Departments of Math and Computer Science

Final Report for
ACRL

Graphing the Kubernetes State Space to Generate Synthetic Data

May 9, 2025

Team Members
Saya Kim-Suzuki
Karina Walker
Jaanvi Chopra
Baltazar Zuniga Ruiz
Maximilian McKnight (Spring Team Lead)
Henry Merrilees (Fall Team Lead)

Advisor
Professor Beth Trushkowsky

Liaison
Dr. David Morrison

Contents

Chapter 1

Introduction

1.1 ACRL

The Applied Computing Research Labs (ACRL), led by Dr. David Morrison, was founded to address scheduling and optimization challenges in distributed systems. Currently, ACRL is focused on addressing the lack of simulation and testing tools for Kubernetes, a platform that automates the management of computing tasks across many servers.

1.2 Problem background

Testing and troubleshooting Kubernetes can be difficult without a way to simulate real-world conditions. To solve this, ACRL developed SimKube, a simulator that creates realistic test environments for Kubernetes. SimKube helps users understand and optimize system behaviors by showing how configurations will perform at scale, reducing the risk of errors in live environments.

Distributed systems comprise many small, isolated, and often highly interdependent parts. Although a distributed architecture offers advantages such as scalability, fault tolerance, and performance, it can be difficult to pinpoint the exact cause of issues such as latency spikes, service degradation, or data inconsistency within the ecosystem—especially while maintaining consistent quality of service. Currently, there are no robust, production-ready solutions for simulating Kubernetes configurations locally, due to the highly complex and distributed nature of the system. As Kubernetes applications grow in size, it becomes impossible to run the entire system on a single machine for testing and debugging. Thus, these applications run on clusters–groups of machines coordinated to work as a single unit. This creates a need for distributed simulation tools like SimKube, which allow developers to reproduce and analyze realistic cluster behavior across multiple nodes without relying on production infrastructure.

1.3 Our Task

ACRL tasked the Harvey Mudd College Clinic team with investigating a graph-based method to generate realistic simulation data for SimKube, ACRL's Kubernetes simulator. Generating high-quality data is important but difficult, as questions such as what constitutes 'realistic' or 'good' data are nontrivial due to a lack of concrete metrics for these characteristics. This data is intended to help users of SimKube simulate variations of their Kubernetes traces at scale in production-like environments, enabling better testing, debugging, and optimization without risking disruption to live systems.

1.4 Overview of Approach

This project was structured as a research effort to explore the potential of graph-based methods to generate realistic simulation data for SimKube. Recognizing the challenging scope of a complete solution, we focused on building a reasonable first attempt at a solution that can inform future work. We first identified the theoretical foundations needed to model Kubernetes states. A Kubernetes state is a specific configuration of a Kubernetes cluster, which includes settings such as the amount of memory and CPU allocated. This information is written as YAML or JSON. We represented these states as a graph by encoding each state as a node, and each state transition as an edge. To capture realistic traits of Kubernetes behavior, we analyzed preexisting Kubernetes data and proposed metrics that reflect the patterns observed in production systems. We then developed code to generate new hypothetical nodes and edges and condense the resulting graph to maintain the most important nodes. Ultimately, we built a workflow that could generate a rudimentary synthetic Kubernetes trace given the data collected by SimKube in any production Kubernetes cluster.

This work serves as the foundation for a broader research initiative focused on how to model the Kubernetes state space. Although we did not completely solve the problem of how to generate realistic synthetic Kubernetes data within the project timeline, our contributions laid the groundwork for an NIST grant proposal submitted by ACRL. We hope that this effort continues beyond our clinic project. Our conclusions highlight potential paths for future implementation, while recognizing that further experimentation and validation will be necessary to integrate these ideas into real-world Kubernetes systems.

Chapter 2

Background Information

This chapter introduces the relevant components of Kubernetes for our project, defines state space graphs and traces, provides a literature review, and covers SimKube's architecture.

2.1 Understanding Kubernetes

To build a strong foundation for our project, we first focused on understanding the core concepts and functionality of Kubernetes. Kubernetes uses a system of clusters, applications, pods, and deployments, among other abstractions that are less relevant to this report.

- A cluster is a group of machines that are bound together to run containerized applications.
- An application is simply any software system that the user wishes to run using the machines in the cluster. In Kubernetes, applications are packaged along with their dependencies into containers, allowing them to run reliably across various computing environments.
- A pod is the smallest deployable unit in Kubernetes and consists of one or more containers that share resources on the same physical machine.
- The Kubernetes state space is the collection of all possible states in which a cluster or individual component can be.

Kubernetes is abstracted into two primary systems. The control plane is the lightweight management system that serves as the front-end interface, listening to commands and deciding what code needs to be run on what computer. The data plane represents the machines that then receive those commands and perform the computationally demanding work.

