

Gecko: Automated Feature Degradation for Cloud Resilience

Kapil Agrawal

University of California, Irvine

Sangeetha Abdu Jyothi

University of California, Irvine & VMware Research

Abstract

Cloud resilience is crucial for cloud operators and the myriad of applications that rely on the cloud. Today, we lack a mechanism that allows cloud operators to gracefully degrade application performance in public clouds. In this paper, we put forward a vision for automated feature degradation with a three-pronged approach. First, we introduce the idea of *diagonal scaling*—pruning an application’s dependency graphs during capacity crunch scenarios—to maximize the availability of critical services. Second, we propose the use of simple and expressive *Criticality Tags* on microservices to convey an application developer’s resilience intent to the cloud provider without any application modifications. Third, we design Gecko, an automated cloud resilience management system that maximizes critical service availability of applications while also taking into account operator objectives, thereby improving the overall resilience of the infrastructure during failures. Using extensive experiments, we show that the Gecko controller running atop Kubernetes can maximize the overall resilience of applications and infrastructure under a broad range of failure scenarios. We also demonstrate that diagonal scaling with Gecko aligns well with microservice architecture using experiments with a real-world application.

1 Introduction

Public and private clouds host a myriad of applications from diverse domains. The resilience of the cloud infrastructure is critical to maintaining business continuity for these applications that rely on the cloud. As cloud infrastructure expands, the number of infrastructure incidents also has seen a significant rise in recent years [8, 46, 61]. A recent study showed that 40% of production incidents encountered are caused by infrastructure failures [40]. Furthermore, the recovery time of infrastructure incidents can be high as it often involves work by on-site personnel [61], often causing massive user-visible outages [8]. This motivates the need to improve the resilience of clouds to withstand failures while ensuring high availability for the applications running on them.

Several solutions have been proposed to improve the resilience of the cloud. At the *infrastructure level*, hyperscale cloud providers stress test the infrastructure to identify the vulnerabilities under large-scale failures. For example, in-house resilience engineering teams at Google (DiRT or Disaster Recovery Testing Program) [26] and Netflix (Chaos Engineering Team) [5] instigate failures in production systems to under-

stand system vulnerabilities. Furthermore, cloud providers maintain rule-based runbooks with a set of tasks that need to be performed during disaster scenarios (migration of data, restarting containers, etc.). Automatic failover during data center-scale failures requires replicas of the entire application in multiple data centers, which incurs significant cost overheads and may not be economically viable.

At the *application level*, resilience solutions range from Chaos engineering tools that stress test the system under various failure scenarios [13, 19, 27, 32] to techniques such as automatic failover between data centers [7]. Chaos engineering tools such as Gremlin [19], ChaosMonkey [32], Istio [27], and Hystrix [13] can automate resilience patterns like retries, timeouts, and circuit breakers to handle local faults.

Several resilience solutions aim for graceful performance degradation or self-adaptation of applications under changing cloud conditions [41, 45, 65]. However, application-level solutions cannot be leveraged by operators since they lack visibility into the applications and, therefore, the information required to coordinate graceful service degradation. Recently, Meta proposed Defcon [51] for graceful degradation at datacenter-scale. However, this solution requires making changes to the applications to expose “knobs” and wrapping application features that can be disabled using these knobs. Such an approach is feasible only with first-party applications controlled by the operator and cannot be employed in public clouds running third-party applications.

In this paper, we put forward a vision for automated graceful feature degradation with a three-pronged approach. First, we introduce a novel scaling technique, Diagonal Scaling, that enables running applications with reduced capabilities under capacity crunch scenarios. Diagonal scaling transforms the optimization landscape of resilience objectives from a scalar to a spectrum of values. Second, we propose Criticality Tags to enable intent-driven automated resilience using diagonal scaling, whereby developers assign resilience labels to microservices to convey their resilience requirements without requiring any application modifications. Third, we build Gecko, an automated cloud resilience management system that converts application-level Criticality Tags and operator-level objectives to actionable decisions for cluster schedulers through criticality-aware planning and scheduling.

We introduce *diagonal scaling* as a fundamental tool for resilience engineering. We define diagonal scaling as the pruning of the application dependency graph by turning off “non-critical” microservices. We demonstrate that microservice architecture, where application functionalities are decom-

posed into microservices that are developed and deployed independently, is naturally amenable to diagonal scaling (§ 4.2). Diagonal scaling allows us to shut down “non-critical” microservices during an infrastructure failure event and reallocate resources to “critical” microservices to maximize critical service availability.

With diagonal scaling, we expand the set of valid application states—beyond the commonly-used notion of the entire application being “on” or “off”—by allowing valid subgraphs of the application dependency graph that maximizes the critical service availability. For example, a chat application could provide text-only connectivity and disable multimedia support when network bandwidth is limited. Diagonal scaling also transforms resilience objective metrics from a single scalar to a range of potential values. For example, the Recovery Time Objective (RTO), a commonly used resilience objective defined as the maximum acceptable time an application can be unavailable, may be expanded to include RTO estimates for sub-graphs. Critical sub-services of an application can have stringent RTO bounds while non-critical sub-services may tolerate looser bounds.

While diagonal scaling can unlock new avenues for leveraging application elasticity, the information about which microservices can be safely turned off to maximize application availability may not be directly available to site operators. To convey the resilience intent of an application to the operator in a simple and expressive manner, we use Criticality Tags. Criticality Tags allow applications to specify the degradable features at the microservice level to the operator. The levels of criticality convey the relative importance of microservices, and, in turn, identify the microservices that can be turned off safely through diagonal scaling. Unlike knobs in Defcon which are integrated into applications, Criticality Tags are specified at the microservice level in configuration files and do not require any modifications to the application.

We build an automated resilience management system, Gecko, that can convert application-level resilience goals to actionable decisions for cluster schedulers. The key goal of Gecko is to maximally satisfy application-level resilience requirements while also taking into account resource availability in capacity-constrained cloud environment after failures. Gecko decouples the application’s core business logic from its resiliency requirements, thereby reducing the engineering complexity required. At its core, Gecko comprises of a criticality-aware planner and scheduler. Planner takes as input application DGs with their resilience tags and generates a list of microservices to activate in the available capacity. Based on the planner’s output, the scheduler generates a sequence of steps (including scheduling, migrating, or shutting down) for running microservices at individual servers (nodes).

We evaluate the performance of Gecko using a large-scale trace dataset from a real-world cluster. We also deploy Gecko on a Kubernetes cluster running real-world microservice-based applications. (Note that Gecko is agnostic to the

underlying cluster scheduler and can support other schedulers [43, 46, 60].) Our results show that Gecko can maximize critical service availability while also satisfying operator objectives. Furthermore, our real-world tests on a 200 CPU Kubernetes cluster show that when the cluster is failed down to 40% of its original capacity, 5 microservice-based application instances resume serving their critical traffic in less than 4 minutes. Thus, we demonstrate that microservice architecture is naturally amenable to diagonal scaling. Production microservice-based Overleaf application [23] worked seamlessly with Gecko, without requiring any modifications.

In summary, we make the following contributions:

- We introduce diagonal scaling as a tool for resilience engineering. Diagonal scaling supports application-level dependence graph pruning and expands the spectrum of resilience objectives.
- We put forward Criticality Tags as a new metric for the cloud to capture application-level resilience intent with per-microservice labels.
- We build Gecko, an automated cloud resilience management system with resilience-aware planning and scheduling heuristics that maximizes application-level resilience goals during capacity-constrained failure scenarios.
- We perform standalone testing of Gecko to demonstrate its broad applicability to maximize service availability for applications while satisfying operator objectives in a 100,000-node cluster.
- On a shared Kubernetes cluster running 5 microservice application instances at a high failure rate of 60%, we show that Gecko ensures high critical service availability.

2 Cloud Resilience

2.1 Motivation

Ensuring cloud resilience under extreme events whose time of occurrence, location, duration, and strength may be difficult to predict is a challenge for cloud operators.

Threats: There exist several threats that affect the availability of cloud resources. This includes natural threats such as extreme weather events [11, 20], including heatwaves, hurricanes, floods, and wildfires. Human errors [1] and faulty operation of automated systems [14] can also result in large-scale outages in the cloud. In addition to extraneous threats, cloud infrastructure may also be stressed during its normal operation. During certain times, the cloud may experience unexpected load spikes, leading to limited availability for some applications. On longer timescales, the process of adding new equipment to the cloud may not keep up with the rate of increase in load. Moreover, the stresses that cloud experiences may be planned (e.g., rack maintenance) or unplanned (e.g., external threats, hardware failures). In short, several factors pose a risk to the reliability and availability of cloud services.

Cloud Resilience Solutions: Cloud resilience solutions aim to improve the availability of cloud services under large-scale failure scenarios.

Application-Level Solutions: Cloud applications are typically represented using dependency graphs to capture the dependencies between interconnected microservices or computation tasks. During disasters, when resources or connectivity is limited at one or more data center locations, a commonly used strategy for improving reliability involves the migration of applications to unaffected locations. This is a formidable task since application DAGs in real-world settings are complex and interlaced [61]. Other well-known techniques are checkpointing and on-demand autoscaling, which also introduces other challenges, such as scarcity of autoscaling groups within a region [6]. Third-party solutions for improving application resilience [13, 19] run chaos engineering experiments under various failure scenarios. While these solutions expose bugs and corner cases, they do not provide a solution to mitigate the risk when large-scale failures strike.

Infrastructure-Level Solutions: Resilience engineering solutions span one or more layers of the systems stack. Several cluster schedulers support georeplication, live migration, and automatic failover [46, 62]. Resilience solutions on storage include designing strategies based on recovery time objective and recovery point objective [44]. They also use a variety of techniques for conflict detection and conflict resolution. Database systems also offer varying levels of consistency ranging from strong to eventual consistency [63].

Large companies stress-test their infrastructure by injecting major failure, e.g., Disaster Recovery Testing (DiRT) event at Google [26], drain tests with Maelstrom at Facebook [61], and Chaos Monkey at Netflix [32]. Each organization typically has its own runbook [61] with a set of tasks (migration of data, restarting containers, etc.) that need to be performed during disaster scenarios. Recently, Meta introduced DEFCON [51], a resilience solution for their private cloud that hosts first-party applications, which employs graceful degradation.

Graceful Degradation in the Cloud: Ideally, operators and application developers want performance to degrade gracefully when resources are constrained. This is often referred to as self-adaptation under changing cloud conditions [51, 52]. Several solutions for graceful degradation exist at both the application level and infrastructure level.

