

---

# AllSim: Simulating and Benchmarking Resource Allocation Policies in Multi-User Systems

---

**Jeroen Berrevoets**  
University of Cambridge

**Daniel Jarrett**  
University of Cambridge

**Alex J. Chan**  
University of Cambridge

**Mihaela van der Schaar**  
University of Cambridge

## Abstract

Numerous real-world systems, ranging from healthcare to energy grids, involve users competing for finite and potentially scarce resources. Designing policies for repeated resource allocation in such real-world systems is challenging for many reasons, including the changing nature of user types and their (possibly urgent) need for resources. Researchers have developed numerous machine learning solutions for determining repeated resource allocation policies in these challenging settings. However, a key limitation has been the absence of good methods and test-beds for benchmarking these policies; almost all resource allocation policies are benchmarked in environments which are either completely synthetic or do not allow *any* deviation from historical data. In this paper we introduce AllSim, which is a benchmarking environment for realistically simulating the impact and utility of policies for resource allocation in systems in which users compete for such scarce resources. Building such a benchmarking environment is challenging because it needs to successfully take into account *the entire collective* of potential users and the impact a resource allocation policy has on all the other users in the system. AllSim’s benchmarking environment is modular (each component being parameterized individually), learnable (informed by historical data), and customizable (adaptable to changing conditions). These, when interacting with an allocation policy, produce a dataset of simulated outcomes for evaluation and comparison of such policies. We believe AllSim is an essential step towards a more systematic evaluation of policies for scarce resource allocation compared to current approaches for benchmarking such methods.

## 1 Introduction

The problem of repeated resource allocation to users with timeliness constraints is ubiquitous in settings ranging from healthcare to engineering systems and even labour markets. This problem becomes even more challenging when the resources are diverse and users may derive different benefits from obtaining a specific resource. In these applications, a resource coordinator or decision maker, is tasked with allocating these diverse resources to a pool of diverse users which arrive or leave over time.

When a new resource arrives, it is the decision maker’s task to assign this coveted resource to one of the users in their current pool. Making such a decision has an enormous impact on the *complete* system: (i) when a resource is allocated to a user, other users now need to wait for the next resource to arrive, thereby impacting their utility (since they are delay-sensitive), (ii) the match between the resource and the user recipient has different valuations, (iii) assigning a resource to a specific user in the pool influences the utilities of *all* the other users in the pool as well, and thereby impacting any subsequent

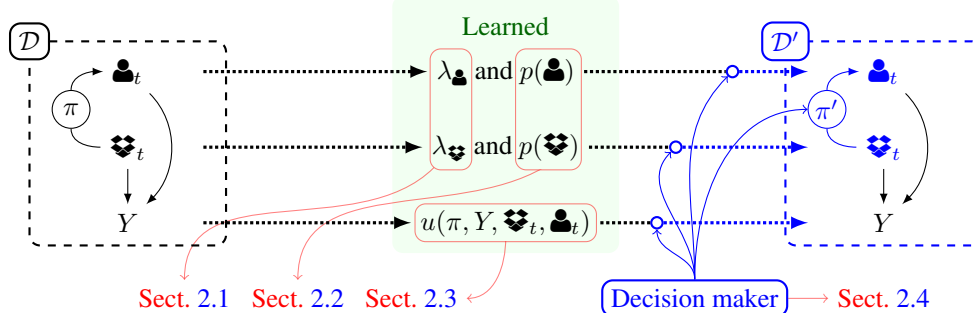


Figure 1: **AllSim overview.** We illustrate how AllSim takes as input a dataset ( $\mathcal{D}$ ) which comprises a set of users ( $\text{♀}$ ) and resources ( $\text{♂}$ ) and outcomes ( $Y$ ). Each of these objects is confounded by an allocation policy  $\pi$ . From this highly complex dataset, AllSim learns a set of separate components (in green): a distribution for users ( $p(\text{♀})$ ) and resources ( $p(\text{♂})$ ), an associated arrival time based on  $t$  ( $\lambda_{\text{♂}}$  and  $\lambda_{\text{♀}}$ ), and a utility ( $u$ ). AllSim then exposes an interface where a decision maker can perturb and influence each component separately, included the allocation policy itself. We highlight these perturbations in blue, resulting in a new dataset ( $\mathcal{D}'$ ) used to measure the effect of each perturbation.

allocations. We note that the above problem scenario is incredibly general. To have an idea of the diversity of situations described as such, we refer to Table 1 where we list a few example situations.

**Resource allocation.** We identify three main challenges one has to overcome when solving problems described by the above. (i) Resources and users are described by multiple (possibly continuous) variables resulting in them being diverse and having complex interactions. (ii) The above are *dynamic non-steady-state scenarios*, which means that at any time the arrival of resources and users may change, the user-specific as well as system-wide utility may change, and even the users and resources themselves may change. (iii) These are multi-user problems, which means that each decision needs to take into account the resource recipient alongside every other user in the system.

**Evaluation.** Given the complex interactions between diverse resources and users, more and more we have to rely on machine learning based allocation policies which model these interactions to optimise a (system-wide) utility [1–4] (cfr. Table 1). While these novel policies receive a lot of research attention, the way in which they are evaluated seems to receive much less while being equally important. In fact, literature introducing these new methods fail to evaluate them against the challenges listed above. We believe the reason is the lack of proper evaluation tools; to our knowledge, there only exist tools that: *Have no diverse resources/users* [5, 6], *Remain steady-state* [7], or *Model single-user systems* [8]. None of them capture the challenges described above.

**Synthetic versus real data.** Another major consideration is the usage of data when evaluating policies. Whenever a real-world dataset is available— comprising users, resources, assignments and outcomes —we have to consider the fact that this dataset is *tainted* by an observational (in-place) policy. We have illustrated this as  $\text{♂}_t \xrightarrow{\pi} \text{♀}_t$  in Figure 1, where the policy is denoted as  $\pi$ . The moment we want to test a policy which is different from the observational policy (e.g.  $\pi'$  in Figure 1), we deviate from the original dataset as the resource to user assignments are (by definition) different.

A solution could be a completely synthetic simulation to evaluate policies. However, given the detailed and diverse descriptions of resources and users, this would introduce too much bias into our evaluation as every detail needs to be manually specified [9]. The latter is of particular importance when testing these novel ML-based policies, as this is exactly what they were built for in the first place.

**Our solution is AllSim.** Illustrated in Figure 1, AllSim learns separate (unbiased) components from historical data which was biased by a previous observational policy,  $\pi$ . These components are exposed as an interface which a decision maker can use to modify the system to fit their purpose. An obvious example of such a modification is to replace the past policy with a different policy. Other examples could be changing the arrival rates of users, resources, changing user and resource types, or even resource efficacy by changing the outcomes ( $Y$ ) while still maintaining detail and realism. These perturbations are illustrated in blue in Figure 1. From AllSim, we sample a new dataset to measure the effect of the practitioner’s modifications on utilities such as fairness, survival, waste, etc.

Table 1: **Example situations.** We list a few example situations that follow the general problem formalism introduced in this paper and the repeated allocation policy used to solve them.

Problem setting	Users	Resources	Allocation policy	Utility
<i>Headhunting</i>	Openings	Applicants	Assignment	Hires (& retention)
<i>Project staffing</i>	Projects	Workers	Staffing	Project success
<i>Organ transplantation</i>	Patients	Donors	Matching	Post/Pre TX survival
<i>Mechanical ventilation</i>	Patients	Ventilators	Triaging	ICU Discharge
<i>Bicycle sharing</i>	Docks	Bicycles	Redistribution	Idle times

**Why are such simulators important for ML research?** The successes in other subareas in machine learning are driven largely by the existence of capable and qualitative simulators [10–13]. With an easy-to-use interface and easily customisable environments, simulators allow researchers to focus on model development rather than creating their own (often conflicting) evaluation protocols. With AllSim, we hope to drive innovation for resource allocation in multi-user problems in healthcare, engineering, economics, etc. The aforementioned simulators are great examples of systematic evaluation across entire research communities, however, they do not: learn realistic and unbiased simulation objects from data, allow for multi-user simulation, or model dynamic non-steady-state scenarios.

**Desiderata.** From Figure 1 we identify three important desiderata: (1) A simulation should extract unbiased components from historical data which was tainted by existing policies; (2) The simulation should infer unbiased outcomes despite having access to only these biased data, which includes long-term impact on system-wide utilities since present allocations influence future allocations, requiring counterfactual inference (to determine outcomes under different allocations). (3) Using the extracted components from (1), a user must be able to perturb and change the components to fit their specific needs to evaluate different policies and settings before being deployed in the real world.

**Contributions** In this work, we present AllSim (Allocation Simulator), a general-purpose open-source framework for performing data-driven simulation of scarce resource allocation policies for *pre-deployment* evaluation. We use modular environment mechanisms to capture a range of environment conditions (e.g. varying arrival rates, sudden shocks, etc.), and provide for componentwise parameters to be learned from historical data, as well as allowing users to further configure parameters for stress testing and sensitivity analysis. Potential outcomes are evaluated using unbiased causal effects methods: Upon interaction with a policy, AllSim outputs a batch dataset detailing all of the simulated outcomes, allowing users to draw their own conclusions over the effectiveness of a policy. Compared to existing work, we believe this simulation framework takes a step towards more methodical evaluation of scarce resource allocation policies.

In Appendix B we compare against other strategies used to evaluate allocation policies. AllSim’s itself is built using ideas from various fields in machine learning which we also review in Appendix B. Furthermore, in Appendix B.1 we review some medical simulations which *seem* related, but are not.

## 2 AllSim

Let  $X \in \mathbb{R}^d$  denote the feature vector of a *user*, and let  $\mathcal{X}(t)$  denote the arrival process of users. At each time  $t$ , let  $\mathbf{X}(t) := \{X_i\}_{i=0}^{N(t)} \sim \mathcal{X}(t)$  give the arrival set of (time-varying) size  $N(t)$ . Likewise, let  $R \in \mathbb{R}^e$  be the feature vector of a *resource*, and let  $\mathcal{R}(t)$  be the arrival process of resources. At each time  $t$ , let  $\mathbf{R}(t) := \{R_j\}_{j=0}^{M(t)} \sim \mathcal{R}(t)$  give the arrival set of (time-varying) size  $M(t)$ .

While we make no assumptions on how users are modelled, we assume that resources are immediately *perishable*—that is, each incoming resource cannot be kept idle, and must be consumed by some user in the same time step. In organ transplantation, for instance, the time between harvesting an organ and transplanting it (“cold ischemia time”) must be minimized to prevent degradation [14–16].

Let  $Y_+ \in \mathbb{R}$  be the outcome of a *matched* user, drawn from the distribution  $\mathcal{Y}(X, R)$  induced by assigning a resource  $R$  to a user  $X$ . At each time  $t$ , let  $\mathbf{Y}_+(t) := \{Y_+ \sim \mathcal{Y}(X_R, R) : R \in \mathbf{R}(t)\}$  give the set of outcomes that result from matching each incoming  $R \in \mathbf{R}(t)$  with its assigned  $X_R$ . Likewise, let  $Y_- \in \mathbb{R}$  be the outcome of an *un-matched* user, drawn from the distribution  $\mathcal{Y}(X, \emptyset)$ . At each time  $t$ , let the set of outcomes for users who are never assigned a resource be given by

$\mathbf{Y}_-(t) := \{Y_- \sim \mathcal{Y}(X, \emptyset) : X \in \mathbf{X}(t), \neg(\exists t' \geq t)(R \in \mathbf{R}(t'), X = X_R)\}$ . (Note that we focus on discrete-time settings (e.g. hours or days), and leave continuous time for future work). Then we have:

**Definition 1 (Repeated Resource Allocation)** Denote an *environment* with the tuple  $\mathcal{E} := (\mathcal{X}, \mathcal{R}, \mathcal{Y})$ . The *repeated resource allocation* problem is to decide which users to assign each incoming resource to—that is, to come up with a *repeated allocation policy*  $\pi : \mathbb{R}^e \times \mathcal{P}(\mathbb{R}^d) \rightarrow \mathbb{R}^d$ , perhaps to optimize some utility defined on the basis of (un-)matched outcomes. For instance, if  $Y$  is a patient’s post-transplantation survival time, we might wish to maximize the average survival time.

With the necessary notation, and a formal definition of a policy’s input and output in Definition 1, we are equipped to introduce each component of AllSim as illustrated in Figure 1. In Sect. 2.4 we also discuss how AllSim’s output can be used to evaluate a new policy (or any other modification from the decision maker). Details regarding the simulation life-cycle can be found in Appendix A.