2.2 State Space Graphs and Traces

We define a Kubernetes state as a valid configuration for a Kubernetes cluster, defining values such as memory, CPU, and replica count (copies of the application requested) for pods within the cluster. Kubernetes stores its current configuration state in YAML or JSON files, so each state can also be considered a configuration file. The Kubernetes state space is the set of all valid Kubernetes configurations. This space is extremely large, so we will only represent subsections. We model the state space as a graph by representing Kubernetes states as nodes, and edges as the corresponding API call or "action" that changes the state. We begin by modeling an input trace, which is a sequence of Kubernetes actions derived from a production cluster. Figure 2.1 shows the sequence of filled circles as a trace, which reflects production behavior. The empty circles are other possible states, and the graph is generated from the trace and all the possible configurations it can take from the trace.

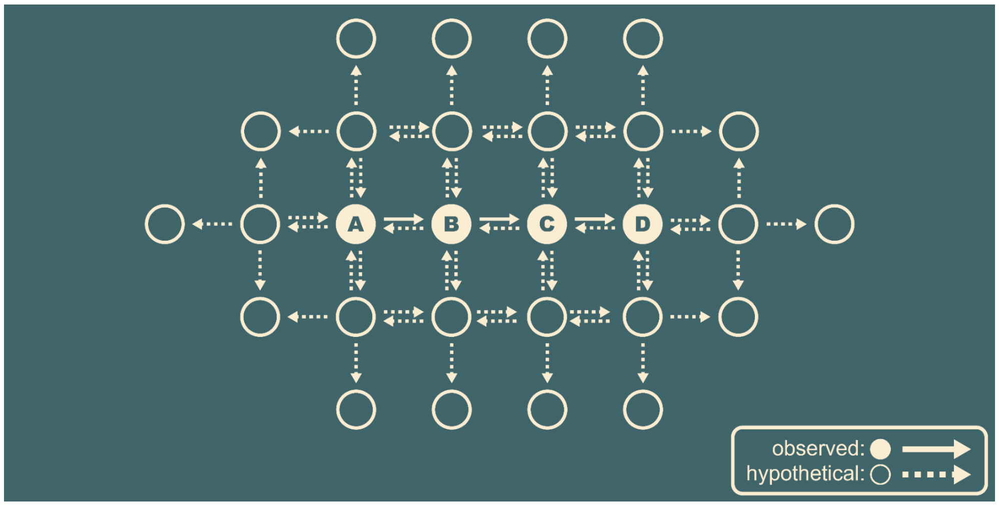


Figure 2.1 Example State Space Graph

Then, from this trace, we can generate a graph by forming alternate histories of actions that theoretically could have been applied to a node to form new nodes. In Figure 2.1, these new nodes are notated as empty circles. And with this new state graph representing only possible actions, any walk on the graph represents a trace that could conceivably be observed in a running Kubernetes system. This abstraction enables us to systematically explore the configuration space of a Kubernetes cluster and evaluate the realism, frequency, or consequences of alternate execution paths.

Because the Kubernetes state space is effectively infinite, we have no hope of modeling it all. The problem can be analogized to trying to understand every combination of foods possible to make a dish for someone, instead it is far more useful to understand what foods are commonly eaten and make something similar to that, but a little bit different so they can try something new but not so adventurous or hard to make that the resulting recipe is unhelpful. We believe synthetic traces that are close to a user's production traces but different enough to represent reasonable alternatives will be the most useful. We call this property representativeness and call a trace that is similar but different, a "representative trace."

2.3 Literature review

Our literature review focused on key papers that provided insights into cluster management and microservices. These papers shaped our understanding of the current challenges and opportunities in Kubernetes simulation. For example, Large-scale Cluster Management at Google with Borg outlined scalability techniques and challenges, offering inspiration for modeling Kubernetes' architecture[1]. We learned that they can modify and reject requests based on policies that a user can set, and that Kubernetes containers are assigned CPU and memory limits. Moreover, in reality, most applications don't consume all of their allocated resources, so total resources allocated to CPU, memory, and storage often exceed the actual physical resources available. The Open-Source Benchmark Suite for Microservices provided an overview of microservices' resource usage and implications for cloud and edge systems, helping to refine our simulation goals[2]. We learned the differences between microservices and monolithic applications, as well as how Kubernetes is most useful in microservice management and is needed

[1]Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[2]Yu Gan et al. *An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems*. Accessed: 2024-09-16. Apr. 2019. URL: https://dl.acm.org/doi/10.1145/3297858.3304013#core-cited-by.

to manage all the small, independent, self-contained software services. We learned that tracing the performance of microservices is difficult due to their complex interdependencies. Importantly, this paper describes DeathStarBench (DSB), several realistic benchmark microservice applications for Kubernetes, which we used to generate our traces. The DSB suite represents several different real-world services and provides a variety of performance challenges. Additionally, Bigger, Longer, and Fewer compared trace data characteristics across organizations, guiding our approach to generating realistic traces for simulation[3]. Reading these papers informed our conclusion that trace data varies widely between companies, and is highly dependent on the size and type of the services they are deploying.

2.4 SimKube Architecture

The four core components of SimKube include a tracer for collecting data on control plane actions, a controller for simulation orchestration, a driver for trace replay, and a CLI tool for user interaction, which together enable realistic Kubernetes control plane behavior simulations on laptops.

