Licence to Scale: A Microservice Simulation Environment for Benchmarking Agentic AI

Christopher Lohse[1,2]
lohsec@tcd.ie

Adrian Selk[2]
adrian.Selk@ibm.com

Amadou Ba[2]
amadouba@ie.ibm.com

Jonas Wahl[3]
jonas.wahl@dfki.de

Marco Ruffini[1]
marco.ruffini@tcd.ie

[1]University of Dublin, Trinity College, Dublin, Ireland
[2]IBM Research Europe, Dublin, Ireland
[3]Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Berlin, Germany

Abstract

Recent advances in large language models (LLMs) have enabled the development of intelligent agents with reasoning and planning capabilities. However, there are two key limitations: the lack of realistic domain specific models that capture the causal system dynamics in which these agents operate and the absence of representative simulation environments combining LLM-agents with reinforcement learning (RL) for rigorous evaluation. The cloud autoscaling problem is a compelling use case for benchmarking AI systems. It allows the use of a causal system model while requiring agents to solve a constrained optimisation problem: minimising resource costs while meeting strict service level objectives (SLOs), with minimal intervention and interpretable actions. We use these characteristics to develop a microservice simulation environment that models the causal relations between CPU usage, memory usage, resource limits, and latency in applications of any scale and topology. It also has the ability to introduce realistic system failures.

Our simulation engine gives agents the 'licence to scale' without doing any harm in real deployments. Furthermore, it provides a realistic and controlled environment for RL agents, making it compatible with standard RL baselines. Our work provides a benchmark environment for the integration of LLMs, agents, causal models, and RL for adaptive decision-making in dynamic, resource-constrained environments.[1]

1 Introduction

In recent years, the breakthrough of large language models (LLMs) has enabled the development of smart agents with reasoning capabilities. These tools have produced good results in various application domains [Timms et al., 2024, Hosseini and Seilani, 2025]. However, as a recent survey by Gao et al. [2024] points out, one issue that still needs to be addressed is how to provide LLM-agents with sufficient domain-specific knowledge of the system dynamics in which they operate. This directly translates into a lack of simulation environments tailored to specific domains, in which to test LLM-based reasoning capabilities in realistic use cases grounded in real system dynamics. In this

[1]The code for the simulation and all the experimental runs can be accessed at https://github.com/itbench-hub/ITBench-Exploratory-Simulator

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Scaling Environments for Agents (SEA).

work, we propose such an environment for cloud autoscaling, a common real-life application of RL, by giving the agent access to the causal latency graph of the application.

Here, the goal is to solve a constrained optimisation problem: keeping the cost of resources as low as possible while maintaining the latency below an agreed service level objectives (SLOs), e.g. latency at a terminal service. As the interventions involved in this use case are costly, it is important to minimise the number of actions and also give and explanation why they have been taken. While some approaches address this problem without a model of the underlying system [Sachidananda and Sivaraman, 2024], other works represent the underlying system with a causal model and apply root cause analyses [Hardt et al., 2024, Ikram et al., 2022]. Nevertheless, automatically scaling resources remains a sufficiently difficult task for (multi)-agentic systems. To address this shortcoming, we make the following contributions:

1. We develop a simulation environment to simulate the load, CPU usage, memory usage and latency of a network of microservices of arbitrary size. It also supports the injection of various system failures such as CPU and memory leaks.
2. We use this environment to set up three challenges of varying difficulty, each accompanied by a gold standard solution.
3. We demonstrate how this environment can be used to evaluate LLM-agents that have access to a system-wide model, and the ability to scale within an interactive setting.
4. We compare multi-agentic solutions with standard RL algorithms in terms of reward, violations and complexity.

2 Background

As our microservice simulation system's model is based on causal methods, to ensure numerical accuracy and realistic values, we provide the necessary background in this section. First, we give a brief introduction to causality, then map it to the microservices use case, and finally explain how multi-agentic AI can be used to solve complex problems.

Causality We employ the structural causal model-based formalisation of cause-and-effect relationships [Pearl, 2009]. The notation $X \rightarrow Y$ means that changes in X lead to changes in Y. For multivariate datasets comprising multiple random variables, the causal relationships among variables can be modelled using a structural causal model (SCM). In an SCM, each variable X is assigned a value based on a function of a subset of its respective causal parents Pa_X, such that $X = f_X(Pa_X, \eta_X)$, where η_X is an independent noise term [Pearl, 1995]. The SCM can be represented graphically as a Directed Acyclic Graph (DAG) $\mathcal{G}$.

Cloud Microservices and Causality With the emergence of cloud computing, the distributed computing paradigm has been popularised. These architectures use a set of independently deployed microservices instead of monolithic applications. While this distributed way of computing has several advantages such as the targeted scaling of parts of the system based on demand, it also poses various challenges. Often, certain SLOs need to be maintained; therefore, it is important to know the causes of high latency in microservice deployments. The services call each other and one can (at least if the call function is synchronous) visualise these dependencies in a directed call graph $\mathcal{G}_C$. The reversed call graph can be seen as an approximation of a potentially causal latency graph of the system [Hardt et al., 2024, Lohse et al., 2025] $\mathcal{G}_L$. As practitioners usually either have direct access to the call graph or it can be retrieved with telemetry tools, we always have an approximation of the latency graph/causal DAG and thus an approximation of the SCM. This characteristic makes it convenient to model the system with an SCM where the latency l_i of a service v_i is modelled as $l_i := f_i(Pa_{li}, \rho_i, \eta'_{vi})$, where Pa_{li} are the causal parent latencies, ρ_i are the associated resources and η'_{vi} is an independent noise term (see Budhathoki et al. [2022], Hardt et al. [2024], Lohse et al. [2025]). Thus, we can say that the latency graph is an approximation of a causal model of the microservice system.

Multi-agentic AI/Systems In multi-agent systems a set of agents $\mathcal{I} = \{1, 2, ...M\}$ aims to satisfy a shared reward $\mathcal{R}(S)$, while only having access to their local observations o_s. Advances in LLMs have expanded their reasoning capabilities, enabling them to be used in new application domains, this is called agentic AI. Agentic systems offer a way to augment LLMs with external tools to enable task decomposition, tool selection and adaptation to changing conditions without the need for retraining the LLM [Lin et al., 2024, Jeyakumar et al., 2024]. A key element of this framework is that the tasks to perform in an environment are decomposed into different agents $\mathbf{A}$ which can access and leverage a number of tools $\mathbf{T}$, enabling them to create complex workflows and integrate various different systems [Hosseini and Seilani, 2025]. This can be combined with multi-agent systems, in which the LLM can act as one or more agents with reasoning capabilities [Li et al., 2024]. We refer to this as multi-agentic sytem or multi-agentic AI.

