing Agents

JavaScript is a language that adopts both functional and object-oriented notions as key abstractions. When implementing JavaScript agents, we can utilize this dichotomy: roughly speaking, an agent's reasoning loop can be defined as a composition of (somewhat impure) functions, whereas the beliefs it maintains are objects. From the belief

³ Cf. Appendix Item 1.

Abstraction	Modeled As	First-Class?	Covered By
Agent	JavaScript Object	✓	
Environment	JavaScript Object	✓	
Belief	JSON (Key, Value)-tuple	✓	
Percept	JSON (Key, Value)-tuple	✗	Belief
Plan	f -tuple: (Head, Body)	✓	
Plan Head	$f: \text{Beliefs} \rightarrow \text{Boolean}$	✓	
Plan Body	$f: \text{Beliefs} \rightarrow \text{Actions}$	✓	
Goal	$f: \text{Beliefs} \rightarrow \text{Boolean}$	✗	Plan Head
Action	JSON Object	✗	Plan
Belief Revision	$f: \text{Beliefs}, \text{Percepts} \rightarrow \text{Beliefs}$	✓	
Plan Deliberation	$f: \text{Beliefs}, \text{Plan} \rightarrow \text{Boolean}$	✗	Agent

Table 1: First-Class (explicit) and second-class (implicit) abstractions of JS-son JavaScript agents; f stands for *function*.

base, the agent can draw inferences and turn them into actions, again utilizing functions that are maintained in a plan library object. This intuition yields a *belief-plan* agent, i.e., a rather simple reasoning loop agent. It is relatively straight-forward to implement a belief-plan agent and its environment from scratch, which we do in Subsection 4.1. In Subsection 4.2, we make use of the JS-son library to implement a slightly more involved agent that utilizes an LLM. The purpose of starting with an example that implements an agent from scratch is to relay a nuanced understanding of agent programming in JavaScript, avoiding any potential obfuscation by non-standard abstractions. The tutorial can help programmers working with “ordinary” JavaScript, alongside mainstream libraries and frameworks, to think in an agent-oriented manner and to ultimately implement their own agent-oriented abstractions. However, readers who wish for a more colloquial tutorial that helps them get started with agent-programming with re-usable abstractions in an easy-going manner may jump to Subsection 4.2 right away.

4.1 A Simple Agent in Vanilla JavaScript

As a minimal example, consider a *porter* agent⁴ that observes whether a door is locked or not, and opens and closes the door based on incoming requests, given the requested state of the door is currently not satisfied.

First, we model the *beliefs* of the agent.

```
const beliefs =
  { door: { locked: true },
    requests: [] }
```

⁴ Cf. Appendix Item 2.

As we can see, the initial state of the belief base models that the door is currently locked, and that no requests are to be handled. Let us now specify the agent's plans: *i*) if the door is not locked and a request to lock the door exists (*head*) then the agent should act to lock the door (*body*); *ii*) if the door is locked and a request to unlock the door exists (*head*) then the agent should act to unlock the door (*body*).

```
const plans = [
  { head:
    beliefs =>
      !beliefs.door.locked &&
      beliefs.requests.includes('lock'),
    body: () => 'lock' },
  { head: beliefs =>
    beliefs.door.locked &&
    beliefs.requests.includes('unlock'),
    body: () => 'unlock' }
]
```

With beliefs and plans specified, we can move on to implement the agent's reasoning loop. We start by implementing the belief revision function, which updates the agent's belief about the state of the door, removes requests that have been successfully executed, and adds new requests that have come in.

```
const revise = (beliefs, percepts) => {
  return {
    door: percepts.door,
    requests: beliefs.requests.filter(
      request =>
        !percepts.executions.includes(request)
    ).concat(percepts.requests)
  }
}
```

Next, we implement the plan deliberation function. The function first determines which bodies of the plans should be activated (*deliberate*) to then execute these bodies (*act*).

```
const deliberate = (beliefs, plans) => {
  return activatedBodies = plans.filter(
    plan =>
      plan.head(beliefs)
  ).map(plan => plan.body(beliefs))
}
```

Finally, we can complete the agent, by *i*) assigning beliefs and plans, as well as the revision and deliberation functions to an agent object, and *ii*) implementing a run function that triggers first belief revision and then plan deliberation.

```
const agent = {
  beliefs,
  plans,
  revise,
  deliberate,
  get run() {
    this.beliefs = this.revise(this.beliefs, percepts)
    return this.deliberate(
      this.beliefs,
      this.plans
    )
  }
}
```

We can observe that we need to make use of a `get` function to access object-level properties and that `percepts` will need to be specified outside of the scope of the

agent object or any of its functions. This indicates that providing generic abstractions in the form of a library may be useful.