SimKube functions as a Kubernetes simulation tool that enables users to evaluate Kubernetes control plane component behavior through safe, isolated testing. Users can record actions (also known as events) from the control plane of their production Kubernetes cluster before SimKube replays them in a simulation environment that operates actual control plane components while mimicking the data plane. This allows simulating decisions the control plane would make for systems containing thousands of computers, with just a fraction of the resources on a single laptop. As a result, SimKube makes it easy for developers to solve problems and test configurations, and analyze Kubernetes component interactions in a fast and cost-effective way.

A trace is a log of relevant events we wish to record. Since the tracing format is not standardized, SimKube has a custom format for storing trace data. Specifically, SimKube watches all configured resources and pods and records any Kubernetes API call, in our graph represented as edges, that would update the configuration. We call these changes trace events. By recording the transition between states instead of the states themselves, we can then reverse-engineer the states when we replay the trace later.

[3]George Amvrosiadis et al. *Bigger, Longer, Fewer: what do cluster jobs look like outside Google?* Accessed: 2024-09-26. Oct. 2017. URL: https://www.pdl.cmu.edu/PDL-FTP/CloudComputing/CMU-PDL-17-104.pdf.

Chapter 3

sk-gen: Our Solution for Synthetic Trace Generation

Our clinic project's mission was to explore methods of synthetic Kubernetes trace generation for use in SimKube. To solve that problem we designed sk-gen, a library and method for synthetic trace generation. sk-gen takes a trace recorded with SimKube and outputs a synthetically generated similar trace. sk-gen has five stages, which will be explained in more detail throughout this chapter.

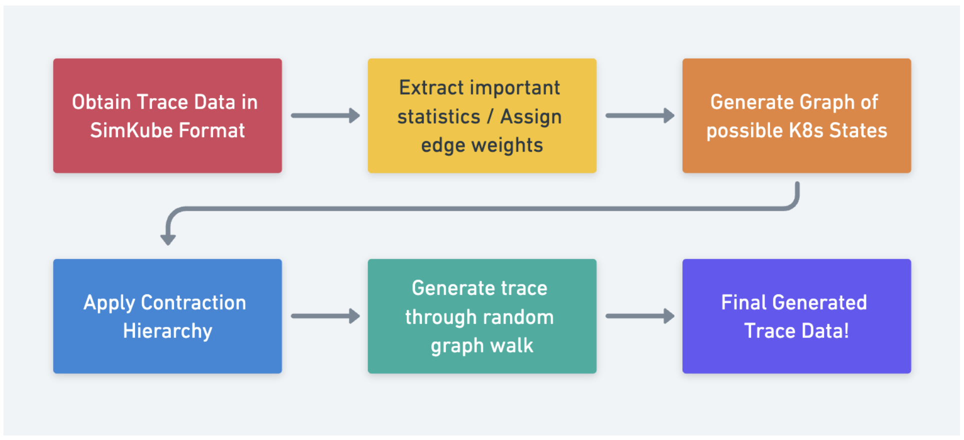

Obtain Trace Data in SimKube Format
Extract important statistics / Assign edge weights
Generate Graph of possible K8s States
Apply Contraction Hierarchy
Generate trace through random graph walk
Final Generated Trace Data!

Figure 3.1 Overview of architecture

The five Stages are:

- Importing a Trace: In this step, we give sk-gen a trace in SimKube format to use as a ground truth to base its synthetic trace on. We model the Kubernetes actions as edges and the resulting states after applying the actions as nodes, creating the beginnings of our graph.
- Extracting Statistics and Assigning Edge Weights: From our ground truth trace, we extract information about the probability of actions. We assign the extracted probabilities as edge weights to the corresponding edges on our graph, making it a Markov chain.
- Graph Expansion: Now that we have turned our original trace into a probabilistic graph we then expand the graph to cover possible valid states that our original trace did not reach. We do this by uniformly applying a series of predefined "atomic" actions, which each change one field (such as memory or CPU allocation) in the configuration.
- Graph Contraction: The expanded form of the graph is unrealistic, as it contains lots of marginal changes to one field of the configuration at a time. We use a technique called contraction hierarchies to combine multiple edges generated by atomic actions into a single, more realistic composite action. This results in a smaller "contracted" graph.
- Trace generation: Now that we have a condensed and more realistic graph of possible states we generate a synthetic trace by taking a random walk along the edges in this graph and recording which actions we take. Because the edges represent valid API changes that could have happened, our extracted trace represents a potential alternative history.

The following sections further describe each of these stages.

3.1 Importing a Trace

sk-gen was designed to use an existing trace as a reference. This is because we don't have a good way of determining what a generally representative trace looks like. We address that lack of data by asking the user to submit a ground truth trace and use that as a reference for what a representative trace might look like. In this import phase, we set up the initial graph by extracting the actions specified in the ground truth trace and applying them to the starting state to get a graph containing only what happened in the trace. From there, we will run the rest of our algorithm.