3 Related Work

In the this section, we review the state of the art in benchmarking RL for autoscaling, as well as benchmarking multi-agentic systems. We also highlight the gaps in related work, which justify the development of our simulation engine.

Benchmarking RL for Autoscaling Optimising resource usage to reduce latency is a common problem for microservice deployments (see, for example, Sachidananda and Sivaraman [2024], Tournaire et al. [2022]). RL methods have been shown to consistently outperform heuristics in this task (see, for example, [Lakshan and Hussain, 2025] for a review of the state of the art). However, none of these papers have evaluated their setups in the same environment with the same reward function, which makes comparing different algorithms complicated. Furthermore, setting up commonly used real test environments such as Deathstarbench [Gan et al., 2019], Trainticket [Li et al., 2022], the Astronomy Shop Demo[2], and the PetShop environment used by Hardt et al. [2024] is a non-trivial task; injecting faults in them even harder. In addition, performing actions in these environments is slow and not well suited for rapidly iterating multiple algorithms in the same setup for use cases of varying difficulty.

Benchmarking multi-agentic Systems As multi-agentic systems are relatively new, they face a reproducibility crisis [Bettini et al., 2024, Gao et al., 2024]; only recently have the first benchmarking tools been released. For the cloud microservice use case, Jha et al. [2025] include various scenarios in an astronomy shop that need to be resolved by a multi-agentic system. However, rather than providing a reward, they measure task completion as a percentage. For other use cases, Bettini et al. [2024] provide a framework to benchmark multi-agentic systems in various environments. A few examples have been deployed in this framework, such as a collective robot learning environment [Bettini et al., 2022] and a multi-agentic problem based on StarCraft [Ellis et al., 2023]. However, more practical and industry-related use cases are lacking.

4 Simulation Setup

In the following section, we outline the setup of our simulation engine. First, we motivate our numerical simulation process. Then, we briefly describe the reward function and formalise the optimisation problem. Finally, we motivate our design choices for the library.

4.1 Simulation Process

As we know, the reversed call graph $\mathcal{G}_c$ can be used to generate the latency graph. If we assume that all services are synchronised and there are no asynchronous function calls, we can use this feature to set up the microservice simulation environment and model realistic behaviours.

The numerical simulation engine follows the structure of an structural vector autoregressive model [Hyvärinen et al., 2010], which is commonly used to simulate time series data with causal dependencies. We simulate latency values as we would expect them to be for the $95th$ percentile of latency over a short time e.g. a minute, which is common when dealing with latency in this setup as we want to catch the worst case scenario [Hardt et al., 2024, Sachidananda and Sivaraman, 2024]. The data for d latency variables is generated by a discrete multivariate process $(\mathbf{X}_t)_{t\in\mathbb{Z}},\ \mathbf{X}_t = (X_t^1, ..., X_t^d)$, compatible with $\mathcal{G}_L$. The process $(\mathbf{X}_t)_{t\in\mathbb{Z}}$ follows the evolution rule

[2] `https://github.com/open-telemetry/opentelemetry-demo`

$$X_t^j = \sum_{i \in \mathrm{pa}_{\mathcal{G}_L}(j)} X_{t-1}^i + a_j X_{t-1}^j + f^j(\rho_t^j, R_t^j) + \eta_t^j, \tag{1}$$

where η_t^j is a choosable noise function and a_j is the autocausation constant. Typical noise functions for latency modelling are truncated exponential and half-normal distributions.[3] The function f^j is a linear function that models the resource use to latency mapping depending on the number of requests R_t and the associated resources ρ_t^j of a service j. $Pa_{\mathcal{G}_L}$ are the causal parents of the service j which can influence the latency for j at time t. We model the system in such a way that resources have an instantaneous effect on the latency while the latency of a parent service is lagged by one timestep. The number of requests R_t^j are generated by a discrete multivariate process $(\mathbf{R}_t)_{t\in\mathbb{Z}}$, $\mathbf{R}_t = (R_t^1, \dots, R_t^d)$, which is compatible with the graph $\mathcal{G}_C$. The number of requests at each node is a constant multiple of the total requests from its parents in $\mathcal{G}_C$:

$$R_t^j = \sum_{i \in \mathrm{pa}_{\mathcal{G}_C}(j)} R_t^i c_j^i, \tag{2}$$

where $c_j \in \mathbb{R}$ is the scaling constant associated with node j for all non-root nodes. For root nodes R_t^k, the number of requests is modelled, as previous work suggests [Niu et al., 2018], as a Poisson random variable with a sinusoidal rate parameter to model daily seasonality [Abdullah et al., 2019]: $R_t^k \sim \mathrm{Poisson}(\lambda_k\,[1 + \sin(\omega_k t + \phi_k)])$, where $\lambda_k > 0$ is the mean rate, ω_k is the frequency, and ϕ_k is the phase shift. We start the simulation after an initial burn in period to assure a stationary distribution. Following the causal Markov assumption [Pearl, 2009], the joint distribution P_X over $(X_1, \dots, X_n)$ factorizes into causal mechanisms:

$$P_X = \prod_{i=1}^{n} P_{X_i|\mathrm{pa}_i}. \tag{3}$$

4.2 Injected System Failures

For the base case the process is modelled sufficiently in Equation 1. However, we are also able to model more complex cases.

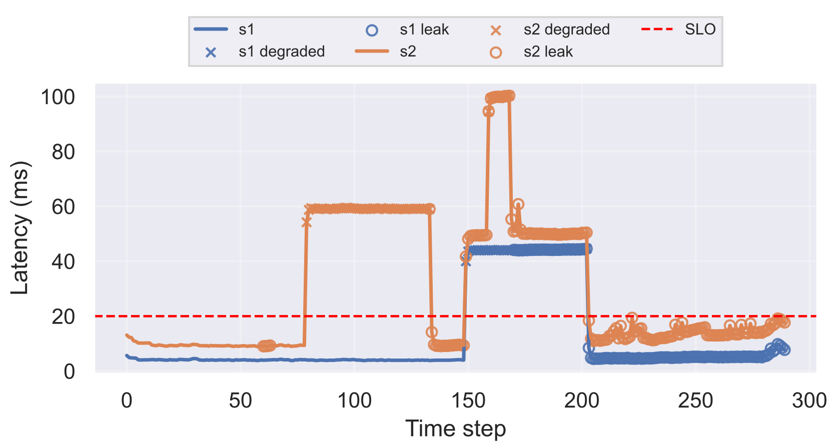


Figure 1: Latency traces for two services with degraded episodes (x) and leak episodes (o) marked; SLO shown as dashed line to highlight distribution shift.

In order to make the simulation more challenging and realistic our engine is capable of injecting various faults at the service level for a latency variable X_i translating to a change in the causal mechanism of X_i.