Application-Level Solutions: Past work introduced graceful degradation solutions in application-specific contexts such as web servers [34, 42], mail services [55], search engines [38], storage systems [36, 47, 66], to name a few. Solutions that can be applied broadly across applications have also been proposed [52, 64]. Load shedding or dropping a fraction of the load is another common method employed at the application level [10, 30, 33]. Circuit breaking [15] in tools such as Istio also serves a similar purpose but at a microservice level. Brownout solutions, that allow the dimming of optional

features, have also been proposed at the application level. However, these require changes to the application [45].

Infrastructure-Level Solutions: Operator-managed graceful degradation is difficult in practice due to several reasons. First, we lack a practical abstraction that allows application developers to convey their requirements to the operators to enable graceful degradation; for example, what services are safe to turn off. Prior work introduced various abstractions which failed to achieve real-world adoption. Relaxation lattice [41] is a complex set of specifications parameterized by a set of constraints. Another recent abstraction, availability knob [56], conveys the availability requirements of an application to the operator. However, this abstraction only allows a coarse-grained representation of the availability needs of an application. Second, most real-world applications are composed of a large number of microservices interconnected in complex patterns. The behavior of these intricate dependency graphs under varying levels of overload is often difficult to predict. This is particularly hard for operators who may not have visibility into the performance metrics of an application.

Defcon [51] circumvents these challenges by modifying applications to include *knobs* that annotate program elements eligible for degradation. This allows the cluster manager to disable these features during emergency scenarios. However, similar to prior brownout solutions [45], this is feasible only with first-party applications where the operator can modify the software. It cannot be employed in public clouds that host third-party applications.

Towards Application-Agnostic Graceful Degradation: We identify the following requirements for a graceful degradation solution tailored for the cloud. First, to facilitate easy practical adoption, the solution should not require changes to current applications. Second, we need an expressive and concise representation that allows application developers to convey their requirements to cloud providers. Third, the cloud providers should be able to gracefully degrade applications without visibility into them, by relying solely on the provided requirements. Finally, the solution should be generalizable; i.e., it should work across applications and cloud platforms. Toward this goal, we propose the notion of diagonal scaling.

2.2 Diagonal Scaling

We put forward *diagonal scaling* as a mechanism for enabling automated resilience in the cloud. We argue that it is safe to prune the application dependency graph to a minimal sub-graph that can serve a large fraction of user requests. We refer to this dependency graph pruning by turning off microservices as diagonal scaling. This can help in improving the overall availability of “critical” services in the infrastructure during capacity crunch scenarios. Note that diagonal scaling is orthogonal to the notion of horizontal scaling (multiple parallel instantiations of containers) and vertical scaling (scaling up/down the resources allocated to containers) and may be

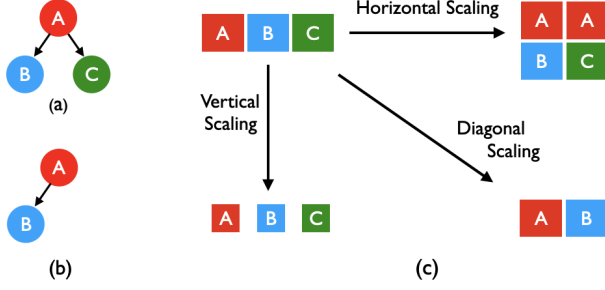


Figure 1: **Diagonal Scaling.** (a) Original application dependency graph (DG) with 3 microservices. (b) Diagonally scaled DG with one microservice removed. (c) Comparison of horizontal, vertical, and diagonal scaling techniques.

employed alongside other scaling techniques. Figure 1 depicts a comparison of these scaling schemes.

To enable application-agnostic graceful degradation with diagonal scaling, we need to answer two questions. First, how can applications indicate their diagonal scaling preferences to the cloud operator? Second, does the notion of diagonal scaling align well with the application architecture today?

Criticality Tags: We introduce *Criticality tags* as a simple yet expressive mechanism to capture the importance or criticality of the microservice to an application. They are represented using criticality levels (C_1, C_2, C_3 , etc.) with a lower number representing higher importance. We tag a microservice as high criticality (e.g., C_1) when it is key in driving the business of an application, and a low criticality such as C_5 to denote a microservice that is “good-to-have.” For example, recommendation services or ads may be of low criticality to an application during failures. By specifying a lower criticality tag, the application is agreeing that in case of a disaster, these microservices may be safely turned off.

Cloud and Application Support: Realizing criticality tags in the cloud does not require additional engineering efforts today. Several orchestrators support tagging, such as *labels* in Kubernetes [18]. These existing tagging mechanisms can be used for criticality tags.

On the application front, we find that the dependency graph-based microservice architecture is naturally amenable to diagonal scaling. The shift from monolithic applications to microservice architecture was inspired by the need for composing an application by combining independent microservices. The loosely coupled microservices are typically developed, deployed, and maintained independently. Hence, a microservice-based application can typically continue to offer services even when some of its component microservices are turned off.

Moreover, microservices are often developed using languages such as Go and Node.js with several in-built fault-tolerance functionalities, such as error handling when downstream RPC calls are unavailable, thereby reducing the effort

of engineering. In § 4, we demonstrate with real-world applications that diagonal scaling is indeed feasible with today’s applications.

Different applications can have varying tolerances to the extent of degradation. We believe that the semantics of calls within an application are best known to the application developers who build them and, therefore, developers should be able to specify what is degradable and what is not.

Opportunities for Criticality Tagging: We identify several scenarios in which diagonal scaling is feasible. Microservices may be deemed to be of lower criticality based on a variety of criteria: the importance of their functionality, frequency of access, energy consumption, or any other developer-specified metric. A microservice may be of low priority for certain services and high priority for some other services within the same application. In such scenarios, the industry practice is to run two instances of the microservice, one handling the low-priority and the other handling high-priority traffic [31]. Here, only the low-priority instance can be given a lower criticality.

Expanding the Resilience Metrics Design Space: Resilience is typically measured using metrics such as Recovery Time Objective (RTO). RTO specifies the maximum duration of time that the application can tolerate being unavailable. Resilience metrics are defined under the assumption that an application is either completely available or unavailable.

With diagonal scaling, we argue that an application can have multiple levels of availability. In addition to the fully “on” state where all microservices in the dependency graph are active, an application could serve a large number of user requests at multiple “valid” intermediate states where a carefully selected subset of its microservices are active. This flexibility introduced by diagonal scaling expands the range of resilience metrics. With diagonal scaling, RTO can be defined at different levels of operation. An application could define a stringent RTO for its critical functionality and a more lenient RTO for its auxiliary services. A broader range of resilience metrics will offer greater flexibility to operators during application deployment, and improve the availability of critical subservices of the application.

3 System Design

In this section, we detail the system design of our automated resilience management system, Gecko. The key goal of the Gecko resilience management system is to maximally satisfy the resilience goals of applications and the cluster operator in the event of large-scale failures. Gecko constantly tracks the cluster state and has access to application dependency graphs with their associated criticality tags. During a failure event, Gecko will generate a new target cluster state based on operator and application goals, along with a list of steps to achieve that state. Figure 2 shows the system architecture of Gecko. Gecko has two key modules: the Planner and the

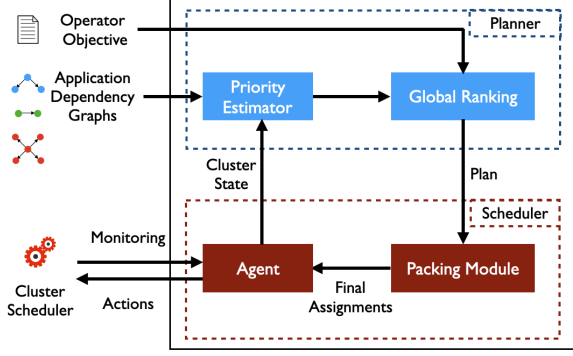


Figure 2: Gecko System Diagram

Scheduler.

Given a failure scenario, the planner generates a new plan for the cluster as an ordered list of microservices in decreasing order of priority. The planner takes as input dependency graphs of active applications and resource allocation of microservices before the failure. The planner then generates a subset of microservices to be enabled in the current environment under constrained resources. The planner consists of two sub-modules. The priority estimator generates an ordered list of microservices within each application while taking into account topological constraints and criticality tags. Next, a global ranking function generates a global ordering of microservices across applications based on the operator’s objective. The ranking function can take as input a broad range of objectives, such as max-min fairness, weighted fairness, maximizing revenue, maximizing application utility, etc.

The scheduler takes as input the plan from the Gecko planner and the cluster state. It then maps individual microservices to nodes in the cluster while taking into account the priority order determined by the planner. The scheduler employs a bin-packing heuristic to obtain the final assignments of microservice to servers.

Gecko allocation adheres to the following requirements:

- (R1) Criticality Awareness:** Within an application, a microservice of lower criticality is scheduled only after all microservices with a criticality higher than it is scheduled.
- (R2) Topology Awareness:** Microservices are ordered based on the application dependency graph.
- (R3) Resource Efficiency:** The resource assignment scheme should efficiently pack microservices within the available cluster capacity.
- (R4) Operator Objective:** The operator objective determines the priority of microservices across applications.

The Integer Linear Program formulation that captures these requirements can be written as follows:

$$\begin{aligned} & \text{Maximize } \sum_i \sum_j F(x_{ij}) \\ & \text{such that} \end{aligned}$$

$$[R1] : x_{ij} \geq x_{ik} \mid \forall m_j, m_k \in app_i \text{ with } C(m_k) > C(m_j)$$

$$[R2] : \sum_{j \in pred_i(m_k)} x_{ij} \geq x_{ik} \mid \forall app_i, \forall m_k \in app_i$$

$$[R3] : \sum_k y_{ijk} = x_{ij} \mid \forall app_i, \forall m_j \in app_i$$

$$[R3] : \sum_i \sum_j R_{ij} * y_{ijk} < S_k \mid \forall S_k, \forall app_i, \forall m_j \in app_i$$

The Boolean variable x_{ij} represents whether a microservice j in application i is activated or not. The Boolean variable y_{ijk} represents whether a microservice j in application i is placed on server k . $C(m_j)$ denotes the criticality level of microservice m_j . R_{ij} represents the resource requirement of microservice m_j belonging to application app_i and $pred_i(m_j)$ refers to its predecessors in the dependency graph. S_k denotes the capacity of the server k . $F(x_{ij})$ represents the global ranking function, which could be fair share, revenue, etc.

Globally, the operator objective determines the priority of microservices across applications. We aim to support a broad range of objectives. We discuss two representative examples—maximizing revenue and max-min fairness. With the maximize revenue objective, each application can have a cost they are willing to pay per criticality tier. The global objective optimizes the total revenue of the provider under capacity-constrained scenarios. With max-min fairness, every app receives a fair share of the cluster resources. The ILP may have additional constraints depending on the operator objective. For example, with max-min fairness, we have an additional constraint on resources allocated to every application.