We still need to implement the environment that provides our agent with percepts and processes its actions.

```
const environment = {
    agent,
    state: {
        door: { locked: true },
        requests: []
    },
    get run() {
        [...Array(steps).keys()].forEach(
            step => {
                this.state.requests =
                    Math.random() < 0.5 ?
                        ['lock'] :
                        ['unlock']
                percepts = this.state
                const exec = this.agent.run
                const actions = exec ? exec : []
                if (actions.includes('lock'))
                    this.state.door.locked = true
                if (actions.includes('unlock'))
                    this.state.door.locked = false
                this.state.executions = actions
            }
        )
    }
}
```

Our environment processes all agent actions by changing the state of the door accordingly and adds pseudo-random requests to either lock or unlock the door, independently of the door's current status. In this way, the environment primitively simulates the presence of additional agents. Note that in this simple example, our environment is a *single agent environment*, i.e., it is not programmed to handle more than one agent. We again exploit the `get` syntax in a way that is arguably not idiomatic.

We can now execute the environment, first specifying the number of environment-agent interactions (`steps`) it should run.

```
const steps = 20
environment.run
```

When running the example, we see that our porter agent processes the requests it receives from the environment as intended, generating the desired actions, which are then further processed accordingly by the environment⁵. When debugging nuances of the example, we observe that nonsensical requests (the locking of locked doors and the unlocking of unlocked doors) “pile up” until they can be meaningfully addressed.

As we have seen, it is relatively straightforward to implement a simple reasoning loop agent in JavaScript. Still, even in this simple example, in which a purely reactive agent handles a straightforward task whose completion merely requires a single reasoning cycle, we see that we can implement abstractions that *i*) guide developers so that they implement agents in a somewhat idiomatic manner and *ii*) reduce overhead and duplicate work during the implementation. Below, we make use of such abstractions to implement a slightly more involved agent.

⁵ Cf. Appendix Item 3.

4.2 An LLM Agent Based on Reusable Abstractions

We implement a reasoning loop agent utilizing an LLM in the following scenario: Our agent acts on behalf of a student who has forgotten to do their homework and aims to submit an excuse request to the environment. The environment either accepts or rejects the request and gives feedback regarding why a request was denied. As the student has several attempts to submit the request, their agent adjusts the excuse request for subsequent attempts based on feedback that the environment has provided.

We first implement the agent's beliefs, by instantiating JS-son `Belief` objects.

```
const beliefs = {
  ...Belief('rejectExps', []),
  ...Belief('excuseAccepted', false),
  ...Belief('name', 'Bart'),
  ...Belief('teacherName', 'Edna Krabappel')
}
```

In our case, the result is roughly the same as when specifying beliefs as JavaScript objects. However, in advanced scenarios, JS-son beliefs have in-built support for specifying revision behavior, such as priorities and inter-belief dependencies, where the value of one belief is inferred from the value(s) of one or several other beliefs.

For the integration with the LLM, we assume a `model` object with a `generatePrompt` function that takes a text as input and returns a `promise` (for asynchronous function call handling), ultimately yielding text output. In our implementation, we rely on the *Gemini LLM* [20], developed and provided by Google.

We specify a prompt template for instantiating a prompt based on the agent's beliefs, considering the agent's name (the name of the person the agent represents), the name of the teacher, and information about why previous excuse attempts (if any) have failed.

```
const genPrompt = beliefs =>
`Can you write a charming yet convincing excuse
for a student who forgot their homework?
The names of teacher and student are
${beliefs.teacherName}, and ${beliefs.name},
respectively (i.e., sign the excuse with ${beliefs.name}).

Consider the following feedback
received from past rejected excuses:

${beliefs.rejectExps.map(exp => ` * ${exp}`).join('\n')}`
```

The prompt template is called in the body of the agent's single plan. The body is activated (by the plan's head) only if no excuse has been accepted so far. The body then constructs the excuse prompt and sends it to the LLM. When the agent has received the generated excuse, it returns the excuse to the environment.

```
Plan(
  beliefs => !beliefs.excuseAccepted,
  async beliefs => {
    const prompt = genPrompt(beliefs)
    const excusePromise =
      await model.generateContent([prompt])
    const excuse = excusePromise.response.text()
    return excuse
  }
)
```

Above, we have instantiated a JS-son `Plan` object; later, the agent will apply a default deliberation function, thus saving us some implementation overhead.

Next, we implement the agent's belief revision function. JS-son's default revision function simply merges percepts into the agent's belief base, with percepts taking precedence over beliefs in case of conflicts. However, for our agent we need a more specific belief revision function that *i*) adds new rejection explanations (if novel) to our array of explanations and *ii*) updates the `excuseAccepted` belief based on the percepts.

```
const reviseBeliefs = (beliefs, percepts) => {
  const rejectExps =
    percepts.rejectExp &&
    !beliefs.rejectExps.includes(percepts.rejectExp) ?
      Array(...beliefs.rejectExps, percepts.rejectExp) :
      beliefs.rejectExps
  return percepts.excuseAccepted ? {
    ...beliefs,
    excuseAccepted: true,
  } : {
    ...beliefs,
    rejectExps
  }
}
```

With our beliefs, plan, and belief revision function, we can then instantiate a JS-son Agent object.

```
const agent = new Agent({ id: 'student', beliefs, [plan], reviseBeliefs })
```

Note that the agent's runner function (next in JS-son terms) is built in, i.e., we do not need to implement it explicitly.