3.2 Extracting Statistics and Assigning Edge Weights

Next, we analyze the given ground truth trace and collect insights about what synthetic traces might look like so that sk-gen could produce trace data with probabilities. Given a ground truth trace, we next infer the probability of an action happening to increase or decrease a resource (memory, CPU, replica counts, etc.). For example, for memory we do this by aggregating the number of actions that increased memory and comparing that to how many actions decreased memory utilization. Dividing the number of actions of a certain type over the total number of actions in the trace gives an approximation for the probability of an event occurring between two Kubernetes configuration states. The probabilities gathered from submitted ground truth traces are then added to our state space graph as the transition probabilities between edges.

3.3 Graph Expansion

Now that we have a small ground graph representing the ground truth trace alongside probabilities of various resources changing, we can add new, hypothetical nodes. We do this by creating a set of pre-defined actions that change exactly one resource by a predefined amount within the configuration. We then look at each node in our graph and add a new edge and corresponding node for every atomic action that results in a valid configuration that is already represented in the graph. The result is that we now have every node possible to create by applying exactly one of our predefined resource changes to the starting graph. We can repeat this multiple times to get an exponentially larger graph; the number of times this is repeated is called the trace length.

Currently, the next possible actions are hard-coded based on the specific subset of properties outlined above: CPU, memory, and replica counts. We implemented basic guardrails on these actions, such as preventing the replica count from falling below one. To generate new nodes, we used the `jq` package, a tool that allows efficient manipulation of JSON objects, since each Kubernetes configuration can be represented by a JSON object. This approach gave us a concrete method for expanding a single trace into a larger simulated state space. Additionally, we implemented functionality to input multiple starting traces, allowing the graph to expand from multiple initial configurations extracted from the same production cluster.

As the Kubernetes API is vast, we decided to narrow our investigation into how CPU, memory, and replica counts of deployments change throughout traces to create a simpler model that we could visualize concretely. While replica counts change linearly, we decided to model CPU and memory with a scaling factor of two, meaning that deployments can only double or halve with each atomic action, as this mirrors what we feel is reasonable based on reviewing other companies' traces. Critically, this also prevents us from setting the trace length infeasibly high to see a representative range of memory sizes. We incorporated this behavior into our sk-gen code by introducing a resource usage parameter. These values were selected early on as they seemed reasonable and not because they were carefully evaluated to be optimal.

3.4 Graph Contraction

The Kubernetes state space graph is very large, owing to the large number of configuration options that exist. Theoretically, our graph should grow at roughly d^ℓ where d is the degree of nodes defined by the number of potential different transitions we model from a node and ℓ is the trace length. Since we must define one transition for each API option we model, it is important that we carefully model API changes and how we search through the state space to extract information that will be useful to the user. Thus, we need a better idea of which parts of the state space are meaningful and which can be ignored; this maps directly onto how effectively our model can generate realistic and useful trace data. Since we are currently generating our graph in a breadth-first way, the growth of our graph surpasses the maximum practical trace length, which is usually millions of steps long.

3.4.1 Motivation for Contraction Hierarchies

In our initial approach, we generate a state graph by repeatedly applying a set of actions, each modifying a single configuration field by a fixed amount. This method has several limitations:

- The Graph is Impractically Large: The expansion process creates a massive state graph because it represents every possible state change as a separate node and transition. This quickly leads to exponential growth in graph size.

- The Graph is Unrealistic: The transitions in this graph only reflect simple changes where each action:
  - Modifies a single field by a fixed amount or by a fixed multiple.
  - Cannot represent changes that involve multiple fields at once.
  - Excludes any transition where a field changes by a variable amount.
- Composite Transitions are Missing: Real-world systems have complex, composite changes where:
  - A single action can change multiple fields simultaneously.
  - The magnitude of change can vary. Our expanded state graph cannot directly represent these composite transitions. To simulate them, we must decompose each composite change into a sequence of single-field modifications, resulting in intermediate states that do not exist in the real system. To reproduce a composite trace event, we must serialize its changes into a series of primitive trace events.
- Incorrect Simulation Behavior: When these primitive trace events are replayed on SimKube, the underlying control plane will observe and potentially react to these intermediate configuration states, producing an outcome that may diverge from the composite trace event being executed. Hence, this could make the simulation inaccurate.
- Exponential Complexity: Generating this graph has exponential time complexity relative to the number of trace events, making it computationally expensive.

Why Merging Trace Events Is Not Enough: We need a method to merge trace events to produce composite trace events in our output trace. Simply merging repeated trace events of the same field is not enough, because this does not account for cases where multiple fields change together. This would only focus on one field at a time and would create an unrealistic pattern, where each field's value would strictly alternate up and down. In reality, configuration fields often change gradually in one direction, such as increasing resource limits as application loads grow.

Understanding the Need for Centrality: When we generate output traces, we treat the state graph like a Markov process. This means that, aside from what we can infer about the past from the current state, the entire history of how we got there doesn't matter when picking the next state for the trace. When choosing the next step, the only thing that matters is the probabilities of where you can go next from where you are now. Also, the more possible actions a state has, the less you can infer about the past from it. However, this discards how "central" a state is: how often it tends to show up in traces. By considering this centrality, we can safely merge less important states, while still building traces that are realistic according to the original graph structure.