We build two LP-based solutions using the above criteria: LPCost (maximizing revenue) and LPFair (max-min fairness). We also add additional constraints to limit the migration of microservices from unaffected nodes and comply with the per-node limit on the number of microservices of underlying cluster schedulers. However, LP-based solutions scale poorly, as we show later in § 5. Hence, we design heuristics for planning and scheduling.

3.1 Gecko Planner

The planner takes as input the application dependency graphs, the criticality tags and resource allocation of microservices, and the current cluster state. The goal of the planner is to generate an ordered list of microservices for instantiation in a cluster after large-scale failures. The sequence is determined in a manner that maximally satisfies the resilience goals of the application while also considering application dependencies and operator objectives such as revenue/fairness.

The Gecko planner has two modules: Priority Estimator and Global Ranking. The priority estimator determines the ranking of microservices *within* an application. The global ranking module generates a ranking of microservices across applications. The high-level planner algorithm is given in Algorithm 1.

Algorithm 1: Criticality-Aware Planning Algorithm

```
1 def Main:
2   AppRank = PriorityEstimator(G, tags)
3   GlobalRank = Sort(AppRank, ClusterObj)
4   return GlobalRank
5 def PriorityEstimator(G, tags):
6   def DFS(node):
7     if node ∈ visited then return
8     visited.add(node)
9     AppRank.append((g.id, node.id))
10    foreach child ∈ node.children do
11      if tags(child) ≥ tags(node) then
12        DFS(child)
13      else Q.insert(child)
14  foreach g ∈ G do
15    Q = InitPriQ(g.sources, key = tags)
16    visited = set()
17    while len(Q) ≠ 0 do
18      curr = Q.pop()
19      DFS(curr)
20  return AppRank
21 def GetGlobalRank(AppRank, Obj):
22  Q = InitPriQ([root for app in AppRank], key =
    Obj)
23  while len(Q) ≠ 0 do
24    (appID, ms) = Q.pop(0)
25    R = R - ms.resources
26    if R ≥ 0 then
27      GlobalRank.append((appID, ms.id))
28      Q.insert(AppRank[appID][ms.idx + 1])
29    else break
30  return GlobalRank
```

Priority Estimator: This module determines the relative priority of microservices within an application based on two factors: the criticality of each microservice, and the application dependency graph, in order to satisfy [R1] and [R2] requirements. The PriorityEstimator function in Algorithm 1 takes as input a list of application dependency graphs G , and the criticality tags, $tags$. Its output is an ordered list, $AppRank$, which denotes the priority order in which nodes need to be activated. Note that this ordering is within each application.

As shown in Algorithm 1, for each graph, we perform a criticality-based graph search to generate an ordered list of microservices. We use the priority queue, Q , to track the pending nodes, sorted based on the key, $tags$ which denote criticality tags. In lines 14-19, we traverse the graph based on a combination of two factors—priority (from high to low) and topological ordering (from root to leaves). Q is initialized

with the root nodes in line 15. In line 18, we pop microservices from Q in the order of criticality. For every popped node, we run a DFS subroutine, lines 6-13, to ensure that its downstream dependencies of higher or equal criticality are activated. The pre-order graph traversal complies with [R2], and using criticality as the key while popping nodes from Q ensures that [R1] is satisfied. We avoid redundant computation by maintaining a visited set, thereby reducing the time complexity for each graph to be similar to a DFS/BFS traversal in $O(V + E)$ time.

Global Ranking: This module takes as inputs the operator objective, Obj , and the ordered lists of microservices per application generated by the priority estimator, $AppRank$. It generates as output, $GlobalRank$, a single ordered list of microservices across all applications, with tuples of the form $\langle AppID, MicroserviceID \rangle$.

We use a priority queue, Q , to track the pending microservices sorted based on the key, Obj . We discuss later in this section how various objectives translate to a sorting order. Q is initialized with the first node in every application's individual priority list in line 22. In each step, the microservice with the highest value based on the operator's objective is popped from the priority queue and added to $GlobalRank$ in lines 24-28. The resource usage of the microservice is deducted from the available capacity in line 25. The next node in the corresponding application's priority list is then added to the priority queue. This process continues until all microservices are visited. Cluster operators can plug in any sorting key or objective in our algorithm.

We discuss two objectives here. (i) *Cost-Based:* The key in the cost-based global ranking is the price per unit resource. Microservices that generate higher revenue are prioritized. (ii) *Fairness-Based:* In the fairness-based ranking, we precompute the max-min fair share of every application based on its total resource requirements. Microservices of applications using the least fraction of their fair share are preferred in each round. The key in this context is the residual resources, which are defined as the current allocation subtracted from the fair share.

3.2 Gecko Scheduler

The scheduler is responsible for mapping microservices to servers based on the ordered list generated by the planner. This formulation is a variation of the well-known bin-packing [3] problem and is known to be NP-hard. Hence, we design a criticality-aware scheduling heuristic. The scheduler has two modules: the packing module, which generates the mapping, and an agent that executes it.

Packing module: The packing module is responsible for mapping the microservices to servers based on the sequence generated by the planner. Note that this module performs all operations on a copy of the cluster state and does not enforce them on the cluster. The final execution is deferred to the Agent. The detailed pseudocode is given in Appendix 2. We

highlight the key steps here. Note that while we present the design assuming a single container for every microservice, our algorithm can also handle multiple replicas (Appendix B).

When a cluster experiences partial failures, a fraction of microservices in the planner list may be already running. The packing module iterates over the ordered list generated by the planner. If the next microservice to be scheduled is already running on a server that has not failed, it is allowed to continue on the same server. If the microservice to be activated was previously running on a failed server, it needs to be rescheduled on an active server. The scheduling heuristic first sorts the active nodes in the decreasing order of available capacity. If the resource requirements of the microservice can be accommodated without migrating any of the other active microservices, it is assigned to the server that has the smallest available capacity larger than the required resources, i.e., the best-fit strategy [4].

If the microservice cannot be accommodated on the existing servers based on the available capacity, the heuristic will proceed to find a migration strategy. It identifies a source server from which microservices can be migrated based on available capacity and the size and count of microservices currently active on the server. Smaller microservices are more likely to be accommodated in the available capacity of other servers. Hence, nodes with large available capacity and a large number of small currently active microservices are preferred. The heuristic then proceeds to migrate the microservices on the node to other active servers. If the heuristic fails to identify a target server for a microservice under both best-fit and migration strategies, it proceeds to delete the currently active services in the reverse order from the planner’s list. The lowest-priority microservices are deleted first. After each deletion, the heuristic attempts best-fit and migration strategies again to find a suitable mapping.

Agent: The agent continuously monitors the cluster state, and when a failure event is detected, reports it to the planner. The agent is also responsible for executing the list of tasks determined by the Gecko scheduler on the cluster scheduler. At a high level, the agent performs three tasks: deleting non-critical microservices, migrating already running microservices, and restarting microservices that were impacted. We use cluster scheduler API to execute these tasks. Additional subroutines are performed with all three tasks, such as draining traffic, scaling up/down the containers, and reconfiguring iptables, which we detail in the implementation section.

4 Implementation

4.1 Gecko System

Gecko system is written in Python. We test Gecko Controller with Kubernetes cluster scheduler, however, our design can work with other cluster schedulers [43, 57, 60] with minor modifications. Gecko Agent monitors the cluster state at 15-

second granularity. This is a tunable parameter. We chose 15 seconds to maintain a low response time while ensuring the Kubernetes cluster is not overwhelmed. Gecko can operate in settings where only some applications are diagonal scaling compliant. We achieve this by using labels on namespaces, tagging only the subscribed applications as "Gecko=enabled". Within the Gecko controller, we store the application dependency graphs and associated criticality tags and resource requirements as NetworkX DiGraph objects [22]. In the packing module, we employ a tree-based data structure, Python’s Sorted Lists [28], to perform insert, search, and delete operations faster than linear time.

Note that the Gecko currently only supports diagonal scaling on stateless services. Stateful workloads require reasoning about consistency which we defer for future work. Nonetheless, stateless workloads comprise more than 60% of large data center machine usage, as reported by real-world data centers [46]. Since Gecko can provide benefits even when a fraction of applications are not diagonal scaling compliant, we expect significant benefits with 60% compliant workloads.

4.2 Diagonal Scaling Practical Experience

In microservice-based architectures, application functionalities are decomposed into microservices, which are typically deployed as isolated containers and composed to serve requests. To demonstrate the feasibility of diagonal scaling in the real world today, we take Overleaf, a production-ready open-sourced application for collaborative editing that serves millions of users, and assess its ability to diagonally scale during disaster scenarios. Overleaf comprises of 14 microservices with various features implemented as separate microservices. For example, spell-check, clsi for compiling, etc. Users typically login, enter into a project, and then perform edits, spell-checks, compiles, etc. Edits typically require low latencies and are done on a WebSocket connection, whereas most other services are implemented as REST calls.

Due to the decoupling of features in Overleaf, the application is directly amenable to diagonal scaling without any modifications. In addition, Overleaf employs several application-level resilience measures, such as error handlers and rate limiters, that allow the application to continue offering services even when non-critical microservices are turned off. This ability to handle failures of downstream microservices, a common practice across production applications, enables these applications to be diagonal scaling compliant.

We next analyze Hotel Reservation (HR) application from DeathStarBench (DSB), a benchmark suite for cloud microservices. Note that this is a demo application with limited functionalities on a research framework. HR has 4 services—search, user, recommendation, and reservations—composed of 19 microservices. Unlike Overleaf, where a user edits on sticky web socket connections, in HR, all connections are stateless RPC calls using a gRPC library [12]. Each logical

microservice, such as reservation, is backed by a database.

Similar to Overleaf, HR isolates non-critical functionalities, like recommendations, as independent microservices. However, HR’s current codebase is not crash-safe, as turning off less critical microservices can result in a front-end crash. Since HR is a demo application, it does not have the error-handling logic necessary for resilience. We wrap these optional calls in conditional if statements to prevent the application from crashing. Next, we analyze individual call graphs in HR and identify opportunities for diagonal scaling. For example, the reservation service invokes the frontend, reserve, user, memcached-reserve, and mongo-reserve microservices. However, we find that the user microservice does not lie in the critical path of the reservation request, i.e., users can make reservations as a guest when the user microservice is unavailable. Therefore, we add error-handling logic to prevent the request from crashing when this microservice is unavailable. In summary, we only had to make less than 100 lines of code changes to make HR diagonal-scaling compliant.