As a prerequisite to implementing the environment, we need to specify the `updateState` function, of which we merely provide a sketch, abstracting from the exact checks executed when determining whether an excuse is acceptable or not.

```
const updateState = async(actions, _, currentState) => {
  const excuse = await actions[0]
  if(!excuse)
    return { ...currentState, rejectExp: '' }
  const state = {
    excuseAccepted: false
  }
  if (...) { // apply rule to analyze excuse text
    state.rejectExp = ... // explain rule violation
  } else if (...) { // apply another rule to analyze excuse text
    state.rejectExp = ... // explain another rule violation
  } else {
    state.excuseAccepted = true
  }
  return state
}
```

As we can see, the function essentially applies a set of rules to assess whether the provided excuse is acceptable or not and, in case of a rule violation, provides an explanation for rejection (for the first rule that is violated, which terminates rule execution). Then, the state of the environment is updated accordingly (which implicitly leads to a corresponding update of the agent's percepts).

Finally, we specify the environment, using the agent, state update function, and an object defining the initial state (excuse not accepted and no explanations for rejections provided so far).

```
const environment = new Environment(
  [agent],
```

```

    {   excuseAccepted: false,
        rejectExp: '' },
    updateState
)
setInterval(() => environment.run(1), 3000)

```

Again, we do not explicitly implement a runner function, as this is provided by default by JS-son's `environment` object. We run the environment in timed intervals. This reduces load on the LLM service API, but also smoothens the execution of a JS-son reasoning loop with asynchronous behavior—in contrast to asynchronous environments, asynchronous behavior within reasoning loops is not supported in a straight-forward, stable manner by JS-son at the time of writing⁶.

While the scenario above has some simplistic aspects to it (such as the application of a few hard-coded rules for the analysis of natural language text), one can argue that it goes into the direction of somewhat realistic use cases. For example, reasoning loop agents utilizing LLMs may be used in the future in the context of Robotic Process Automation (RPA), i.e., for the automation of simple form submissions and related tasks human users tend to find tedious, and LLMs may help brushing over some shortcomings in the corresponding form requirements and system interfaces. At the same time, the example also hints at the fact that agents utilizing LLMs may be employed for socially questionable use-cases that do not lead to long-term sustainable progress.

5 Implementing Multi-Agent Systems

Obviously, agents typically do not only interact with their environment and passive artifacts therein, but also with other agents. Hence, in multi-agent scenarios, the environment serves as a middle-ware, relaying communications between agents. In some centralized scenarios, such as simple games and agent-based simulations, the environment may serve agents in a *round robin* manner, first providing percepts to agents one-by-one, while always waiting for the termination of their reasoning loop to register their actions; the actions are then processed in a batch, i.e., the environment's state update is executed after all agents have acted (Figure 2a). However, in more realistic applications, potentially (logically or physically) distributed agents must be able to act asynchronously. Then, the environment processes agent actions immediately when they are received, *i.e.*, asynchronously across agents (Figure 2b). Below, we sketch two multi-agent examples, implementing centralized (synchronous, Subsection 5.1) and logically distributed (asynchronous, Subsection 5.2) MAS, respectively. For the sake of conciseness, we provide code examples to the extent they are illustrative and point to online material for the complete program code.

⁶ Cf. Appendix Item 4.

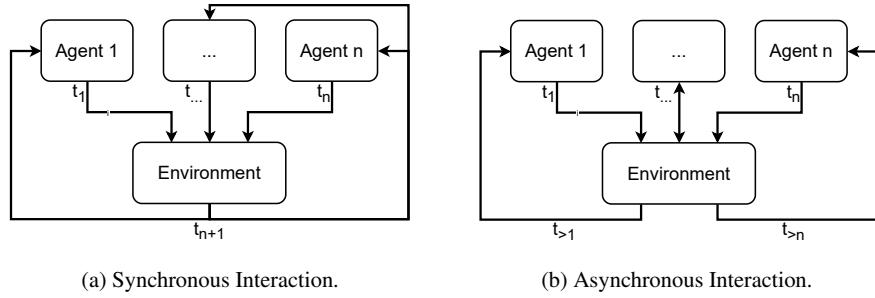


Fig. 2: Multi-agent environments: synchronously scheduled environment, for round-robin based simulations or games vs. asynchronous environments for distributed MAS.

5.1 Centralized Multi-Agent Systems

As our *centralized* example, we implement a *game of life* multi-agent system that runs in a Web browser. Conway’s *Game of Life* (GoL) is a well-known example of a simplistic multi-agent simulation, formally grounded in cellular automata, illustrating how intriguing patterns can emerge from societies of agents that each individually follow very simple rules [8]. Specifically, a GoL is a grid of agents, each of which is either *active* or *inactive* and changes, in discrete time, its status (from active to inactive or vice versa) based on its own current status and the statuses of its direct neighbors:

- If the agent is already active, it will stay so if at least two and at most three of its neighbors are active.
 - Otherwise, the agent turns active only if exactly three of its neighbors are active.