## 2.1 Arrival of users and resources

$\lambda_{\bullet}$  and  $\lambda_{\heartsuit}$

There are two necessary ingredients that comprise the arrival of new users, and new resources: the amount ( $N(t)$  and  $M(t)$ , respectively), and the description ( $X_i$  and  $R_j$ , respectively). Each is modelled differently. Before we sample the user and resource description, we first sample the amount of each arriving at time  $t$  from an associated arrival process— i.e., in this subsection we will focus on  $N(t)$  and  $M(t)$ . We first introduce the structure of the arrival processes, and explain how their parameters can be learned from data and modified by a decision maker to setup the environment.

First, we stress that  $N(t)$  and  $M(t)$  are not necessarily sampled from *constant* arrival processes. Instead, we want the user and resource arrivals to change over time either completely or per user/resource type, which we will explain in more detail below. To accommodate this, we split each arrival process into a product of separate arrival processes which we combine into  $\mathcal{X}(t)$  and  $\mathcal{R}(t)$  as:

$$\hat{\mathcal{X}}(t, \theta_x) = \hat{\mathcal{X}}_1(t, \theta_{1,x}) \times \cdots \times \hat{\mathcal{X}}_K(t, \theta_{K,x}), \quad (1)$$

$$\hat{\mathcal{R}}(t, \theta_r) = \hat{\mathcal{R}}_1(t, \theta_{1,r}) \times \cdots \times \hat{\mathcal{R}}_L(t, \theta_{L,r}), \quad (2)$$

where each individual arrival process in  $\hat{\mathcal{X}}_k$  and  $\hat{\mathcal{R}}_l$  is parameterised with (learnable) parameters  $\theta_{k,x}$  and  $\theta_{l,r}$  with  $k \in [K]$  and  $l \in [L]$ , respectively. Each factor corresponds with some (learned or predefined) user-type (Equation (1)) and resource-type (Equation (2)). Having these factors allows us to model increasing numbers of, for example, older/younger patients entering a transplant wait-list.

In order for  $\mathcal{X}(t)$  and  $\mathcal{R}(t)$  to change over time, we let their parameterisation,  $\theta_x$  and  $\theta_r$ , change in  $t$ . As an example, we can set the arrival processes to Poisson processes (we refer to Appendix E for other examples) with arrival rates  $\theta_x = \lambda_k(t)$  and  $\theta_r = \lambda_l(t)$ , which we can both model over time as,

$$\lambda_k(t) = \nu_k \lambda_k(0) g_k(t), \quad (3)$$

$$\lambda_l(t) = \nu_l \lambda_l(0) g_l(t), \quad (4)$$

with  $\lambda_k(t), \lambda_l(t) \in \mathbb{R}_+$ , and  $\nu_k, \nu_l \in \mathbb{R}_+$  as a normalising constant such that the sum of all  $\lambda_k$  equal some overall arrival rate  $a_x$ , and similarly, the sum of all  $\lambda_l$  equal some overall arrival rate  $a_r$ . Lastly,  $g_k$  and  $g_l$  are continuous functions that simulate a user-specified drift. Note that these  $g$  can also be a combination of multiple drift scenarios, or can be shared across different  $k, l$ . Having  $g$ , allows practitioners to very accurately describe the non-stationarity they wish to test for. Optionally,  $\nu_k$  and  $\nu_l$  can be kept fixed throughout the simulation such that  $a_x$  and  $a_r$  vary as does  $g_{k,l}(t)$ , or it can be recomputed for every step  $t$ , such that  $a_x$  and  $a_r$  are kept fixed throughout the simulation.

As such, we have a set of arrival rates,  $\Lambda_x = [\lambda_1, \dots, \lambda_K]$ , with  $\sum_k \lambda_k = \alpha_x$  with  $\alpha_x \in \mathbb{R}_+$  as the total arrival rate of recipients. The advantage of splitting  $\alpha_x$  into multiple  $\lambda_k$ , is that we can finetune the arrival of certain recipient types, yet allow comparison between  $\alpha_x$  and  $\alpha_r$  (the total arrival rate for resources). For example, the  $k^{\text{th}}$  recipient type may be completely absent when a policy is launched, but over time it gradually enters the system, increasing  $\alpha_x$  as a whole. Naturally, we also model the arrival of resources as we have for recipients, but left it out of discussion for clarity.

Learning  $\theta_{x,r}$  naturally depends on the choice of arrival process. In our setups below we use a Poisson process and have either: (i) learned the dynamic parameters  $\lambda_{k,l}(t)$  as in Equations (3) and (4) using polynomial regression over a time-windowed average of incoming users and resources— over all

data to compute a correct  $\nu_{k,l}$ , as well as per predefined condition; or (ii) have predefined an arrival function and drift functions,  $g$ , to illustrate a scenario where one wishes to test a prespecified scenario.

## 2.2 New users and resources

$p(\text{👤})$  and  $p(\text{👤👤})$

From  $\hat{\mathcal{X}}(t)$  and  $\hat{\mathcal{R}}(t)$  we sample  $N(t)$  and  $M(t)$ , respectively. Of course, we need to provide the tested policies with more than just an *amount* of users and resources arriving at time  $t$ . Furthermore, when working with user and resource types (using the decomposition in Equations (1) and (2)), we have  $N(t) = \sum_k N_k(t)$  and  $M(t) = \sum_l M_l(t)$ , where each  $N_k(t)$  and  $M_l(t)$  represents an amount of users and resources *per type*. As such, we need these types to sample detailed descriptions of each.

When a recipient or a resource arrives, we sample them from a distribution denoted  $p_{\theta_x}(X)$  for the recipients, and  $p_{\theta_r}(R)$  for the resources. These distributions are either learnt from data, or shared as an open-source (but privatised) distribution. For the user-distributions we learn from  $\bigcup_t \mathbf{X}_t$ , and similarly, for the resource-distributions we learn from  $\bigcup_t \mathbf{R}(t)$ . Since both remain independent from the past policy (no policy determines which users and resources arrive in the system), we can use any (conditional) generative model to learn these distributions as we are not required to de-bias these data.

Of course, we need to be able to sample specific user and resource *types*. For this we require *conditional* generative models, where the condition corresponds with a type:  $p_{\theta_x}(X)$  becomes  $p_{\theta_x}(X|k)$ , and similarly  $p_{\theta_r}(R)$  becomes  $p_{\theta_r}(R|l)$ . In case we wish to use an unconditional generative model, we can simply learn multiple:  $p_{\theta_{k,x}}(X)$  for each  $k \in [K]$ , and similarly for  $p_{\theta_{l,r}}(R)$  for each  $l \in [L]$ .

Interestingly, we do not need to account for any variability (learned nor specified) over time, since this is completely modelled through the arrival processes in Sect. 2.1. In particular, whenever we wish one type,  $k$  to dominate others, we simply increase  $g_k(t)$  in Equations (3) and (4). In case we only want one type to appear after  $t$  in the simulation, we set  $g_k = 0$  for  $t' < t$  and increase it for  $t'' > t$ .

## 2.3 Utility

$u(\pi, Y, \text{👤}_t, \text{👤👤}_t)$

The final component in AllSim, as per Figure 1, are the utilities: functions of the policy ( $\pi$ ), the users and resources ( $X$  and  $R$ ), and crucially, the allocation outcomes ( $Y$ ). Given the previous sections, all that remains are the outcomes and how we can combine each element into a new dataset,  $\mathcal{D}'$ , with *counterfactual* outcomes,  $Y'$ .

As the outcome is a function of the resource and its recipient, inference is a hard problem as allocations suggested by the tested policy deviate from historical data which was collected under a different policy (i.e., they are counterfactual). Consequentially, some combinations are less observed in the original data, illustrated in Figure 2. In Figure 2 we illustrate two policies,  $\pi$  and  $\pi'$  which result in different datasets  $\mathcal{D}$  and  $\mathcal{D}'$ . The latter ( $\mathcal{D}'$ ) is what we wish to provide with AllSim, using only data from the former ( $\mathcal{D}$ ).

**Counterfactual inference.** AllSim handles this difficult problem by using a counterfactual estimator. Counterfactual methods correct for allocation bias explicitly [2]. In particular, these methods aim to make an unbiased prediction of *the potential outcome*, associated with some treatment (or resource). We are interested in counterfactual methods that model the potential outcomes for the recipients when they are (not) allocated a resource. A counterfactual estimator then “completes” the dataset ( $\mathcal{D}'$ ) as,

$$\mathbf{Y}(t) = \mathbb{E}[\hat{\mathbf{Y}}(\mathbf{R}(t)) | \mathbf{X}_\pi(t)], \quad (5)$$

where  $\hat{\mathbf{Y}}(\mathbf{R}(t))$  is the estimated potential outcome, using methodology known in the potential outcomes literature [17–22], and  $\mathbf{X}_\pi(t)$  are the recipients selected by a policy  $\pi$  at time  $t$ . The potential outcome is a random variable depicting the (possibly alternative) outcome when the user receives the resource,  $R(t)$ . Note that this is not the same as simply conditioning the outcome

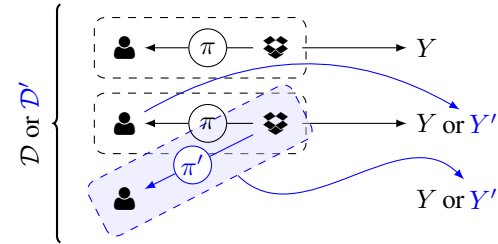


Figure 2: **Allocation policies bias data.** Above illustration depicts two policies:  $\pi$  and  $\pi'$ . Each policy is tasked with assigning resources to users as per Definition 1. Besides users and resources, we observe an outcome,  $Y$ . Despite observing the same users and resources, a different policy results in *completely* different outcomes,  $Y'$ , and data,  $\mathcal{D}'$ .

variables on the users, for the reasons outlined above: conditioning using only biased data will lead to biased estimates for the outcome variable. Hence, literature on counterfactual inference introduced the potential outcomes notation in Equation (5) to differentiate between  $Y(R(t))$  and  $Y|R(t)$ . We provide a comprehensive overview of counterfactual methods and literature in Appendix H.

## 2.4 Putting it all together

Decision maker

We have now discussed each component in the middle section of Figure 1. What remains are the decision maker’s perturbations, and finally, combining each component into a new dataset,  $\mathcal{D}'$ .

**Perturbations.** From Figure 1 we learn that a decision maker can make three types of perturbations: (i) they can replace the original policy,  $\pi$ , with a new (alternative) policy,  $\pi'$ ; (ii) they can change the utility function,  $u$ , which takes as argument a dataset comprised of users, resources, outcomes, and a policy; and lastly, (iii) they can change the types, as well as the amount, of users and resources entering the system. Given these perturbations, the policy is allowed to act in a different environment.

Changing the policy in (i) is done simply by implementing the new policy according to the simulation interface (discussed in the next section). We stress once more that this paper does not provide guidance for allocation policies nor does it propose a new policy of any kind. In fact, the policies used in the following section are tried and tested policies, currently in use in practice. Changing the utility function for (ii) is easily done in AllSim as running the simulation does not depend at all on the chosen utility function! As AllSim provides a completely counterfactual dataset,  $\mathcal{D}'$ , the utility is computed *post-hoc* which allows us to always fall back on the generated dataset. Finally, perturbing arrivals (iii) is already discussed in Sect. 2.1; the arrival processes are perturbed through  $g$ .

**Sampling data.** Equations (1) to (4) provides us with  $\mathbf{X}(t)$  and  $\mathbf{R}(t)$ . Equation (5) provides us with an estimated potential outcome given  $X(t)$ ,  $R(t)$  and their allocations using  $\pi'$ . AllSim then carefully indicates a timestamp for each combination and then presents the decision maker with a new dataset:

$$\mathcal{D}' := \{(X, R, \hat{Y}(R), t)_i : i = 1, \dots, N\}.$$

Having a new (counterfactual) dataset based on  $\pi'$ ,  $\mathcal{D}'$ , allows to easily calculate various performance utilities of interest, which the decision maker can use to evaluate the allocation policy, *pre-deployment*:

**Definition 2 (Pre-Deployment Evaluation)** Let  $f : \prod_k \mathbb{R}^k \times \dots \rightarrow \mathbb{M}$  denote an *evaluation metric* mapping a sequence of outcomes  $\{\mathbf{Y}(t)\}_{t=1, \dots}$  to some space of *evaluation outcome*  $\mathbb{M}$  (e.g. for the average survival time, this would simply be  $\mathbb{R}$ ), where  $\mathbf{Y}(t) := \mathbf{Y}_+(t) \cup \mathbf{Y}_-(t)$ . Given a problem  $\mathcal{E}$  and policy  $\pi$ , the *pre-deployment evaluation* problem is to compute statistics of the distribution  $\mathcal{F}_{\mathcal{E}, \pi}$  of evaluation outcomes  $f(\{\mathbf{Y}(t)\}_{t=1, \dots})$ ; commonly, this would be the mean  $\mathbb{E}_{\mathcal{E}, \pi}[f(\{\mathbf{Y}(t)\}_{t=1, \dots})]$ .