5 Evaluation

We evaluate Gecko and answer the following questions:

- Can Gecko achieve operator objectives such as fairness and revenue while also maximizing application availability?
- Can Gecko perform well at scale in clusters with over 100,000 servers and across real-world application dependency graphs with thousands of microservices?
- Can diagonal scaling mitigate the impact of failure events across diverse cloud applications and failure rates?

5.1 Setup

Workloads: We evaluate Gecko using large-scale real-world data center traces. We also demonstrate the feasibility of diagonal scaling with Gecko using Cloudlab [25] experiments involving real-world microservice-based applications.

We use a dataset consisting of over twenty million call graphs collected over a period of seven days from clusters at Alibaba [48]. To evaluate Gecko at scale, we derive 18 application dependency graphs of varying sizes from 10 to 3000s of microservices from Alibaba’s cluster traces by following their analysis methodology [49]. Since these traces do not include the CPU/memory usage, we evaluate two realistic resource models to approximate the resource requirements of each microservice: (i) resource as a function of calls-per-minute [50], another study from Alibaba on the same dataset, and (ii) Resources sampled from a long-tailed distribution model as specified in Azure Bin-packing traces [2].

To demonstrate the feasibility of diagonal scaling in the real world, we use two applications—Overleaf [23] and Hotel Reservation (HR) from DeathStarBench [39]. Overleaf already supports diagonal scaling, whereas in HR, we make

the application diagonal scaling compliant with minimal modifications (§ 4). For load generation, we use publicly available load generators [58, 59] for Overleaf and wrk2 [39] for HR. We determine the resource requirements of microservices by running the applications under varying loads; for example, different levels of edit, compile, etc. for Overleaf.

Schemes: We evaluate two variants of Gecko—GeckoFair (operator objective is max-min fairness) and GeckoCost (operator objective is to maximize revenue). We compare Gecko variants with four baselines: Default, Priority, Fair, and No Diagonal Scaling. The *Default* baseline represents the default behavior of Kubernetes when failures strike, without information on criticality tags and topology of the application DGs. *Priority* represents a variant of Kubernetes baseline with PodPriority [17], wherein Kubernetes scheduler ensures that pods with higher criticality tags are scheduled first. *Fair* is criticality unaware but is topology-aware and also ensures max-min fairness across applications. *No diagonal scaling* is a scheme where applications cannot function when their resource requirement is not met, i.e., they are not degradable.

Operator Metrics: We report operator metrics such as utilization, revenue, and resource fairness at varying failure rates. Revenue is computed as a function of whether a microservice is activated or not when failures strike. Fairness is measured as the deviation from max-min fairness. We decompose the deviation fairness measure into two parts: positive deviation (using more resources than fair share) and negative deviation (using fewer resources than fair share).

Environments: We run experiments in two settings: (i) Standalone Gecko evaluation which takes as input a cluster state and generates an output. Since we use this setting for testing at scale, we do not interface it with a Kubernetes controller. We test up to a scale of 100,000 nodes in this setting with 1000 instances of real-world applications obtained from Alibaba cluster traces. The applications vary in size from tens to thousands of microservices. A detailed analysis of the workload is presented in Appendix E (ii) Cloudlab-based Kubernetes cluster of 200 CPUs, with 64-bit Intel Quad Core Xeon E5530 processor d710 machines, on Ubuntu 22.04 LTS as a real-world multi-tenant setting. We use multiple instantiations of Overleaf and Hotel Reservation applications, each with its own load generator and unique criticality tag assignments, for evaluations in the real-world cluster. Since the datasets are widely different, we discuss metrics and tagging schemes used in each context.

Standalone Gecko settings: Due to a lack of visibility into Alibaba’s application graphs, we define criticality in a best-effort manner. We introduce two schemes: (i) service-level tagging and (ii) frequency-based tagging. A service is a set of microservices that together offer a useful functionality. In *service-level tagging*, we identify the most frequently invoked services and assign all the component microservices as C_1 . In *frequency-based tagging*, we use a linear program to

Application	Metric	Application	Metric
Overleaf0	document-edits	HR0	search
Overleaf1	versions	HR1	reserve
Overleaf2	downloads		

Table 1: Resilience objective of individual applications. For example, we say Overleaf0’s resilience goal as satisfied if the throughput of document-edits remains unaffected.

find the top microservices that can serve specified target percentile requests. We generate both service-level tagging and frequency-based tagging at 50th and 90th percentile, denoted as P50 and P90, respectively.

In addition, in all schemes, we tag a tiny fraction of infrequently invoked services that are randomly chosen as highly critical. This is to account for critical background services that are infrequent, such as garbage collection routines that ensure requests can continue serving without performance degradation. We define an application’s **critical service availability goals** as met when all the microservices of C_1 are running and unmet if even a single C_1 microservice was not activated by Gecko and other baselines.

Gecko on Clouddlab Settings: We employ the experiment design methodology proposed in Chaos Engineering [24, 29], of defining the steady state metric for each application for real-world testing. For example, for Overleaf0, we mark the steady state metric as edits per second. We treat an application’s resilience goal as satisfied if the throughput of the steady state metric is retained after a failure event. In Table 1, we assign different steady-state metrics to different instances of applications to ensure heterogeneity in resilience goals.

We then tag individual microservices based on the definition of steady state metric. For example, in HR1, reservation is the primary or steady state metric, and hence we tag the corresponding microservices as C_1 . Although the user’s credentials are fetched during reservation, we tag the user microservice as C_5 since reservations can be made as guests too. All other microservices are tagged to lower criticalities. Note that stateful workloads such as mongodb [21] are running separately, as is typical practice adopted by cloud applications when deploying stateful and stateless workloads [48]. In the real-world setting, we define an application’s **critical service availability goals** as met when its critical services (listed in Table 1) continue to run and unmet otherwise.

To show the impact of degrading low-criticality features, we augment the load generator to compute a utility. Each microservice is assigned a utility value that aligns with its criticality. The utility of a service is the sum of the utilities of the component microservices. When some of the microservices are turned off, the utility decreases accordingly.

5.2 Standalone Gecko Evaluation

In Figure 3, we compare the performance of Gecko against the four baselines on a 100,000-node cluster using the Alibaba

trace dataset at a cluster utilization of 90%. We report the best result here and add all other results in Appendix D.

Gecko offers high critical service availability. Figure 3(a) shows the critical service availability under varying levels of failure rates. We normalize the availability with respect to that of the unaffected cluster state and report the average across all applications at each failure level. Default (which does not have criticality awareness, dependency awareness, or packing efficiency) performs the worst. Priority performs poorly due to a lack of inter-app prioritization, which results in a few applications with many high-criticality microservices using most of the resources. Fair’s topology- and fairness-aware resource allocation leads to better availability than Priority, yet it suffers performance deterioration due to a lack of criticality awareness. Gecko variants with intra- and inter-app prioritization offer high availability.

GeckoCost maximizes revenue. Figure 3(b) reports normalized revenue with respect to the cluster before failure. GeckoCost offers superior performance due to its ability to pack more microservices efficiently while explicitly maximizing revenue. Since Fair and GeckoFair are designed for fairness, they naturally perform poorly at high failure levels.

GeckoFair offers the highest fairness. In Figure 3 (c), we report the deviations from fair share across three failure levels of 10%, 50%, and 90%. Ideally, we want the deviation to be close to zero. A negative fair share occurs when an application receives fewer resources than its max-min fair allocation, and a positive fair share when it receives more. We observe that with varying failure rates, GeckoFair has the lowest total deviation. Due to the indivisibility of microservices within an application and the inability to activate beyond fair-share, Fair deviates more on the negative side. Since GeckoFair follows a relaxation of the strict fair share criterion, it can achieve much less deviation on both sides. Other schemes perform poorly due to the inability to enforce inter-app fairness.

Gecko scales well to real-world cluster sizes. We evaluate the scalability of Gecko against baselines by measuring their run-time performance in Figure 4. We run the evaluation on a Linux machine with 24 physical cores and 48 logical processors. LP variants do not scale to cluster sizes beyond 1000 servers, even with applications with less than 20 microservices. Gecko can scale to 100,000 servers while handling application graph sizes up to 3000 microservices, with comparable runtime performance as Default.

5.3 Gecko with Kubernetes on Clouddlab

We run 5 workloads—3 Overleaf instances (Overleaf0, Overleaf1, Overleaf2) and 2 Hotel Reservation (HR) instances (HR0, HR1)—on a Kubernetes cluster with 200 CPUs. We first show how Gecko balances operator objectives and application resilience. Figures 5 (a) and (b) show the performance when the cluster capacity is reduced to 42%. All results are

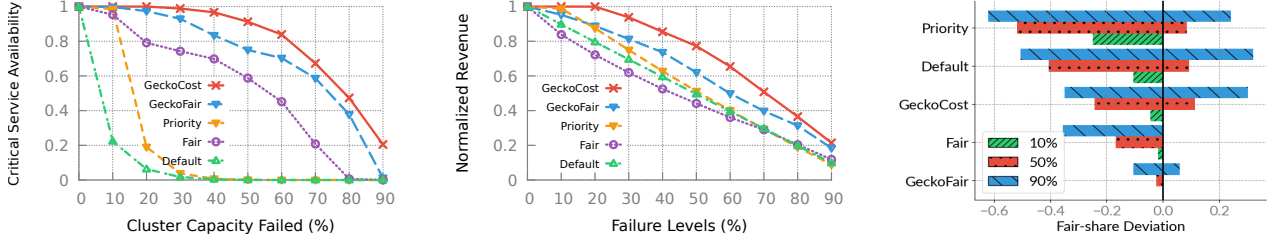


Figure 3: Gecko evaluated on Alibaba with Service-Level-P90 criticality tagging scheme and Calls-Per-Minute (CPM) based resource assignment scheme on a 100000-node cluster. (a) Aggregate critical service availability across applications at different capacity failure scenarios shows that GeckoFair and GeckoCost activate more critical services consistently. (b) Normalized revenue shows that GeckoCost maximizes revenue. (c) Deviation from fair-share shows that GeckoFair has least deviation.

