Cooperative Graceful Degradation In Containerized Clouds

Kapil Agrawal
University of California, Irvine
Irvine, USA
kapila1@uci.edu

Sangeetha Abdu Jyothi
University of California, Irvine & VMware Research
Irvine, USA
sangeetha.aj@uci.edu

Abstract

Cloud resilience is crucial for cloud operators and the myriad of applications that rely on the cloud. Today, we lack a mechanism that enables cloud operators to perform graceful degradation of applications while satisfying the application's availability requirements. In this paper, we put forward a vision for automated cloud resilience management with cooperative graceful degradation between applications and cloud operators. First, we investigate techniques for graceful degradation and identify an opportunity for cooperative graceful degradation in public clouds. Second, leveraging criticality tags on containers, we propose diagonal scaling—turning off non-critical containers during capacity crunch scenarios—to maximize the availability of critical services. Third, we design Phoenix, an automated cloud resilience management system that maximizes critical service availability of applications while also considering operator objectives, thereby improving the overall resilience of the infrastructure during failures. We experimentally show that the Phoenix controller running atop Kubernetes can improve critical service availability by up to 2× during large-scale failures. Phoenix can handle failures in a cluster of 100,000 nodes within 10 seconds. We also develop AdaptLab, an open-source resilience benchmarking framework that can emulate realistic cloud environments with real-world application dependency graphs.

CCS Concepts: • **Computer systems organization** → **Reliability**; **Cloud computing**.

Keywords: Cloud resilience; graceful degradation; Service Level Objectives (SLOs)

ACM Reference Format:
Kapil Agrawal and Sangeetha Abdu Jyothi. 2025. Cooperative Graceful Degradation In Containerized Clouds. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3669940.3707244

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707244

1 Introduction

Public and private clouds host a myriad of applications from diverse domains. The resilience of the cloud infrastructure is crucial for maintaining the business continuity of these applications. However, as cloud infrastructure expands, the number of infrastructure incidents has also been rising significantly [1–3], with nearly 40% of production incidents caused by infrastructure failures [4]. The recovery time of such infrastructure incidents is typically long, as it involves work by on-site personnel [3], leading to massive user-visible outages and revenue loss [2, 5–7].

Cloud operators and applications employ a variety of solutions to improve availability during failures. The solutions on the *operator front* can be broadly classified into three categories based on when they are employed—proactive, mitigation, and recovery. Proactively, large-scale cloud providers conduct risk assessment [8–10] and disaster-driven capacity planning [11–14], typically adding sufficient redundancy across all components of the infrastructure [1, 15, 16]. Cloud providers also perform disaster audits proactively, using stress tests to identify vulnerabilities in the infrastructure [17–19]. During failures, cloud providers employ a range of mitigation solutions to minimize the impact [1, 3, 20–28]. For example, cloud operators maintain rule-based runbooks with tasks that must be performed during disaster scenarios (migration of data, restarting containers, etc.) [3]. Post-disaster recovery include damage assessment [29], on-site repairs [3], and restoration [30–32].

On the *application front*, the key goal is graceful degradation during failures, also referred to as self-adaptation [33–43]. Several application-level resilience products [44–53] that applications can readily incorporate to contain failures provide out-of-box capabilities, such as circuit-breakers [48] and rate-limiters [47]. Furthermore, open-source chaos testing tools [54–56] can proactively test the efficacy of resilience patterns to detect any undesirable behaviors.

Today, cloud resilience management mainly involves independently operating solutions at the operator and application levels, particularly in public clouds. Operator solutions typically treat applications as blackboxes [21, 23] and rely only on infrastructure-level signals to mitigate the impact of failures. For example, a recent solution [23] identifies VMs that can be potentially throttled by inferring their criticality through analysis of access patterns. While choosing mitigation actions, another system, Narya [21], assumes that a single long VM outage is preferable to applications over multiple short ones. These black-box solutions that are agnostic of application requirements could inadvertently terminate critical components of the application and affect its availability significantly. Thus, the current practice of employing independent resilience solutions at infrastructure and application levels hurts overall availability.

We ask an alternative question: Is cooperative graceful degradation feasible in public clouds while maintaining applications' black-box nature? If applications could share their resilience requirements without divulging business logic, operators could incorporate these requirements into infrastructure-level resilience solutions. Recently, Meta [37] demonstrated the benefits of a cooperative degradation approach in their *private* cloud, leveraging visibility and control over both applications and infrastructure. In Defcon [37], application developers incorporate knobs to expose degradable features controlled by the infrastructure, allowing non-critical features to be turned off during capacity crunch scenarios. However, this approach is not feasible in public clouds, as it requires application modifications.

We argue that cooperative graceful degradation is indeed feasible in public clouds without compromising the black-box nature of applications. We examine the design space for realizing graceful degradation (§ 3) and find that prior solutions have explored the two extremes, i.e., requiring fine-grained application changes [37] or treating the application as a complete black-box [21, 23]. We make the case that exposing the resilience requirements at the *container level* gives operators better visibility without revealing the application's business logic. Thus, the widely adopted container paradigm serves as an ideal abstraction for implementing cooperative graceful degradation. Furthermore, in microservice-based applications, where functionality is decomposed into independently developed and deployed containerized microservices, container-level degradation is a natural fit.

In this paper, we present Phoenix, a cooperative graceful degradation framework for containerized clouds hosting microservice-based applications. Phoenix's key goal is to satisfy application-level resilience requirements maximally while also considering operator objectives and resource availability in capacity-constrained cloud environments following failures. On the application front, Phoenix relies on a simple and expressive abstraction to convey the application's resilience requirements—*Criticality Tags* on containers. On the operator front, the Phoenix automated resilience management system converts application-level criticality tags and operator-level objectives to actionable capacity reallocation decisions for cluster schedulers [57–59].

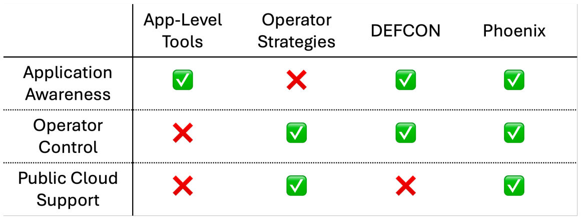
At its core, Phoenix comprises a criticality-aware planner and scheduler. During a failure event, the planner takes as input the container information of applications, along with their criticality tags, to generate a prioritized list of microservices to activate within the available capacity. This process has two steps. First, the planner creates an ordered list of containers at the per-application level based on criticality tags. Second, it ranks containers across applications based on cloud operator objectives, such as fairness or revenue targets. Based on the planner's globally ordered list, the scheduler then generates a sequence of actions (including scheduling, migrating, or shutting down) for running microservices at remaining healthy servers.

Cooperative graceful degradation with Phoenix offers several benefits. First, a fast response is critical during disaster events. A centralized resilience mechanism with access to application-level criticality information can speed up the response time and improve availability. Second, by specifying their acceptable degraded states using various levels of criticality to the cloud—beyond the commonly-used notion of the entire application being "on" or "off"—applications can transform their resilience objectives from a single scalar value 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 intermediate RTO requirements for degraded states. Critical sub-services of an application can have stringent RTO bounds, while non-critical sub-services may allow more flexibility. Finally, at the infrastructure level, several large-scale failures within a data center, which would otherwise require failing over to a backup data center, can be handled in place.

We deploy Phoenix on a Kubernetes cluster running *Overleaf* [60], a real-world microservice-based document editing application, and the Hotel Reservation application from DeathStarBench [61] to demonstrate the feasibility and benefits of cooperative degradation. [1] We also develop AdaptLab, a resilience benchmarking platform, to emulate realistic large cloud environments of sizes up to 100,000 nodes running real-world microservice application dependency graphs obtained using Alibaba traces. Benchmarking results show that Phoenix's cooperative graceful degradation can maximize critical service availability across applications while also satisfying operator objectives, outperforming non-cooperative baselines. Our AdaptLab simulations show that the planning time of Phoenix is under 10 seconds for a 100,000-node cluster. On a real-world 200 CPU Kubernetes cluster, we observe that the end-to-end time taken by Phoenix to provide full recovery for all applications—including deletions, restarts, etc.—is under 4 minutes.

While Phoenix takes the first step towards practical cooperative degradation in large-scale public clouds, the focus in this paper is on stateless workloads, which account for over

[1] Note that Phoenix is agnostic to the underlying cluster scheduler and can support other schedulers [57, 58, 62, 63].



	App-Level Tools	Operator Strategies	DEFCON	Phoenix
Application Awareness	✅	❌	✅	✅
Operator Control	❌	✅	✅	✅
Public Cloud Support	❌	✅	❌	✅

Figure 1. Table comparing features of app-level tools [33, 41–43, 45–49, 71–82], operator strategies [21, 23], DEFCON [37] and Phoenix.

60% of resource utilization in large data centers [1]. Extending cooperative degradation to stateful workloads requires substantial research and is left as future work.