Note that we have defined  $f$  in terms of the *sequence* of per-period outcomes such that it gives maximum flexibility: Depending on how individual outcomes  $Y$  are defined, we can measure point estimates (e.g. the mean survival), compare subpopulations (e.g. whether some types of recipients systematically receive more favourable outcomes), examine trends (e.g. whether outcomes degrade as the types of recipients arriving change), or potentially investigate more complex hypotheses.

## 3 AllSim Interface & Examples

Given the formal definition of AllSim presented in Sect. 2 (and further in Appendix A), we now introduce AllSim’s programming interface and use it directly to provide some experimental results. We split this section in two major parts: first, we show the type of analysis AllSim can do for us, as well as how to tailor AllSim to the decision maker’s needs; and then, we show how realistic the AllSim simulations are, compared to the factual data; we show that AllSim models realistic systems.

### 3.1 Example analysis and decision-maker specifications

As a first example, let us showcase an analysis to illustrate the possible impact AllSim may have in practice. Throughout this section, we will use the open-access United Network for Organ Sharing (UNOS) dataset which comprises 25 years of liver-to-patient allocation. Importantly, we had to make zero adjustments to our framework to fully capture these data, showcasing the generality of the

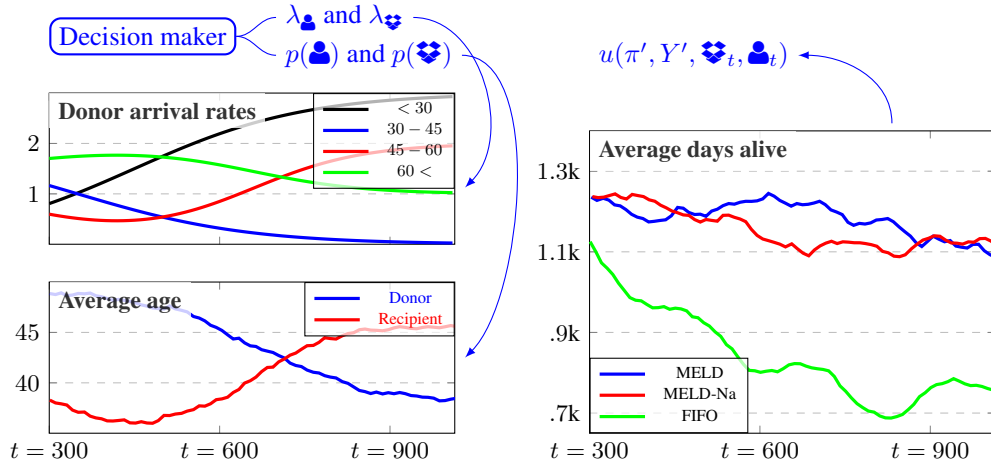


Figure 4: **Specifying a simulation using AllSim.** In the above, a decision maker defines a set of donor arrival rates, based on age ( $\lambda_{\text{♂}}$  and  $\lambda_{\text{♀}}$ ). Using these very simple, but custom, arrival rates, we see a direct influence in the user and resource distributions ( $p_{\text{♂}}$  and  $p_{\text{♀}}$ ). These perturbations constitute as perturbations of type (iii) as per Sect. 2.4. Finally, the decision maker tries out three different policies: MELD, MELD-na, and FIFO, which constitute as perturbation type (i). The result of these policies is shown on the right. The reported averages are windowed over 300 samples.

AllSim framework. We only use UNOS data until 2019 which, interestingly, predates the COVID-19 global pandemic. As such, it is impossible to evaluate policies using only these data: we need AllSim to model a counterfactual scenario that mimics what we saw during the pandemic to test a policy.

In Figure 3 we ran the MELD-Na policy in two hypothetical scenarios: one where COVID-19 happens (which resulted in a 50% drop in the donor liver arrival rates [23–25]), and one where it doesn't. With AllSim we can model each scenario confidently. For this particular example, we fix the seed of AllSim and only change the supply of organs by giving two different resource arrival processes:

```

1 def covid(t):
2     if t < 600:
3         return .5
4     else:
5         return .25
6
7 def no_covid(t):
8     return .5

```

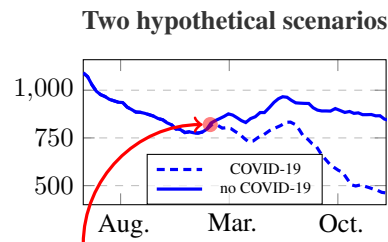


Figure 3: **Two hypothetical scenarios.** We require AllSim to evaluate a policy (e.g. MELD-Na) in hypothetical (counterfactual) scenarios. The x-axis is time, and the y-axis indicates survival time.

Having illustrated the power of AllSim, let us now show how a decision maker may introduce their perturbations (such as the `covid` and `no_covid` arrival processes from above) into the AllSim simulator. For this, we will provide the donor-organ system with two specific perturbations: (1) we will change the policies (from MELD, to MELD-na, and a simple FIFO policy), and (2) we will increase the user age and decrease the donor age. These two perturbations respectively illustrate perturbation types (i) and (iii), listed in Sect. 2.4 (recall that perturbation type (ii) was changing the utility which is done *after* a counterfactual dataset is sampled and hence does not require testing).

Consider Figure 4 which shows the resulting simulation and the found utility when perturbing the organ arrival rates as well as the allocation policies. The take-away from this experiment is not the performance of the policy (although, reassuringly, MELD and MELD-na<sup>1</sup> do outperform FIFO). Instead, we learn that increasing user age results in dropping the survival time, regardless of the donor age which is decreasing. This is not surprising, as older transplant patients simply have less time to live, whether they get an organ or not. Which leads to the next question: is AllSim realistic?

<sup>1</sup>Which are the policies used in Europe and the USA for donor liver allocation.

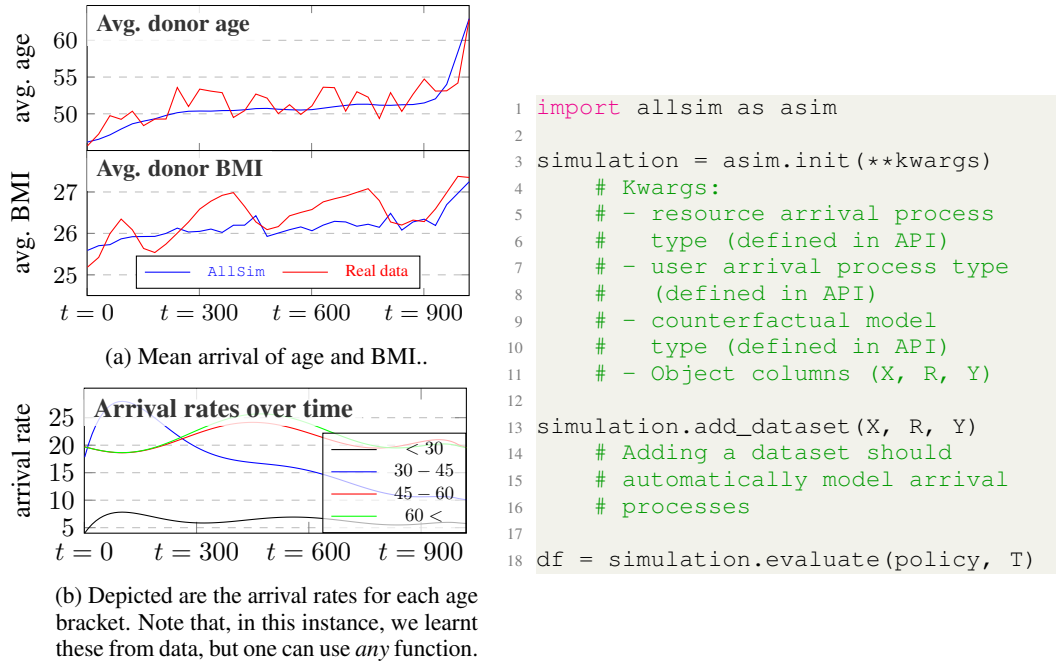


Figure 5: **AllSim (easily) simulates realistic environments.** Using real-world data on donor organs, we let AllSim model 3 years of organ arrivals and compare it with the actual arrival as reported in the data. In Figures 5a and 5b we show AllSim’s output (in donor age and BMI), given the code on the right. With minimal code, a simple condition (4 age brackets), and conservative models (polynomial regression to fit the arrival rates, and a Gaussian kernel density to model the organ densities), we find that AllSim accurately models the actual (real-world) arrival of organs as reported in the UNOS data.

### 3.2 AllSim’s realism

In this section, we will learn an AllSim configuration purely from the UNOS data (i.e. without a decision maker’s input), such that we can compare AllSim’s output side-by-side with what actually happened in UNOS. If they match up, we confirm that AllSim can output realistic scenarios (as UNOS is a real dataset). However, before we do so, we first show *how* we use AllSim from a programming perspective and configure appropriate arrival processes and densities for this particular use case.

First we determine how many users and resources we need to sample, once we know the amount we sample them from a generative model. The former is modeled through a Poisson arrival rate that changes over time, and the second is sampled from some learnt density. Of course, a user can implement their own arrival process by inheriting from the abstract `ArrivalProcess` class.

Importantly, we need to be able to condition the density on some pre-specified characteristic of the object of interest. For example, one may be interested in modelling the arrival of harvested organs of older patients distinctly from younger patients. An example of this is provided in Figure 5, where we show the changing resources coming in the system, alongside the code that generated the result.

**Object densities.** Before discussing a temporal arrival rate, we first discuss modelling the object’s densities. Consider `lns 3–12` in the righthand side of Figure 5. Using this code, we first define what we want to condition on, using a `Condition` object: in this case we formulate age brackets. With the `KDEDensity` class, which is a subclass of the abstract `Density` class, we can automatically model a density, conditioned on these age brackets. Each `Density` object implements a `fit` and `sample` function, which is used to sample new objects by the `System`, which we discuss next.

**Arrival processes.** Using a `Density`, we move on to `lns 14–26`, where we first build a system of multiple arrival processes, one for each discrete condition as in Equations (1) and (2). In particular, we define a `PoissonProcess` for each condition (or age bracket), which is then provided to a `PoissonSystem`. Using the `PoissonSystem`, we can sample the arriving objects for each  $t$  in



ln 25. Note that we also model `alpha`, returning the overall arrival rate, such that the system can calculate an appropriate  $\nu$ . Figure 5a shows that `AllSim` accurately models the arriving objects.

With the arrival processes coded above together with a counterfactual `Inference` object, we compose a `Simulation` object—the main interaction interface. In particular, one defines a set of arrival rates (such as in Figure 5b) for both users and resources (cfr. ln 13–17) to create a simulation:

```
1 simulation = asim.init(resource_process, patient_process, inference,
    columns)
```

With that simulation, a practitioner can instantiate a `Policy`, which implements the `add` and `select` methods. For example, we have implemented the MELD policy [26], which is a widely known and used ScRAP for liver allocation. Using the `simulation`, we can generate a simulated dataset:

```
1 df = simulation.evaluate(policy=meld_policy, T=T)
```

Where `df` is a Pandas `DataFrame` [27, 28]. Naturally, `df` contains an enormous amount of information w.r.t. the `policy`'s allocations in our environment. As such, we have included only a subset of the potential results in Figure 4. Additional results and details can be found in Appendices C and H. Ultimately, the practitioner determines appropriate analysis, settings, and performance metrics.

### 3.3 Beyond Organs

`AllSim` is a general purpose simulator which evaluates scarce resource allocation policies. While we have mainly focused on organ-transplantation so far, `AllSim` is also applicable in other settings. To illustrate, we show how one can implement a vaccine distribution policy evaluation system in `AllSim`. This use-case illustrates the few adjustments one has to make compared to the organ-allocation problem. Specifically, in vaccine distribution, each resource is the same and arrives in batches. Furthermore, the type of patient-in-need is also much broader (in fact, they cover the entire population). Yet, `AllSim` is perfectly capable of modelling this scenario given the following:

- Batch arrival requires a multiplier: if the Poisson process samples a value of 2 on one day, we could simply interpret this as two batches of 1000 doses, i.e. multiply by batch content.
- We no longer require a resource density as vaccines are not unique, contrasting organ allocation. This is implemented as a dummy-density that always returns 1 (or the vaccine amount).
- The broader patient-type is achieved by retraining the recipient-density on the entire population.