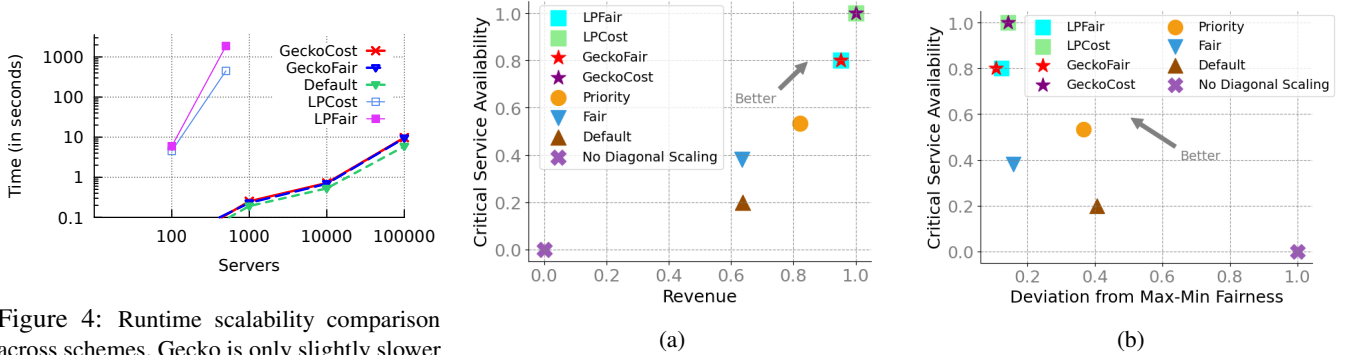


Figure 4: Runtime scalability comparison across schemes. Gecko is only slightly slower than Default. LP does not scale beyond 1000 nodes. We run the evaluation on a Linux machine with 24 physical cores and 48 logical processors.

Figure 5: Gecko evaluated on a Kubernetes cluster with 5 application instances and cluster capacity reduced to 42%. Critical service availability across 5 applications (with heterogeneous goals listed in Table 1) is shown. The x-axis shows two cluster operator objectives (a) Revenue and (b) Deviation from Fairness

reported by averaging across 5 trials. On the y-axis, we plot the critical service availability, i.e., availability of services with target steady-state metrics listed in Table 1. On the x-axis, we plot the application objective—revenue in (a) and fair share deviation in (b). We make the following observations.

Diagonal scaling improves overall performance. As shown in Figures 5 (a) and (b), no diagonal scaling has poor performance. When applications are not diagonally scalable, they can no longer run in resource-constrained environments. On the other hand, diagonal scaling enables applications to continue running in reduced capacities.

Criticality tags can improve application’s resilience. As shown in Figures 5 (a) and (b), all the schemes that leverage criticality tags (Gecko variants, LP variants, and Priority) perform better on the y-axis (resilience goals) ensuring that critical service availability is maintained in low-resource environments. The baselines that are not criticality aware (Default and Fair) perform poorly on availability.

Gecko can work across different operator objectives. Along with maintaining a high availability, Gecko variants are able to maximize across two cluster operator objectives: Revenue and Fairness, as shown in Figures 5 (a) and (b). Both

LPCost and LPFair outperform other baselines on availability in Figures 5 (a) and (b). Gecko closely matches the performance of the optimal LP under both objectives.

We next report the aggregate resource consumption per criticality level across 5 instances. The resource division between C_1 and non-critical (C_2 and lower) is $\approx 60:40$. Furthermore, the load is such that all C_1 require $\approx 40\%$ of cluster capacity, and all applications together require $\approx 70\%$ of cluster capacity. In this experiment, we introduce failure rates up to $\approx 42\%$, below which C_1 microservices could fail. We compare GeckoCost against the Kubernetes Default mechanism. For details on the division of resources by criticalities, refer to Appendix D. To emulate failure events in the cluster, we stop the kubelet process [16] on the failed nodes.

Gecko meets the availability targets. Figure 6 (a) and (b) show two runs with Gecko and Default, respectively. On the y-axis, we report the critical service availability. The x-axis shows the event timeline, aggregated at 30-second intervals. To ensure a fair comparison, we use the same detection mechanism for both schemes. We show that Gecko, in a few minutes, recovers from the dip bringing back the steady state of each application at 400th second mark, whereas Default only rises

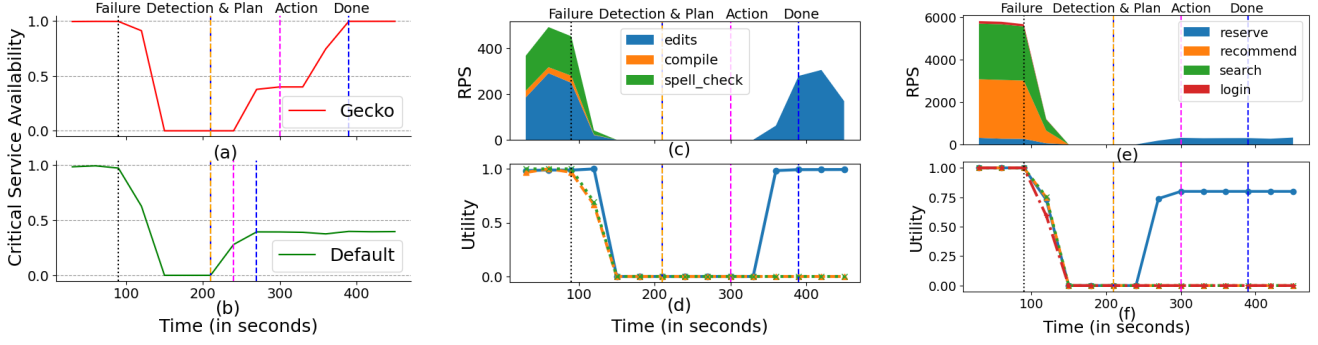


Figure 6: Diagonally scaling a multi-tenant cluster with 5 microservice application instances using Gecko. (a) and (b) show the benefits of Gecko over Default when cluster capacity reduces to 40%. (c) and (e) Demonstrates diagonal scaling on Overleaf and HR, respectively, where critical service throughput (requests per second) is retained while non-critical services are turned off during resource-crunch scenarios. (d) and (f) show end-user utility degradation of different services under diagonal scaling. (f) demonstrates the degradation of option yet "good-to-have" features as end-user utility drops to 0.8 for reservations as a result of degradation of a non-critical call to user microservice allowing reservations to be made as guests.

to 0.4 i.e., only 2 out of 5 applications' steady states are recovered. Gecko detects the failure after 100 seconds (orange line) and prepares a plan almost instantly (blue line overlapping on orange). The Gecko agent then issues commands to the Kubernetes cluster (magenta). Finally, the agent marks the cluster state as recovered (blue) when Kubernetes reports all the desired pods as running. The time elapsed between executing action (magenta) and done (blue) can vary depending on the pod startup times.

Gecko recovery delay is slightly longer but tolerable. In Figure 6 (a), Gecko takes longer to reach the steady state after failure. This is because Gecko turns off those microservices that are non-critical to free up capacity for the failed critical services. Default, on the other hand, being criticality unaware, tries to reschedule as many microservices in a random order. The delay in detection is primarily due to Kubernetes `list_node` API, which updates a node as "Not Ready" only after it misses three consecutive heartbeats (≈ 40 seconds). In addition, Gecko adds a 30-second delay to ensure that we fetch all the unhealthy nodes. This is a configurable delay. Furthermore, the rise in the critical service availability of Gecko in Figure 6 (a) at 300th second mark shows that 2 applications recovered early. This time can vary depending on pod startup, image pull times, etc.

Overleaf can diagonally scale without any modifications.

We now zoom in into one of the instances of Overleaf to show how Gecko performs diagonal scaling in Overleaf, shown in 6 (c) and (d), for the same time-varying run reported in 6 (a). Figure 6 (c) shows the throughput as requests per second on the y-axis and time on the x-axis of three request types, namely compile, edits, and spell-check, plotted as a stacked chart. The long lull (whitespace) in (c) and (d) represents that the application went down due to the ingress service "web" being impacted by the failure. We observe that the throughput of

edit requests recovers by turning off two non-critical services, based on our definition of resilience goal in 1 for Overleaf0. The plot (d) shows how the end-user's utility falls for compile and spell-check after recovery, while ensuring maximum utility for edits. Note that we instrument locust scripts to assign a utility score (value that an end-user receives) when a request succeeds. Utility is 0 when a request fails. Without any code modifications in Overleaf, Gecko degrades non-critical services, such as spell-check and compile, to ensure Overleaf's throughput is recovered.

Diagonal scaling enables the dropping of optional features.

In Figure 6 (c), entire services are turned off. We next provide another example where requests can continue to be served in a degraded mode by dropping optional yet "good-to-have" features. Figure 6 (e) shows how non-critical services are turned off by Gecko while ensuring the throughput of reservation is maintained. Furthermore, in (f), we observe that reservation's utility was decreased to 0.8, showing that the end-user utility of reservation drops. This is a result of turning off a non-critical downstream call to the user, allowing reservation to be made as a guest. Note that when we partially prune a service, the performance may degrade due to timeouts. However, we do not observe any performance degradations, as shown in our P95 latency measurements in § 5.4.

5.4 Microbenchmarking Results

Gecko is resource-efficient. Figure 7 shows the cluster capacity utilized under various failure rates by Gecko planner, Gecko scheduler (output obtained after planner and scheduler), and the default scheduler. We observe that Gecko scheduler consistently outperforms the Default scheduling of Kubernetes, demonstrating the superior packing efficiency of Gecko. Moreover, the drop in cluster utilization from planner

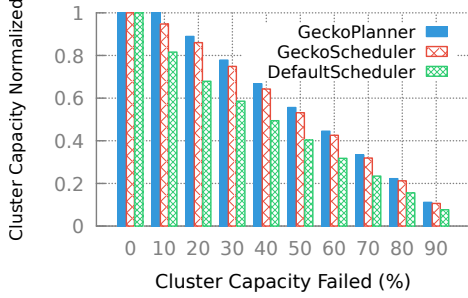


Figure 7: **Breakdown of Gecko performance.** Loss of utilization with only Gecko planner compared with both planner and scheduler enabled shows that both modules are highly efficient in Gecko.

Applications	Services	P95 Latencies (in ms)	
		Before	After
Overleaf	edits	141	144
	compile	4317.9	–
	spell_check	2296.7	–
HR	reserve	55.33	50.11
	recommend	47.43	–
	search	53.26	–
	login	41.8	–

Table 2: End-to-end 95th percentile (P95) latencies for services in HotelReservation and Overleaf before and after diagonal scaling.

to scheduler output is minimal.

Gecko offers high performance for critical services. Table 2 reports the P95 latencies (in ms) before and after diagonal scaling. The services that were pruned through diagonal scaling are denoted as “–”. Overleaf does not exhibit any partial pruning because its request types do not have any optional microservices. While the service ‘reserve’ of HR is partially pruned (turning off one downstream microservice), the performance overheads are minimal. This is because HR uses gRPC for inter-process communication, which builds on top of HTTP/2 protocol and leverages its ability to detect failed connections, thereby failing fast without incurring timeout overheads [12].

In summary, we make the following observations:

- Gecko can offer high critical service availability for applications while maximizing operator objectives (maximizing revenue/minimizing fairness deviations). (Fig. 3)
- Gecko scales well to real-world data center sizes. (Fig. 3)
- We demonstrate that diagonal scaling is practical with today’s applications, using Overleaf as an example. (Fig. 6)

6 Discussions and Future Work

We now discuss challenges that may arise with Gecko.