In summary, we make the following contributions:

- We make a case for cooperative graceful degradation in public cloud environments.
- We develop Phoenix, an automated cloud resilience management system that employs resilience-aware planning and scheduling to maximize both application-level resilience and operator objectives during capacity-constrained failure scenarios.
- We develop AdaptLab, a resilience benchmarking platform that can emulate disasters of varying failure rates in realistic cloud environments with real-world microservice workloads.
- Phoenix controller running two microservice applications, Overleaf [60] and HotelReservation from DeathStarBench [61], in a 200 CPU Kubernetes cluster improves critical service availability by up to 2× under large-scale failures and provides full recovery in under 4 minutes.
- Phoenix can generate new plans for clusters with up to 100,000 nodes in under 10 seconds.

2 Background and Related Work

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 planned and unplanned threats that affect the availability of cloud resources. This includes natural threats such as severe weather events [64, 65], including heatwaves [6], hurricanes [3], floods [66], fires [10], and, in the extreme case, solar storms [67]. Human errors [7] and faulty operation of automated systems [4, 68] can also result in large-scale outages in the cloud. In addition to extraneous threats, during its normal operation, the cloud may experience unexpected load spikes, leading to power outages [23, 24, 27], hardware failures [21, 22, 25, 69, 70]. On longer timescales, the process of adding new equipment to the cloud may not keep up with the rate of increase in load [9]. In short, several factors pose a risk to the reliability and availability of cloud services. We refer to a large-scale failure due to any of the above causes as a "disaster".

Cloud Resilience Solutions: Infrastructure-level resilience solutions can be broadly classified into pro-active (before disaster), mitigation (during disaster), and recovery (post-disaster). Pre-disaster resilience solutions include risk assessment [8, 9] to assess failure modes [10], capacity planning [12, 14, 83] by adding sufficient redundancy on components such as power distribution units, cooling units, servers, and network [16, 29, 84]. Several large cloud providers have in-house disaster readiness teams [17, 18] which conduct regular disaster audits [19] by stress testing all layers of the cloud stack. During disaster, mitigation actions include failover [1], enabling new capacity [8], soft reboots [21], fail in-place [22], load-shaping [23, 24], etc. For large-scale failures such as hurricanes, data center draining [3, 85] have been proposed. Finally, post-disaster solutions include on-site repairs [3] and recovery [30, 31], to name a few.

Graceful Degradation: Our focus is on a class of mitigation solutions known as graceful degradation [35], where systems are designed to continue functioning with reduced performance or limited features during capacity-crunch scenarios. This is also referred to as self-adaptation [37, 39]. Graceful degradation may be independently employed by the application or the infrastructure, or through coordination between both, as shown in Figure 1.

Application-level Graceful Degradation: Past work introduced graceful degradation solutions for web servers [41, 42], mail services [73], search engines [77], and storage systems [74–76]. Additionally, there exist solutions broadly applicable across applications [39, 40]. Load shedding, or dropping a fraction of the load, is a common method employed at the application level [43, 78–82]. Brownout solutions [33, 71], on the other hand, allow the dimming of optional features at the application level. Microservice-based applications also leverage out-of-box circuit breakers [86] from commercial tools [45, 48, 49, 72] to degrade non-critical containers. These application-focused solutions perform degradations *obliviously*, i.e., applications respond to capacity crunch scenarios without awareness of the extent of failures in the underlying infrastructure.

Infrastructure-level Graceful Degradation: In the infrastructure-only context, public clouds treat applications as blackboxes and rely solely on infrastructure-level signals for graceful degradation. For example, Kumbhare et al. [23] infer the criticality of VMs by analyzing diurnal behaviors in access patterns and throttle non-user-facing ones. Similarly, other solutions [21, 25] choose potential mitigation actions (such as live migration) based on heuristics that are agnostic to application requirements. In addition, several infrastructure tools and techniques exist for application degradation, such as pod preemption based on pod priority in Kubernetes [87], and resilience guidebooks employed by cloud providers [88–91]. However, these solutions remain mostly unaware of application requirements. Moreover, current solutions do not support coordinated site-wide degradation policies.

Co-operative Graceful Degradation: In cooperative degradation, cloud operators take mitigation actions with application awareness. Cooperative degradation offers faster response times and better resource efficiency since degradations are orchestrated at the cloud level with full visibility into infrastructure failures and application flexibilities. Recently, Meta introduced Defcon [37], a resilience solution that employs cooperative graceful degradation in their first-party cloud, leveraging their visibility into both applications and the infrastructure. Defcon involves modification of Meta's own cloud applications to include *knobs* that annotate program elements eligible for degradation. This allows the cluster manager to disable low-priority features during disaster scenarios. However, this approach requiring application-level modifications are not well-suited for public clouds.

Towards Practical Cooperative Graceful Degradation in Public Clouds: The key goal of graceful degradation is to minimize the impact on application availability. A practical cooperative graceful degradation solution will not only consider application-level impact but will also leverage application awareness at the cloud level to improve response time and efficiency by *prioritizing critical workloads over non-critical ones*. However, we face two key challenges in achieving this goal in public clouds today.

First, we need a standardized interface through which applications can clearly express their resilience requirements. An effective interface for cooperative graceful degradation in public clouds should (a) be general, easy to convey, and straightforward to adopt for a wide range of applications without requiring any application modifications, and (b) provide concise, actionable information for cloud operators. Critical functions may be identified internally or externally to the application. Defcon's approach is internal, with in-application knobs for degradation control. However, this approach is only suited for first-party clouds with complete application visibility and presents significant adoption barriers in public cloud environments. Hence, in public clouds, an interface that identifies critical components externally for black-box applications is essential.

Second, to enable automated resilience management, cloud operators need straightforward mechanisms to reallocate capacity from non-critical to critical workloads dynamically. While Defcon's knobs can help reduce server load, they lack explicit controls for dynamic capacity reallocation and introduce complexities in estimating resource savings. We need an appropriate level of granularity for resilience specifications that will enable seamless capacity reallocation by resilience management systems.

We make the case that containers offer an ideal abstraction to enable cooperative degradation in public clouds. In widely adopted microservice-based architectures, application functionality is decomposed into microservices, each deployed as an isolated container. These microservices, developed and deployed independently, are naturally suited to container-level degradation. Containerized degradation offers several advantages. First, it enables criticality specification at the container level without modifying or exposing application logic. Second, current container-based frameworks already support tagging at the container level; we can leverage this existing capability for resilience specification, thereby facilitating easy adoption. Finally, the resource savings from turning off containers are straightforward to estimate from container specifications, giving resilience management systems clear visibility into potential capacity reallocation gains. Building on these insights, we put forward the notion of diagonal scaling as a method to achieve effective, cooperative degradation in public clouds.

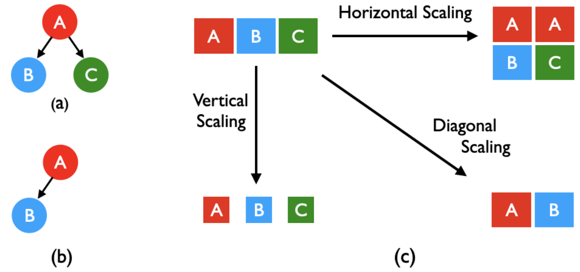


Figure 2. Diagonal Scaling. (a) Original application dependency graph (DG) with 3 microservices. (b) Diagonally scaled DG with one microservice removed. (c) Comparison of horizontal, vertical, and diagonal scaling techniques.

3 Diagonal Scaling

We put forward *diagonal scaling*, a graceful degradation technique at the container level that involves pruning applications by turning off microservices. This can help in improving the overall availability of "critical" services in the infrastructure during capacity crunch scenarios. 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 employed alongside other scaling techniques. Figure 2 depicts a comparison of these scaling schemes.

To enable graceful degradation with diagonal scaling, we first need a mechanism for applications to indicate their preferences to the operator.

Criticality Tags: We introduce *Criticality tags* as a simple yet expressive mechanism to indicate the importance or criticality of the microservice in an application. They capture criticality levels (C_1, C_2, C_3, etc.), with a lower number representing higher importance. We tag a container 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 when it is "good-to-have." For example, a document-editing application such as Overleaf may indicate chat as having lower criticality. By specifying a lower criticality tag, the application is agreeing that in case of a disaster, these microservices may be safely turned off. We can leverage existing tagging mechanisms in cluster management frameworks [57–59, 62] to indicate criticality tags.

3.1 Implications of Diagonal Scaling

Diagonal scaling expands the resilience metrics space, thereby enhancing the granularity of resilience management. Resilience is typically measured using metrics such as the Recovery Time Objective (RTO), which specifies the maximum duration an application can tolerate being unavailable. Traditionally, resilience metrics are defined under the assumption that an application is either fully available or unavailable.

With diagonal scaling, an application can operate at multiple levels of availability. In addition to the fully "on" state, where all components are active, an application can serve a large number of user requests at multiple valid intermediate states, with only a carefully selected subset of components active. Thus, with diagonal scaling, RTO can be defined for various 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 provides operators with greater flexibility to maintain the availability of critical subservices during failures.

3.2 Practical adoption

We demonstrate diagonal scaling in a real-world application, Overleaf [60]. We also discuss challenges with practical adoption and potential opportunities.