3.4.2 Contraction Hierarchies

To overcome these problems, we use Contraction Hierarchies, a technique inspired by optimizing shortest-path finding in road networks.

A road network can be encoded as a graph, with nodes corresponding to points in physical space at which road segments (edges) intersect. Although both a highway interchange and a driveway entrance may be encoded as a node with, for instance, three edges (one for each attached navigable direction to/from the road-segment intersection), the interchange is generally considered more central than the driveway entrance. The sense of "centrality" most relevant to the purposes of shortest-path optimization is that we expect more (temporal) shortest paths to route through the interchange. By "contracting" the driveway entrance, that is deleting its intersection node and adding "shortcut" edges weighted to preserve all shortest paths between the contracted node's neighbors pairwise, the routing algorithm can search through the main road without the overhead of visiting an intermediate road that is relevant to only a small number of users.

Similarly, we can simplify our state graph by contracting less central states by removing them and replacing them with shortcut edges that maintain the correct transitions, and preserving composite transitions by carefully choosing which states to merge, we can retain the realistic behavior of multiple fields changing together.

To effectively apply Contraction Hierarchies, we must decide the order in which the states are contracted. We chose the Nested Dissection algorithm because it is an efficient method for determining the order in which nodes are removed from the graph. This method is also popular when implementing contraction hierarchies for their traditional use case, road networks. This order, known as the "contraction order", is important because it affects how quickly we can find the shortest paths later. As nodes are removed, the remaining graph is called the "core graph" at each step. The final result is a "Contraction Hierarchy", which is a combination of all core graphs. The process is divided into two phases:

- Optimization Phase: We create the Contraction Hierarchy by choosing the best contraction order using Nested Dissection and building the necessary connections (shortcut edges) to preserve the shortest paths.
- Query Phase: We use the Contraction Hierarchy to look up the shortest paths between two nodes.

3.4.3 Nested Dissection

Nested dissection is a graph partitioning strategy designed to efficiently determine a good contraction order. It works by:

- Finding an approximately minimal "separator" set of nodes. These are nodes that, when removed, split the two roughly equal disconnected halves; that is, they do not contain connections between each other.
- The algorithm then marks the separator nodes to be contracted last, as they are in some sense most "central".
- Then we recursively apply the same process to the two disconnected halves. Because there are no connections between the two halves created after the separator is removed, this means that each can be considered an independent ordering problem, so we can continue to recurse on each of them until we reach a base case[1].

Why Separators are Central: The separator nodes are considered "central" because they lie on many possible paths between nodes in the graph. Contracting last ensures that the less central nodes, which are less critical for path-finding, are removed first. This approach mirrors the idea that the more frequently used paths should be preserved as long as possible.

Finding the Partitioning with METIS: To identify an optimal separator, we need to partition the graph. As the partitioning step is NP-Hard, meaning it is computationally expensive to solve exactly, and we cannot enforce an upper bound on the number of nodes in the graph, we approximate the partitioning step with the METIS algorithm[2]. This algorithm approximates

[1]Julian Dibbelt, Ben Strasser, and Dorothea Wagner. *Customizable Contraction Hierarchies*. arXiv:1402.0402 [cs]. Aug. 2015. DOI: 10.48550/arXiv.1402.0402. URL: http://arxiv.org/abs/1402.0402 (visited on 05/07/2025).

[2]George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. en. In: *SIAM Journal on Scientific Computing* 20.1 (Jan. 1998), pp. 359–392. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/S1064827595287997. URL: http://epubs.siam.org/doi/10.1137/S1064827595287997 (visited on 05/07/2025).

the partitioning by dividing the graph into two halves. The cut-set consisted of the edges that connect the two halves of the partitioned graph, and a vertex cover is the set of nodes such that every edge in the graph is connected to at least one of these nodes. The separator is then chosen as the minimum vertex cover of the cut-set[3], as the smallest possible set of nodes that covers all the edges in the cut-set.

3.4.4 Adapting Contraction Hierarchies to Trace Generation

The Contraction Hierarchies method conventionally improves the shortest-path search performance by adding shortcuts to the original graph while preserving all original nodes. This approach ensures that the shortest paths are maintained in fewer intermediate steps. However, we aim to preserve the most probable paths in a Markov chain while permanently omitting contracted nodes. This requires several key adaptations.

Selective Use of Core Graphs: Rather than using the full Contraction Hierarchy, we use the *i*th core graph, where *i* is tuned to achieve the degree of contraction to reduce the graph to any desired resource footprint. This allows us to control the degree of contraction, balancing the graph size against the retention of important paths.

No Traditional Query Phase: Unlike conventional Contraction Hierarchies, we do not have a query phase for shortest-path lookups. Instead, we use Contraction Hierarchies to retain probable transitions in the Markov process. While we care about retaining the most probable path, we intentionally use less probable paths in our trace extraction process, so we want to retain some of those as well.