Intuitively, agents thrive in areas that are neither underpopulated nor overcrowded.

We can implement the agent behavior we have sketched above as a single plan.

```
Plan() => true,
beliefs => {
    const neighborActivity =
        determineNeighborActivity(
            beliefs.index, beliefs.activityArray
        )
    const isActive =
        beliefs.activityArray[beliefs.index]
    return (
        isActive &&
        neighborActivity >= 2 &&
        neighborActivity < 4
    ) || neighborActivity === 3 ?
        { nextRound: 'active' } :
        { nextRound: 'inActive' }
})
```

Note that the plan is always active (its head necessarily evaluates to true).

The plan's body depends on the following beliefs:

- The agent's index in the grid;
- The activity statuses of all agents (i.e., assigned by their indices in the grid).

Also, the agent makes use of the inference rule/function

`determineNeighborActivity` that, given its index and the activity statuses of all agents, determines the number of active neighbors.

Given our plan, we can generate the entire grid of agents by using a simple function that assigns the agents' initial beliefs based on an `initialActivity` array that specifies the initial state of the agents (for example, in order to guarantee the emergence of a specific pattern).

```
const generateAgents = initialActivity =>
  initialActivity.map((value, index) => {
    const beliefs = {
      ...Belief('index', index),
      ...Belief('activityArray', initialActivity)
    }
    return new Agent(index, beliefs, {}, [plan])
  })
}
```

Next, we need to specify the environment's behavior. Here, it is crucial to assure that updates are propagated *at the end of a given round* (cf. Figure 2a). Accordingly, our environment distinguishes between *previous* and *next* agent activity statuses, right from the start when specifying the initial state.

```
const generateState = initialActivity => ({
  previousActivity: initialActivity,
  nextActivity: []
})
```

Accordingly, the environment's state update is propagated only when the last agent in a grid has submitted its action.

```
const updateState = (actions, agentId, currentState) => {
  const stateUpdate = { ...currentState }
  const agentActive =
    actions.some(action =>
      action.nextRound === 'active'
    )
  stateUpdate.nextActivity.push(agentActive)
  if (agentId === currentState.previousActivity.length - 1) {
    return {
      previousActivity: stateUpdate.nextActivity,
      nextActivity: []
    }
  }
  return stateUpdate
}
```

We also specify a state filter that exposes the agents only to the grid's “previous” status, i.e., not to the status that is currently undergoing an update.

```
const stateFilter = state => (
  { activityArray: state.previousActivity }
)
```

Finally, we can generate the GoL's initial activity grid and instantiate the environment.

```
const initialActivity = generateInitialActivity()
const env = new Environment(
  generateAgents(initialActivity),
```

```

    generateState(initialActivity),
    updateState,
    render,
    stateFilter
)

```

With this we have finalized the specification of the GoL. We can, for example, attach the MAS to a rendering function that produces/manipulates HTML elements and run it in a Web browser⁷.

5.2 Distributed Multi-Agent Systems

Next, we describe how to implement a simple logically distributed MAS, wherein agents interact asynchronously. For this, we again go back to our initial *porter* agent example. This time, however, we implement an MAS and not a single-agent system, featuring two agents in addition to the porter:

- A *paranoid* agent, who always requests that the door be locked when it is unlocked;
- A *claustrophobic* agent, who always requests that the door be unlocked when it is locked.

The porter agent always satisfies the latest request issued by either agent and thus iteratively unlocks and locks the door (until the MAS is terminated). In our example, the porter and paranoid agents, as well as the environment, run on a Web socket server, whereas the claustrophobe agent runs on a client. As some of the agents and the environment are executed in a single thread, we may technically consider the scenario a *hybrid* case (a mix of Figures 2a and 2b) instead of a fully distributed case. In order to manage interactions with the claustrophobe, the environment maintains a *shadow agent*, which serves as an interface to the client, abstracting away the details of the client and the agent running on it. Like this, it is in principle possible to integrate JS-son MAS with agents implemented using other libraries and approaches.

As the agents are straightforwardly implemented, analogously to how the porter agent is implemented as described in Subsection 4.1 (though here using the JS-son library and not vanilla JavaScript), we focus below on what is required for managing the distributed nature of the MAS. This means we assume that on server side we have:

- The two local agents, *porter* and *claustrophobe*, who have the same simple set of beliefs (*beliefs*) about the environment, cf. Subsection 4.1;
- The environment's state (*state*) and state update function (*update*);
- *ws*, a Web socket connection object that allows us to listen to open connections (*.on('open', ...)*) and messages (*.on('message', ...)*), and send messages (*.send(...)*).

What remain to be implemented are the shadow agent, environment-client communications, and the client agent.

⁷ Cf. Appendix Item 5.