Real-World Application Demonstration: Overleaf [60] is a shared Latex editing environment. It comprises 14 microservices, such as spell-check and clsi. Users typically log in, open a project, and then perform edits and compiles on their Latex documents. Edits, which require low latencies, are implemented as web socket connections, whereas most other services are implemented as REST calls. Due to the decoupling of features in Overleaf, we observe that the application functions smoothly even when some non-critical microservices, such as chat and project tagging, are turned off. We first define a key business metric in Overleaf: edits made per second. We then evaluate the impact of turning microservices off on this key metric. We tag microservices that contribute highly to the metric as C_1 and microservices with low impact as C_5. We demonstrate that Overleaf can work seamlessly when C_5 microservices are turned off (§6).

Practical Challenges and Opportunities

Rule-Based Criticality Tagging: We identify practical opportunities for rule-based criticality tagging.
(i) *Microservices serving a single upstream caller*: We analyze the microservice dataset from Alibaba [3]. We derive dependency graphs of 18 applications from this dataset, consisting of over 20 million call graphs following the dependency mining methodology [4]. We find that 74% of microservices in the top 4 applications, and 82% across all 18 applications are invoked by a single upstream microservice. These "single-upstream" stub microservices can be safely degraded if marked as low criticality by the upstream caller without causing any cascading failures. A similar analysis on Meta's microservice deployment fleet showed that 60% of microservices are ML-inference based, serving a single upstream microservice [92]. We posit that these stub microservices are suitable candidates for criticality tagging.
(ii) Frequency-based criticality tagging: Among 18 applications in Alibaba traces, we find that most requests are served by four applications. Using an LP, we determine for each application the minimal set of microservices required to serve the maximal number of requests (Appendix G in our technical report [93]). We find that in the most popular application, which serves over 1.3 million requests and contains 3000 microservices, more than 80% of requests can be served by enabling only 3% microservices (90 microservices). Other applications also exhibit similar behavior, indicating a large skew. We argue that a frequency-based classification of microservices can be useful for applications to identify which microservices are more critical than other microservices.

Automated Criticality Tagging and Testing: While manual tagging is feasible in small applications, it can be tedious for large ones built by several teams. We envision application developers leveraging their system logs to infer criticalities using learning-based schemes. While data-driven insights are helpful, application developers may need to run additional tests to assess the effectiveness of their criticality tagging. Furthermore, application developers may need to override and tag known high-criticality low-frequency microservices manually. To assess criticality tagging effectiveness, we also build a chaos-testing framework (§5).

Support for Flexible Adoption of Tagging: Not all applications may be amenable to container-level degradation. For example, when a single microservice contains both critical and non-critical functionalities. Therefore, our system design does not require all applications or all microservices within an application to be assigned criticality tags. In such cases, all untagged microservices will be deemed to be of the highest criticality level.

Our goal is to leverage the benefits of cooperative degradation on the fraction of existing applications which can diagonally scale. Nonetheless, the advent of modern cloud development architectures such as Service Weaver [94] shows promise in that they propose a container runtime framework that will offload how application binaries are packed and shipped to the container runtime. In such cases, developers can specify the criticality on the code-interface level which can then be leveraged by the container-runtime policy to separate critical and non-critical containers for improving the resilience of the application.

4 System Design

We present the design of Phoenix that performs application-aware resilience management at data center scale. Phoenix's key goal is to satisfy the resilience objectives of applications and operators maximally in the event of large-scale failures. Phoenix takes into account the following considerations:

(R1) Application Requirements: Within an application, microservices should be enabled based on the order of criticality. When dependency graphs are available, microservices are ordered based on both dependencies and criticality.

(R2) Operator Objectives: While criticality tags enable to prioritization of containers *within* an application, the operator objectives determine the resource allocation *across* applications. Phoenix should support a variety of operator objectives (e.g., fairness, revenue maximization).

(R3) Resource Efficiency: Phoenix should efficiently pack microservices within the available cluster capacity.

(R4) Fast Response Time: Phoenix should be responsive to disasters in the order of seconds at the data center scale.

(R5) Broad Deployability: Phoenix should be broadly deployable across applications and data center environments. It should be able to handle clusters with a mix of applications that may/may not be diagonally scalable and may/may not have dependency graphs.

We design the resilience management system as an independent layer that interacts with cluster schedulers. Phoenix constantly tracks the cluster state and, during a failure event, generates a new target state based on operator and application goals, which is conveyed to the cluster scheduler. This decoupling of resilience management from the cluster scheduler enables Phoenix to be easily portable across cluster schedulers. It also allows us to maintain a simpler design at each of the individual layers while also supporting independent evolution at both layers. Finally, this separation will enable future extensions of Phoenix to handle resilience across the cluster scheduler, the network controller, and the storage controller.

We first develop a Linear Program (LP) that adheres to the above design requirements.

Linear Program Formulation: We use the following variables in the LP. The Boolean variable x_{ij} represents whether a microservice j in application i is activated. 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 — lower numbers, C_1, imply higher criticality. 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. The Linear Program (LP) formulation that captures Phoenix requirements can be written as follows:

$$Maximize \sum_i \sum_j F(x_{ij})$$

$$x_{ij} \geq x_{ik} \mid \forall m_j, m_k \in app_i \; with \; C(m_k) > C(m_j) \quad (1)$$

$$\sum_{j \in pred_i(m_k)} x_{ij} \geq x_{ik} \mid \forall app_i, \; \forall m_k \in app_i \quad (2)$$

$$\sum_k y_{ijk} = x_{ij} \mid \forall app_i, \; \forall m_j \in app_i \quad (3)$$

$$\sum_i \sum_j R_{ij} * y_{ijk} < S_k \mid \forall S_k, \; \forall app_i, \; \forall m_j \in app_i \quad (4)$$

Eq. (1) specifies criticality constraints within an application, where nodes are activated in the order of criticality. For example, C_1 microservices are activated before activating C_2. Note that this constraint applies within each application only and not across applications. Eq. (2) imposes topological constraints wherein for each node m_k, at least one predecessor of m_k must be activated. This implies that in the application dependency graph, there is at least one activated path leading to every activated node. This constraint is optional when the service dependency graph for an application is missing. Eq. (3) specifies that a microservice m_j of app_i must be placed in at most one server. Eq. (4) imposes that the total assigned load on a server k, is less than its capacity, S_k.

Global Objectives: Globally, the operator objective, F, determines the resource allocation across applications. The operator has the flexibility to define any monotonically increasing function F as an objective.

We consider two candidates for F: 1) Fairness-based and 2) Revenue-based. In the fairness-based objective, the goal is to distribute R units of resources among n applications such that each application receives a fair share R/n. If an application's demand is larger than R/n, it is allocated at least (but possibly more than) this share. However, if the job's demand is smaller, the excess resources may be allocated to other jobs to improve overall resource usage. We formulate a water-filling [95–97] based fairness objective to implement this. In the revenue-based objective, the LP prioritizes applications with a willingness to pay a higher price per unit of resource. We refer to these two LP formulations as LPFair and LPCost, respectively. The mathematical formulations of these LPs can be found in Appendix C of our technical report [93].

Our LP formulation is designed to be generic and adaptable, allowing the addition of various constraints relevant to cloud operators. Examples of such constraints include limiting the migration of microservices from unaffected nodes or adhering to per-node microservice limits imposed by underlying cluster schedulers. However, as we demonstrate in § 6, LP-based solutions scale poorly to today's cluster

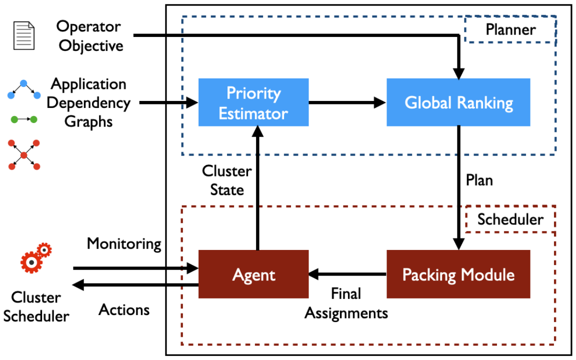


Figure 3. Phoenix System Diagram

sizes. Hence, we use the LP as a guide to design the Phoenix system.

Phoenix: The Phoenix system (Figure 3) has two key components: a Planner and a Scheduler. The planner generates a plan for allocating resources to microservices during a failure scenario. The planner takes as input microservice-level information of active applications in a standardized format, including criticality tags and resource allocation before failure. The planner then generates a subset of microservices to be enabled by considering the aggregate resource availability in the cluster. The scheduler is responsible for generating a mapping from microservices to nodes in the cluster and enforcing this plan in the correct sequence by interfacing with the cluster scheduler.

4.1 Phoenix Planner

The planner consists of two sub-modules (Alg. 1). The *Priority Estimator* generates an ordered list of microservices per application based on application requirements: criticality tags and (optionally) application dependency graphs. The *Global Ranking* submodule generates a globally ordered list of microservices based on the operator objective and application-level ranking generated by the priority estimator.