These implementation details are relatively simple to implement using `AllSim`'s modular API. While not necessarily a problem in vaccine distribution, recipient arrival in the ICU in a setting of infectious disease (such as COVID-19), is definitely different compared to the organ-allocation setting. With organ-allocation, we can safely assume a Poisson process for recipient arrival as recipients enter the system independently. This is not true for infectious diseases: one recipient arriving may indicate higher infection rates. As such, recipients *do not* arrive independently, motivating `AllSim`'s design.

It is clear that above scenario can no longer rely on a Poisson arrival process for new recipients entering the system. Instead, accurately modelling a situation of infectious disease could be done using a Hawkes process. To illustrate, we include some code below showing exactly how one may go about including such a Hawkes process in `AllSim` (replacing the Poisson processes used earlier).

```
1 class HawkesProcess(PoissonProcess):
2     def __init__(self,
3                 lam: float=.1,
4                 update_lam: Callable[[int], float]=lambda t: t,
5                 delta: float=.1,
6                 a: float=.2):
7         assert a >= 0, "a should be larger than or equal to 0"
8         assert delta > 0, "delta should be larger than 0"
9
10        super().__init__(lam, update_lam)
11
12        self.a, self.delta, self._samples = a, delta, []
13
```

```

14     def get_lam_unnormalized(self, t: int) -> float:
15         return self._baseline_lam + np.sum(
16             self.a * self.beta * np.exp(-beta * (t - self._samples[
17                 self._samples < t])))
18
19     def progress(self, t: int, neu: float=1) -> int:
20         self.lam = neu * self.get_lam_unnormalized(t) # eqs. (5, 6)
21         sample = np.random.poisson(lam=self.lam)
22         self._samples.append(sample)
23         return sample

```

**Allocation policies from machine learning and OR.** It seems that both the ML [1–3, 29–34] and OR [35–41] community is focused more and more on this important class of problems– which is fantastic! But it also warrants careful evaluation. Furthermore, if we find that the evaluation strategies in medicine (which generally propose linear combinations of features [26, 42] or simple CoxPH models [43–46]) have shortcomings, then this is certainly the case for much more complicated strategies introduced in ML or OR. In fact, a recent survey confirmed exactly this concern: [47, cfr. Limitations of ML in transplant medicine]. It is in these extended scenarios where AllSim could help.

Naturally, problems solved by the OR community concern a *variant* of the general problem presented in this paper. For example, Balseiro et al. [40] are concerned with distributing a *fixed* set of resources, to a varying set of incoming users. While different, such problems can still be modelled in AllSim. In the specific case of Balseiro et al. [40], resources are not unique (they represent an amount) and require much less machinery than what we require to model the varying resource scenario. Specifically, one can model the remaining amount of resources as an attribute in our `Policy` class.

## 4 Conclusion

AllSim provides the means to perform standardised evaluation of repeated resource allocation policies in non-steady-state environments. While our experiments focus on organ-transplantation for the sake of exhibition, Appendix G illustrates AllSim for COVID-19 vaccine distribution, an example outside organ-transplantation. We believe that AllSim’s generality and modularity allows for *sensible* adoption in a wide range of application areas. Furthermore, having standardised evaluation will encourage research in this very important and impactful domain spanning many application areas.

Conducting research in repeated resource allocation requires consideration of a policy’s societal impact. While we believe AllSim will *aid* (rather than negatively impact) in this respect (by offering more than simple aggregate statistics), in Appendix F we provide a section dedicated to this topic.

**Ethical research.** We envisage AllSim as a tool to *help* accurate and standardised evaluation of repeated resource allocation policies, however emphasise that any finding would need to be further verified by a human expert or in some cases by a clinical trial. Ultimately, the decision on whether or not to trust a decision making tool is up to the acting decision-maker and ethics board. We hope that AllSim can help in any way to facilitate that decision, but stress that suggestions or evaluation always require critical assessment, as is the case for any research. We also refer the reader to Appendix F for a more thorough discussion on the potential societal impact of systems such as AllSim.

**Reproducibility.** To encourage reproducibility, we have included all our code to reproduce the presented results (as well as those in Appendix C). It should be clear from this paper, that reproducibility is actually one of the main reasons for doing this type of research in the first place. Furthermore, we have included a detailed discussion on how to use our simulation in Appendices D and E.

## Acknowledgements

We would like to thank our many collaborating clinicians, and in particular, Alexander Gimson, for many interesting discussions leading to this work. JB is funded by the W.D. Armstrong Trust, DJ is funded by Alzheimer’s Research UK (ARUK), and AC is funded by Microsoft Research.

## References

- [1] Jeroen Berrevoets, Ahmed Alaa, Zhaozhi Qian, James Jordon, Alexander ES Gimson, and Mihaela Van Der Schaar. Learning queueing policies for organ transplantation allocation using interpretable counterfactual survival analysis. In *International Conference on Machine Learning*, pages 792–802. PMLR, 2021.
- [2] Jeroen Berrevoets, James Jordon, Ioana Bica, Alexander Gimson, and Mihaela van der Schaar. OrganITE: Optimal transplant donor organ offering using an individual treatment effect. In *Advances in Neural Information Processing Systems*, volume 33, pages 20037–20050. Curran Associates, Inc., 2020.
- [3] Jinsung Yoon, Ahmed Alaa, Martin Cadeiras, and Mihaela Van Der Schaar. Personalized donor-recipient matching for organ transplantation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.
- [4] Jinsung Yoon, William R Zame, Amitava Banerjee, Martin Cadeiras, Ahmed M Alaa, and Mihaela van der Schaar. Personalized survival predictions via trees of predictors: An application to cardiac transplantation. *PloS one*, 13(3):e0194985, 2018.
- [5] Michael Balmer, Marcel Rieser, Konrad Meister, David Charypar, Nicolas Lefebvre, and Kai Nagel. Matsim-t: Architecture and simulation times. In *Multi-agent systems for traffic and transportation engineering*, pages 57–78. IGI Global, 2009.
- [6] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *2009 international conference on high performance computing & simulation*, pages 1–11. IEEE, 2009.
- [7] Kartik Ahuja and Mihaela Van der Schaar. Dynamic matching and allocation of tasks. *ACM Transactions on Economics and Computation (TEAC)*, 7(4):1–27, 2019.
- [8] Alex J. Chan, Ioana Bica, Alihan Hüyük, Daniel Jarrett, and Mihaela van der Schaar. The medkit-learn(ing) environment: Medical decision modelling through simulation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021. URL <https://openreview.net/forum?id=AyF90B1yESX>.
- [9] Jeroen Berrevoets, Krzysztof Kacprzyk, Zhaozhi Qian, and Mihaela van der Schaar. Causal deep learning. *arXiv preprint arXiv:2303.02186*, 2023.
- [10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.
- [11] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.
- [12] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.
- [13] Charles Beattie, Thomas Köppe, Edgar A Duñez-Guzmán, and Joel Z Leibo. Deepmind lab2d. *arXiv preprint arXiv:2011.07027*, 2020.
- [14] Agnes Debout, Yohann Foucher, Katy Trébern-Launay, Christophe Legendre, Henri Kreis, Georges Mourad, Valérie Garrigue, Emmanuel Morelon, Fanny Buron, Lionel Rostaing, et al. Each additional hour of cold ischemia time significantly increases the risk of graft failure and mortality following renal transplantation. *Kidney international*, 87(2):343–349, 2015.
- [15] J Adam van der Vliet and Michiel C Warlé. The need to reduce cold ischemia time in kidney transplantation. *Current opinion in organ transplantation*, 18(2):174–178, 2013.
- [16] James E Stahl, Jennifer E Kreke, Fawaz Ali Abdul Malek, Andrew J Schaefer, and Joseph Vacanti. Consequences of cold-ischemia time on primary nonfunction and patient and graft survival in liver transplantation: a meta-analysis. *PloS one*, 3(6):e2468, 2008.
- [17] Jersey Neyman. Sur les applications de la théorie des probabilités aux expériences agricoles: Essai des principes. *Roczniki Nauk Rolniczych*, 10:1–51, 1923.
- [18] Donald B Rubin. Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of educational Psychology*, 66(5):688, 1974.

- [19] Jeroen Berrevoets, Fergus Imrie, Trent Kyono, James Jordon, and Mihaela van der Schaar. To impute or not to impute? missing data in treatment effect estimation. In Francisco Ruiz, Jennifer Dy, and Jan-Willem van de Meent, editors, *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pages 3568–3590. PMLR, 25–27 Apr 2023. URL <https://proceedings.mlr.press/v206/berrevoets23a.html>.
- [20] Jeroen Berrevoets, Alicia Curth, Ioana Bica, Eoin McKinney, and Mihaela van der Schaar. Disentangled counterfactual recurrent networks for treatment effect inference over time. *arXiv preprint arXiv:2112.03811*, 2021.
- [21] Alicia Curth and Mihaela van der Schaar. Nonparametric estimation of heterogeneous treatment effects: From theory to learning algorithms. In *International Conference on Artificial Intelligence and Statistics*, pages 1810–1818. PMLR, 2021.
- [22] Ioana Bica, Ahmed Alaa, and Mihaela Van Der Schaar. Time series deconfounder: Estimating treatment effects over time in the presence of hidden confounders. In *International Conference on Machine Learning*, pages 884–895. PMLR, 2020.
- [23] C Ahn, H Amer, D Anglicheau, N Ascher, C Baan, B Bat-Ireedui, Thierry Berney, MGH Betjes, S Bichu, H Birn, et al. Global transplantation covid report march 2020. *Transplantation*, 2020.
- [24] Ezekiel J. Emanuel, Govind Persad, Ross Upshur, Beatriz Thome, Michael Parker, Aaron Glickman, Cathy Zhang, Connor Boyle, Maxwell Smith, and James P. Phillips. Fair allocation of scarce medical resources in the time of covid-19. *New England Journal of Medicine*, 382(21): 2049–2055, 2020. doi: 10.1056/NEJMsb2005114. URL <https://doi.org/10.1056/NEJMsb2005114>.
- [25] Marco Vergano, Guido Bertolini, Alberto Giannini, Giuseppe R. Gristina, Sergio Livigni, Giovanni Mistraletti, Luigi Riccioni, and Flavia Petrini. Clinical ethics recommendations for the allocation of intensive care treatments in exceptional, resource-limited circumstances: the italian perspective during the COVID-19 epidemic. *Critical Care*, 24(165), 2020. doi: <https://doi.org/10.1186/s13054-020-02891-w>. URL <https://doi.org/10.1186/s13054-020-02891-w>.
- [26] Patrick S Kamath, Russell H Wiesner, Michael Malinchoc, Walter Kremers, Terry M Therneau, Catherine L Kosberg, Gennaro D’Amico, E Rolland Dickson, and W Ray Kim. A model to predict survival in patients with end-stage liver disease. *Hepatology*, 33(2):464–470, 2001.
- [27] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.
- [28] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL <https://doi.org/10.5281/zenodo.3509134>.
- [29] Katie L Connor, Eoin D O’Sullivan, Lorna P Marson, Stephen J Wigmore, and Ewen M Harrison. The future role of machine learning in clinical transplantation. *Transplantation*, 105(4):723–735, 2021.
- [30] Dennis Medved, Pierre Nugues, and Johan Nilsson. Simulating the outcome of heart allocation policies using deep neural networks. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 6141–6144. IEEE, 2018.
- [31] Manuel Dorado-Moreno, María Pérez-Ortiz, Pedro A Gutiérrez, Rubén Ciria, Javier Briceño, and César Hervás-Martínez. Dynamically weighted evolutionary ordinal neural network for solving an imbalanced liver transplantation problem. *Artificial Intelligence in Medicine*, 77: 1–11, 2017.
- [32] Dennis Medved, Mattias Ohlsson, Peter Höglund, Bodil Andersson, Pierre Nugues, and Johan Nilsson. Improving prediction of heart transplantation outcome using deep learning techniques. *Scientific reports*, 8(1):1–9, 2018.
- [33] Kyung Don Yoo, Junhyug Noh, Hajeong Lee, Dong Ki Kim, Chun Soo Lim, Young Hoon Kim, Jung Pyo Lee, Gunhee Kim, and Yon Su Kim. A machine learning approach using survival statistics to predict graft survival in kidney transplant recipients: a multicenter cohort study. *Scientific reports*, 7(1):1–12, 2017.