Changing Resource Requirements: As we perform graceful

degradation in real-time with user traffic, user behavior may change, thereby changing the resource requirements of microservices. For example, when a video service is turned off, users may shift towards a chat service changing the resource requirements of microservices invoked in chat. Existing load balancing schemes in clusters, including vertical/horizontal scaling, can adapt to these variations. In the future, we can extend Gecko to leverage learned resource profiles [35, 53, 54].

Broader Applications of Gecko: We demonstrate the benefits of Gecko with large-scale partial failure scenarios within a data center. Gecko can also be useful in other settings. Gecko can also help with migration from one data center to another, or between on-premise and cloud instantiations of the same user. Another use case involves updating the servers within a production data center by migrating out microservices running on these servers. In an active data center, Gecko can be given a list of servers to be drained and it can successfully reschedule the active application components in these servers to others. Gecko helps to automate the process of draining and reloading in such scenarios.

Extending to Other Tags and Infrastructure Controllers:

Gecko focuses on criticality tags and stateless applications. A future direction involves accommodating stateful applications and more expressive resilience tags such as consistency tags. In addition, we envision extending automated resilience management to other infrastructure controllers such as the network controller. This consideration also inspired the separation of the planner and the scheduler in Gecko. In the eventual automated resilience management system, the planner will have a broader view across servers, the network, and the storage system. The scheduler will then generate a sequence of steps across the cluster manager, network controller, storage controller, etc.

Incorrect Criticality Tags: When applications tag their microservices incorrectly, the impact is limited to that application alone and not the entire cluster. Developers may use chaos tests to verify their tagging.

7 Conclusion

The resilience of cloud infrastructure is crucial for cloud operators and cloud applications alike. We put forward diagonal scaling as a novel strategy for resilience engineering. Our automated cloud resilience management system, Gecko, which leverages diagonal scaling, can seamlessly handle a broad range of failure scenarios to improve the overall resilience of applications without sacrificing utilization or fairness. We show that current microservice-based applications are amenable to diagonal scaling through our real-world experiments. Our work opens several new directions for research, including the design of new resilience tags and extensions of automated resilience management to stateful applications and other infrastructure controllers.

References

- [1] AWS Internet Outage Cause Human Error Incorrect Command. <https://www.vox.com/2017/3/2/14792636/amazon-aws-internet-outage-cause-human-error-incorrect-command>. (Accessed on 04/04/2023).
- [2] Azure Trace for Packing 2020. <https://github.com/Azure/AzurePublicDataset/blob/master/AzureTracesForPacking2020.md>. (Accessed on 04/04/2023).
- [3] Best-fit bin packing. https://en.wikipedia.org/wiki/Best-fit_bin_packing. (Accessed on 04/06/2023).
- [4] Bin packing problem. https://en.wikipedia.org/wiki/Bin_packing_problem. (Accessed on 04/06/2023).
- [5] Chaos Engineering. <https://netflixtechblog.com/tagged/chaos-engineering>. (Accessed on 04/04/2023).
- [6] Disney + Hotstar and their tale with scalability . <https://www.linkedin.com/pulse/disney-hotstar-tale-scalability-achyutha-rao-sathvick/>. (Accessed on 04/06/2023).
- [7] Failover with AWS. <https://docs.aws.amazon.com/whitepapers/latest/web-application-hosting-best-practices/failover-with-aws.html>. (Accessed on 04/04/2023).
- [8] Fault tolerance through optimal workload placement. <https://engineering.fb.com/2020/09/08/data-center-engineering/fault-tolerance-through-optimal-workload-placement/>. (Accessed on 03/12/2023).
- [9] Feature Toggles (aka Feature Flags). <https://martinfowler.com/articles/feature-toggles.html>. (Accessed on 11/26/2023).
- [10] Google - site reliability engineering. https://sre.google/sre-book/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degradation. (Accessed on 12/05/2023).
- [11] Google cloud service health. <https://bit.ly/46WTJLb>. (Accessed on 12/02/2023).
- [12] gRPC: Identifying Failed Connections. <https://grpc.io/blog/grpc-on-http2/#identifying-failed-connections>. (Accessed on 11/26/2023).
- [13] Improving the Resilience of your Software: a Practical Approach. <https://medium.com/ssense-tech/improving-the-resilience-of-your-software-a-practical-approach-9ca8952e09bd>. (Accessed on 05/12/2022).
- [14] Incident Review – Google Cloud Outage has Widespread Downstream Impact. <https://www.catchpoint.com/blog/incident-review-google-cloud-outage>. (Accessed on 04/04/2023).
- [15] Istiodie 1.4 / circuit breaking. <https://istio.io/v1.4/docs/tasks/traffic-management/circuit-breaking/>. (Accessed on 12/05/2023).
- [16] Kubelet. <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>. (Accessed on 11/23/2023).
- [17] Kubernetes: Pod Priority and Preemption. <https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/>. (Accessed on 04/04/2023).
- [18] Labels and Selectors. <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>. (Accessed on 05/12/2023).
- [19] Manage reliability to a higher standard with Grem-lin. <https://www.gremlin.com>. (Accessed on 04/04/2023).
- [20] Microsoft Blames "Severe" Weather for Azure Cloud Outage. <https://www.datacenterknowledge.com/uptime/microsoft-blames-severe-weather-azure-cloud-outage>. (Accessed on 12/01/2023).
- [21] MongoDB. <https://www.mongodb.com>. (Accessed on 10/31/2022).
- [22] NetworkX. <https://networkx.org>. (Accessed on 12/06/2023).
- [23] Overleaf: LaTeX, Evolved The easy to use, online, collaborative LaTeX editor. <https://www.overleaf.com>. (Accessed on 12/06/2023).
- [24] Principles of Chaos Engineering. <https://principlesofchaos.org>. (Accessed on 11/13/2023).
- [25] Profile emulab-ops/k8s. <https://www.cloudlab.us/show-profile.php?project=emulab-ops&profile=k8s>. (Accessed on 09/11/2022).
- [26] Shrinking the time to mitigate production incidents—CRE life lessons. <https://cloud.google.com/blog/products/management-tools/shrinking-the-time-to-mitigate-production-incidents>. (Accessed on 04/04/2023).

- [27] Simplify observability, traffic management, security, and policy with the leading service mesh. <https://istio.io>. (Accessed on 04/04/2023).
- [28] Sorted Containers. <https://grantjenks.com/docs/sortedcontainers/introduction.html#sorted-list>. (Accessed on 09/25/2023).
- [29] SPS: the Pulse of Netflix Streaming. <https://netflixtechblog.com/sps-the-pulse-of-netflix-streaming-ae4db0e05f8a>. (Accessed on 11/13/2023).
- [30] Target group load shedding for application load balancer | networking & content delivery. <https://aws.amazon.com/blogs/networking-and-content-delivery/target-group-load-shedding-for-application-load-balancer/>. (Accessed on 12/05/2023).
- [31] The Bulkhead Pattern. <https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead>. (Accessed on 12/06/2023).
- [32] The Netflix Simian Army. <http://techblog.netflix.com/2011/07/netflix-simian-army.html>. (Accessed on 04/03/2023).
- [33] Using load shedding to avoid overload. <https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/>. (Accessed on 12/05/2023).
- [34] Tarek F Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks*, 31(11-16):1563–1577, 1999.
- [35] Romil Bhardwaj, Kirthivasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro: {Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 623–643. USENIX Association, 2023.
- [36] Houssein-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301. IEEE, 2012.
- [37] Andrea Detti, Ludovico Funari, and Luca Petrucci. μ bench: an open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, 2023.
- [38] Shuai Ding, Sreenivas Gollapudi, Samuel Jeong, Krishnam Kenthapadi, and Alexandros Ntoulas. Indexing strategies for graceful degradation of search quality. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 575–584, 2011.
- [39] 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.
- [40] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 126–141, 2022.
- [41] Maurice P Herlihy and Jeannette M Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.
- [42] Hideaki Hibino, Kenichi Kourai, and S Shiba. Difference of degradation schemes among operating systems: Experimental analysis for web application servers. In *Workshop on Dependable Software, Tools and Methods, Yokohama, Japan*. Citeseer, 2005.
- [43] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.
- [44] Kimberly Keeton, Cipriano A Santos, Dirk Beyer, Jeffrey S Chase, John Wilkes, et al. Designing for disasters. In *FAST*, volume 4, pages 59–62, 2004.
- [45] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodríguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711, 2014.
- [46] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 553–569, 2021.

- [47] Jingqiang Lin, Bo Luo, Jiwu Jing, and Xiaokun Zhang. Grade: Graceful degradation in byzantine quorum systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 171–180. IEEE, 2012.
- [48] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.
- [49] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.
- [50] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: Microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 355–369, 2022.
- [51] Justin J Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, et al. Defcon: Preventing overload with graceful feature degradation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 607–622, 2023.
- [52] Jeremy Philippe, Noel De Palma, Fabienne Boyer, and et Olivier Gruber. Self-adaptation of service level in distributed systems. *Software: Practice and Experience*, 40(3):259–283, 2010.
- [53] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 805–825, 2020.
- [54] Krzysztof Rządca, Paweł Findeisen, Jacek Swiderski, Przemysław Zych, Przemysław Broniek, Jarek Kusmierek, Paweł Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [55] Yasushi Saito, Brian N Bershad, and Henry M Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. *ACM SIGOPS Operating Systems Review*, 33(5):1–15, 1999.
- [56] Mohammad Shahradd and David Wentzlaff. Availability knob: Flexible user-defined availability in the cloud. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 42–56, 2016.
- [57] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803, 2020.
- [58] Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. Sharelatex on the edge: Evaluation of the hybrid core/edge deployment of a microservices-based application. In *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets*, pages 8–15, 2018.
- [59] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 14–27, 2017.
- [60] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.
- [61] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 373–389, 2018.
- [62] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.
- [63] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.
- [64] J Robert von Behren, Eric A Brewer, Nikita Borisov, Michael Chen, Matt Welsh, Josh MacDonald, Jeremy Lau, and David E Culler. Ninja: A framework for network services. In *USENIX Annual Technical Conference, General Track*, pages 87–102, 2002.

- [65] Minxian Xu and Rajkumar Buyya. Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 52(1):1–27, 2019.
- [66] Lidong Zhou, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Roy Levin, and Chandramohan A Thekkath. Graceful degradation via versions: specifications and implementations. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 264–273, 2007.