Let us start with the former. For this, we implement a higher-order function that generates our custom runner function that we will give to our shadow agent. The function the generator returns will send the belief update to the remote agent (instead of applying it to a locally running instance).

```
const nextGen = agentId => function next (beliefs) {
  ws.send(JSON.stringify(beliefs))
  return global.actionRequests[agentId]
}
```

Now, we can instantiate the shadow agent, using JS-son's `RemoteAgent` abstraction.

```
const claustrophobe = new RemoteAgent(
  'claustrophobe',
  beliefs,
  nextGen(agentId)
)
```

Let us move to environment-client communications. For this, we first implement a higher-order function generating a `runner` for the environment. Because we need the Web socket client to be available in the context of this function, we create the 3rd-order function `runnerGen`. The `runner` higher-order function provides a context in which the environment's `run` function is executed. In our case, an execution cycle of the environment is triggered when a message of our remote agent arrives via the Web socket connection.

```
const runnerGen = ws => run => () => {
  ws.on('message', message => {
    const jMessage = JSON.parse(message)
    if (jMessage.agentId && jMessage.actions)
      global.actionRequests[jMessage.agentId] = jMessage.actions
    run()
  })
}
```

On server side, we now have all components for implementing the environment (and running it).

```
const environment = new Environment(
  [paranoid, claustrophobe, porter],
  state,
  update,
  state => console.log(state),
  state => state,
  runnerGen(ws)
)
environment.run()
```

Finally, we implement the client agent. Again, we abstract from agent-internal details, assuming we have a `claustrophobe` agent, whose reasoning loop we can execute.

To enable the Web socket server integration, we instantiate a client and send an initial opening request to the server to trigger the execution of the environment's loop.

```
ws.on('open', () => {
  ws.send(JSON.stringify({ agentId, actions: [[]] }))
})
```

Finally, we implement the Web socket listener that receives messages from the environment and triggers the agent's reasoning loop using these messages as the percepts.

```
ws.on('message', message => {
  const jMessage = JSON.parse(message)
  if (jMessage === Object(jMessage)) {
```

```

    const actions = claustrophobe.next(jMessage)
    ws.send(JSON.stringify({ agentId, actions }))
  }
})

```

We can then start server and client to run the MAS (leading to a loop in which the porter agent iteratively attempts to satisfy the wishes of the two other agents)⁸.

6 Applicability in Different Ecosystems

Given the ubiquity of the language, JavaScript agents can be deployed across a broad range of environments and devices. Below, we sketch the relevance of, and some example applications for, JavaScript agents in different ecosystems that can inspire more elaborate applications, as well as in-depth research on related open challenges.

Agents in Web browsers and other client-side environments. The most classical deployment target for JavaScript agents is the Web browser. We have show-cased a simple application in Section 4. In more serious application scenarios, JavaScript agents and MAS may, for example, run in client environments in order to execute simulations that run directly on user devices, thus off-loading resource consumption from server to client, and potentially facilitating share-ability, e.g., via Web links. An example of such an agent-based simulation is presented in [12, 13] (implemented in JS-son); the simulation has been used in a human-computer interaction study. Analogously, JavaScript agents can be deployed to desktop and mobile applications using frameworks like Electron⁹ and Ionic¹⁰, respectively.

Agents in Node.js. The implementation of the *porter* and *LLM* examples in Sections 4 and 5 run on Node.js, which is frequently used for executing JavaScript on Web servers (or as a build tool). More broadly, with Node.js, JavaScript agents can be deployed to most servers and personal computing devices, e.g. devices that run Windows, MacOS, or common Linux distributions.

Agents in Jupyter notebooks. With the rise of data science as a practical discipline, technologies that blend programming and data analysis work, such as Jupyter notebooks, have become important tools for scientists as well as for data analysts in industry. An example of a simple integration of JS-son-based JavaScript agents and Jupyter notebooks is presented in [10], highlighting the interplay of Python and JavaScript¹¹. Future work could investigate more advanced use cases involving additional, mature technologies for fully integrating JavaScript into data science environments, such as Deno¹².

⁸ Cf. Appendix Item 6.

⁹ <https://github.com/electron/electron>, accessed at 2025-02-13.

¹⁰ <https://github.com/ionic-team/ionic-framework>, accessed at 2025-02-13.

¹¹ Cf. Appendix Item 7.

¹² <https://docs.deno.com/runtime/reference/cli/jupyter/>, accessed at 2025-02-13.

Agents in Function-as-a-Service environments. JavaScript is frequently used in Function-as-a-Service (FaaS) environments. FaaS offerings facilitate the development and deployment of micro-services by allowing developers to run relatively small and often stateless programs in the cloud with minimal management and maintenance overhead, as the FaaS provider takes care of all abstractions below business logic level (as well as of infrastructure). An example tutorial showcasing agents—in this case a simple agent-based simulation—that can be deployed to FaaS environments is available in the JS-son project repository¹³.

LLM agents. One of the examples presented in Section 4 makes use of an LLM, thus relating to the emerging research trend of LLM agents, i.e., the integration of reasoning loop agents (in the broader sense) and LLMs. Arguably, common narratives around LLM agents in academia and industry pay little attention to the history of research on reasoning loop agents and AOP. Accordingly, there may be a research gap between LLM agent design patterns and long-running research on AOP. Future research could analyze this gap comprehensively, potentially identifying ways in which AOP can address challenges of LLM agents. In this context, pragmatic, practice-oriented reasoning loop approaches like the ones presented in this chapter may be a first, preliminary starting point for bridging the gap.