[3] as shown in https://www.pywhy.org/dowhy/v0.12/example_notebooks/gcm_rca_microservice_architecture.html

Following Hardt et al. [2024] we model this as a distribution shift for a variables X_T, indexed by a change set T, where

$$P_X^T = \prod_{j \in T} \tilde{P}_{X_j|\mathrm{pa}_j} \prod_{j \notin T} P_{X_j|\mathrm{pa}_j}. \tag{4}$$

The change of the causal mechanism for X_i originates from the following type of failures, which we implemented following Hardt et al. [2024], Mariani et al. [2018], Ikram et al. [2022]:

Spikes: CPU and memory usage typically follow a predictable distribution for a constant number of requests. However, when the request rate changes drastically, this distribution shifts, causing sudden spikes in CPU and/or memory usage. This can lead to increased latency at the terminal node. Spikes can be simulated by passing `enable_random_spikes = True` to the constructor of a service object.

Memory Leak: In contrast to spikes, a memory leak leads to a gradual, monotonic increase in memory usage. In real-world scenarios, this may be caused by poorly written code that fails to release allocated memory. We simulate this behaviour by introducing a degradation probability that increases memory usage at each time step by a specified amount. This can be enabled by passing `enable_memory_leak = True` and setting both `memory_leak_probability` and `leak_recovery_probability`.

CPU Hog/Leak: Similarly, a CPU leak causes CPU usage to increase steadily over time. This behavior can be enabled with `enable_cpu_leak = True`, along with appropriate values for `cpu_leak_rate`, `cpu_leak_probability`, and `leak_recovery_probability`.

Service Degradation: Service degradation is a general fault model that simulates generic service failures unrelated to CPU or memory usage. When `enable_degradation = True`, the service may fail at each time step with a defined `degradation_probability`, and it can recover according to the specified `recovery_probability`.

These fault behaviours can be introduced by passing the corresponding parameters to the constructor of a service object. Figure 1 shows how the distribution shift can look like for a simple example. The full setup and a more detailed figure can be found in Appendix A.

4.3 Reinforcement Learning Problem Formulation

We slightly adapt the reward function introduced by [Sachidananda and Sivaraman, 2024] by including two scaling constants selectable by the user: α (for weighting SLO violations more) and β (for punishing high resource use):

$$\mathcal{R}(l_{target}, \alpha, \beta, S) \triangleq \alpha \cdot min((l_{target} - l_{obs}), 0) - \beta \cdot \mathcal{P}(S). \tag{5}$$

The variable l_{target} is the specified maximum latency (SLO), l_{obs} is the observed latency and S is the state of the environment, $\mathcal{P}$ is a penalty function quantifying the used resources. We include both constants to control the scale of the reward function; however, since only the ratio β/α matters, the same results can be obtained by fixing $\alpha = 1$ and keeping only β.

The overall goal here is to keep the observed latency l_{obs} e.g. at the interaction point of a microservice application like a web interface below a certain (agreed) time, while using as few resources/electricity as possible, making it a constrained optimization problem. As modelling resource autoscaling with continuous actions would result in a very large action space, we only allow decremental or incremental vertical scaling (adding more resources to a service) and horizontal scaling (duplicating the service) at a **single** service i. If we denote memory actions by $\mathcal{M}$, CPU actions by $\mathcal{C}$ and pod actions by $\mathcal{H}$, we get the action space

$$\mathcal{A} = \{\varnothing\} \cup \{(i, k, \delta) \mid i \in \{1, \dots, N\},\ k \in \{\mathcal{C}, \mathcal{M}, \mathcal{H}\},\ \delta \in \{-1, +1\}\} \tag{6}$$

for N services, where $\varnothing$ means not performing any action. In our simulation, we scale the CPU up or down by adding or subtracting 0.1, and we add or subtract 128MB for memory. As the number of actions $|\mathcal{A}| = 1 + 6N$ the action space linearly increases with the size of the environment.

If an oracle model tells us which service is the root cause for a latency violation, the setup slightly changes and we employ a local reward function $\mathcal{R}_i$ for service i. This means that we only consider $\mathcal{R}(S_i)$ as the resource penalty while the remaining part of the function remains unchanged. In this case, the number of actions shrinks to $|\mathcal{A}_i| = 7$ as i in Equation 6 is set to a fixed value. In this reward-action-setup, LLM-based agentic AI could be utilized to orchestrate resource scaling by first identifying the resource to scale, and then by operating on a limited per-service action space A_i. For the states of the environment (that change with the actions taken) $\mathcal{S}$, we set the limits for CPU, pods and memory. The environment also provides observation of the resource usage at $\mathcal{U}_t$, which can also be used as an input for an agent. The goal of the agent is to learn a policy π that maps all states $\mathcal{S}$ and observations $\mathcal{U}_t$ to actions $a \in \mathcal{A}$ such that the expected reward is maximized.

4.4 Simulation Design Choices

The simulation engine is designed using a modular, object-oriented programming style. In line with best practice, we have incorporated various Python libraries, including NetworkX [Hagberg et al., 2008], NumPy [Harris et al., 2020] and SciPy [Virtanen et al., 2020] to reuse as many functions as possible. The RL environment is configured using Gymnasium [Towers et al., 2024], providing an effective setup process and simplifying the testing of various RL algorithms. Each service within the environment is represented as a Python object that can have a parent service, thereby defining the system's topology. The workflow of the system is shown in Figure 2.

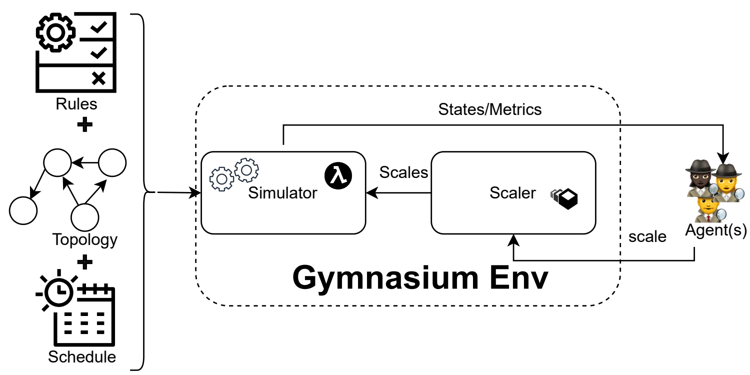


Figure 2: Overview of the Simulation Architecture used with the Gymnasium Environment. The system allows one or more agents to interact with the environment. At each timestep t, the agents take actions a_t that are sent to the *Scaler*, which adjusts resource allocations accordingly. The *Simulator* within the Gymnasium Environment provides observations l_{obs}, target latencies l_{target}, the value of the reward function $\mathcal{R}$, the states $\mathcal{S}$, and the causal latency graph $\mathcal{G}_L$ back to the agents.