Algorithm 2: Packing Heuristic

Input: P : Planner Output List, A : Assignments,
Output: A : Final assignment of microservices to server

Data: C : Cluster State object

```
1 def Main:
2   foreach  $ms \in P$  do
3     if  $ms \notin A$  then
4        $node = \text{GetBestFit}(ms, A)$ 
5       if  $node$  is None then  $node =$ 
6         RepackMicroservicesToFit( $ms, A$ )
7       if  $node$  is None then  $node =$ 
8         DeleteLowerRanksToFit( $ms, A$ )
9       if  $node$  is None then break
10      else UpdateClusterState( $node, A$ )
11
12  return A
13
14 def GetBestFit( $ms, A$ ):
15    $best = \text{None}$ 
16   foreach  $node \in C.nodes$  do
17     if  $node.size \geq ms.size$  and  $node.empty \leq$ 
18        $best.empty$  then  $best = node$ 
19   return  $best$ 
20
21 def RepackMicroservicesToFit( $ms, A$ ):
22   foreach  $node \in \text{sorted}(A.nodes(key =$ 
23      $\text{RemainingCapacity}, reverse = \text{True}))$  do
24     if  $\text{IsMigrationFeasible}(node)$  then
25       RepackMicroservices( $node$ )
26     return  $node$ 
27   else continue
28   return None
29
30 def DeleteLowerRanksToFit( $ms, A$ ):
31   foreach
32      $loms \in \text{sorted}(A.podsrnk, reverse = \text{True})$  do
33      $node = \text{DeletePod}(loms)$ 
34     if  $ms.size \leq node.size$  then return  $node$ 
35     else continue
36   return None
```

A Packing Heuristic Pseudocode

We now provide the pseudocode for the bin-packing heuristic in Algorithm 2. The main function runs in the following manner: First, it traverses over the list of microservices outputted by the planner. If a microservice is not already scheduled, then it first calls the best-fit subroutine [4] to find the node with the least remaining capacity that can accomodate the microservice. If a node is found then we proceed to line 8 of updating the cluster state and proceeding to the next microservice. If a node is not found, then we proceed to the repacking strategy.

The repacking strategy iterates over the nodes (in the order of most empty to least empty). And it checks the feasibility of migrating smaller sized microservices on to other target nodes. If such a node is found which can free up capacity by repacking smaller microservices to other nodes then it does the migration, as shown in Line -17 and returns the candidate node else it returns None, i.e., it cannot find any node in the list.

Finally, if repacking also does not work then we perform deletions of lower ranked microservices to free up capacity for the higher ranked microservice to be scheduled. If such a node is freed up then we return the node else we return none.

B Extending to Multiple Replicas

The LP treats a microservice as a single instance in formulation described in Section 3. Here, we provide a mechanism to extend it such that each microservice can have multiple replicas, as is common practice. Towards this, Gecko Planner algorithm requires no changes as it does not operate on the microservice instance information but only requires the resource information as a whole. On the bin-packing heuristic side, we can easily modify our implementation to treat a microservice as active or not if all microservices replicas are activated. Specifically, when we obtain a list from the planner, we traverse based on the ranking of microservices, for every microservice we see whether all the replicas are running or not. If not we assign them using the three-pronged bin-packing heuristic strategy of best-fit, repacking and deletion. If in case one of the replicas, does not fit, we do not extend beyond and terminate the for loop by deleting all microservice replicas.

C Microservice Deletion, Migration, and Restarts in Kubernetes

In this subsection, we detail the fine-grained processes involved in microservice deletion, migration, and restarts at the Kubernetes level when the Gecko agent issues the corresponding commands.

Deletion: When a graceful shutdown request is issued, the cluster scheduler starts by removing service endpoints on each server where the microservice is running to avoid any incoming requests. Next, the cluster scheduler scales the replicas gradually down by issuing SIGTERM commands. These containers then complete any outstanding requests and then terminate. However, if a container doesn't terminate after a threshold time limit is reached, a SIGKILL command request is issued. Currently, Gecko Agent supports draining two types of traffic: 1. Stateless Connections and 2. Websocket connections. In addition to the deletion of microservices, Gecko Agent is extensible for running other user-defined subroutines pre-deletion or post-deletion to manage upstream or downstream dependencies. These can include toggling knobs or

feature flags [9, 51] in the upstream or downstream services to dynamically adjust the behavior of upstream/downstream microservice in run-time without rebuilding binaries. It is important to note that the Gecko Agent only supports executing operations on the cluster level and does not perform any application-level changes.

Migrations: Migrations typically involve the replacement of a microservice from one server to another to improve the resource efficiency in the cluster. However, depending on cluster operators, we make migrations optional because migrations can incur additional latency overheads, increasing the time to reach the desired state. Migrations in Gecko are currently performed in two stages currently, first by restarting the microservice on the target server, reconfiguring the service, and then deleting the old microservice. We employ a similar traffic-draining strategy by rerouting the requests to the new instantiated microservice.

Restarts: Restarting a microservice means that the microservice was impacted as a result of a failure event. We perform restarting in a few steps. We begin by cleaning up stale entries on the ip tables. We then instantiate the container(s) on the desired servers first. Once the containers are running, we reconfigure the service to start serving traffic. Similar to deletion, the Gecko agent is extensible to perform any pre-restart or post-restart subroutines that application developers provide.

D Detailed Workload Settings

In this section, we list down the results for the different resource models we considered and Criticality tagging schemes when running Alibaba DGs on a 100,000 node server on the simulator.

D.1 Criticality Tagging

Due to lack of visibility into applications semantics, we perform a best-effort criticality tagging scheme that closely resembles any practical scenario. Specifically, we start by tagging the services first and the service criticality then propagates naturally to microservices. There are two types of services which we deem are critical: 1. user-facing popular services such as video streaming for Netflix whose going down can hurt the business and 2. Back-end services that are infrequently invoked yet perform several critical tasks to ensure the popular services keep on running without any performance or user-visible degradations. Following this insight, we tag the top 50/90 percentile service in the alibaba-cluster traces as C_1 and randomly tag from the tail a few more services as C_1 . All other services are randomly tagged between C_2 - C_{10} . This way of criticality tagging is what we deem as Service-Level Criticality Tagging. We propose two variants of this tagging P50 and P90. Furthermore, if a microservice

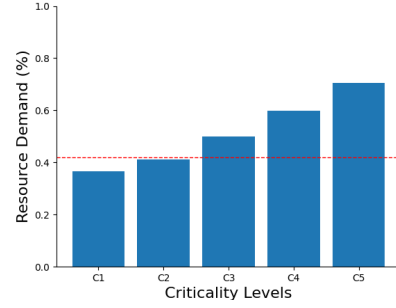


Figure 8: **Breakdown of resources across criticalities for the real-world experiment.**

is invoked in two downstream services, we take the highest service. This way of tagging is conservative, because we end up with a lot of C_1 services and very few microservices are tagged as non- C_1 . To get a better tagging, we leverage the insight that not all microservices are invoked when a service is invoked as shown by Alibaba [48]. Therefore, using LP based methods we obtain the minimal subset of microservices that can help satisfy 50/90 percentile of the requests. Similarly, we also tag some more microservices in the bottom set with high criticality to preserve the notion that backend microservices that are infrequently invoked are critical too. This we dub as Frequency-Based Tagging. Similar to Service-level tagging, we propose two variants of P50 and P90.

D.2 Resource Models

:

We use two resource models in our setting. A calls-per-minute (CPM) based resource model which estimates the resource usage of a microservice as a linear function of CPM. For a more general evaluation, we also use a Long-tailed division scheme of resources as is seen in datacenter workloads. We report the results in these two resource schemes.

In summary, we get 4 (criticality tagging schemes) * 2 resource schemes that we report here. Consistently, in all cases Gecko outperforms other baselines.

E Alibaba Analysis

E.1 Real-World Dependency Graph Analysis

To better understand real-world application usage patterns, we analyze the microservice dependency graphs dataset from Alibaba [48]. The dependencies are characterized using call graphs, where each call graph corresponds to a single user request and is denoted as a directed graph containing all the calls between microservices triggered by that request. The dataset consists of over twenty million call graphs collected over a period of 7 days from Kubernetes clusters at Alibaba.

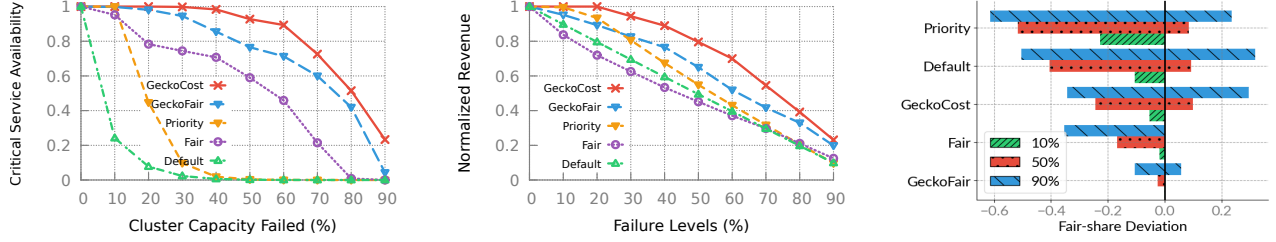


Figure 9: **Gecko evaluated on Alibaba with Service-Level-P50 criticality tagging scheme and CPM based resource assignment scheme on a 100000-node cluster** (a) Aggregate critical service availability across application at different capacity failure scenarios shows that GeckoFair and GeckoCost activates more paths consistently than baselines. (b) Revenue achieved. (c) Fair-share deviation with respect to water-fill fairness.

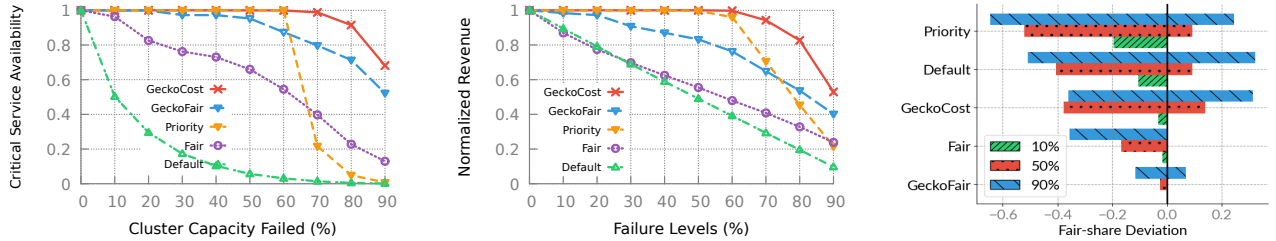


Figure 10: **Gecko evaluated on Alibaba with Frequency-Based-P50 criticality tagging scheme and CPM based resource assignment scheme on a 100000-node cluster** (a) Aggregate critical service availability across application at different capacity failure scenarios shows that GeckoFair and GeckoCost activates more paths consistently than baselines. (b) Revenue achieved. (c) Fair-share deviation with respect to water-fill fairness.