- [34] Alex J. Chan, Alicia Curth, and Mihaela van der Schaar. Inverse online learning: Understanding non-stationary and reactionary policies. In *International Conference on Learning Representations*, 2021.
- [35] Dimitris Bertsimas, Vivek F Farias, and Nikolaos Trichakis. Fairness, efficiency, and flexibility in organ allocation for kidney transplantation. *Operations Research*, 61(1):73–87, 2013.
- [36] Dimitris Bertsimas, Vivek F Farias, and Nikolaos Trichakis. On the efficiency-fairness trade-off. *Management Science*, 58(12):2234–2250, 2012.
- [37] Amir Elalouf, Yael Perlman, and Uri Yechiali. A double-ended queueing model for dynamic allocation of live organs based on a best-fit criterion. *Applied Mathematical Modelling*, 60: 179–191, 2018.
- [38] John P Dickerson, Ariel D Procaccia, and Tuomas Sandholm. Failure-aware kidney exchange. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 323–340, 2013.
- [39] Mustafa Akan, Oguzhan Alagoz, Baris Ata, Fatih Safa Erenay, and Adnan Said. A broader view of designing the liver allocation system. *Operations research*, 60(4):757–770, 2012.
- [40] Santiago Balseiro, Haihao Lu, and Vahab Mirrokni. Dual mirror descent for online allocation problems. In *International Conference on Machine Learning*, pages 613–628. PMLR, 2020.
- [41] Jeroen Berrevoets, Sam Verboven, and Wouter Verbeke. Treatment effect optimisation in dynamic environments. *Journal of Causal Inference*, 10(1):106–122, 2022. doi: doi:10.1515/jci-2020-0009. URL <https://doi.org/10.1515/jci-2020-0009>.
- [42] Andres E Ruf, Walter K Kremers, Lila L Chavez, Valeria I Descalzi, Luis G Podesta, and Federico G Villamil. Addition of serum sodium into the meld score predicts waiting list mortality better than meld alone. *Liver Transplantation*, 11(3):336–343, 2005.
- [43] Uri Kartoun. Towards optimally replacing the current version of meld. *Journal of Hepatology*, 2022.
- [44] James Neuberger, Alex Gimson, Mervyn Davies, Murat Akyol, John O’Grady, Andrew Burroughs, Mark Hudson, UK Blood, et al. Selection of patients for liver transplantation and allocation of donated livers in the uk. *Gut*, 57(2):252–257, 2008.
- [45] David Goldberg, Alejandro Mantero, Craig Newcomb, Cindy Delgado, Kimberly Forde, David Kaplan, Binu John, Nadine Nuchovich, Barbara Dominguez, Ezekiel Emanuel, et al. Development and validation of a model to predict long-term survival after liver transplantation. *Liver transplantation*, 27(6):797–807, 2021.
- [46] W Ray Kim, Ajitha Mannalithara, Julie K Heimbach, Patrick S Kamath, Sumeet K Asrani, Scott W Biggins, Nicholas L Wood, Sommer E Gentry, and Allison J Kwong. Meld 3.0: the model for end-stage liver disease updated for the modern era. *Gastroenterology*, 161(6): 1887–1895, 2021.
- [47] Neta Gotlieb, Amirhossein Azhie, Divya Sharma, Ashley Spann, Nan-Ji Suo, Jason Tran, Ani Orchanian-Cheff, Bo Wang, Anna Goldenberg, Michael Chassé, et al. The promise of machine learning applications in solid organ transplantation. *NPJ digital medicine*, 5(1):1–13, 2022.
- [48] Tennison Liu, Zhaozhi Qian, Jeroen Berrevoets, and Mihaela van der Schaar. Goggle: Generative modelling for tabular data by learning relational structure. In *The Eleventh International Conference on Learning Representations*, 2022.
- [49] Doina Precup. Eligibility traces for off-policy policy evaluation. *Computer Science Department Faculty Publication Series*, page 80, 2000.
- [50] Philip S Thomas. *Safe reinforcement learning*. PhD thesis, University of Massachusetts Libraries, 2015.
- [51] Miroslav Dudík, John Langford, and Lihong Li. Doubly robust policy evaluation and learning. *arXiv preprint arXiv:1103.4601*, 2011.
- [52] John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.
- [53] Michael JD Powell and J Swann. Weighted uniform sampling—a monte carlo technique for reducing variance. *IMA Journal of Applied Mathematics*, 2(3):228–236, 1966.
- [54] Yash Chandak, Scott Niekum, Bruno Castro da Silva, Erik Learned-Miller, Emma Brunskill, and Philip S Thomas. Universal off-policy evaluation. *arXiv preprint arXiv:2104.12820*, 2021.

- [55] Alizée Pace, Alex J. Chan, and Mihaela van der Schaar. Poetree: Interpretable policy learning with adaptive decision trees. In *International Conference on Learning Representations*, 2021.
- [56] Keisuke Hirano, Guido W Imbens, and Geert Ridder. Efficient estimation of average treatment effects using the estimated propensity score. *Econometrica*, 71(4):1161–1189, 2003.
- [57] Donald B Rubin. Estimating causal effects from large data sets using propensity scores. *Annals of internal medicine*, 127(8\_Part\_2):757–763, 1997.
- [58] Guido W Imbens and Donald B Rubin. *Causal inference in statistics, social, and biomedical sciences*. Cambridge University Press, 2015.
- [59] Guido W Imbens and Donald B Rubin. Rubin causal model. In *Microeconometrics*, pages 229–241. Springer, 2010.
- [60] Steven Piantadosi. *Clinical trials: a methodologic perspective*. John Wiley & Sons, 2017.
- [61] Stuart J Pocock. *Clinical trials: a practical approach*. John Wiley & Sons, 2013.
- [62] Lawrence M Friedman, Curt D Furberg, David L DeMets, David M Reboussin, and Christopher B Granger. *Fundamentals of clinical trials*. Springer, 2015.
- [63] Rebecca DerSimonian and Nan Laird. Meta-analysis in clinical trials. *Controlled clinical trials*, 7(3):177–188, 1986.
- [64] Ahmed Alaa, Alex J. Chan, and Mihaela van der Schaar. Generative time-series modeling with fourier flows. In *International Conference on Learning Representations*, 2020.
- [65] Joel Z. Leibo, Edgar Dué nez Guzmán, Alexander Sasha Vezhnevets, John P. Agapiou, Peter Sunehag, Raphael Koster, Jayd Matyas, Charles Beattie, Igor Mordatch, and Thore Graepel. Scalable evaluation of multi-agent reinforcement learning with melting pot. In *International Conference on Machine Learning*, pages 6187–6199. PMLR, 2021.
- [66] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.
- [67] Tom Stepleton. The pycolab game engine, 2017.
- [68] Alex J. Chan and Mihaela van der Schaar. Scalable bayesian inverse reinforcement learning. In *International Conference on Learning Representations*, 2020.
- [69] Paul W Holland. Statistics and causal inference. *Journal of the American statistical Association*, 81(396):945–960, 1986.
- [70] Vikram Kilambi, Kevin Bui, and Sanjay Mehrotra. Livsim: an open-source simulation software platform for community research and development for liver allocation policies. *Transplantation*, 102(2), 2018.
- [71] Mohd Shoaib, Utkarsh Prabhakar, Sumit Mahlawat, and Varun Ramamohan. A discrete-event simulation model of the kidney transplantation system in rajasthan, india. *Health Systems*, 11(1):30–47, 2022.
- [72] Shoaib Mohd, Navonil Mustafee, Karan Madan, and Varun Ramamohan. Leveraging healthcare facility network simulations for capacity planning and facility location in a pandemic. *Available at SSRN 3794811*, 2021.
- [73] Cecilia Nardini. The ethics of clinical trials. *Ecancermedicalscience*, 8, 2014.
- [74] Jack Cuzick, Robert Edwards, and Nereo Segnan. Adjusting for non-compliance and contamination in randomized clinical trials. *Statistics in medicine*, 16(9):1017–1029, 1997.
- [75] David Maxwell Chickering and Judea Pearl. A clinician’s tool for analyzing non-compliance. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1269–1276, 1996.
- [76] Oswald Nitski, Amirhossein Azhie, Fakhar Ali Qazi-Arisar, Xueqi Wang, Shihao Ma, Leslie Lilly, Kymberly D Watt, Josh Levitsky, Sumeet K Asrani, Douglas S Lee, et al. Long-term mortality risk stratification of liver transplant recipients: real-time application of deep learning algorithms on longitudinal data. *The Lancet Digital Health*, 3(5):e295–e305, 2021.
- [77] Ina Jochmans, Marieke van Rosmalen, Jacques Pirenne, and Undine Samuel. Adult liver allocation in eurotransplant. *Transplantation*, 101(7):1542–1550, 2017.

- [78] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.
- [79] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [80] Sören R Künnel, Jasjeet S Sekhon, Peter J Bickel, and Bin Yu. Metalearners for estimating heterogeneous treatment effects using machine learning. *Proceedings of the national academy of sciences*, 116(10):4156–4165, 2019.
- [81] Ioana Bica, James Jordon, and Mihaela van der Schaar. Estimating the effects of continuous-valued interventions using generative adversarial networks. *Advances in Neural Information Processing Systems*, 33:16434–16445, 2020.
- [82] Xinkun Nie and Stefan Wager. Quasi-oracle estimation of heterogeneous treatment effects. *Biometrika*, 108(2):299–319, 2021.
- [83] Edward H Kennedy. Optimal doubly robust estimation of heterogeneous causal effects. *arXiv preprint arXiv:2004.14497*, 2020.
- [84] Scott Powers, Junyang Qian, Kenneth Jung, Alejandro Schuler, Nigam H Shah, Trevor Hastie, and Robert Tibshirani. Some methods for heterogeneous treatment effect estimation in high dimensions. *Statistics in medicine*, 37(11):1767–1787, 2018.

## Checklist

1. For all authors...
  - (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
  - (b) Did you describe the limitations of your work? [Yes]
  - (c) Did you discuss any potential negative societal impacts of your work? [Yes]
  - (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
  - (a) Did you state the full set of assumptions of all theoretical results? [N/A]
  - (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments (e.g. for benchmarks)...
  - (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
  - (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
  - (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
  - (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [N/A] These are non compute-intensive experiments
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
  - (a) If your work uses existing assets, did you cite the creators? [Yes]
  - (b) Did you mention the license of the assets? [Yes] When using the code
  - (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
  - (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
  - (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] All used data is public
5. If you used crowdsourcing or conducted research with human subjects...
  - (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
  - (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
  - (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]



# Appendix: AllSim

## Table of Contents

---

<b>A Simulation Life-cycle</b>	<b>17</b>
A.1 (i) Sample users and resources	17
A.2 (ii) notify policy of users	18
A.3 (iii) allocate resources	18
A.4 (iv) consume resources	19
<b>B Extended Related Work</b>	<b>19</b>
B.1 Medical simulation	20
<b>C Extended results</b>	<b>21</b>
C.1 Running a simulation	21
C.2 Analysing AllSim’s output.	21
<b>D Details on AllSim</b>	<b>22</b>
D.1 Neat functionality	22
D.2 Package structure	23
<b>E Using AllSim</b>	<b>24</b>
E.1 Getting started	26
<b>F Social impact</b>	<b>27</b>
<b>G AllSim in a vaccine distribution scenario</b>	<b>27</b>
G.1 Online Learning	29
G.2 Low-level API example	29
<b>H Counterfactual Inference</b>	<b>29</b>

---

## A Simulation Life-cycle

Using AllSim’s definition in Sect. 2, we will formalise a high-level simulation life-cycle. In simple terms, the simulation loop will model the interaction between  $\mathcal{E}$  and the `policy` for each consecutive day ( $t = 1, 2, \dots$ ) until some pre-specified end ( $T$ ). We have organised this section around Algorithm 1 where we connect each important line with a component in Figure 6 and discuss it in a separate subsection. In Figure 6 we illustrate the interaction between each component, and clarify which parts of the simulation can be learned, and which parts should be parameterised by the practitioner.