One can further parse certain rules, such as the distribution that the latency follows, the defined SLO, the resource limit and the behaviour of the services, e.g. whether they are memory- or CPU-dependent, as well as the initial values for all resource usage and latency. The influence of the number of pods on the observed latency is modelled using exponential decay, so that the effect on latency decreases as the number of pods increases. The influence of CPU and memory usage on latency depends on whether a service is CPU- or memory-dependent. If a service is close to its resource limit the influence of CPU or memory on its latency increases exponentially. One can further define a schedule for spikes and set the probabilities for service failures, as described earlier. Figure 2 highlights how this setup can be used to create the simulation engine with the pre-defined rules. In the gymnasium class, a scaler can then modify the environment's resources and interact with potential agents, who also receive the environment's state S and the reward function's value $\mathcal{R}$. A detailed list of all the customisable initialisation parameters of the environment can be found in Appendix B in subsection B.1 and an example setup of a small environment is given in subsection B.2.

5 Assessing Agents in the Environment

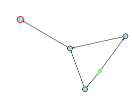
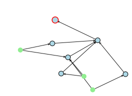
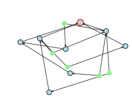
5.1 Setup

To highlight the capabilities of our simulation environment, we define three challenges of differing difficulty that can be solved by a multi-agentic system, either with or without RL, as well as with standard RL-agents, such as a deep Q-network [Mnih et al., 2015]. In the following, we describe these challenges as well as the four agents we test on them.

Simulation challenges As three evaluation scenarios, we use our simulation engine to setup an easy, an intermediate, and a hard challenge. The easy challenge is a small system, which could, for example, be an online shop. The intermediate challenge has slightly more services, one injected system failure, and could model a social media website. The hard challenge could represent a more challenging enterprise financial platform, with two system failures injected. A short explanation of the challenges is given in Table 1, and a more detailed explanation is provided in Appendix C.

Master Policy Each of these challenges has one master solution, or at least a solution that performs as well as possible in the case of the hard example. We include this master policy as a benchmark method in our results. The master policy has access to information that the agents do not, such as degradation and leak status, and which services have the greatest impact on reducing latency. In the easy environment, the master policy increases the number of pods for the central `shopping-cart`. In the intermediate challenge, the master policy increases CPU with each step up to a certain amount in the CPU-leaking `user-service`. For the hard challenge, it increases the number of pods to the maximum for the central `ml-model` pod, provided there is no failure. It also increases the number of pods if `compliance-db` is degraded, and increases the CPU limit for the `fraud-detection` service if it is leaking. This rule-based master policy can be considered the gold standard, although there may be ways to do better.

Table 1: setup of the three experiments, d is number of nodes/Services, $|\mathcal{V}|$ number of edges, l_{target} the SLO and $\mathcal{G}_L$ the causal Latency Graph (blue nodes depend on memory, blue on CPU, terminal service has red border.

	Use case	Challenge	d	$\|\mathcal{V}\|$	l_{target}	$\mathcal{G}_L$
Easy	E-commerce Platform	The system is slightly underprovisioned. If some services are not scaled, the SLO is violated. Some central services have a higher impact and are smarter to scale.	5	5	35ms	
Intermediate	Social Media Platform	The system does not violate the SLO without failures. Injected Failure: CPU of the central `user-service` has a high change to leak CPU quickly leading to an SLO violation.	9	11	90ms	
Hard	Enterprise Financial Platform	The system has three issues: is underprovisioned always leading to an SLO, has two injected failures: The compliance DB can degrade and the fraud-detection service can leak CPU.	12	17	135ms	

LLM-multi-agent To demonstrate the usefulness of our simulation for benchmarking multi-agentic systems using LLMs, we have implemented a two-agent tool that interacts with our simulation.

This is to demonstrate our engine's ability to interact with such systems. As shown in Figure 3, two agents orchestrate the scaling process: a *Reasoning Agent* and a *Resource Scaling Agent*. The

Reasoning Agent is based on the GPT-OSS LLM [Wallace et al., 2025] and determines which resources should be scaled, while the *Resource Scaling Agent* enforces these decisions.

We pre-train local per-service DQN-agent [Mnih et al., 2015] for 500 steps (one step is acting for $t = 150$ time steps in the environment) isolated from each other while operating on $\mathcal{A}_i$ and a local reward function $\mathcal{R}_i$ only having access to the local resource states, giving it access to the resource usage $\mathcal{U}_t^i$ for each service. We will not go in detail how the training of the DQN works and refer to Mnih et al. [2015] for this. At each time step t, the Gymnasium environment outputs observations $O_t = \{l_{\text{obs}}, l_{\text{target}}, \mathcal{A}_t, \mathcal{S}, \mathcal{U}_t\}$, where $\mathcal{U}$ is the observed resource usage. The *Reasoning Agent* uses O_t to compute the latency margin $\Delta L_t = l_{\text{obs}} - l_{\text{target}}$, evaluates SLO compliance, and infers the most impacted service $\hat{s}_t$ using $\mathcal{G}_L$ and $\mathcal{U}_t$ as inputs for the prompt. The complete prompt can be found in appendix subsection D.1.

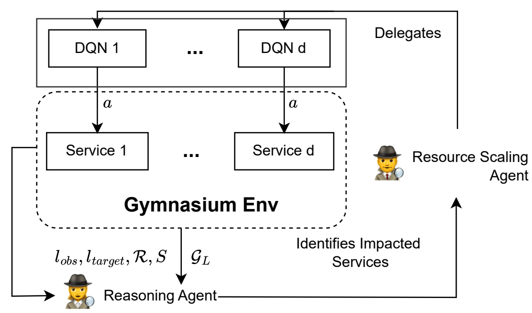


Figure 3: Overview of the simple example multi-agentic system used on the Benchmark Environment.

The reasoning agent selects a service i and passes its state $\hat{s}_t$ to its corresponding pre-trained DQN-agent of service i, which selects an action $a \in \mathcal{A}_i$ in the reduced per service action space as described in appendix subsection 4.3. The different tools and agents are registered with the LangChain agent interface [Mavroudis, 2024], enabling the language model to call them directly during the reasoning loop of the agent.

Global Deep-Q-Network We also implement a global DQN-agent [Mnih et al., 2015] that scales the complete system without having access to the causal graph operating on the complete action space $\mathcal{A}$, but having access to the states $\mathcal{S}_t$, resource usage $\mathcal{U}_t$. and satisfying the global reward $\mathcal{R}$. The DQN is trained for 2000 steps on the full environment where one step is again acting for $t = 150$ time steps in the environment.