Priority Estimator: This module determines the relative priority of containers within an application based on their criticality and dependencies. The Priority Estimator (in Algorithm 1) takes as input an ‘app‘ object representing the list of containers and criticality tags and optionally dependency graphs (DGs) when available. Its output is an ordered list, *AppRank*, which denotes the priority order in which microservices need to be activated. Note that this ordering is within each application.

When DGs are unavailable (line 13), containers are ordered based on criticality from highest to lowest (lines 18 and 19). For applications with DGs, we populate a priority queue, Q, with source nodes i.e., entry microservices with no inbound edges (line 14). In lines 14-19, we traverse the graph based on a combination of two factors—criticality (from high to low) and topological ordering (from root to leaves). The pre-order graph traversal ensures that the ordering of nodes is topology-aware such that no microservices are activated without at least one predecessor (satisfying Eq. (2) in the ILP formulation). Using criticality as the key while popping nodes from Q ensures that our orderings are criticality-aware, satisfying Eq. (1) in the ILP formulation. We avoid redundant computation by maintaining a visited set (not shown in the algorithm) to reduce the time complexity for each graph to be similar to a DFS/BFS traversal in $O(V + E)$ time.

Algorithm 1: Criticality-Aware Planning Algorithm

```
1  Function Main:
2      AppRank = PriorityEstimator(apps, tags)
3      GlobalRank = Sort (AppRank,ClusterObj)
4      return GlobalRank
5  def PriorityEstimator(apps, tags):
6      def DFS(node):
7          if node ∈ visited then return
8          AppRank.append((app.id, node.id))
9          foreach child ∈ node.children do
10             if tags(child) ≥ tags(node) then
                   DFS(child)
11             else Q.insert(child)
12     foreach app ∈ apps do
13         if app.G ≠ NULL then
14             Q = InitPriQ(G.src_nodes, key=tags)
15             while len(Q) ≠ 0 do
16                 DFS(Q.pop())
17         else
18             foreach node ∈ sorted(app.nodes) do
19                 AppRank.append((app.id, node.id))
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, msID)
28             Q.insert(AppRank[appID][ms.idx+1])
29     else break
30     return GlobalRank
```

Global Ranking: This module leverages cluster operator objectives to obtain a global ordering of microservices across all applications. As shown in line 21 in Algorithm 1, this module takes as inputs the operator objective, Obj, and the ordered lists of containers per application generated by the priority estimator, *AppRank*. The output is *GlobalRank*, an ordered list of containers across all applications. Obj in Algorithm 1 is an operator-defined Python method that takes as input the list of applications, container resources, and current assignment state, and outputs a score. This scoring function captures the objective by which cloud operators prioritize and rank microservices *across applications*, such as fairness or revenue maximization.

We use a priority queue, Q, to track the containers sorted based on the operator objective. Q is initialized with the first node in every application's priority list in line 22. In each step, the container 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 container is deducted from the available capacity in line 25. The next node in the corresponding application's priority list is added to the priority queue. This process continues until all containers are visited.

Phoenix planner supports a broad range of operator objectives. Following the LP formulation, we implement two operator objectives: *(i) Cost-Based:* Containers that generate higher revenue are prioritized. The key in the cost-based global ranking is the price per unit resource. *(ii) Fairness-Based*: Those containers are allocated resources in every round whose resulting deviation is least from the precomputed fair-share.

4.2 Phoenix Scheduler

The scheduler is responsible for mapping containers to servers based on the ordered list generated by the planner. This formulation is a variation of the well-known bin-packing problem [98] 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 containers to servers based on the ordered list 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 B of technical report [93]. We highlight the key steps here.

When a cluster experiences partial failures, a fraction of containers in the planner list may be already running. The packing module iterates over the ordered list generated by the planner. If the next container to be scheduled is already running, it can continue on the same server. If the container to be activated has failed, it must be rescheduled on an active server. The scheduling heuristic first sorts the active servers in the decreasing order of available capacity. If the resource requirements of the container can be accommodated without migrating any of the other active containers, it is assigned to the server that has the smallest available capacity larger than the required resources, i.e., the best-fit strategy [99].

If the container 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 containers can be migrated based on available server capacity and the size and count of containers currently active on the server. Smaller containers are more likely to be accommodated by other servers. Hence, servers with large available capacity and a large number of small currently active containers are preferred. The heuristic then migrates the containers on the server to other active servers. If the heuristic fails to identify a target server for a container 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 containers 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 Phoenix scheduler on the cluster scheduler. At a high level, the agent performs three tasks: deleting non-critical containers, migrating already running containers, and restarting containers impacted by a failure. We use cluster scheduler API to execute these tasks. Additional subroutines performed with all three tasks include draining traffic, scaling up/down the containers, and reconfiguring iptables, detailed in the implementation section.

5 Implementation

The Phoenix Controller is written in Python and is available as open source [100]. We test Phoenix Controller with Kubernetes cluster scheduler. Note that our design can also work with other cluster schedulers [58, 62, 63] with minor modifications. The Phoenix 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. The Phoenix controller maintains the deployment information and associated criticality tags. It also maintains application dependency graphs as NetworkX DiGraph objects [101]. In the packing module, we employ a tree-based data structure, Python's Sorted Lists [102], to perform insert, search, and delete operations faster than linear time.

Fault Tolerance: Phoenix is a lightweight module that can tolerate unplanned failures. While Phoenix maintains the information of criticality tags and DGs in memory, these inputs are also persisted on a storage service that can be fetched on-demand. When failures occur, leading to a sudden crash for Phoenix, it can simply restart on a healthy node, pull the inputs from a persistent store, and resume operation.

Chaos Testing Service: Since an application development life-cycle consists of regular rollouts, rollbacks, version updates, etc., we also build a managed chaos testing suite in Phoenix to improve developer productivity. This testing service injects failures to verify the correct behavior of an application under the specified criticality tags. It takes as input the application's deployment files (such as YAML, and TOML), an end-to-end load-generator (such as Locust or wrk2), and a utility function. The utility function computes a score on the logs generated by the load generator to measure the quality of the application's outputs. Before pushing deployments to production, this service can conduct tests at different degrees of failure and report the results to developers.

Partial Tagging: Phoenix can operate when only a subset of applications are diagonal scaling compliant. We achieve this by using labels on namespaces, tagging only the subscribed applications as "phoenix=enabled". Furthermore, applications can also partially tag their degradable containers. Phoenix, by default, assumes the criticality to be highest when no criticality tags on deployments are specified.

Stateless Workloads: As noted in § 1, Phoenix currently only supports diagonal scaling on stateless services. Nonetheless, stateless workloads comprise more than 60% of large data center machine usage, as reported by real-world data centers [1]. We expect significant benefits at current levels since Phoenix can provide benefits even when a fraction of applications are not diagonal scaling compliant.

Diagonal Scaling Practical Experience: We now discuss our process of adapting HotelReservation (HR)—a microservice based reservation application from DeathStarBench [61]—to support diagonal scaling. Since Overleaf is diagonal-scaling compliant, we first inspect its code to apply our learning to HR. Our inspection of Overleaf's code showed that it is crash-proof. When a functionality, such as spell-check or chat, is turned off, Overleaf can continue serving requests with reduced functionality, thereby remaining compliant with diagonal scaling requirements. This resilience is achieved through several application-level measures, including error handlers that wrap downstream calls to silently handle failures and allow binaries to execute successfully.

In contrast, HR is primarily designed as a demonstration application for research purposes and, therefore, lacks robust error-handling mechanisms. Although HR, like Overleaf, separates non-critical features into distinct microservices, it is not entirely crash-proof; disabling certain non-essential microservices can still result in user-visible failures. For example, the initialization of the front-end server depends on the availability and connectivity of downstream microservices like search, profile, user, reservation, and recommendation. To enable diagonal scaling, the front end should remain stable even if a low-criticality service, such as recommendation, is turned off. To address this, we implement error-handling logic to prevent request crashes when a downstream microservice, such as the user microservice, is unavailable.

6 Evaluation

We evaluate Phoenix to answer the following questions:

- Can Phoenix's cooperative degradation improve the availability of cloud applications?
- Can Phoenix perform well at scale in clusters with over 100,000 servers and across real-world application dependency graphs with thousands of microservices?
- What are the performance implications of Phoenix at both the cloud operator level (time to mitigate) and at the application level (performance degradation)?

We conduct our experiments across two distinct environments: (1) CloudLab, where we deploy two microservice-based workloads on a cluster with 200 CPU cores; and (2) AdaptLab, our benchmarking platform, to simulate sub-data-center failures in realistic large-scale cloud environments up to 100,000 nodes.

Baselines: We evaluate two variants of Phoenix—Phoenix-Fair (operator objective is max-min fairness) and Phoenix-Cost (operator objective is to maximize revenue) as discussed in §4—with their corresponding LP formulations, LPCost and LPFair. We also compare Phoenix with two non-cooperative degradation baselines. We implement a fairness-based resource redistribution scheme, *Fair*, which does not take into account criticality tags. In the second non-cooperative baseline scheme, *Priority*, applications expose criticality tags, but the operator does not enforce any per-application quotas. The baseline *Default* represents the default behavior of Kubernetes without information on criticality tags or cluster operator objectives.