Towards agents on constrained devices. A generally acknowledged research challenge at the intersection of AI and software systems engineering is moving somewhat “intelligent” behavior to the edge of networks, i.e., closer to the problems at hand but also to less resourceful, *constrained* computing devices. In this context, research has explored the deployment of agents and use of AOP [21, 15]. Indeed, agents implemented using a slimmed-down version of JS-son have experimentally been deployed to *Espruino*s, microcontroller-based devices that support the execution of JavaScript [9]. However, JS-son is not optimized for performance and resource utilization. Indeed, one can intuitively gauge that JS-son’s reasoning loop cycle with its nested and otherwise intertwined function calls can lead to inefficient programs. This is less of a problem in environments such as Web browsers, where resources are abundant and efficiency is, commonly, an afterthought. In contrast, on constrained devices, JavaScript agents compete with program code in lower-level languages, thus raising the bar in terms of performance requirements.

7 Conclusions

In this chapter, we have provided an overview of how to implement reasoning loop agents and multi-agent systems in JavaScript. We have highlighted how such JavaScript agents can be implemented with relative ease, either from scratch in vanilla JavaScript, or using *JS-son*, a small agent programming library. Such agents can then be straightforwardly deployed to a range of technology ecosystems, such as Web front-ends, Node.js back-ends and Jupyter notebook-based data science environments. We hope that our guide provides a useful starting point for implementing agents and MAS in JavaScript, for

¹³ Cf. Appendix Item 8.

researchers working on the foundations of agent technologies, applied scientists and practitioners who implement agents and MAS for practical purposes, as well as for educators teaching agent and multi-agent systems to diverse student groups. Specifically, yet not exhaustively, we recommend considering the application of JavaScript and JS-son agents in the following broad scenarios.

Teaching AOP to non-computer science students. Traditional AOP languages, such as AgentSpeak [16], adopt the logic programming paradigm, which arguably makes them difficult to learn for beginners who lack computer science knowledge. One can reasonably assume that JavaScript (or, alternatively, Python) is more accessible for “non-technical” students, who may, for example, start learning about artificial intelligence primarily from the perspectives of the social sciences and humanities, i.e., from disciplines such as philosophy and psychology. While Python may have emerged as the programming language of choice in such contexts, having another mainstream language available for teaching purposes not only allows broadening *polyglot* perspectives on programming languages but also facilitates the implementation of agents and MAS with a focus on different use cases, such as in *Web* contexts where the choice of Python as a programming language is less natural.

Creating shareable agent-based simulations. A key advantage of JavaScript-based simulations implemented with libraries like JS-son is that they can be deployed as “static” Web pages that do not require any back-end (neither a server, nor a “serverless” environment). This facilitates sharing of simulations (and entire simulators), for example as a means to provide interactive science demonstrators using agent technologies [12].

Implementing light-weight agents with heterogeneous deployment targets.

As demonstrated above, JavaScript reasoning loop agents can be deployed to a broad range of environment types, such as function-as-a-service providers, Web front-ends, and (possibly constrained) IoT devices. Use cases that require the deployment of conceptually somewhat similar agents to a range of these (and other) environment types may benefit from JavaScript and JS-son agents that can be easily moved between environments, potentially even at run-time.

Beyond these use-case suggestions for JavaScript agents, the conceptual part of this chapter may provide a starting point for closing the gap between reasoning loop agents in the traditional sense and the pragmatic implementation of agents in mainstream programming languages and software technology ecosystems. Still, the long-term ambition to establish agent-oriented abstractions in the mainstream of programming fundamentals requires continuous efforts. Accordingly, we consider the following two lines of future work particularly relevant.

Engineering design patterns for reasoning loop agents. With the emergence of LLM agents, *agent design patterns* have emerged as important practical abstractions¹⁴. An apparent advantage of these design patterns is their focus on easy-to-understand, generally applicable and purely conceptual/informal abstractions, and the resulting broad appeal to software engineers and even non-programmers. These

¹⁴ Cf. <https://www.anthropic.com/research/building-effective-agents>, accessed at 2025-02-21.

patterns tend to be detached from the academic AOP literature. Future research can explore *i*) to what extent similarly intuitive, high-level design patterns can be compiled from the AOP literature and *ii*) how the AOP literature can contribute to LLM agent design patterns and, more broadly, ground the current hype around agentic AI. In this context, one could set out to refine *design patterns* for reasoning loop agents in a language-agnostic manner, targeted towards practical desiderata.

Software frameworks for industry-scale JavaScript agents. The concepts and abstractions for implementing JavaScript agents presented in this chapter are rooted in engineering intuition and pragmatism. Still, some design choices may stray too far from what is considered *idiomatic* in either JavaScript and AOP, and can thus benefit from further refinement. On the entirely technology-oriented side, future work may set out to further mature so far somewhat “academic” frameworks and libraries. For example, improvements to JS-son—or to entirely novel JavaScript-based agent programming libraries—may provide better support for asynchronous behaviors within reasoning loops and could be optimized for modern tools and build systems that utilize JavaScript extensions such as TypeScript.