Mapping Probabilities to Weights: The standard Contraction Hierarchies method depends on shortest-path algorithms, where edge weights are combined additively. In contrast, the transition probabilities of a Markov process are combined by multiplication. To adapt this, we transform transition probabilities p into edge weights compatible with the shortest-path algorithm using the following transformation of taking the logarithm and inverting to obtain: $-\log p$, which is also commonly known as surprisals, where more probable transitions have lower weights. This preserves paths of maximal rather than minimal probability.

Preserving Probability Consistency: When contracting a node, we confirm, the sum of outgoing transition probabilities from any state is one.

[3]Karypis and Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs".

This is crucial for ensuring that our graph remains a valid Markov process. Specifically, when a node is contracted, we must take care during contraction to redistribute the probability of "orphaned" edges (edges that lose their endpoints due to contraction). This redistribution ensures that the total probability of outgoing transitions from any remaining node is correctly maintained.

3.5 Trace Generation

The sk-gen workflow creates synthetic trace data from real Kubernetes traces generated by SimKube. The final output results from conducting random walks across the contracted graph, which we do through a Markov chain model where, at each node, we decide which edge to take by using the edge weights as probabilities and doing random sampling. This means that more likely transitions occur more frequently, creating realistic behavior patterns. The generated synthetic traces mimic Kubernetes behavior patterns without duplicating the original input data. The generated traces serve testing purposes without needing access to confidential production data. Such purposes could include testing how a Kubernetes deployment will react under different conditions, such as different resources allocated or varying workloads. This is a safer and more flexible way to evaluate system performance.

Chapter 4

Trace Data Collection

The previous chapter introduced sk-gen as our solution to create synthetic Kubernetes traces for SimKube through graph-based modeling. The effectiveness of this approach depends entirely on having realistic data available. The graph expansion and contraction phases of sk-gen use statistical patterns from real traces to assign edge weights, which produce plausible synthetic alternatives. Acquiring production trace data became a major development obstacle throughout our project. Our synthetic trace's quality and realism would significantly improve from better reference data, even though sk-gen can generate plausible outputs from limited inputs. This chapter explains our search for available trace sources, the standardization challenges we faced, and the alternative methods we created to simulate real-world traces using the Death Star Benchmark's social media simulation. Our goal was to verify our synthetic generation approaches while building a stronger probabilistic modeling base for sk-gen.

4.1 Exploration of the Public Trace Data from Alibaba

As described in Section 3.1, sk-gen requires a ground truth trace as a reference point to build its graph model. The quality of this real trace data directly impacts the realism of the synthetic outputs generated. Similarly, Section 3.2 details how sk-gen derives transition probabilities from these ground-truth traces to create a Markov chain model. To support these critical components of our system, we needed to understand how real Kubernetes deployments behave in production environments.

sk-gen models Kubernetes node state transitions as a graph, where each node represents a specific resource configuration (e.g., a combination of memory, CPU, replicas, and GPU), and edges represent actions that increment or decrement these resource dimensions. To assign realistic probabilities to these transitions, as required by our edge weighting process in Section 3.2, we needed to analyze how resource usage typically evolves in production systems.

To understand this, we conducted a detailed analysis of real-world Kubernetes trace data. Specifically, we used publicly available Alibaba traces that have information on node-level resource usage, including CPU and memory consumption over time. Our goal in exploring this dataset was to extract edge weight probabilities between cluster states in our graph that could guide the creation of more realistic synthetic traces and add those probability values to our sk-gen code.

4.1.1 Data Translator

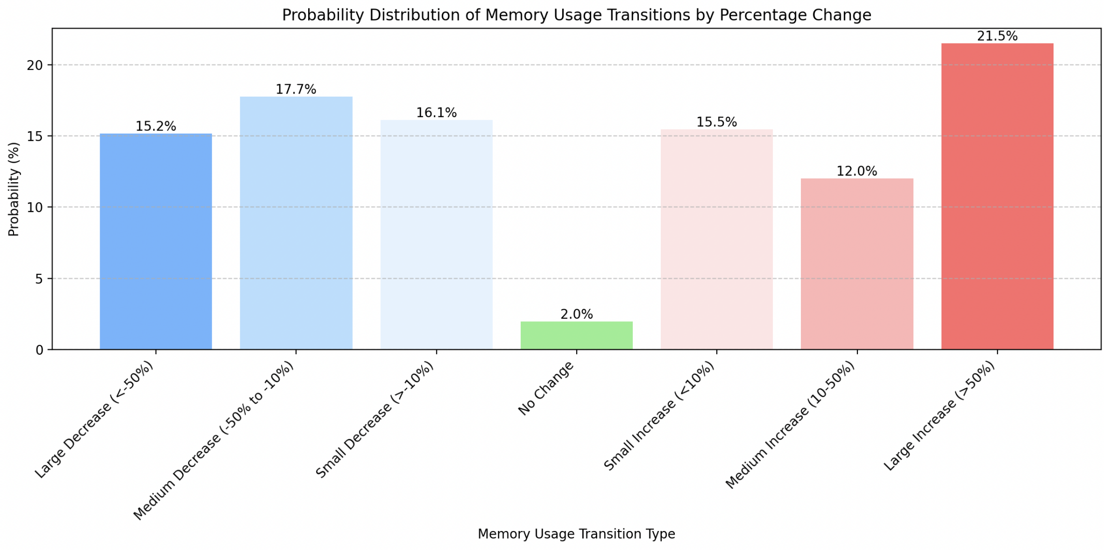
To ensure our Alibaba data worked with the existing SimKube code, we wrote code to translate the Alibaba trace format into the SimKube trace format. Because of the size of the Alibaba data, this first involved transferring it to a DuckDB database, which allowed us to retrieve and query the data fast and efficiently. The conversion script then took the Alibaba data stored in DuckDB and converted it into Kubernetes Deployment objects by translating CPU and memory allocation changes into Kubernetes resource specifications. The script also logged the timestamps to produce properly formatted trace events with timestamps that SimKube could understand.