### A.1 (i) Sample users and resources

The first step in Algorithm 1 comprises the first and second component in Figure 6, repeated below,

$$\underbrace{\lambda_x, \lambda_r}_{\text{arrival amount}} \underbrace{\left( \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{matrix} \right) \#}_{\text{sample objects}} \rightsquigarrow \mathbf{X}(t), \mathbf{R}(t).$$

Essentially, the goal in ln 4 of Algorithm 1 is to translate the arrival rates to a set of users and resources, trademarked by their features sets. With above formulation, we model the arrival processes as a two-step approach: (1) sample the amount of objects we may expect, (2) sample the objects from a distribution. Indicated in Figure 6, only some of these component are learned, and others user-defined. While not limited to, our examples focus on tabular data. Hence, any density strategy

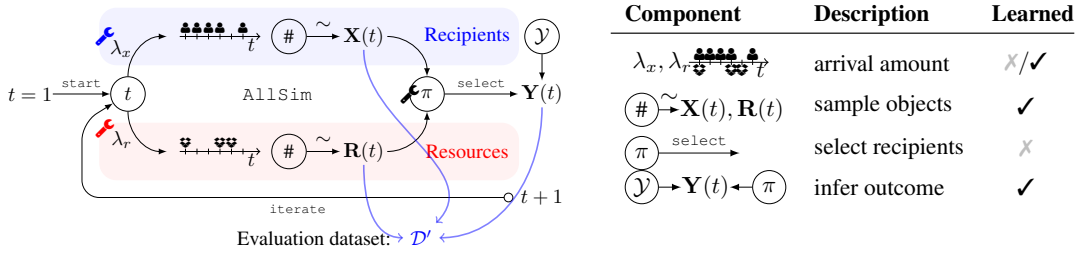


Figure 6: **Simulation life-cycle of AllSim.** In the leftmost part of this figure, we have illustrated the simulation life-cycle. In the rightmost part, we list each component and indicate whether or not they are learned from data. *Simulation life-cycle.* First, we sample the amount of each object arrives on day  $t$ . The expected amount is indicated by  $\lambda_x$  and  $\lambda_r$  for the users, and resources, respectively. The expected amount should be interpreted as an arrival rate. Once the amount of objects is specified, they are sampled from a distribution  $\mathbf{X}(t)$  and  $\mathbf{R}(t)$ . Next, each object is presented to the policy, we wish to test, and the policy replies by selecting a set of users whom the resources will be presented to. Finally, the simulation evaluates the policy’s decision, by providing an inferred outcome. Wrench icons (⚒) and an ✗’s in the table indicate which components can be user-specified, others (with ✓) are learned. Finally,  $\mathbf{X}(t)$ ,  $\mathbf{R}(t)$ , and  $\mathbf{Y}(t)$  compose a dataset  $\mathcal{D}'$  for evaluation.

**Algorithm 1: Main simulation loop.** This simulation life-cycle acts as a section overview for Appendix A. In our main text we discuss how each line is simulated.

**input** : Environment,  $\mathcal{E}$ ; A resource allocation policy denoted as `pol`

**output** : Policy runtime summary

```

1 Start,  $t = 0$ ;
2 while simulation runs do
3    $t \leftarrow t + 1$ ; /* iterate time */
4    $\mathbf{X}(t), \mathbf{R}(t) \sim \mathcal{E}(t)$ ; /* (i) sample users and resources (Appendix A.1) */
5   pol.add( $\mathbf{X}(t)$ ); /* (ii) notify policy of users (Appendix A.2) */
6    $\mathbf{X}_{\mathbf{R}(t)} \leftarrow \text{pol.select}(\mathbf{R}(t))$ ;
   /* (iii) allocate resources (Appendix A.3) */
7    $\mathbf{Y}_t \sim \mathcal{Y}_{\mathcal{E}}(\mathbf{X}_{\mathbf{R}(t)}, \mathbf{R}(t))$ ; /* (iv) consume resources (Appendix A.4) */
8 end

```

should handle such data. This is not an easy challenge, since the structure in tabular data is best respected [9, 48].

### A.2 (ii) notify policy of users

After the simulation has sampled  $\mathbf{X}(t)$  and  $\mathbf{R}(t)$ , the `policy.add( $\mathbf{X}(t)$ )` function is called. Specifically, this notifies the policy of the arrival of new users, which in Figure 6 corresponds to,

$$\mathbf{X}(t), \mathbf{R}(t) \rightarrow \pi$$

Note that the above is simply a service provided by the simulation to the policy, and is thus not a learned. The reason for this is, that the organisation of in-need-users is entirely up to the policy. For example, more traditional policies may maintain only one priority queue [44], whereas more novel policies may maintain multiple, based on recipient and resource types [1]. As such AllSim remains agnostic to the tested policy, resulting in a more general-purpose framework.

### A.3 (iii) allocate resources

Like Appendix A.2, step (iii) is entirely managed by the tested policy, and in Figure 6 corresponds with,

$$\pi \xrightarrow{\text{select}}$$

Selecting which users that get to consume the resource, is generally what differentiates policies. When a set of users is selected (through a `policy.select( $\mathbf{R}(t)$ )` call in the python interface, AllSim retains every piece of information associated with the select-call, which is used to evaluate.

Having the simulation retain all this information, allows AllSim to output a dataset of past policy behaviour, much like the original (real-world) dataset we provided to the simulation in order to learn

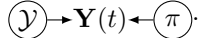
Table 2: **Overview of related work.** We categorise our related work in four categories: off-policy learning, causal inference, clinical simulations, simulations for evaluating reinforcement learning agents. For each category we provide the most prominent method of calculating policy performance, and whether the categories take into account four major questions: (i) is the method tunable to new settings?; (ii) can we evaluate using data?; (iii) are the performance estimates unbiased?; and (iv) can we evaluate beyond simple aggregate descriptive statistics?

	Citations	Perf. calculation	(i)	(ii)	(iii)	(iv)
Off-policy learning	[49–55]	$\frac{p_{\pi'}(R X)}{p_{\pi}(R X)} Y \sim \mathcal{D}^{\pi}$	✗	✓	✓	✗
Causal inference	[56–59]	$p(R X)^{-1} Y \sim \mathcal{D}^{\pi}$	✗	✓	✓	✗
Clinical simulations	[44, 60–64]	$\mathbb{E}_{\mathcal{D}^{\pi}}[Y X]$	✗	✓	✗	✓
Reinforcement learning	[10–13, 65–68]	$\mathbb{E}_{\text{sim}}[v(Y) X, R]$	✓	✗	✓	✓
AllSim	(ours)	$\mathbb{E}_{\mathcal{D}^{\pi'} \sim \mathcal{X}' \times \mathcal{R}'}[Y X]$	✓	✓	✓	✓

the various components. In essence, with AllSim we are able to sample a synthetic dataset of a counterfactual scenario such that we are able to test a policy of interest, *as if it were already in use*.

#### A.4 (iv) consume resources

The final component in AllSim is the inference-component, yielding an outcome after a resource was consumed by a user. Importantly, this component allows us to evaluate a policy, despite it deviating from the data used to learn the simulation. In Figure 6, this component is illustrated as,



The outcome is a function of the resource and its recipient, making inference hard as some combinations are less observed in the original data. Essentially, the tested policy may make out-of-distribution combinations, as the simulation is learnt from data that was collected under some other policy, illustrated in Figure 2 where two policies,  $\pi$  and  $\pi'$  result in different datasets  $\mathcal{D}$  and  $\mathcal{D}'$ .

**Counterfactual inference.** AllSim handles this by using a counterfactual estimator to infer allocation outcomes as they deal with this issue explicitly. Counterfactual methods aim to make an unbiased prediction of the potential outcome, associated with some treatment (or resource). We are interested in counterfactual methods that model the potential outcomes for the recipients when they are/are not allocated a resource. A counterfactual estimator then “completes” the simulated dataset as,

$$Y(t) = \mathbb{E}[\hat{Y}(\mathbf{R}(t)) | \mathbf{X}_{\pi}(t)], \quad (6)$$


where  $\hat{Y}(\mathbf{R}(t))$  is the estimated potential outcome, using methodology known in the potential outcomes literature, and  $\mathbf{X}_{\pi}(t)$  are the recipients selected by  $\pi$  at time  $t$ . Equation (6) provides the recipient-resource pair with an estimated outcome, and presents the practitioner with a dataset:

$$\mathcal{D}^{\pi} := \{(X, R, \hat{Y}(R), t)_i : i = 1, \dots, N\}.$$

$\mathcal{D}^{\pi}$  allows to easily calculate clinical measures of performance, as we demonstrate in Sect. 3. More information regarding these counterfactual models is provided in Appendix H.

## B Extended Related Work

We have summarised work related to ours in four major categories: (a) Off-policy learning and evaluation, (b) causal inference, (c) clinical simulations and trials, and (d) simulations for evaluating reinforcement learning (RL) agents. A high-level summary of these areas can be found in Table 2. For a discussion on related work in the clinical domain, we refer to our Appendix B.

**(a) Off-policy evaluation and (b) causal inference.** Data are collected under some active resource allocation policy, which determines how resources are paired with users, which in turn determine the outcomes we get to observe. Clearly, our setting is connected to off-policy evaluation, as we wish to evaluate a policy that is different from the policy that collected the available data. Figure 2 illustrates different matches (depicted as ) made across different policies resulting in different outcomes:  $Y$  or  $Y'$ , and datasets,  $\mathcal{D}$  and  $\mathcal{D}'$ . The difficulty of this situation, is that we observe either  $\mathcal{D}$  or  $\mathcal{D}'$ , *not both*, relating to the potential outcomes setup [17, 18, 69]. Using  $\mathcal{D}$  to evaluate  $\pi$  simply means calculating some descriptive statistics. In contrast, evaluating  $\pi'$  on  $\mathcal{D}$  is much more involved, as the same statistics would yield a biased estimate.

One way to calculate, for example the average  $Y' \sim \mathcal{D}$ , is to weigh each sample:  $g(\text{person}, \text{policy})Y \sim \mathcal{D}$ . In the literature on off-policy evaluation (and learning), this is known as *importance sampling*, where each sample is weighted according to the likelihood of it belonging in  $\mathcal{D}'$ . Interestingly, the exact same strategy is also widely known in counterfactual learning and causal inference, under the name *inverse propensity weighting* (IPW) [51]. The key difference between both, is the target distribution. Where importance sampling transforms the estimate from one policy’s distribution, to another; IPW transforms from a policy’s distribution, to an unbiased estimate (i.e. to a  $\frac{1}{N}$  weighting for averages).

Shown in Table 2, neither IPW nor importance sampling, provide a solution to evaluating scarce resource allocation policies. One reason for this is that the outcome is a (weighted) estimate of only one aggregate descriptive statistic (for example, expected outcome or reward). When we want additional estimates, one requires a different weighting scheme [54]. Furthermore, when we wish to test a policy in a different environment, where for example the recipients’ or resources’ distributions change, evaluating a policy becomes increasingly more difficult if one wishes to rely on IPW or importance sampling due to the increased differences in likelihood density. Furthermore, evaluation goes much beyond aggregate statistics, as we may be interested in, for example, demographic differences between recipients and non-recipients (Appendix C includes an example using AllSim).

**(c) Clinical evaluation and trials.** A naive method to evaluating these policies, is to model  $Y'$  using simple (linear [44]) regression. Any combination  $\text{person} \xleftarrow{\pi'} \text{policy}$ , through a new policy,  $\pi'$  has an estimated outcome  $\hat{Y}'$  using a regression model, trained on  $\mathcal{D}$ . Herein lies the problem: the regression model is still biased to  $\pi$ . In particular,  $\mathcal{D}$  simply does not contain pairs that  $\pi'$  would make, and therefore has trouble estimating  $Y'$ . By using causal methodology (Appendix A.4), this is one key area where AllSim improves upon contemporary clinical simulations [44, 70–72], as these only “replay” the past without the possibility of changing the environment characteristics nor allow counterfactual inference.

Clinicians do have another way to account for this: clinical trials [60, 61]. Clinical trials estimate a causal estimand, which could then be used to “complete” the dataset as a naive regression model would above. However, two problems arise: setting up a clinical trial can sometimes be considered unethical (especially when dealing with scarce resources, and more specific research questions) [73], or clinical trials may suffer from compliance issues which will still result in biased outcomes [74, 75].

**(d) Simulations for evaluating RL agents.** A final set of solutions is that of simulations in which an RL agent can act and learn. Naturally, there are differences between policies for RL and scarce resource allocation, such as the importance of future reward which implies correlated consecutive states, and a fixed action space. However, there are also similarities, such as the definition of a policy (being a set of rules which take into account a context or state), and the online nature of the problem.

Literature on RL is blessed with some of the most well-known and actively maintained libraries for evaluating a wide range of RL algorithms [10, 11, 67]. However, other than being inapplicable in our setting (for the reasons included above), they also have one other major downside: *they do not learn from data*. While these simulations model some very interesting environments, each aspect of the simulation is hard-coded. In order to evaluate a policy that is to be deployed in a real-world setting, we have to test it to the specifics of the environment of interest; to be defined by the practitioners developing the policy. In fact, we consider this a major reason for developing and using AllSim.

AllSim marks a significant advance over the current state-of-the-art: currently *no evaluation technique* allows a practitioner to *change the behaviour of the evaluation environment*, despite the fact that the actual environment most certainly will change in the future; nor do contemporary evaluation techniques account for potential bias from previous policies, resulting in poor performance estimates.