Lazy-agent The simplest benchmark algorithm is the 'lazy-agent': it does nothing but sits and observes, failing to accomplish its mission of satisfying the SLO. However, this benchmark is really important, as it allows us to see whether an agent that performs actions is better than doing nothing.

Evaluation setup To compare the different approaches, we set the four trained agents to compete in three challenges. As the LLM-based agent is very slow at inference (taking one hour to perform 100 steps in the simulation), for now, we only perform 100 steps in the environment per trained agent. We fix the random seed to keep results comparable. This means that the results must be treated with care, as the environment is also slightly different every time. If the environment is to be used for thorough benchmarking of different agents, we recommend running multiple initialisations of the environment. For the each agent, we report the average reward, percentage of violations of l_{target} in percent and number of actions/interventions taken in the environment.

5.2 Results

In Table 2, we report the results for the four agents across the three environments. For the **easy** environment, the *master policy* achieves the target with only a two actions and no violations. The *LLM-multi-agent* also succeeds but incurs a 3% violation rate and requires three actions. This is because the model selects a slightly less ideal service to scale, resulting in additional actions. The *global DQN* consistently performs a large number of actions (100 per run) but shows a 0% violation rate. The *lazy-agent* performs no actions, leading to violations in 23% of cases.

Table 2: Comparison of Master Policy, Global DQN Model, LLM-multi-agent, and Lazy-agent baseline on the three different Simulation Tasks.

	Easy			Intermediate			Hard		
	Reward	Violations	Actions	Reward	Violations	Actions	Reward	Violations	Actions
Master Policy	-5.93 ± 0.03	0 %	2	-14.18 ± 3.85	8 %	25	-242.23 ± 124.87	79 %	28
Global DQN	-6.63 ± 0.42	0 %	100	-14.94 ± 5.52	8 %	100	-298.68 ± 132.46	95 %	100
LLM-multi-agent	-6.01 ± 0.63	3 %	3	-15.31 ± 1.70	7 %	77	-262.63 ± 99.03	87 %	100
Lazy-agent	-6.86 ± 3.71	23 %	0	-19.73 ± 6.94	57 %	0	-283.08 ± 120.98	85 %	0

Evaluated on the **intermediate** environment, the *Master Policy* again achieves 8violations with only 25 actions. The *LLM-multi-agent* has slighly less violations (7) but requires more than twice as many actions (77). The *global DQN* performs 100 actions per run but still has a 8% violation rate as well. The *Lazy-agent*, performing no actions, violates the SLO in 57 of cases. Operating in the **hard** environment, all agents fail to fully meet the SLO. The *Master Policy* performs best, with an 79% violation rate, while the other agents perform worse with violation rates close to 100%; the LLm-multi-agent performs better than the global DQN but both are worse than not doing any actions. We note that the *LLM-multi-agent* was sometimes able to diagnose one of the causes of the high latency, but its paired DQN-service agent failed to select the correct scaling action.

6 Discussion, Conclusion and Future Work

In this paper, we present a causality-driven simulation environment for cloud microservices, which serves as a benchmark environment for multi-agentic systems on realistic, constrained optimisation problems. Our benchmark simulation engine provides a range of capabilities for setting-up customisable environments of different sizes and complexities. Our framework can model both global reward optimisation with a large number of actions and local reward optimisation operating on a reduced per-service action space. It also enables failures to be introduced into the system that can be attributed to one service and lead to a distribution shift. A notable feature of our simulator is the ability to provide agents with a causal system model, which could potentially enable advanced reasoning based on the model and recent observations. Furthermore, we provide three templates for challenges in environments of varying difficulty and size that can be used to fairly compare different (multi)-agentic tools. Although our simulation attempts to model the use case realistically, it is still inherently limited by the constraints of simulation, and therefore does not provide a fully realistic representation of a real system.

We further demonstrate how one can leverage the information and simulation capabilities of the environment to test and benchmark a simple LLM powered multi-agentic system against three other algorithms. The LLM-multi-agent system is not a sophisticated approach aimed at solving the challenges in the environment; it merely serves to demonstrate how a multi-agentic system could utilise the information provided by our environment. Our experiments in our three challenges indicate that they are difficult enough to challenge a simple multi-agentic system that has to come up with the way to solve the challenges. The fact that our gold standard outperforms every other model still leaves the challenge for algorithmic improvements on this application open. However, due to the limited number of evaluation steps for the agents in the environments, the results should be treated with caution; a more extensive evaluation is required.

Future work could address the limitations of our approach by attempting to integrate or enrich the simulation system with real-system data. Furthermore, more sophisticated agents could be developed using our system. For example, causal root cause attribution techniques could be utilised to identify the service responsible for the distribution shift, and this information could be provided to an orchestrator agent. Alternatively, agent(s)/tools could be retrained while a global reward is optimised or a domain specific pre-trained LLM-model could be used. Additionally, our simulation environment could be integrated with existing benchmarking libraries.

To summarise, our environment gives every agent a ‘licence to scale’. They have to adapt dynamically to a changing environment conditions and optimise a target under pressure, while being rigorously assets in a simulation, where conditions can be shaken through exploratory action, yet real systems are not stirred.

Acknowledgements

C.L. and A.B. were partially supported by the European Union's Horizon Research and Innovation program under AC3 project and grant agreement No 101093129.
J.W. received support from the German Federal Ministry of Education and Research (BMBF) as part of the project MAC-MERLin (Grant Agreement No. 16IW24007).

References

Muhammad Abdullah, Waheed Iqbal, Abdelkarim Erradi, and Faisal Bukhari. Learning predictive autoscaling policies for cloud-hosted microservices using trace-driven modeling. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 119–126. IEEE, 2019.

Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. Vmas: A vectorized multi-agent simulator for collective robot learning. *The 16th International Symposium on Distributed Autonomous Robotic Systems*, 2022.

Matteo Bettini, Amanda Prorok, and Vincent Moens. Benchmarl: Benchmarking multi-agent reinforcement learning. *Journal of Machine Learning Research*, 25(217):1–10, 2024.

Kailash Budhathoki, Lenon Minorics, Patrick Bloebaum, and Dominik Janzing. Causal structure-based root cause analysis of outliers. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 2357–2369. PMLR, 17–23 Jul 2022. URL `https://proceedings.mlr.press/v162/budhathoki22a.html`.

Benjamin Ellis, Jonathan Cook, Skander Moalla, Mikayel Samvelyan, Mingfei Sun, Anuj Mahajan, Jakob Nicolaus Foerster, and Shimon Whiteson. SMACv2: An improved benchmark for cooperative multi-agent reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL `https://openreview.net/forum?id=5OjLGiJW3u`.

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 3–18, 2019.