4.1.2 Data Analysis Findings

We analyzed the Alibaba trace data to identify patterns and distinguishing features of real trace data. Through analyzing the Alibaba trace data, we identified patterns in how CPU and memory usage change between different Kubernetes configurations. For instance, we discovered that while the magnitude of memory changes varied, large memory increases were the most common transition observed. However, aggregating the number of actions that increase or decrease memory, we found the probabilities of memory increasing or decreasing were roughly equal, about 50 percent for each. Figure 4.1 shows the frequency of the various magnitudes of memory changes that were most common from one Kubernetes state to another.

We used these results to populate the transition probabilities in our state space graph as outlined in Section 3.2, enabling us to construct a Markov chain model: a sequence of events where the probability of each event

Figure 4.1 Example Alibaba Data Exploration Finding



depends only on the current state. Because Kubernetes configurations are declarative, the next state is based on the current state, where Kubernetes figures out how to transition from one state to another.

By integrating these probabilities into our sk-gen code base, we enhanced the realism and utility of the synthetic traces for testing and experimentation. While the Alibaba dataset does not represent all Kubernetes deployments, it is an example of how trace data can inform probabilistic modeling of system states. Moreover, the methodology we developed is generalizable: we created a script (attached to our code base) that can extract similar statistical patterns from any trace dataset formatted in the SimKube style, thereby automating the integration of empirical data into synthetic trace generation.

Thus, using the Alibaba trace data provided a practical basis for refining our synthetic trace generation approach. It demonstrated how existing traces can be systematically analyzed and applied to synthetic traces.

4.1.3 Difficulties Sourcing Data

We found that there is no accepted standard of what constitutes a Kubernetes trace. As a result, there is variability in what information a trace may contain about a given cluster across various datasets. The company Alibaba is rare in that it has publicly shared trace data. But even with the Alibaba data, the trace formats are inconsistent. For example, the trace datasets vary across years. In 2017, they published metrics on batch jobs and containers. By contrast, in 2022, they published a trace on pods and microservices, where the metrics they looked at were incomparable. Traces normally contain sensitive information, so organizations are reticent to share, leading to a lack of publicly accessible or comparable datasets for research purposes. Alongside a lack of standards on what a trace is, the performance characteristics and workload traces measure have been shown to vary greatly by the size of the company and their use case[1]. The lack of available, comparable, and diverse trace data presents a barrier to validating our work, as it prevents us from establishing a baseline for what a real trace looks like. A greater volume and diversity of data would make creating something approaching “realistic” trace data easier. Thus, we are using the Alibaba data as an example of real trace data, but it is important to note that it is not representative of all trace data and is just an example. Our project would greatly benefit from access to more trace data.

4.2 Getting Traces from DSB

Having realistic trace data is important for this project because it can give us a sense of what characteristics are common in Kubernetes systems. We can use these insights to tweak our synthetic data generation algorithm. To help acquire more data, we turned to DeathStarBench: a suite of realistic cloud applications designed to model the behavior of microservice-based systems. One of its components is a social media application. This application simulates the internal operations of a large-scale social networking platform by reproducing typical user actions such as logging in, posting content, and liking posts.

When the DeathStarBench social network is deployed on a Kubernetes cluster, SimKube runs alongside it to capture a trace. SimKube does this by monitoring pod activity, service requests, and network communication within the cluster during the execution of the workload.

A benefit of using DeathStarBench is the ability to generate a wide range of realistic traces by varying input parameters such as request rates, workload durations, and types of user interactions. This flexibility allows us to explore how different conditions affect system performance. Because the traces are generated locally and reproducibly, we are not dependent

[1] Amvrosiadis et al., *Bigger, Longer, Fewer: what do cluster jobs look like outside Google?*

on companies making their production traces publicly available. Therefore, DeathStarBench offers a controlled environment that closely mirrors real-world systems, making it an effective and accessible tool for collecting traces.