We hope that our practice-oriented overview of how to implement reasoning loop agents in JavaScript inspires students, researchers, and practitioners to try out agent-oriented abstractions in JavaScript, and vice versa, JavaScript as a language for the agent programming tool-box. Ultimately, we would wish that others develop their own approaches and idioms, and ideally even advance the research directions sketched above.

Acknowledgements The author thanks the many researchers in the agents and multi-agent systems community who have helped with feedback and discussions about engineering agents and multi-agent systems, in JavaScript, on the Web, and beyond. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Appendix

The appendix contains links to code examples and documentation, sometimes accompanied by brief remarks.

1. JS-son Documentation

The JS-son documentation is available at <https://js-son.readthedocs.io/en/latest/>, accessed at 2025-02-12.

2. Jason Room Example

Our example is based on one of the standard examples of the seminal *Jason* AgentSpeak interpreter: <https://github.com/jason-lang/jason/tree/main/examples/room>, accessed at 2025-12-02. Our simple example here is a partial, single-agent implementation.

3. Vanilla JavaScript Agent

The example is, with additional console log statements for the sake of illustration, available at <https://gist.github.com/TimKam/>

2825ff18367090eb2e9bbd4db224723c (*accessed at 2025-02-18*) and can, for instance, be executed by pasting the entire content of the corresponding file into a Web browser console.

4. JS-son LLM Agent

The example's code is available at <https://github.com/TimKam/JS-son/tree/master/examples/llm>, *accessed at 2025-02-19*.

5. JS-son Centralized MAS

The example's code, also covering the browser-based implementation, is available at <https://github.com/TimKam/JS-son/tree/master/examples/web>, *accessed at 2025-02-14*.

6. JS-son Distributed MAS

The example's code (including all agent and environment internals that we skip in the walk-through we provide in this chapter) is available at <https://github.com/TimKam/JS-son/tree/master/examples/distributed>, *accessed at 2025-02-18*.

7. JS-son Agents and Jupyter Notebooks

The example's code is available at <https://github.com/TimKam/JS-son/tree/master/examples/jupyter>, *accessed at 2025-02-13*.

8. Serverless JS-son Agents

The example's code is available at <https://github.com/TimKam/JS-son/tree/master/examples/serverless>, *accessed at 2025-02-12*.

References

- Baiardi, M., Burattini, S., Ciatto, G., Pianini, D.: Jakta: BDI agent-oriented programming in pure kotlin. In: V. Malvone, A. Murano (eds.) Multi-Agent Systems - 20th European Conference, EUMAS 2023, Naples, Italy, September 14-15, 2023, Proceedings, *Lecture Notes in Computer Science*, vol. 14282, pp. 49–65. Springer (2023). DOI 10.1007/978-3-031-43264-4\4. URL https://doi.org/10.1007/978-3-031-43264-4_4
- Boissier, O., Bordini, R.H., HÄEbner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. *Science of Computer Programming* **78**(6), 747 – 761 (2013). DOI <https://doi.org/10.1016/j.scico.2011.10.004>. URL <http://www.sciencedirect.com/science/article/pii/S016764231100181X>. Special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011) & Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments
- Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology). John Wiley & Sons, Inc., USA (2007)
- Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259 (2017). DOI 10.17487/RFC8259. URL <https://www.rfc-editor.org/info/rfc8259>
- Briola, D., Ferrando, A., Mascardi, V.: Fantastic mass and where to find them: First results and lesson learned. In: A. Ciortea, M. Dastani, J. Luo (eds.) Engineering Multi-Agent Systems - 11th International Workshop, 2023, London, UK, May 29-30, 2023, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 14378, pp. 233–252. Springer (2023). DOI 10.1007/978-3-031-48539-8_16. URL https://doi.org/10.1007/978-3-031-48539-8_16
- Cass, S.: The top programming languages 2024. *IEEE Spectrum* (2024). URL <https://spectrum.ieee.org/ibm-quantum-computer-2668978269>