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

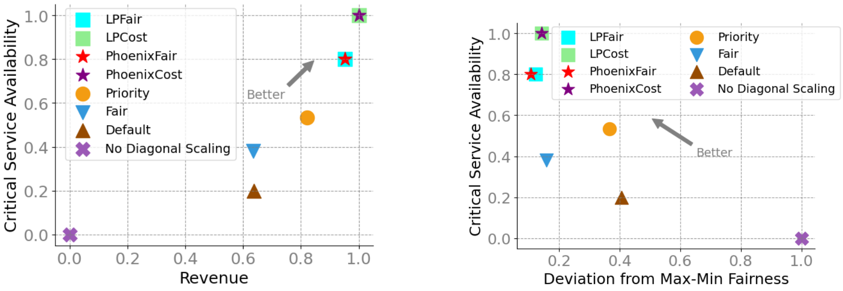
6.1 Phoenix with Kubernetes on CloudLab

We deploy Overleaf [60] and the Hotel Reservation (HR) system from DeathStarBench [61] on a CloudLab cluster with 200 CPUs, using d710 machines equipped with 64-bit Intel Quad Core Xeon E5530 processors and running Ubuntu 22.04 LTS, to simulate a real-world multi-tenant environment. Overleaf inherently supports diagonal scaling (as detailed in § 3), whereas HR requires minimal modifications to achieve diagonal scaling compliance (outlined in § 5).

Experimental Setup: We run five instances across the two applications, as shown in Table 4. For load generation, we use publicly available load generators for Overleaf [103, 104] and

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

Figure 4. Resilience objective of individual applications. For example, we say Overleaf0's resilience goal is satisfied if the served requests-per-second (RPS) of document-edits remains unaffected.



(a) Revenue

(b) Deviation from Fairness

Figure 5. Resilience schemes evaluated on a Kubernetes cluster with cluster capacity reduced to 42%. Critical service availability across microservice applications (with heterogeneous goals based on Table 4) is shown. The x-axis shows the operator objectives.

wrk2 [61] for HR. For each instance, we tweak the parameters so each application's resource distribution across containers is different. For example, different levels of edits, spell-checks, versioning, etc. We determine the resource requirements of containers by running their respective load generators.

Criticality Tagging: Following chaos engineering methodology [105, 106], we first define the steady state for each application, representing the application's "healthy" operational behavior. Any disruption to this steady state indicates a failure to meet the application's critical service goal. For each application, we designate one primary service, listed in Table 4, as the most critical; this service's throughput defines the application's steady state. We label the microservices supporting this critical service as C_1 for each application. All other microservices are assigned lower criticality levels.

Application Metrics: We define an application's *critical service availability goal* as met when the requests per second (RPS) of the critical service are retained after a failure event and unmet otherwise. We assume stateful workloads such as MongoDB [107] are running on a separate stateful cluster, as is standard practice adopted by cloud applications today [108], running stateful and stateless workloads in separate clusters. To show the impact of degrading low-criticality features, we adopt the approach proposed in Fox et. al, 1999 [35] of harvest and yield by assigning a utility (harvest) to each use-case. 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.

Experimental Setup: In all CloudLab experiments described below, we report results by reducing the cluster capacity to 42% (i.e., the breaking point, going below which violates the minimum resource required for maintaining critical service (Appendix F.1 in technical report [93]). The y-axis reports the critical service availability; it takes a value of 1 when all C_1 microservices of the critical service listed in Figure 4 are active and 0 otherwise. The x-axis shows two operator objectives: revenue in (a) and fair share deviation in (b).

Cooperative Degradation allows applications to withstand failures in-place. The × marker (in purple) in Figures 5 (a) and (b) represents no diagonal scaling, i.e., when applications cannot adapt to a resource crunch to resume operations. In contrast, all other schemes have non-zero service availability and continue to operate at reduced capacity.

Phoenix has broad applicability both across applications and cloud environments. Figures 5 (a) and (b) show Phoenix's ability to maximize different operator objectives: revenue and fairness, respectively. Phoenix achieves superior performance under both objectives.

Phoenix performs targeted recovery of critical services. We now report qualitative results by showing a real-world run. In this experiment, we measure the performance of PhoenixCost against the Kubernetes Default mechanism. To emulate actual failure events in the cloud, we stop the Kubelet process [109] on the failed nodes and restart it after 10 minutes. We use the same detection mechanism, outlined in §5 for both schemes. We mark events using time-markers at the top of the plot. Figures 6 (a) and (b) show critical service availability of two runs with Phoenix and Default.

When a failure occurs (t1), Phoenix detects it after 100 seconds and prepares a plan almost instantly (t2). At t2, the Phoenix agent then starts issuing the commands to the Kubernetes cluster to reach the target state (t3). The agent marks the cluster state as recovered when the desired state is reached (t4). The time elapsed between executing action (t3) and completion (t4) can vary depending on the pod deletion and startup times. At 1500s, the nodes come back online (t5). Notice that Default only resumes its operation once all nodes are recovered at 1500s mark. Phoenix's target-driven recovery allows all 5/5 applications to retain critical service availability, whereas Default is able to satisfy only 2/5 (40%).

From the same run in 6 (a), we zoom in on two applications, Overleaf0 and HR1, and report requests per second served

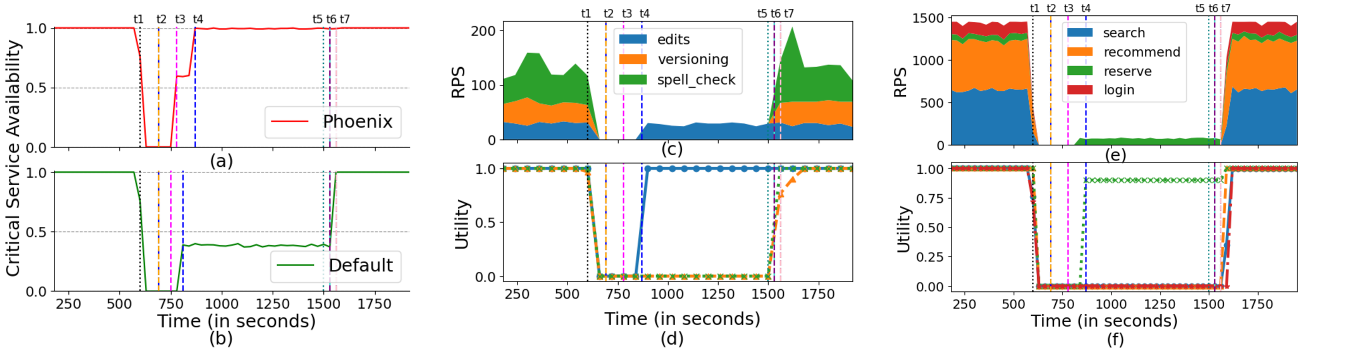


Figure 6. Diagonally scaling a multi-tenant cluster with microservice application instances using Phoenix. **(a)** and **(b)** show the benefits of Phoenix over Default when cluster capacity reduces to 40%. **(c)** and **(e)** demonstrate diagonal scaling on Overleaf and HR, respectively, where critical 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 optional yet "good-to-have" features as end-user utility drops to 0.8 for "reserve".

for each of the services. Figure 6 (c) shows the throughput as requests per second of three request types, namely edits, spell-check, and versioning, plotted as a stacked chart for Overleaf0. The whitespace (between 600s and 900s, t1 - t4) in (c) represents application downtime due to frontend failure. We observe that the throughput of edit requests, the critical service for Overleaf0 (Table 4), recovers in under 4 minutes. 10 minutes later (at the 1500s mark), when failed nodes recover, Phoenix can instantly detect the capacity increase and spawn non-critical services. The sharp rise in the spell_check service immediately following the recovery is due to the pending edits during service downtime.

Figure 6 (d) reports the end-user's utility. Utility falls for versioning and spell-check when nodes fail, while edits maintain high utility. As noted above, we instrument load generator scripts to ascribe a utility score for each successful request. Utility is 0 when a request fails. Without any code modifications in Overleaf, Phoenix degrades non-critical services, such as spell-check and versioning, to ensure fast recovery of Overleaf's critical services.

Pruning call-graphs for partial utility. We provide another example where a service can continue serving requests in a degraded mode by dropping optional yet "good-to-have" features. Figure 6(e) shows how non-critical services are turned off by Phoenix while ensuring the throughput of "reserve" is maintained. Furthermore, in Figure 6(f), we observe that the "reserve" 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 reservations to be made as a guest. When the nodes recover, the "reserve" utility returns to 1. Note that when we partially prune a service, the performance may degrade due to timeouts. However, we do not observe any performance degradation in our P95 latency measurement (Appendix H in technical report [93]).

6.2 AdaptLab Phoenix Evaluation

We develop AdaptLab, a resilience benchmarking platform to emulate failures in large-scale clusters. We evaluate Phoenix and baselines at scale using real-world traces from Alibaba clusters [108] on AdaptLab. The Alibaba dataset contains over twenty million call graphs collected over a seven-day period [108]. Using the methodology from Luo et al., 2022 [110], we derive 18 application dependency graphs of varying sizes (ranging from 10 to 3,000 microservices). Since these traces do not include specific CPU/memory usage data for each microservice or information on criticality assignment, we experiment with two resource allocation models and two criticality assignment models within this environment.