## B.1 Medical simulation

Let us discuss how the literature which introduces novel allocation policies, in particular to donor-organ allocation. From this, we observe that they all come up with their own unique simulation in order to validate their proposal.

**Allocation policies from medicine.** As an example, consider the following papers introduced in the medical domain (we focus here on the liver allocation setting as this is also where we focused on in our paper): [43–46, 76, 77]. Each paper, all coming from different research groups, provide a custom simulation to validate their allocation policy. While some focus only on gathering test data (such as

the recent [46]), others construct a simulation by simply iterating over the patients in the sequence they arrived in reality (such as [44]). While we would note some flaws in the way these policies are evaluated (e.g. counterfactual trajectories when allocations misalign), the truly striking observation is that *each evaluation strategy is different!* Not one paper reuses the same simulation; AllSim may change this going forward.

**Allocation policies from machine learning and OR.** The same is true for the ML [1–3, 29–33] and OR [35–40] communities. It seems that both the ML and OR community is focused more and more on this important problem– which is fantastic! But it also warrants careful evaluation. Furthermore, if we find that the evaluation strategies in medicine (which generally propose linear combinations of features [26, 42] or simple CoxPH models [43–46]) have shortcomings, then this is certainly the case for much more complicated strategies introduced in ML or OR. In fact, a recent survey confirmed exactly that: [47, cfr. Limitations of ML in transplant medicine].

Naturally, there problems solved by the OR community concern a *variant* of the general problem presented in this paper. For example, [40] are concerned with distributing a *fixed* set of resources, to a varying set of incoming users. While different, such problems can still be modelled in AllSim. In the specific case of [40], the resources are not unique (they are represented by an amount) and require much less machinery than what we require to model the varying resource scenario. Specifically, one can model the remaining amount of resources as an attribute in our `Policy` class.

**SimUnet.** While indeed related, SimUnet is in fact very different from AllSim. In particular, from the online README document (see <https://unos.org/wp-content/uploads/1-page-SimUnet.pdf>) it can be seen that SimUnet is more concerned to simulate *offers* from the transplant clinician’s point of view; not *allocations* from an overarching healthcare system (such as UNOS, or the NHS).

## C Extended results

### C.1 Running a simulation

**Tuning the simulation.** Running and composing a simulation, is as simple as defining how we wish to let the supply of resources and recipients evolve in  $t$ . Such evolution is expressed as a changing arrival rate (cfr. Equations (3) and (4)). In particular, one only needs to define  $\lambda_{x,r}$ , normalisation is handled by AllSim. However, even there does AllSim offer detailed specification possibilities. Like  $\lambda_{x,r}$ , we can specify a custom function for  $\alpha_{x,r}$  also. Normalisation, will then respect the value of  $\alpha$  at time  $t$ . For example, if we wish the total arrival rate of all resources to remain constant at 5, we simply set  $\alpha_r$  to a constant function:

```
1 alpha_r = lambda t: 5
```

This way, no matter how complicated our functions for  $\lambda_r$ , are, we know that the total arrival rate, across all conditioned distributions for the resources, we remain fixed at 5.

**Inference.** One additional component to AllSim, is its `Inference` module. In our simulations, we relied on OrganITE [2], where we adopted the original source-code provided by the authors, to our `Inference` module. Having a causal model, allows to have unbiased estimates for the resource to recipient pairing’s outcome (column “Y” in Table 3).

Building a custom `Inference` object, is as simple as inheriting from `Inference`, which requires the user to implement the `infer(x: np.ndarray, r: np.ndarray) -> Any` function.

### C.2 Analysing AllSim’s output.

Running `df = simulation.simulate(policy, T=T)` yields a `DataFrame` object, containing each match made by the `policy`, in the environment simulated by the `ss.Simulation` object. Consider Table 3 for an excerpt of the output for which we provided some aggregate results in Figure 4. Table 3 is exactly the type of data one may expect when learning about a policy, or even learning a new (ML-driven) policy [2].

Given the output of our simulation, we can see each variable, both donor as well as recipient progress over time. For example, one may be interested how each gender evolves w.r.t. allocations, with the

Table 3: **Example output.** When donor covariates are NaN, we know a patient died on the wait list, as they did not receive a donor organ. Note that, we only display some covariates, we have included code in our submission, should the interested reader want to inspect the complete DataFrame.

Donor covariates			Recipient covariates		Gender	Y	Day
Age	BMI	INR	Sodium	Creatinine			
28.1628	27.7080	4.8144	138.10	11.351	M	1792.9	1
NaN	NaN	0.5663	137.71	0.4535	F	5.0	7
NaN	NaN	6.694	135.63	1.7323	F	24.948	26
34.893	21.852	3.8375	134.15	7.9500	M	257.79	1
...							

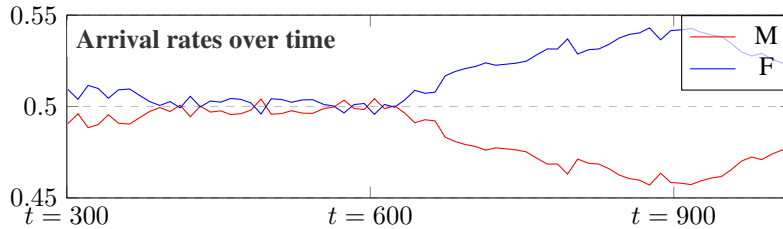


Figure 7: **Recipient-gender in function of time.** Given the MELD policy, and the changing supply of resources (cfr. Figure 4, we find that the recipient-gender changes slightly. Note that, this type of analysis is easily done given AllSim, as we can easily run analysis on the provided DataFrame object, returned by our simulation.

changing supply of resources (as in Figure 4). Naturally, this is but one example of what one can achieve in terms of analysis on MELD (or any other policy), using AllSim. As such, we encourage the reader to browse through the provided code, in order to get a sense of what the possibilities really are when using AllSim.

## D Details on AllSim

We provide some additional detail on AllSim’s functionality, structure, and future goals.

Please find the code for AllSim online at:

<https://github.com/jeroenbe/allsim>  
 or  
<https://github.com/vanderschaarlab>

### D.1 Neat functionality

**One-hot encoded variables.** In heterogeneous data, one may expect a mixture of continuous and categorical variables. Good practice to handle categorical variables, is to one-hot encode them into columns of ones and zeroes, for each category. Most density estimation methods, do not automatically sample these mixtures of continuous and categorical data. Despite its performance and simplicity, a standard Kernel Density estimator, like the one we use in our simulation, cannot handle these data out-of-the-box. However, our implementation *does*. One only needs to provide the specific groups of columns which compose the categorical variables.

For example, the gender column for the recipient may be one-hot encoded into: "GENDER\_0" and "GENDER\_1". When fitting our KDEDensity, we simply provide these columns to the one\_hot\_encoded=["GENDER\_0", "GENDER\_1"] parameter, and the KDEDensity automatically transforms these columns, in such a way that only one of them (being the one with maximum probability), is translated into 1.

**Auto-scaling.** Dealing with a normalised arrival rate can be tricky to determine up front. As such, AllSim does this automatically. By solely providing an

alpha: Callable[[int], float], AllSim's PoissonSystem objects will normalise any output that the System's PoissonProcesses may have. Naturally, normalisation will depend on the chosen process, and as such does not propagate up to the general System nor Process. If, for example, one wishes to implement an alternative point process (not a Poisson process), normalisation may look differently.

**Automatic conditioning.** Another nice functionality provided in AllSim is automatic conditioning, in case no Condition is provided. A Density does this, by first clustering the data (from which it will later learn its density), into  $K$  clusters. Naturally, one will have to provide the amount of clusters before the Density is able to automatically condition. Having these separate clusters, still allows defining arrival rates and processes for each cluster center, i.e.  $K$  must match the amount of Processes are provided to the System.

## D.2 Package structure

The AllSim package structure is as follows:

```

1 |_ src
2   |_ infer.py
3   |_ sim.py
4   |_ outcome
5     |_ counterfactual_inference.py
6     |_ counterfactual_models.py
7   |_ policies
8     |_ base.py
9     |_ policy.py

```

The two main components are in infer.py and sim.py, where infer.py contains everything related to Density, Process, and their subclasses; and sim.py concerns the Simulation class.

In outcome and policy, we provide basic implementations of some well known policies and counterfactual models. Note that these are actually non-essential to AllSim, as they can be completely user-defined, as long as they respect the Inference and Policy abstract classes, provided in the respective folders.

**Inheritance.** The base classes (Density, Policy, and Inference) can all be implemented by the user. Consider following class structures:

```

1   class Density:
2       def __init__(self
3           condition: Condition, # see below for more details
4           K: int=1,
5           drop: np.ndarray=np.ndarray(['condition'])):
6           ...
7
8       @abstractmethod
9       def sample(self, n: int=1) -> np.ndarray:
10          ...
11
12      @abstractmethod
13      def fit(self, D: pd.DataFrame) -> None:
14          ...
15
16      class Policy:
17          def __init__(self,
18              self,
19              name: str,
20              initial_waitlist: np.ndarray,
21              dm: OrganDataModule): # a standardised datamodule should the
22              policy want to learn from data
23          ...
24
25      @abstractmethod:
26      def select(self, resources: np.ndarray) -> Tuple[np.ndarray, np.
27          ndarray]:

```

```

26         # returns recipients and resources, matched
27         ...
28
29     @abstractmethod
30     def add(self, x: np.ndarray) -> None:
31         # allows policy to add recipients to internal waitlist (if
needed)
32         ...
33
34     @abstractmethod
35     def remove(self, x: np.ndarray) -> None:
36         # allows policy to remove recipients, when for example they
die
37         ...
38
39     class Inference
40     def __init__(self,
41                 model: Any, # the Inference class acts as a wrapper for
any model type
42                 mean: float=0, # mean and std are necessary to scale the
outcomes, assuming
43                 std: float=1): # standard scaling
44         ...
45
46     def __call__(self, x: np.ndarray, r: np.ndarray, *args: Any, **
kwargs: Any) ->Any:
47         return self.infer(x, r, *args, **kwargs)
48
49     @abstractmethod
50     def infer(self, x: np.ndarray, r: np.ndarray, *args: Any, **
kwargs: Any) -> Any:
51         ...

```

The Condition class then, wraps a function from a set of labels in the dataset, to a numerical value. One may choose to combine multiple variables, or just one, as we have with the "AGE" variable in Figures 4 and 5. Please find an example initialisation in Figure 5. While one *can* inherit from the Condition class, we would advise to implement a custom function (as the lambda function we had), and provide it to the Condition object.

**Future goals & open-source** Our goal is for AllSim to be a benchmarking standard when evaluating ScRAPs. An important milestone for this, is to completely open-source our simulation. Doing so, allows other researchers to scrutinise, enhance, and discuss how we should move beyond what we can do today. Below, we provide two points we believe our community should discuss.

## E Using AllSim

Here we provide the basic code, that will generate Figure 4. Naturally, the complete code is provided in the supplemental materials.

```

1     # load data
2     X, R, Y = custom_load_function(data)
3     organite = load_organite_model(location) # should load an implemented
Inference object
4
5     # DENSITY LEARNING
6     # RESOURCES
7     bins_r = [30, 45, 60]
8     def condition_function_r(age):
9         return np.digitize(age, bins=bins_r).item()
10
11
12     condition_r = infer.Condition(
13         labels=['AGE'],
14         function=condition_function_r,

```



```

15     options=len(bins_r) + 1
16 )
17
18 kde_r = infer.KDEDensity(condition=condition_r, K=condition_r.options
19 )
20 kde_r.fit(R, one_hot_encoded=groups_r)
21
22 # RECIPIENTS
23 bins_x = [30, 45, 60]
24
25 def condition_function_x(bilir):
26     return np.digitize(bilir, bins=bins_x).item()
27
28 condition_x = infer.Condition(
29     labels=['AGE'],
30     function=condition_function_x,
31     options=len(bins_x) + 1
32 )
33
34 kde_x = infer.KDEDensity(condition=condition_x, K=condition_x.options
35 )
36 kde_x.fit(X, one_hot_encoded=groups_x)
37
38 # BUILD THE SYSTEMS
39 resource_system, patient_system = dict(), dict()
40
41 def update_lam_0(t):
42     return (1 / (1+np.exp(-(t-450)/150))) * 3
43
44 def update_lam_1(t):
45     return (1 / (1+np.exp((t-350)/150))) * 2
46
47 def update_lam_2(t):
48     a = (1 / (1+np.exp((t-150)/150))) * 2
49     b = (1 / (1+np.exp(-(t-650)/100))) * 2
50     return (a + b)
51
52 def update_lam_3(t):
53     a = (1 / (1+np.exp(-(t-150)/150)))
54     b = (1 / (1+np.exp((t-650)/100)))
55     return (a + b)
56
57
58 resource_system[0] = infer.PoissonProcess(update_lam=update_lam_0)
59 resource_system[1] = infer.PoissonProcess(update_lam=update_lam_1)
60 resource_system[2] = infer.PoissonProcess(update_lam=update_lam_2)
61 resource_system[3] = infer.PoissonProcess(update_lam=update_lam_3)
62
63
64 patient_system[3] = infer.PoissonProcess(update_lam=update_lam_0)
65 patient_system[2] = infer.PoissonProcess(update_lam=update_lam_1)
66 patient_system[1] = infer.PoissonProcess(update_lam=update_lam_2)
67 patient_system[0] = infer.PoissonProcess(update_lam=update_lam_3)
68
69
70 resource_process = infer.PoissonSystem(
71     density=kde_r,
72     system=resource_system,
73     alpha=lambda t: 5,
74     normalize=True)
75
76 patient_process = infer.PoissonSystem(
77     density=kde_x,