7. Ciortea, A., Boissier, O., Ricci, A.: Engineering world-wide multi-agent systems with hypermedia. In: D. Weyns, V. Mascardi, A. Ricci (eds.) *Engineering Multi-Agent Systems - 6th International Workshop, EMAS 2018, Stockholm, Sweden, July 14-15, 2018, Revised Selected Papers, Lecture Notes in Computer Science*, vol. 11375, pp. 285–301. Springer (2018). DOI 10.1007/978-3-030-25693-7_15
8. Gradner, M.: The fantastic combinations of john conway's new solitaire game life. *Scientific American* **223**(4), 120–123 (1970)
9. Kampik, T., Gomez, A., Ciortea, A., Mayer, S.: Autonomous agents on the edge of things. In: F. Dignum, A. Lomuscio, U. Endriss, A. Nowé (eds.) *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pp. 1767–1769. ACM (2021). DOI 10.5555/3463952.3464231. URL <https://www.ifaamas.org/Proceedings/aamas2021/pdfs/p1767.pdf>
10. Kampik, T., Nieves, J.C.: Js-son - A lean, extensible javascript agent programming library. In: L.A. Dennis, R.H. Bordini, Y. Lespérance (eds.) *Engineering Multi-Agent Systems - 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13-14, 2019, Revised Selected Papers, Lecture Notes in Computer Science*, vol. 12058, pp. 215–234. Springer (2019). DOI 10.1007/978-3-030-51417-4_11
11. Mascardi, V., Weyns, D., Ricci, A., Earle, C.B., Casals, A., Challenger, M., Chopra, A., Ciortea, A., Dennis, L.A., Díaz, Á.F., Fallah-Seghrouchni, A.E., Ferrando, A., Fredlund, L.Å., Giunchiglia, E., Guessoum, Z., Günay, A., Hindriks, K., Iglesias, C.A., Logan, B., Kampik, T., Kardas, G., Koeman, V.J., Larsen, J.B., Mayer, S., Méndez, T., Méndez, T., Nieves, J.C., Seidita, V., Tezel, B.T., Varga, L.Z., Winikoff, M.: *Engineering Multi-Agent Systems: State of Affairs and the Road Ahead. SIGSOFT Engineering Notes (SEN)* (2019)
12. Mualla, Y., Kampik, T., Tchappi, I.H., Najjar, A., Galland, S., Nicolle, C.: Explainable agents as static web pages: UAV simulation example. In: D. Calvaresi, A. Najjar, M. Winikoff, K. Främling (eds.) *Explainable, Transparent Autonomous Agents and Multi-Agent Systems - Second International Workshop, EXTRAAMAS 2020, Auckland, New Zealand, May 9-13, 2020, Revised Selected Papers, Lecture Notes in Computer Science*, vol. 12175, pp. 149–154. Springer (2020). DOI 10.1007/978-3-030-51924-7_9. URL https://doi.org/10.1007/978-3-030-51924-7_9
13. Mualla, Y., Tchappi, I., Kampik, T., Najjar, A., Calvaresi, D., Abbas-Turki, A., Galland, S., Nicolle, C.: The quest of parsimonious XAI: A human-agent architecture for explanation formulation. *Artif. Intell.* **302**, 103573 (2022). DOI 10.1016/J.ARTINT.2021.103573. URL <https://doi.org/10.1016/j.artint.2021.103573>
14. O'Neill, E., Collier, R.W.: Exploiting service-discovery and openapi in multi-agent microservices (MAMS) applications. In: A. Ciortea, M. Dastani, J. Luo (eds.) *Engineering Multi-Agent Systems - 11th International Workshop, EMAS 2023, London, UK, May 29-30, 2023, Revised Selected Papers, Lecture Notes in Computer Science*, vol. 14378, pp. 78–84. Springer (2023). DOI 10.1007/978-3-031-48539-8_5. URL https://doi.org/10.1007/978-3-031-48539-8_5
15. Ramanathan, G., Gomez, A., Mayer, S.: Learnings from implementation of a BDI agent-based battery-less wireless sensor. *CoRR abs/2406.17303* (2024). DOI 10.48550/ARXIV.2406.17303. URL <https://doi.org/10.48550/arXiv.2406.17303>
16. Rao, A.S.: Agentspeak(I): BDI agents speak out in a logical computable language. In: W.V. de Velde, J.W. Perram (eds.) *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings, Lecture Notes in Computer Science*, vol. 1038, pp. 42–55. Springer (1996). DOI 10.1007/BFb0031845. URL <https://doi.org/10.1007/BFb0031845>
17. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: J. Allen, R. Fikes, E. Sandewall (eds.) *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pp. 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991)
18. Russell, S., Norvig, P.: *Artificial Intelligence: A Modern Approach*, 3 edn. Prentice Hall (2010)
19. Shoham, Y.: Agent-oriented programming. *Artificial Intelligence* **60**(1), 51–92 (1993). DOI [https://doi.org/10.1016/0004-3702\(93\)90034-9](https://doi.org/10.1016/0004-3702(93)90034-9). URL <https://www.sciencedirect.com/science/article/pii/0004370293900349>

20. Team, G., Anil, R., Borgeaud, S., Alayrac, J.B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A.M., Hauth, A., Millican, K., et al.: Gemini: a family of highly capable multimodal models. arXiv preprint arXiv:2312.11805 (2023)
21. Vente, S., Kimmig, A., Preece, A., Cerutti, F.: Increasing negotiation performance at the edge of the network. In: N. Bassiliades, G. Chalkiadakis, D. de Jonge (eds.) Multi-Agent Systems and Agreement Technologies, pp. 351–365. Springer International Publishing, Cham (2020)
22. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W.X., Wei, Z., Wen, J.: A survey on large language model based autonomous agents. *Frontiers of Computer Science* **18**(6), 186345 (2024). DOI 10.1007/s11704-024-40231-1. URL <https://doi.org/10.1007/s11704-024-40231-1>