Resource Assignment: We test two realistic resource models to approximate the resource requirements of each microservice: (i) resources as a function of calls-per-minute, proposed by another study from Alibaba on the same dataset [111], and (ii) resources sampled from a long-tailed distribution model as specified in Azure Bin-packing traces [112].

Criticality Tagging: We develop two schemes for criticality tagging in AdaptLab for Alibaba's application graphs: (i) service-level tagging and (ii) frequency-based tagging. Service, in this context, refers to 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 find the top microservices that can serve specified target percentile requests (Appendix G in technical report [93]). We generate both service-level tagging and frequency-based tagging at 50^{th} and 90^{th} 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.

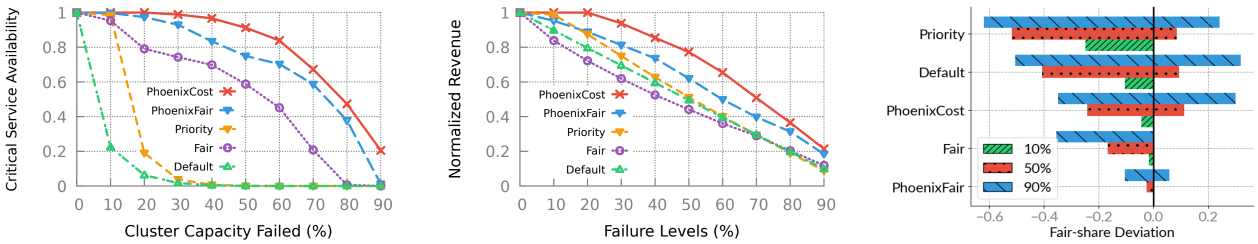


Figure 7. Resilience schemes evaluated on AdaptLab using Alibaba traces, Service-Level-P90 criticality tagging scheme, and Calls-Per-Minute (CPM) based resource assignment scheme in a 100,000-node cluster. (a) Aggregate critical service availability across applications at different capacity failure scenarios shows that PhoenixFair and PhoenixCost activate more critical services consistently. (b) Normalized revenue shows that PhoenixCost maximizes revenue. (c) Deviation from fair-share shows that PhoenixFair has the least deviation.

Application Metrics: We define an application's *critical service availability goals* as met when "all" C_1 microservices are running.

We evaluate Phoenix at scale using the AdaptLab simulator and report the observations, with all results averaged across 5 trials. Here, we report findings from the P90 service-level criticality tagging scheme and the CPM-based resource allocation model. Additional results are available in Appendix F of our technical report [93]. The baselines, LPCost and LPFair, are excluded due to poor scalability (refer Figure 8(b)).

Cooperative degradation improves the overall critical service availability. In Figure 7(a), we report critical service availability (C_1 containers activated) across applications under different failure rates. We normalize the availability with respect to the unaffected cluster state and report the average across all applications at each failure level. We observe that Phoenix's cooperative degradation outperforms the two non-cooperative degradation strategies, Fair and Priority. Priority performs poorly because it lacks an operator-level signal of inter-app prioritization, which results in a few applications with many high-criticality microservices using most of the resources. Fair's fairness-aware resource allocation leads to better availability than Priority, yet it suffers performance deterioration due to a lack of criticality awareness. Phoenix, with intra- and inter-app prioritization, offers high availability. Default (which lacks criticality, dependency, or packing efficiency awareness) performs the worst.

Phoenix maximizes operator-level objectives. Figure 7(b) reports normalized revenue with respect to the state before failure. PhoenixCost offers superior performance due to its efficient packing while explicitly maximizing revenue. The fairness-based schemes perform poorly. Figure 7(c) shows the deviations from fair share across three failure levels of 10%, 50%, and 90%. Ideally, the deviation from fair share must be 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. With varying failure rates, PhoenixFair has the lowest total deviation. Due to the indivisibility of microservices within an application and the inability to activate beyond fair-share, Fair has a high negative deviation. Since PhoenixFair follows a relaxed fair share criterion, it can achieve lower deviation on both sides. Other schemes perform poorly due to the inability to enforce inter-app fairness.

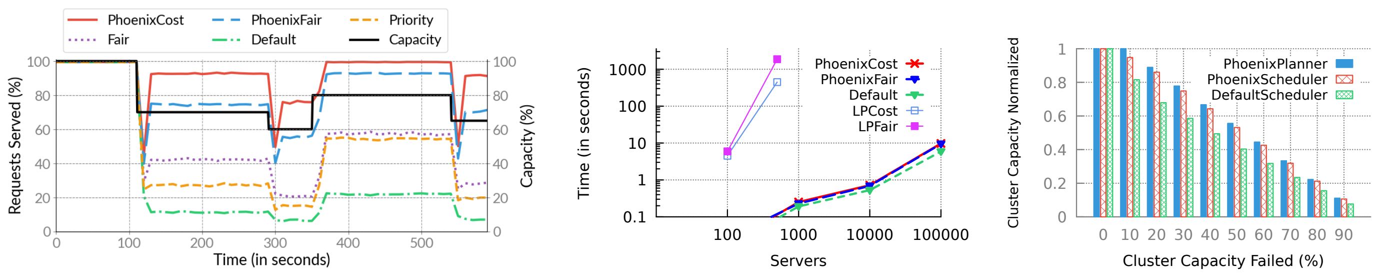
Application can meet their critical RTOs under cloud failures with Phoenix's Cooperative Degradation: Figure 8 (a) reports the requests served (y-axis) vs. time (x-axis) by replaying Alibaba traces in AdaptLab. As capacity varies (shown by the solid black line), we see the benefits of Phoenix's cooperative degradation over its non-cooperative counterparts. Phoenix serves 2× requests in comparison to non-cooperative baselines: Fair and Priority.

Phoenix scales well to real-world cluster sizes. Figure 8 (b) reports the time overheads incurred by Phoenix and baselines on a Linux machine with 24 physical cores and 48 logical processors. LP variants do not scale beyond 1000-server clusters, even with applications with less than 20 microservices. Phoenix's time overheads are comparable to Default, taking less than 10 seconds on 100,000 servers while handing large application sizes up to 3000 microservices.

Phoenix is resource-efficient. Figure 8 (c) shows the cluster capacity utilized under various failure rates by Phoenix planner, Phoenix scheduler (output obtained after planner and scheduler), and the Default scheduler. We observe that the Phoenix scheduler consistently outperforms the default scheduling of Kubernetes, demonstrating superior packing efficiency. Moreover, the drop in cluster utilization from planner to scheduler output is minimal.

In summary, we make the following observations:

- Phoenix's cooperative degradation approach provides high availability and targeted recovery of critical services across applications while maximizing operator objectives compared to non-cooperative approaches (Figures 5 and 7)
- Phoenix's design of diagonal scaling is practical with today's applications, using Overleaf as an example. (Fig. 6)
- Phoenix scales well to real-world data center sizes. (Fig. 7)



(a) (b) (c)

Figure 8. (a) As the cluster capacity varies significantly over a span of 10 minutes, Phoenix can recover quickly and serve nearly 2× user requests compared to baselines. Simulation on a 10,000-node cluster by replaying real-world Alibaba traces. (b) AdaptLab benchmarking on a Linux machine with 24 cores. Phoenix is only slightly slower than Default. The LP does not scale beyond 1000 nodes. (c) **Breakdown of Phoenix performance.** Loss of utilization with only Phoenix planner compared with both planner and scheduler enabled shows that both modules are highly efficient.

7 Discussion and Limitations

Phoenix takes the first steps towards application-agnostic automated resilience management in public clouds. However, several challenges remain to be solved in this setting.

Stateful Workloads: Phoenix is currently restricted to stateless workloads, based on the premise that containers can be safely terminated and restarted to resume serving requests. Expanding these degradation controls to include stateful workloads poses substantial challenges, primarily the need to reliably persist state across container restarts.

Large-scale adoption: To achieve broad adoption, Phoenix needs to be complemented with a modular application design that can be easily decoupled at deployment time into critical and non-critical workloads. Future work should focus on (1) applying learning-based methods to system logs to modularize existing applications by distinguishing critical from non-critical components, and (2) creating container runtime frameworks that delegate packaging and deployment decisions to automated platforms, such as Service Weaver [94].

Other degradation modes: Phoenix's container-level degradation is orthogonal to other degradation modes supported by various applications, such as request-level shedding and Quality of Service (QoS) degradation. Phoenix can be combined with these complementary resilience solutions in the future to achieve better overall efficiency.

Dynamic Criticality Tagging: Phoenix currently employs static criticality tags for containers. Some applications may benefit from dynamic tags that adjust based on contextual factors, such as time of day or user behavior, to guide real-time decisions on microservice degradation. Future work could expand Phoenix by introducing criticality tagging APIs that allow applications to assign criticality tags dynamically.