Chen Gao, Xiaochong Lan, Nian Li, Yuan Yuan, Jingtao Ding, Zhilun Zhou, Fengli Xu, and Yong Li. Large language models empowered agent-based modeling and simulation: A survey and perspectives. *Humanities and Social Sciences Communications*, 11(1):1–24, 2024.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

Michaela Hardt, William Roy Orchard, Patrick Blöbaum, Elke Kirschbaum, and Shiva Kasiviswanathan. The petshop dataset — finding causes of performance issues across microservices. In Francesco Locatello and Vanessa Didelez, editors, *Proceedings of the Third Conference on Causal Learning and Reasoning*, volume 236 of *Proceedings of Machine Learning Research*, pages 957–978. PMLR, 01–03 Apr 2024. URL `https://proceedings.mlr.press/v236/hardt24a.html`.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

Soodeh Hosseini and Hossein Seilani. The role of agentic ai in shaping a smart future: A systematic review. *Array*, page 100399, 2025.

Aapo Hyvärinen, Kun Zhang, Shohei Shimizu, and Patrik O Hoyer. Estimation of a structural vector autoregression model using non-gaussianity. *Journal of Machine Learning Research*, 11(5), 2010.

Azam Ikram, Sarthak Chakraborty, Subrata Mitra, Shiv Saini, Saurabh Bagchi, and Murat Kocaoglu. Root cause analysis of failures in microservices through causal discovery. *Advances in Neural Information Processing Systems*, 35:31158–31170, 2022.

Shankar Kumar Jeyakumar, Alaa Alameer Ahmad, and Adrian Garret Gabriel. Advancing agentic systems: Dynamic task decomposition, tool integration and evaluation using novel metrics and dataset. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Saurabh Jha, Rohan R. Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, Oishik Chatterjee, Pranjal Gupta, Suranjana Samanta, Pooja Aggarwal, Rong Lee, Jae wook Ahn, Debanjana Kar, Amit Paradkar, Yu Deng, Pratibha Moogi, Prateeti Mohapatra, Naoki Abe, Chandrasekhar Narayanaswami, Tianyin Xu, Lav R. Varshney, Ruchi Mahindru, Anca Sailer, Laura Shwartz, Daby Sow, Nicholas C. M. Fuller, and Ruchir Puri. ITBench: Evaluating AI agents across diverse real-world IT automation tasks. In *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=jP59rz1bZk`.

Sivagnanasothy Lakshan and Sahdiya Hussain. A review of ai-driven techniques for cost optimization in kubernetes environments. In *2025 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, pages 1–5. IEEE, 2025.

Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27(1):25, 2022.

Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9, 2024.

Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony Cohn, and Janet B Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. *arXiv preprint arXiv:2402.02805*, 2024.

Christopher Lohse, Diego Tsutsumi, Amadou Ba, Pavithra Harsha, Chitra Subramanian, Martin Straesser, and Marco Ruffini. Causal latency modelling for cloud microservices. In *IEEE International Conference on Cloud Computing*, 2025.

Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273. IEEE, 2018.

Vasilios Mavroudis. Langchain v0.3. *Preprints*, November 2024. doi: 10.20944/preprints202411.0566.v1. URL `https://doi.org/10.20944/preprints202411.0566.v1`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Yipei Niu, Fangming Liu, and Zongpeng Li. Load balancing across microservices. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 198–206. IEEE, 2018.

Judea Pearl. Causal diagrams for empirical research. *Biometrika*, 82(4):669–688, 1995.

Judea Pearl. *Causality*. Cambridge university press, 2009.

Vighnesh Sachidananda and Anirudh Sivaraman. Erlang: Application-aware autoscaling for cloud microservices. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 888–923, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704376. doi: 10.1145/3627703.3650084. URL https://doi.org/10.1145/3627703.3650084.

Alexander Timms, Abigail Langbridge, and Fearghal O'Donncha. Agentic anomaly detection for shipping. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Thomas Tournaire, Yue Jin, Armen Aghasaryan, Hind Castel-Taleb, and Emmanuel Hyon. Factored reinforcement learning for auto-scaling in tandem queues. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2022.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Eric Wallace, Olivia Watkins, Miles Wang, Kai Chen, and Chris Koch. Estimating worst-case frontier risks of open-weight llms. *arXiv preprint arXiv:2508.03153*, 2025.

A Example of Injected Failures and Distribution Shifts

Figure 4 shows the Behaviour of a two-service microservice environment with induced faults over `max_steps=300`. `service1` can become degraded (`degradation_probability=0.005`, `recovery_probability=0.02`, `degradation_latency_penalty=40`) and develop a CPU leak (`enable_cpu_leak=true`, `cpu_leak_probability=0.01`, `leak_recovery_probability=0.001`, `cpu_leak_rate=0.01` cores/step), while `service2` can become degraded (`degradation_latency_penalty=50`) and develop a memory leak (`enable_memory_leak=true`, `memory_leak_probability=0.01`, `leak_recovery_probability=0.005`, `memory_leak_rate=5` MB/step). Top-left: per-service latencies with degraded episodes highlighted and the SLO target at `target_latency=20` ms. Top-right: CPU usage vs. limits showing gradual growth from the CPU leak in `service1`. Bottom-left: memory usage vs. limits showing gradual growth from the memory leak in `service2`. Bottom-right: the failure timeline (red = degraded, orange = leak) aligned to exogenous spikes (`spike_schedule` at steps `50` for `service1` and `150` for `service2`, `enable_random_spikes=false`), and the corresponding step-wise reward signal (`alpha=5.0`, `beta=1.0`) reflecting SLO violations. In this example the Memory/CPU is scaled when reaching 80% usage.

Figure 4: Example of Induced System Failures for a two service System

B Environment Configuration

B.1 Parameters of the Simulation

Table 3: Parameters for `SimulatedService` and `SpikeMicroserviceEnv`.