```

```

78     system=patient_system,
79     alpha=lambda t: 7,
80     normalize=True)
81
82     organite.model.eval()
83
84     simulation = sim.Sim(
85         resource_system=resource_process,
86         patient_system=patient_process,
87         inference=organite
88     )
89
90     policy = MELD(
91         name='MELD', initial_waitlist=simulation._internal_waitlist, dm=
92         dm
93     )
94     df = simulation.simulate(policy, T=1021)

```

## E.1 Getting started

To use AllSim, a user requires at least a dataset of the following type:  $\mathcal{D} := \{(X_t, R_u, Y) : t, u \in \mathbb{N}_+\}$ , with  $t, u$  indicating the time of arrival. In principle, this is sufficient to start using AllSim already. In fact, this is exactly what we provided AllSim in our experiment in Figure 4. Let us elaborate:

AllSim requires to specify the following components:

1. A counterfactual model
2. The patient and resource arrival processes
3. The patient and resource densities

Components 1 and 3 are easily implemented using existing packages like `econml` (<https://econml.azurewebsites.net>) and `scikit-learn` (<https://scikit-learn.org/stable/>), respectively. They only require to use the `model.fit` API to be applied to the users' data:

```

1     counterf = econml.BaseCateEstimator()
2     counterf.fit(Y, R, X)
3
4
5     density_X, density_R = allsim.infer.KDEDensity(), allsim.infer.
6     KDEDensity()
7     density_X.fit(X)
8     density_R.fit(R)

```

The only place where we need AllSim specifically is when we create the arrival processes, and the eventual simulation. Creating an arrival process on data, first requires us to regress the arrival probability on time:

```

1     from sklearn.preprocessing import PolynomialFeatures
2     from sklearn.linear_model import LinearRegression
3     from sklearn.pipeline import Pipeline
4
5     func_X = function(Pipeline([
6         ('poly', PolynomialFeatures(degree=6)),
7         ('linear', LinearRegression(fit_intercept=True))]))
8
9     # with amount() a function that returns how many of X arrived at t in
10    D
11    func_X.fit(t, amount(X, t))
12
13    # with the arrival processes

```

```

13 patient_process = allsim.infer.PoissonProcess(func_X)
14 resource_process = allsim.infer.PoissonProcess(func_R)
15
16 patient_system = allsim.infer.PoissonSystem(density_X,
17 patient_process)
18 resource_system = allsim.infer.PoissonSystem(density_R,
19 resource_process)

```

The simulation is then created as:

```

1 simulation = asim.sim.Sim(
2     resource_system=resource_system,
3     patient_system=patient_system,
4     inference=counterf
5 )

```

The only thing left is to actually run the policy against our created simulation:

```

1 df = simulation.simulate(policy, T=100) # thats it!

```

## F Social impact

It is very clear that machine learning has the potential to transform healthcare. Its success both in other domains and already within healthcare is very promising. ML-based policies have the potential to extend lives. However, as with almost any other machine learning method, there are risks associated with their deployment in a real healthcare setting. Before any (experimental) ML-based policy (or even non-ML-based policies for that matter) are to be deployed, they require thorough testing.

We believe AllSim will enable practitioners to leverage their data and evaluate their policies in a rigorous manner. AllSim fully recognises the difficulties associated with evaluating policies that diverge from the data-generating policy and aims to mitigate these difficulties by employing tried and tested methods from causality. Naturally, there are some caveats: AllSim inherits the potential assumptions made by the counterfactual method, furthermore, if real data is used, it is crucial it is at least *somewhat* related to the environment the tested policy will end up operating in. If for example, one aims to evaluate a policy on a domain that is completely unrelated, AllSim's learned simulation will provide the tested policy with unrealistic resources, recipients, and arrivals.

Essentially, we believe AllSim may benefit (medical) society in the following (non-exhaustive) ways:

- We propose a shared evaluation benchmark for policies stemming from a wide variety of fields. Our focus is naturally ML, but fields such as OR and decision making can absolutely contribute. This allows the testing of policies for problems initially thought outside the scope of their original design.
- AllSim allows a shared platform for comparing results. Such a platform is missing in the allocation literature which until now had no formal way of comparing across different niche problems.
- AllSim allows much more thorough testing. Specifically, AllSim is able to stress test policies under extreme (unforeseen) scenarios. This is an important part of testing which existing policies did not go through.

## G AllSim in a vaccine distribution scenario

AllSim is a general purpose simulator which evaluates scarce resource allocation policies. While we mainly focus on organ-transplantation in our main text, we show in this section that AllSim is also applicable in other settings. To illustrate, we show how one can implement a vaccine distribution policy evaluation system in AllSim. This use-case will show how few adjustments one has to make with respect to the presented settings in our main text.

Some projects use AllSim Already:

- OrganITE and OrganSync (two organ allocation projects) are evaluated with an early iteration of AllSim [1, 2].
- OrganITE is currently being evaluated using this new version of AllSim, on different data and scenarios. This is a vital part of the process to get OrganITE in the hands of transplant centers.

Compared to the organ-allocation problem, in vaccine distribution, each resource is the same and they arrive in batches. Furthermore, the type of patient-in-need is also much broader (in fact, they cover the entire population). Yet, AllSim is perfectly capable of modelling this scenario given the following:

- Batch arrival simply requires a multiplier. For example, if the Poisson process samples a value of 2 on one day, we could simply interpret this as two batches of 1000 doses.
- As all vaccines are the same, we no longer require a density of resources as we required for organ allocation. This can be done by implementing a dummy-density that always returns 1 (or the amount of vaccine).
- A broader patient-type in AllSim is achieved by retraining the density of recipients over the entire population.

These implementation details are relatively simple to implement and easily done using AllSim's modular API.

While not necessarily a problem in vaccine distribution, recipient arrival in the ICU in a setting of infectious disease (such as COVID-19), is definitely different as compared to the organ-allocation setting. With organ-allocation, we can safely assume a Poisson process for recipient arrival as recipients enter the system independently. This is of course not true in an infectious disease scenario: one recipient arriving may indicate higher infection rate. As such, recipients *do not* arrive independently.

With the above, it is clear that we can no longer rely on a Poisson arrival process for recipients entering the system. Instead, to accurately model a situation of infectious disease, we recommend using a Hawkes process. To further illustrate, we include some code below showing exactly how one may go about including such a Hawkes process in AllSim.

```

1  class HawkesProcess(PoissonProcess):
2      def __init__(
3          self,
4          lam: float=.1,
5          update_lam: Callable[[int], float]=lambda t: t,
6          delta: float=.1,
7          a: float=.2
8      ):
9
10         assert a >= 0, "a should be larger than or equal to 0"
11         assert delta > 0, "delta should be larger than 0"
12
13         super().__init__(lam, update_lam)
14
15         self.a, self.delta = a, delta
16         self._samples = []
17
18         def get_lam_unnormalized(self, t: int) -> float:
19             return self._baseline_lam + np.sum(
20                 self.a * self.beta * np.exp(-beta * (t - self._samples[
21                     self._samples < t])))
22
23         def progress(self, t: int, neu: float=1) -> int:
24             self.lam = neu * self.get_lam_unnormalized(t) # eqs. (5, 6)
25             sample = np.random.poisson(lam=self.lam)
26             self._samples.append(sample)
27             return sample

```

Naturally, if recipient arrival is indeed dependent on previous arrivals, simply learning the arrival process (as we have for figure 3) should model such a self-exciting process automatically. Furthermore,

the above implementation is a simple linear univariate Hawkes process. We refer to `hawkeslib` (<https://github.com/canerturkmen/hawkeslib>) for implementations of more types of Hawkes processes.

## G.1 Online Learning

Consider the scenario where we wish to test a policy that learns continuously from newly presented users and resources (and of course outcomes). To allow for this, we present the `Policy.feedback(X, R, Y)` function. While it is not required to implement, the `simulation` calls this function whenever an outcome is simulated. This allows the policy to learn from its feedback.

The simple steps one needs to do are:

1. Implement a novel `Policy` subclass, which includes an `Inference` attribute (of the shape  $\mathcal{X} \times \mathcal{R} \rightarrow \mathcal{Y}$ ).
2. Implement the `Policy.feedback` function to allow learning from the novel information (on outcomes) provided by the simulation

There are numerous algorithms which efficiently learn in the above framework [78, 79], which we recommend the future reader to consider.

## G.2 Low-level API example

```
1 import allsim as asim
2
3 bins = [30, 45, 60]
4 f = lambda age: np.digitize(
5     age, bins=bins).item()
6
7 condition = asim.infer.Condition(
8     labels=["AGE"], function=f, options=4)
9
10 kde = asim.KDEDensity(
11     condition=condition)
12 kde.fit(R) # R: res. pd.DataFrame
13
14 system = {
15     0: asim.PoissonProcess(
16         update_lam=regressors[0]),
17     1: ...
18 }
19 poisson_system = asim.PoissonSystem(
20     density=kde, system=system,
21     alpha=alpha, normalize=True)
22
23 stream = DataFrame(
24     columns=[*R_columns, "t"])
25 for t in T:
26     sample = poisson_system(t)
27     stream = stream.append(sample)
```

## H Counterfactual Inference

`AllSim` relies for a large part on counterfactual inference. As real-world data is collected under an active policy, testing an alternative policy would almost immediately diverge from the allocations made in the data. As such, we have to *infer* an outcome from a pair made by the tested policy. If the tested policy is indeed different from the active policy, then we are unlikely to find a comparable pair in the data. The above is a question most considered in research on counterfactual inference [69].

Of course there are many models that perform counterfactual inference. As such, we provide a brief overview of the type of models one may resort to. Naturally, this list is non-exhaustive, but may guide a user to a model fit for their use-case.

Broadly speaking, we recognise a few "meta-categories", or *meta-learners* [21, 80]. The are as follows:

- **T-Learner.** Simplest is to learn a model (such as a neural net or random forest) for each treatment. In the vaccine distribution case that would mean learning a model on recipients who haven't received a vaccine, and a different model on recipients who have received a vaccine.
- **S-Learner.** Contrasting the above, an S-Learning learns one model, where the treatment is considered part of the covariate set. This allows for a more flexible treatment, such as continuous [81] or multivariate treatments [1]. In our main-text we use OrganITE as a counterfactual model [2], which can be considered an S-Learner.
- **X-Learner.** Used specifically for estimating the treatment effect directly, an X-Learner first learns the outcome functions (such as the T-Learner), then imputes the dataset with the completed treatment effect (or the estimated counterfactual outcome), and then learns the treatment effect with a third model directly on the completed samples. While useful for treatment effect estimating, performing counterfactual inference with an X-Learner would default back to either an S-Learner or T-Learner.
- **R-Learner.** An R-Learner, like the X-Learner, estimates the treatment effect directly [82]. Specifically, it learns the outcome functions as well as a propensity estimate. Using these two models, the R-Learner optimises a custom loss function based on the propensity, the outcome model, and a cross-validation setup.
- **DR-Learner.** The DR-Learner or *doubly robust*-learner is an iteration of the X-Learner [83]. Like the X-Learner the DR-learner first estimates the outcome models, then completes the dataset to learn a treatment effects model using standard supervised learning. Then the DR-Learner repeats this process using the treatment effects model from the first step.

While there are more meta-learners than what we reported above (e.g. U-Learner [82] or CW-Learner [84]), but they are much less adopted and unlike the S-Learner and T-Learner not fit for estimating the counterfactual outcomes as they, like the X-Learner, R-Learner, and DR-Learner, fit the effect function directly. We point the interested reader to the following papers: [21, 80, 82]; or to the following open-source libraries for various implementations: `causal-ml` (<https://github.com/uber/causalml>), or `econml` (<https://github.com/microsoft/EconML>).