Dynamic Resource Profiling: Phoenix relies on deployment specifications (such as YAML or TOML) to estimate the capacity freed during degradation. However, degrading user-facing services can influence user behavior, which in turn can change resource demands. This presents an opportunity to extend Phoenix's design with a learning mechanism to adapt to changing resource profiles [113–115].

Adversarial or Incorrect Criticality Tags: In dynamic environments with frequent container additions and updates, developers may employ pre-deployment checks using chaos tests to verify tagging and prevent the deployment of incorrect criticality tags. Furthermore, independent tools that can verify the correctness of criticality tags at the application level and the robustness of operator objectives at the infrastructure level can be devised in the future. Operators can employ policies such as resource fairness to limit the impact of incorrect tags.

8 Conclusion

In this paper, we introduce, diagonal scaling, a cooperative graceful degradation technique that involves turning off non-critical containers to mitigate the impact of large-scale failures in public clouds. We design an automated resilience management system, Phoenix, that leverages diagonal scaling with criticality tags to simultaneously meet application resilience requirements and operator objectives. We lay the foundation for future research in cloud resilience management with our open-source platform. We identify several open challenges in this space: developing automated resilience management in stateful settings, dynamic criticality tagging, learning-based resource saving estimation, etc.

9 Acknowledgements

We sincerely thank our shepherd, Tianyin Xu, and the anonymous reviewers for their insightful feedback. Additionally, we thank Sujata Banerjee, Lalith Suresh, Saurabh Bagchi, and Sharad Agarwal for their early feedback, which helped shape our paper.

References

[1] 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.

[2] 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).

[3] 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.

[4] 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.

[5] AWS outage: What happens when the world's largest cloud service provider goes offline? https://techwireasia.com/06/2023/what-happens-when-the-worlds-largest-cloud-service-provider-goes-offline/. (Accessed on 04/04/2023).

[6] Google's London data center outage during heatwave caused by "simultaneous failure of multiple, redundant cooling systems". https://www.datacenterdynamics.com/en/news/googles-london-data-center-outage-during-heatwave-caused-by-simultaneous-failure-of-multiple-redundant-cooling-systems/. (Accessed on 04/21/2024).

[7] 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).

[8] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based planning of network changes in evolving data centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 414–429, 2019.

[9] Yiting Xia, Ying Zhang, Zhizhen Zhong, Guanqing Yan, Chiunlin Lim, Satyajeet Singh Ahuja, Soshant Bali, Alexander Nikolaidis, Kimia Ghobadi, and Manya Ghobadi. A social network under social distancing: risk-driven backbone management during covid-19 and beyond. In *18th USENIX Symposium on Networked Systems Design and Implementation*, pages 217–231. USENIX Association, 2021.

[10] Managing Failure Modes in Microservice Architectures. https://www.infoq.com/presentations/microservices-failure-modes/. (Accessed on 05/12/2022).

[11] 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.

[12] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Maroo Lieuw, Justin Meza, Scott Michelson, et al. Global capacity management with flux. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 589–606, 2023.

[13] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 258–271, 2021.

[14] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. Ras: continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 505–520, 2021.

[15] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[16] TIA 942. https://tiaonline.org/standards-the-key-to-improving-data-center-resilience-efficiency-and-sustainability/#:~:text=One%20of%20the%20most%20significant,are%20always%20available%20when%20needed. (Accessed on 04/21/2024).

[17] 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).

[18] Chaos Engineering. https://netflixtechblog.com/tagged/chaos-engineering. (Accessed on 04/04/2023).

[19] Verify the resilience of your workloads using Chaos Engineering. https://aws.amazon.com/blogs/architecture/verify-the-resilience-of-your-workloads-using-chaos-engineering/. (Accessed on 04/04/2023).

[20] Failover with AWS. https://docs.aws.amazon.com/whitepapers/latest/web-application-hosting-best-practices/failover-with-aws.html. (Accessed on 04/04/2023).

[21] Sebastien Levy, Randolph Yao, Youjiang Wu, Yingnong Dang, Peng Huang, Zheng Mu, Pu Zhao, Tarun Ramani, Naga Govindaraju, Xukun Li, et al. Predictive and adaptive failure mitigation to avert production cloud {VM} interruptions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1155–1170, 2020.

[22] Jialun Lyu, Marisa You, Celine Irvene, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, Savyasachi Samal, Ioannis Manousakis, Lisa Hsu, et al. Hyrax:{Fail-in-Place} server operation in cloud platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 287–304, 2023.

[23] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, et al. {Prediction-Based} power oversubscription in cloud platforms. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 473–487, 2021.

[24] Shaohong Li, Xi Wang, Faria Kalim, Xiao Zhang, Sangeetha Abdu Jyothi, Karan Grover, Vasileios Kontorinis, Nina Narodytska, Owolabi Legunsen, Sreekumar Kodakara, et al. Thunderbolt:{Throughput-Optimized},{Quality-of-Service-Aware} power capping at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1241–1255, 2020.

[25] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[26] Leonardo Piga, Iyswarya Narayanan, Aditya Sundarrajan, Matt Skach, Qingyuan Deng, Biswadip Maity, Manoj Chakkaravarthy, Alison Huang, Abhishek Dhanotia, and Parth Malani. Expanding datacenter capacity with dvfs boosting: A safe and scalable deployment experience. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 150–165, 2024.

[27] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. *ACM SIGARCH Computer Architecture News*, 44(3):469–480, 2016.

[28] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with {BLASTSHIELD}. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 325–338, 2022.

[29] Luiz André Barroso, U Holzle, and P Ranganathan. The datacenter as a computer. *Morgan Claypool 2013*, 2018.

[30] Quang Tran Minh, Kien Nguyen, Cristian Borcea, and Shigeki Yamada. On-the-fly establishment of multihop wireless access networks for disaster recovery. *IEEE Communications Magazine*, 52(10):60–66, 2014.

[31] Quang Tran Minh, Yoshitaka Shibata, Cristian Borcea, and Shigeki Yamada. On-site configuration of disaster recovery access networks made easy. *Ad Hoc Networks*, 40:46–60, 2016.

[32] Hwaiyu Geng and Masatoshi Kajimoto. Lessons learned from natural disasters and preparedness of data centers. *Data Center Handbook*, pages 659–667, 2015.

[33] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711, 2014.

[34] Maurice P Herlihy and Jeannette M Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.

[35] Armando Fox and Eric A Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178. IEEE, 1999.

[36] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, pages 15–28, 2012.

[37] 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.

[38] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[39] 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.

[40] 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.

[41] Tarek F Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks*, 31(11-16):1563–1577, 1999.

[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] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, 2018.

[44] SpringBooth. https://spring.io/projects/spring-boot. (Accessed on 04/21/2024).

[45] Resilience4j. https://resilience4j.readme.io/docs/getting-started-3. (Accessed on 04/21/2024).

[46] GoBackoff. https://github.com/cenkalti/backoff. (Accessed on 04/21/2024).

[47] GoLimiter. https://github.com/ulule/limiter. (Accessed on 04/21/2024).

[48] Hystrix. https://github.com/Netflix/Hystrix. (Accessed on 04/21/2024).

[49] Simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io. (Accessed on 04/04/2023).

[50] Sentinel. https://github.com/alibaba/Sentinel. (Accessed on 04/21/2024).

[51] gRPC: Identifying Failed Connections. https://grpc.io/blog/grpc-on-http2/#identifying-failed-connections. (Accessed on 11/26/2023).

[52] Wire: Automatic Dependency Injection in Go. https://go.dev/blog/wire. (Accessed on 04/06/2023).

[53] Distributed Systems Safety Research. https://jepsen.io. (Accessed on 04/03/2023).

[54] The Netflix Simian Army. http://techblog.netflix.com/2011/07/netflix-simian-army.html. (Accessed on 04/03/2023).

[55] LitmusChaos: Open Source Chaos Engineering platform. https://litmuschaos.io. (Accessed on 04/04/2023).

[56] Manage reliability to a higher standard with Gremlin. https://www.gremlin.com. (Accessed on 04/04/2023).

[57] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020.

[58] 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.

[59] Production Grade Container Orchestration. https://kubernetes.io. (Accessed on 06/11/2022).

[60] Overleaf: LaTeX, Evolved The easy to use, online, collaborative LaTeX editor. https://www.overleaf.com. (Accessed on 12/06/2023).

[61] 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.

[62] 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.

[63] 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.

[64] 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).

[65] Google cloud service health. https://bit.ly/46WTJLb. (Accessed on 12/02/2023).

[66] Designing for failure: Architecting resilient systems on AWS. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Designing_for_failure_Architecting_resilient_systems_on_AWS_ARC335-R1.pdf. (Accessed on 05/12/2022).

[67] Sangeetha Abdu Jyothi. Solar Superstorms: Planning for an Internet Apocalypse. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 692–704, 2021.

[68] Incident Review – Google Cloud Outage has Widespread Downstream Impact. https://www.catchpoint.com/blog/incident-review-google-cloud-outage. (Accessed on 04/04/2023).

[69] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-Anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In

Proceedings of the ACM symposium on cloud computing, pages 1–14, 2014.

[70] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 1–16, 2016.

[71] 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.

[72] Consul. https://www.consul.io. (Accessed on 10/31/2022).