We analyze one-day worth of data from this dataset and derive the corresponding application dependency graphs [37, 49]. Characteristics of the 18 application dependency graphs are shown in Figure 16.

In Figure 16a, we show that a large number of applications have small dependency graphs with dozens of microservices, and they serve a large fraction of the user requests. In Figure 16b, we analyze the call graph size distribution of the top four applications serving the most user requests. We observe that most call graphs (subgraphs of the corresponding application dependency graphs) only contain a small fraction of the microservices in the application DG. For example, in App1 which contains more than 3000 microservices, over 80% of call graphs contain fewer than 10 microservices only. However, different call graphs may include different subsets of microservices. Hence, we conduct another analysis to understand the overlap of microservices *across* call graphs for each application.

Using a Linear Program, we estimate the number of call graphs that can be fully activated with a given fixed number of microservices. For each application, we vary the number of microservices that are allowed to be activated and evaluate the maximum fraction of user requests that can be supported at each size. In Figure 16c, we show that most applications can serve a large fraction of users with a small fraction of microservices activated. In App1 which serves over 1,300,000

requests and contains over 3000 microservices, more than 80% user requests can be served by enabling only 3% microservices (90 microservices in the DG). Similar pattern also holds for smaller applications. For example, App2 with about 50 microservices can serve 90% of its user requests with less than 10 microservices enabled.

The long-tailed distribution of application call graphs implies that most applications can continue to serve a large fraction of user requests even when several microservices in their dependency graphs are turned off. We argue that this property of call graphs can be leveraged to improve application availability during capacity crunch scenarios by turning off non-critical microservices.

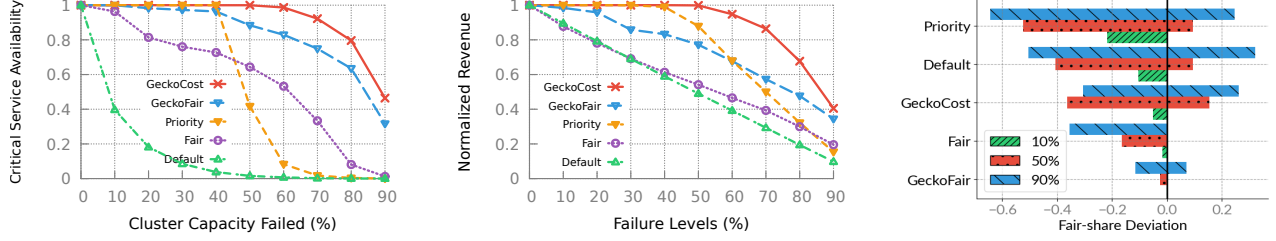


Figure 11: **Gecko evaluated on Alibaba with Frequency-Based-P90 criticality tagging scheme and CPM based resource assignment scheme on a 100000-node cluster** (a) Aggregate critical service availability across application at different capacity failure scenarios shows that GeckoFair and GeckoCost activates more paths consistently than baselines. (b) Revenue achieved. (c) Fair-share deviation with respect to water-fill fairness.

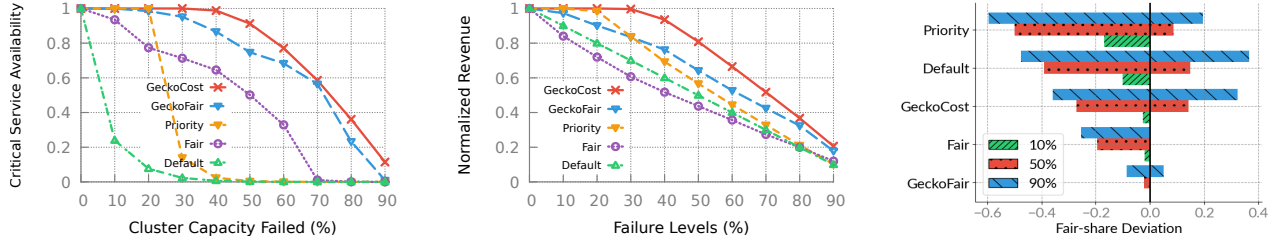


Figure 12: **Gecko evaluated on Alibaba with Service-Level-P50 criticality tagging scheme and LongTailed based resource assignment scheme on a 100000-node cluster** (a) Aggregate critical service availability across application at different capacity failure scenarios shows that GeckoFair and GeckoCost activates more paths consistently than baselines. (b) Revenue achieved. (c) Fair-share deviation with respect to water-fill fairness.

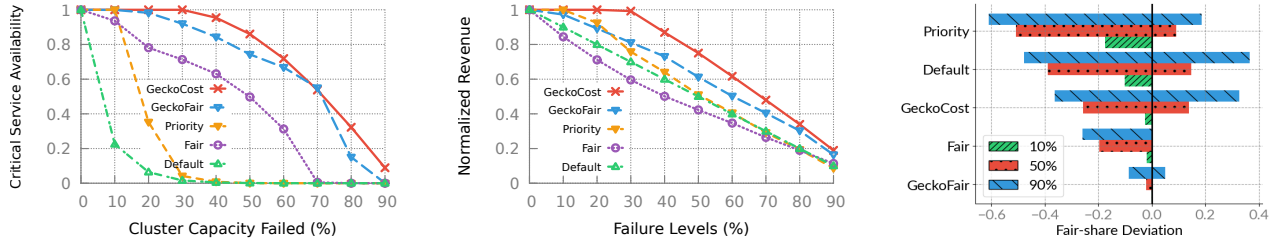


Figure 13: **Gecko evaluated on Alibaba with Service-Level-P90 criticality tagging scheme and LongTailed based resource assignment scheme on a 100000-node cluster** (a) Aggregate critical service availability across application at different capacity failure scenarios shows that GeckoFair and GeckoCost activates more paths consistently than baselines. (b) Revenue achieved. (c) Fair-share deviation with respect to water-fill fairness.

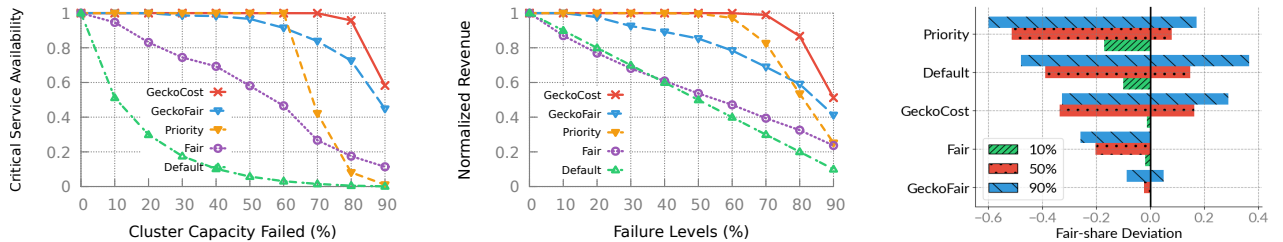


Figure 14: **Gecko evaluated on Alibaba with Freq-Based-P50 criticality tagging scheme and LongTailed based resource assignment scheme on a 100000-node cluster** (a) Aggregate critical service availability across application at different capacity failure scenarios shows that GeckoFair and GeckoCost activates more paths consistently than baselines. (b) Revenue achieved. (c) Fair-share deviation with respect to water-fill fairness.

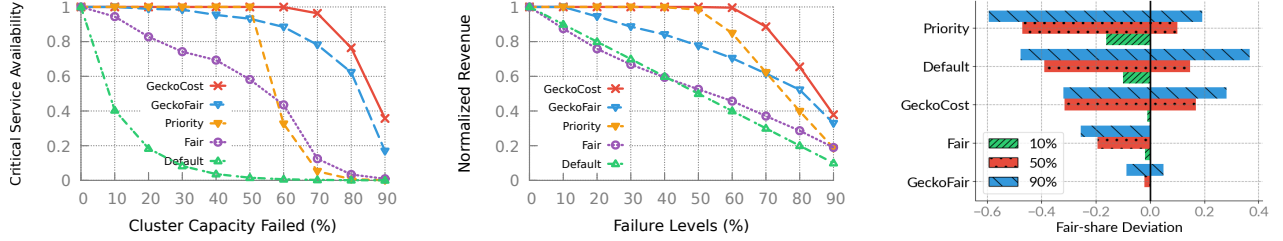


Figure 15: **Gecko evaluated on Alibaba with Freq-Based-P90 criticality tagging scheme and LongTailed based resource assignment scheme on a 100000-node cluster** (a) Aggregate critical service availability across application at different capacity failure scenarios shows that GeckoFair and GeckoCost activates more paths consistently than baselines. (b) Revenue achieved. (c) Fair-share deviation with respect to water-fill fairness.

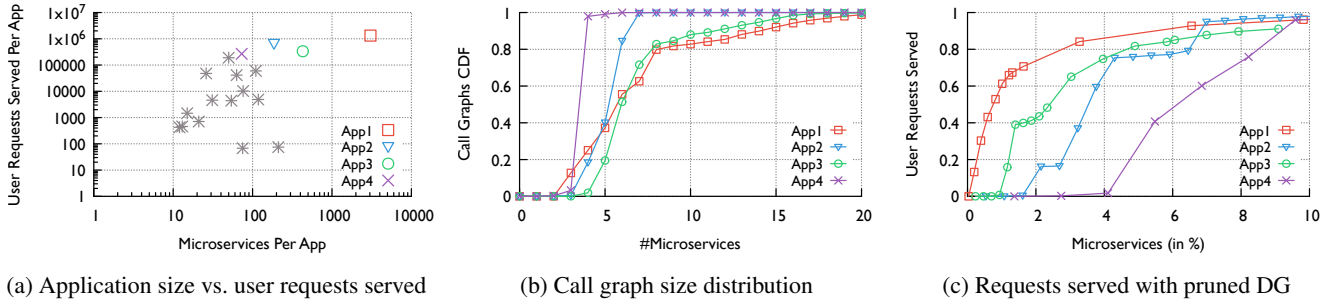


Figure 16: **Analysis of Alibaba workloads:** We analyze a real-world Alibaba dataset [48] over a period of one day and analyze the application characteristics. (a) Analysis of the dependency graph size vs the number of user requests served by 18 applications in the dataset shows that a large number of applications have small DGs. The largest four applications are highlighted using unique markers. (b) Call graph size distribution of the top four applications shows that most call graphs are small. (c) The distribution of number of user requests that can be served within a fixed number of microservices enabled in a pruned dependency graph shows that a large fraction of user requests can be served by enabling a small fraction of microservices for all applications.