Parameter	Default	Description
General Service Parameters		
`obs`	—	Observation object containing the current environment state.
`name`	—	Name of the service.
`max_memory`	—	Maximum memory allocation for the service.
`max_cpu`	—	Maximum CPU allocation for the service.
`max_pods`	—	Maximum number of pods for the service.
`lat_func`	—	Function for computing latency noise.
`auto_regressive_coef`	0.1	Coefficient for auto-regressive latency behaviour.
`parent_services`	`[]`	List of parent services that send requests to this service.
`cpu_dependent`	`True`	Whether the latency is more affected by CPU usage than memory.
`workload_config`	`None`	Workload configuration for the service.
`initial_load`	`None`	Initial load, only for entry point service.
`base_request_rate`	`None`	Base request rate, only for entry point service.
`services`	—	List of simulated services in the environment.
`terminal_service`	—	Name of the terminal service in the environment.
Latency Parameters		
`pod_influence_decay`	2	Exponential decay factor for the effect of pods on latency.
`target_latency`	—	Latency target for the environment.
Workload and Spike Parameters		
`enable_random_spikes`	`True`	Whether random traffic spikes are enabled.
`spike_schedule`	`None`	Predefined schedule for traffic spikes.
Degradation Parameters		
`enable_degradation`	`False`	Enable degradation events in the service.
`degradation_probability`	0.001	Probability of entering a degraded state at each step.
`recovery_probability`	0.01	Probability of recovering from degradation at each step.
`degradation_latency_penalty`	50.0	Additional latency (ms) when degraded.
Leak Parameters		
`enable_cpu_leak`	`False`	Enable uncontrolled CPU usage growth.
`enable_memory_leak`	`False`	Enable uncontrolled memory usage growth.
`cpu_leak_probability`	0.001	Probability of starting CPU leak.
`memory_leak_probability`	0.001	Probability of starting memory leak.
`leak_recovery_probability`	0.005	Probability of fixing a leak per step.
`cpu_leak_rate`	0.002	CPU usage increase rate when leaking.
`memory_leak_rate`	2.0	Memory usage increase rate when leaking.
Environment Parameters		
`alpha`	1.0	Reward function parameter α.
`beta`	1.0	Reward function parameter β.
`max_steps`	500	Maximum number of simulation steps.

B.2 Example Initialization of a small environment

```python
service1 = SimulatedService(  # Service 1: Can develop CPU leaks
    Observation(
        cpu_limit=0.5,
        memory_limit=512.0,
        n_pods=1,
        latency=1,
        memory_use=100.0,
        cpu_use=0.05,
        energy_use=0,
    ),
    name="service1",
    max_memory=2048,
    max_cpu=2.0,
    max_pods=3,
    lat_func=partial(truncexpon.rvs, loc=0, size=1, b=1, scale=0.5),
    initial_load=30.0,
    enable_degradation=True,
    degradation_probability=0.005,
    recovery_probability=0.02,
    degradation_latency_penalty=40.0,
    enable_cpu_leak=True,
    cpu_leak_probability=0.01,
    leak_recovery_probability=0.001,
    cpu_leak_rate=0.01,  # 0.01 cores per step
    base_request_rate=5,
)
service2 = SimulatedService(  # Service 2: Can develop memory leaks
    Observation(
        cpu_limit=0.3,
        memory_limit=512,
        n_pods=1,
        latency=1,
        memory_use=100,
        cpu_use=0.03,
        energy_use=0,
    ),
    name="service2",
    max_memory=1028,
    max_cpu=1.0,
    max_pods=3,
    lat_func=partial(truncexpon.rvs, loc=0, size=1, b=1, scale=0.5),
    parent_services=[service1],
    initial_load=30.0,
    enable_degradation=True,  # Degradation
    degradation_probability=0.005,
    recovery_probability=0.02,
    degradation_latency_penalty=50.0,  # Memory leak
    enable_memory_leak=True,
    memory_leak_probability=0.01,
    leak_recovery_probability=0.005,
    memory_leak_rate=5.0,  # 5MB per step
    base_request_rate=5,
)
env = SpikeMicroserviceEnv(
    services=[service1, service2],
    terminal_service="service2",
    target_latency=20.0,
    alpha=5.0,
    beta=1.0,
    max_steps=300,
)
```

Listing 1: Example Initilaization of Environemnt

C Example Environments

C.1 Simple

The simple simualtion vizualised in Figure 5 is a small-scale online shop system wit 5 services and 5 edges in a mostly linear chain. The `inventory-db` (memory-dependent) is the root, feeding both `shopping-cart` and `product-catalog` (CPU-dependent). The `api-gateway` (CPU-dependent) aggregates both and sends results to the terminal `frontend` (CPU-dependent), where latency is measured. Target latency $l_{target} = 30\,\mathtt{ms}$. Random spikes (`enable_random_spikes=True`) are enabled, but no degradation (`enable_degradation=False`) or leaks (`enable_cpu_leak=False`, `enable_memory_leak=False`) occur. System is slightly underprovisioned. Scaling central services like `shopping-card` yields the largest SLO impact.

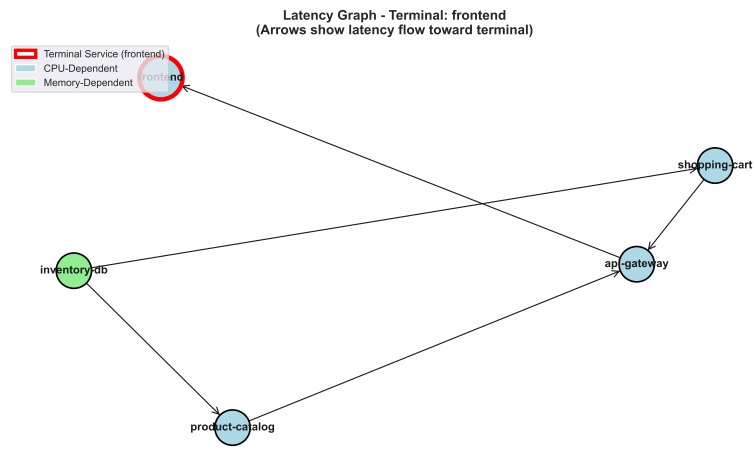


Figure 5: Causal Graph of the easy Simulation

C.2 Intermediate

Social Media System with 9 services and 11 edges in a more complex causal graph as shown in Figure 6. The terminal `web-frontend` (CPU-dependent) receives traffic via `load-balancer`, which aggregates from `feed-generator`, `user-service`, `content-service`, and `notification`. Backends include `auth-db` and `media-storage` (memory-dependent), `message-queue` (memory-dependent, `enable_degradation=True`), and multiple CPU-dependent services. The `user-service` has a high `cpu_leak_probability=0.5`, making it the main SLO risk. Target latency $l_{target} = 35\,\mathtt{ms}$, with `enable_random_spikes=True`. The system does not initially violate the SLO, but sustained CPU leaks or degradation events require targeted scaling, particularly of `user-service`.