[73] 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.

[74] Houssem-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.

[75] 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.

[76] 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.

[77] Shuai Ding, Sreenivas Gollapudi, Samuel Ieong, Krishnaram 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.

[78] Google - site reliability engineering. https://sre.google/sre-book/addressing-cascading-failures/#xref_cascading-failure_load-shed-graceful-degredation. (Accessed on 12/05/2023).

[79] 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).

[80] Using load shedding to avoid overload. https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/. (Accessed on 12/05/2023).

[81] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[82] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for {μs-scale}{RPCs} with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314, 2020.

[83] Satyajeet Singh Ahuja, Varun Gupta, Vinayak Dangui, Soshant Bali, Abishek Gopalan, Hao Zhong, Petr Lapukhov, Yiting Xia, and Ying Zhang. Capacity-efficient and uncertainty-resilient backbone network planning with hose. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 547–559, 2021.

[84] Marek Denis, Yuanjun Yao, Ashley Hatch, Qin Zhang, Chiun Lin Lim, Shuqiang Zhang, Kyle Sugrue, Henry Kwok, Mikel Jimenez Fernandez, Petr Lapukhov, et al. Ebb: Reliable and evolvable express backbone network in meta. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 346–359, 2023.

[85] That time we unplugged a data center to test our disaster readiness. https://dropbox.tech/infrastructure/disaster-readiness-test-failover-blackhole-sjc. (Accessed on 04/21/2024).

[86] Istioldie 1.4 / circuit breaking. https://istio.io/v1.4/docs/tasks/traffic-management/circuit-breaking/. (Accessed on 12/05/2023).

[87] Kubernetes: Pod Priority and Preemption. https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/. (Accessed on 04/04/2023).

[88] Disaster recovery planning guidebook . https://cloud.google.com/architecture/dr-scenarios-planning-guide. (Accessed on 10/24/2024).

[89] Patterns for enabling data persistence. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/enabling-patterns.html. (Accessed on 09/11/2022).

[90] Shared Responsibility Model for Resilience . https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/shared-responsibility-model-for-resiliency.html. (Accessed on 10/24/2024).

[91] Composite SLAs . https://learn.microsoft.com/en-us/azure/well-architected/reliability/metrics#understand-service-level-agreements. (Accessed on 10/24/2024).

[92] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[93] Kapil Agrawal and Sangeetha Abdu Jyothi. Cooperative graceful degradation in containerized clouds, 2024. Available at https://arxiv.org/abs/2312.12809.

[94] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 110–117, 2023.

[95] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 1–14, 2015.

[96] Fair Bandwidth Allocation. https://www.comm.utoronto.ca/~jorg/teaching/ece1545/schedslides/bw-allocation.pdf, month = , year = , note = (Accessed on 06/12/2023).

[97] Bozidar Radunovic and Jean-Yves Le Boudec. A unified framework for max-min and min-max fairness with applications. *IEEE/ACM Transactions on networking*, 15(5):1073–1083, 2007.

[98] Best-fit bin packing. https://en.wikipedia.org/wiki/Best-fit_bin_packing. (Accessed on 04/06/2023).

[99] Bin packing problem. https://en.wikipedia.org/wiki/Bin_packing_problem. (Accessed on 04/06/2023).

[100] Phoenix. https://github.com/NetSAIL-UCI/Phoenix. (Accessed on 10/29/2024).

[101] NetworkX. https://networkx.org. (Accessed on 12/06/2023).

[102] Sorted Containers. https://grantjenks.com/docs/sortedcontainers/introduction.html#sorted-list. (Accessed on 09/25/2023).

[103] 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.

[104] 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.

[105] Principles of Chaos Engineering. https://principlesofchaos.org. (Accessed on 11/13/2023).

[106] SPS: the Pulse of Netflix Streaming. https://netflixtechblog.com/sps-the-pulse-of-netflix-streaming-ae4db0e05f8a. (Accessed on 11/13/2023).

[107] MongoDB. https://www.mongodb.com. (Accessed on 10/31/2022).

[108] 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.

[109] Kubelet. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/. (Accessed on 11/23/2023).

[110] 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.

[111] 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.

[112] Azure Trace for Packing 2020. https://github.com/Azure/AzurePublicDataset/blob/master/AzureTracesForPacking2020.md. (Accessed on 04/04/2023).

[113] Romil Bhardwaj, Kirthevasan 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.

[114] 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.

[115] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel 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.

A Artifact Appendix

A.1 Abstract

We make our code available on GitHub, including the automated resilience management system Phoenix and the resilience benchmarking platform AdaptLab.

Our artifact enables users to deploy Phoenix on a Kubernetes cluster with two applications: Overleaf, a real-world collaborative document editing application, and HotelReservation from DeathStarBench. We include additional scripts for a Cloudlab-specific deployment.

AdaptLab, our benchmarking platform, can emulate realistic public cloud environments of diverse cluster sizes. AdaptLab is a comprehensive testbed for benchmarking various resilience solutions under different failure rates, offering support for key metrics such as critical service availability, operator metrics such as cluster utilization, fairness, and revenue, and systems overheads such as time for adaptation. Our AdaptLab demonstration uses real-world microservice dependency graphs derived from Alibaba cluster traces. Finally, AdaptLab is extensible and can support the development and testing of new degradation policies, providing a foundation to advance cloud resilience research.

A.2 Artifact check-list (meta-information)

- **Program:** Phoenix, our automated resilience management system, and AdaptLab, our resilience benchmarking platform.
- **Compilation:** Compatible with Python 3.10 (or higher), and tested with Python 3.10.
- **Data set:** Alibaba cluster microservices traces 2021.
- **Run-time environment:** Phoenix deployed atop a Kubernetes cluster on CloudLab. AdaptLab is tested on a Linux machine running Ubuntu 20.04.4 LTS with 48 cores.
- **Hardware:** AdaptLab experiments are conducted on a Linux machine with 48 cores running Ubuntu 20.04.4 LTS. Overleaf and HotelReservation experiments are conducted on a CloudLab cluster of 25 nodes with d710 machines.
- **Metrics:** We test Phoenix and baselines on two operator-level objectives: 1) Revenue and 2) Fairness. We introduce critical service availability to measure resiliency scores for each application. We also measure system overheads such as time taken to determine the target cluster state, packing efficiency, etc.
- **Experiments:** Scripts are available in the `plotscripts` folder. Detailed instructions are in the `README.md` file.
- **How much disk space is required (approximately)?:** 15 GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 5-7 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Yes, Apache License version 2.0.
- **Archived (provide DOI)?:** Yes, we have archived our code on Zenodo: https://doi.org/10.5281/zenodo.14483674

A.3 Description

How to access: Phoenix's source code is available on GitHub: https://github.com/NetSAIL-UCI/Phoenix

Hardware dependencies: While there are no special hardware requirements for running AdaptLab, the experiments were conducted on an Intel(R) Xeon(R) Gold 6246 CPU @ 3.30GHz Linux machine running Ubuntu 20.04.4 LTS with 48 cores. Next, we require a CloudLab cluster of 25 d710 machines to test the Phoenix Controller on real-world microservices.

Software dependencies: Gurobi Optimizer and Apache Spark are required. The installation instructions are specified in the GitHub repository.

Datasets: We analyze the Alibaba cluster microservice traces (2021) and extract 18 applications following the methodology cited in Luo et al., 2022 [87]. The dataset can be found here: https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021

A.4 Installation

Please clone the GitHub repository and install Phoenix's dependencies. We have prepared the required Python packages. Please run the script to install the packages as follows: `pip install requirements.txt`

Users should also install the Gurobi optimizer and obtain the Gurobi license following instructions in this link: https://www.gurobi.com/features/academic-named-user-license/.

The Apache Spark package is required for Alibaba trace extraction (however, this step is not critical for reproducing results in this paper). Steps for installing Apache Spark can be found here: https://spark.apache.org/downloads.html.

A.5 Experiment workflow

Phoenix is evaluated in two settings: 1) *Real world*: a 25-node CloudLab Kubernetes cluster running five instances of two microservice applications, and 2) *AdaptLab Simulation*: a 100K-node simulated cluster with Alibaba cluster traces. In the CloudLab setting, we simulate a failure where 58% of the cluster is failed. We then measure Phoenix's performance on its ability to maintain critical service availability and compare it with various baselines.

Next, we simulate a real-world public cloud cluster with 100,000 nodes running real-world microservice application dependency graphs obtained using Alibaba traces. This experiment demonstrates how, at a large scale, Phoenix is able to outperform other baselines, showing Phoenix's efficacy in real-world cloud environments.

A.6 Evaluation and Expected Results

We conduct a comprehensive evaluation of Phoenix across real-world and simulation settings. First, we evaluate Phoenix's ability to maximally satisfy the application's resilience objective using the Critical Service Availability metric. We expect Phoenix's performance to be superior to the baselines. Second, we assess Phoenix's effectiveness in achieving operator objectives using two objectives: cost and fairness. Phoenix should be able to outperform other baselines. Finally, we measure Phoenix's time overhead and expect it to be within seconds for real-world size clusters (100K servers).