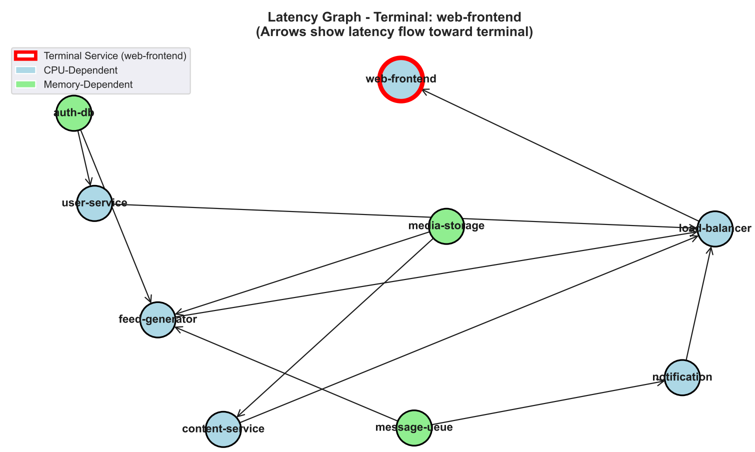


Figure 6: Causal Graph of the intermediate Simulation

C.3 Hard

Enterprise Financial Platform System with 12services and 17 edges in a multi-tier DAG. The terminal `api-gateway` (CPU-dependent) aggregates from `auth-service`, `trading-engine`, `payment-processor`, and `reporting-service`. Backends include memory-dependent `user-db`, `transaction-db`, `price-feed`, `ml-model`, and `compliance-db`. The platform is intentionally underprovisioned: with `enable_random_spikes=True` it tends to breach the SLO $l_{target} = 100\,\mathrm{ms}$ unless scaled. Two injected failures dominate: `compliance-db` can degrade (`enable_degradation=True`) and `fraud-detection` can leak CPU (`enable_cpu_leak=True`, `cpu_leak_probability=0.2`, `cpu_leak_rate=0.09`). The gold-standard policy prioritizes scaling across tiers: increase pods for a healthy `ml-model`, add pods when `compliance-db` is degraded, and raise `fraud-detection` CPU while leaking.

System with 12 services and 17 edges in a multi-tier DAG. The terminal `api-gateway` (CPU-dependent) aggregates from `auth-service`, `trading-engine`, `payment-processor`, and `reporting-service`. Backends include memory-dependent `user-db`, `transaction-db`, `price-feed`, `ml-model`, and `compliance-db`. The platform is intentionally underprovisioned: with `enable_random_spikes=True` it tends to breach the SLO $l_{target} = 100\,\mathrm{ms}$ unless scaled. Two injected failures dominate: `compliance-db` can degrade (`enable_degradation=True`) and `fraud-detection` can leak CPU (`enable_cpu_leak=True`, `cpu_leak_probability=0.2`, `cpu_leak_rate=0.09`). The gold-standard policy prioritizes scaling across tiers: increase pods for a healthy `ml-model`, add pods when `compliance-db` is degraded, and raise `fraud-detection` CPU while leaking—capturing the need for diagnosis and targeted mitigation in a complex graph.

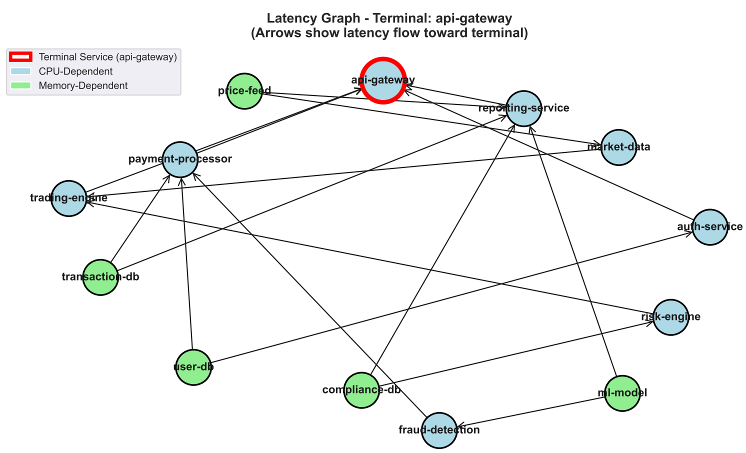


Figure 7: Causal Graph of the hard Simulation

D LLM Prompts and Response

D.1 Prompt

```
"""You are an expert in Microservice System Analysis and your task is to identify
    which services need scaling actions the most.
The system consists of multiple services that are connected and propagate latency
    through the system. Our aim is to keep the latency in the
terminal service, {state['terminal_service']}, below the Service Level Objective
    (SLO) threshold of {state['target_latency']}ms. At the same time,
we want to waste as little resources as possible. Scaling up or down is not always
    sufficient if the SLO is already satisfied by a high margin (over 10%
    satisfied).
Doing as little interventions as possible is desirable.

Consider service dependencies (the causal graph) when prioritizing services.

CURRENT SITUATION:
- SLO Status: {'VIOLATED' if impact['slo_violated'] else 'SATISFIED'} (target:
    {state['target_latency']}ms, actual: {impact['terminal_latency']}ms)
- Terminal Service: {state['terminal_service'],}

CAUSAL GRAPH of latency dependencies ('parent', 'child'):
{state['causal_graph']}

RESOURCE ISSUES:
- Critical (>80% of limit): {[s['service'] for s in resources['critical_services']]}
- High (>60% of limit): {[s['service'] for s in resources['high_util_services']]}
- Moderate and low usage services usually do not need scaling unless latency is
    near SLO.

TASK: Select the TOP 3 services that need scaling action, considering:
1. Services with resource leaks need high attention.
2. High-impact services affecting SLO the most should be prioritized.
3. Services with critical resource usage need scaling to reduce resource pressure.
4. Only recommend scaling if SLO is violated or near violation, keeping scaling
    costs in mind. Also recommend scaling if headroom can save resources.
```

Listing 2: Prompt provided to reasoning agent

D.2 Example Response of Reasoning Agent


```json
{
  "services_to_scale":[
    {
      "service":"frontend",
      "priority":1,
      "reason":"The frontend service is currently violating the SLO threshold
   and has a high impact score, indicating that it is affecting the overall
   latency of the system."
    },
    {
      "service":"api-gateway",
      "priority":2,
      "reason":"The api-gateway service has a significant impact on the system's
   latency and is also a critical component in terms of downstream dependencies."
    },
    {
      "service":"product-catalog",
      "priority":3,
      "reason":"The product-catalog service has a moderate impact score and is
   also experiencing high CPU utilization, which could be indicative of resource
   leaks or inefficiencies that need to be addressed."
    }
  ],
  "system_status":{
    "slo_violated":true,
    "terminal_latency":25.254812240600586,
    "target_latency":25.0
  },
  "analysis_summary":{
    "total_services":5,
    "unhealthy_services":0,
    "top_bottleneck":"frontend"
  },
  "debug_info":{
    "impact_node":{
      "terminal_latency":25.254812240600586,
      "slo_violated":true,
      "num_services_analyzed":5
    },
    "resource_node":{
      "critical_count":0,
      "high_count":0,
      "leaking_count":0,
      "degraded_count":0
    },
    "llm_response":"Based on the provided data, I would recommend the following
   services for scaling actions:\n\n[\n  {\n    \"service\": \"frontend\",\n
   \"priority\": 1,\n    \"reason\": \"The frontend service is currently
   violatin..."
  }
}
```


Listing 3: Response example