The traces collected from running DSB give us more examples of real-world trace data. Since trace data is very hard to access and there are not many publicly available sources from companies, generating our trace data is very helpful to understand what traces can look like in the real world and how they vary. We could theoretically compare our synthetically generated trace data from sk-gen and help us identify whether the trace data we generated has similar features, which we discuss further in Section 6.3. Running DSB to generate multiple different traces provides us with numerous examples to compare our synthetic traces.

Chapter 5

Further Work

This chapter details extensions for our project, focusing on how we can improve our methods. We begin by discussing improvements to our DSB data generation process, including expanding parallel trace generation. We then address the challenge of getting SimKube-compatible production data. Then, we outline approaches for evaluating the realism of our synthetic traces. We also propose enhancements to our contraction hierarchy model, exploring metric-dependent heuristics and multi-phase contraction. Finally, we explore how better visualization tools would help make our complex state space graphs more interpretable.

5.1 Improving DSB Data Generation

DSB currently runs a single instance at a time, making it hard to collect a large number of sample DSB traces. Additionally, our Python scripts, made to generate traces from DSB, only run the social media benchmark; they do not run the other benchmarks. A possible next step is to expand the benchmark to automatically generate multiple DSB traces in parallel. It could also be interesting to gather traces across more of the DSB benchmarks, other than just social media.

5.2 Getting SimKube Production Data

As mentioned, our search for data did not result in us finding a SimKube-compatible trace. ACRL asked a few organizations if we could run SimKube and were largely rejected. One next step would be to get production data from a SimKube trace to test our results more thoroughly. ACRL is already in talks to set up SimKube in one major company, and it would be useful to have production traces from more than one company.

5.3 Evaluating How Realistic Our Synthetic Traces Are

To evaluate realism, noting that there is no one-size-fits-all generated trace data. Because different applications of Kubernetes have different needs and specifications, trace data can look very different across organizations. Thus, the first step in evaluating realism is deciding which metrics to focus on, such as CPU utilization, and comparing the statistics of these metrics (the change over time, distribution, range) between real trace data and the generated trace data. We can thus say that the closer these statistics on important metrics match, the more realistic the data. Another, more direct way to evaluate realism is by taking the first part of an existing real trace and making this the input to our algorithm to generate trace data. That way, we can have a more direct comparison between one-to-one traces. Not only that, we can observe at which point the synthetic trace deviates from the real trace and use this as a signal to improve our generation code.

5.4 Extensions to Contraction Hierarchy Modeling

Nested Dissection depends only on the topology of the graph, however, other "metric-dependent" contraction heuristics take into account the weights. The existing code implementation is a generic heuristic, so these could be implemented and easily tested to determine whether sensitivity to probabilities can make for a better contraction. Additionally, saving space by deleting nodes is not meaningful when the maximum resource usage is determined by the input graph. We could experiment with graph synthesis, which would not be possible otherwise, by proceeding with multiple expansion-contraction phases in sequence before trace generation.

5.5 Visualization

While we made an early visualization of the Kubernetes state space, our current code is hard to visualize due to the sheer size of the graphs produced and the information encoded in each node. The representation uses YAML, which is not easy to visualize as a graph. More work could be done to extract only the necessary information from the graphs sk-gen produces to create a visual of all the relationships between nodes in the state space graph.

Chapter 6

Conclusion of Our Findings

This chapter presents a review of our project's achievements and their significance. We begin by evaluating how we met our initial success criteria. Then we discuss the broader contributions of our work to the field of Kubernetes simulation and trace generation, and how this project supported a grant proposal that is directly based on our findings.

6.1 Meeting Our Success Metrics

Our project aimed to explore the viability of using graph-based methods to model a Kubernetes state space, to generate synthetic trace data, and to develop ways to validate the realism of these traces. Throughout the project, we have successfully met each of these goals.

First, we demonstrated that a graph-based model of Kubernetes configurations is viable. We built a state space graph representing transitions between Kubernetes states, integrating probabilistic transitions extracted from real-world data for memory and replica counts. This model allowed us to simulate changes in a system state over time in a structured way, supporting our initial hypothesis that graph-based methods could help capture the complexity of Kubernetes state changes.

Secondly, we developed a synthetic trace generation pipeline (sk-gen) that uses contraction hierarchies to efficiently traverse the Kubernetes state space and produce traces. To inform the realism of these traces, we conducted an in-depth statistical analysis of Alibaba's publicly available traces and extracted empirical transition probabilities. We also built tools to automate this process, making it easier to adapt our method to new datasets in the future.

Overall, we achieved the goals we laid out at the beginning of the project. We provided a practical proof of concept for realistic trace generation and laid the groundwork for future evaluation and refinement as more real-world data becomes available.

6.2 Contributions to This Problem Space as a Whole

Beyond meeting our immediate project goals, the work we did contributes to the broader research effort of developing better simulation and training tools for Kubernetes environments. By demonstrating that Kubernetes configurations can be modeled as a graph and that realistic synthetic traces can be generated through probabilistic transitions, we provide a new approach for understanding and simulating complex distributed systems.

Our exploration of contraction hierarchies as a method for scaling synthetic trace generation offers a promising avenue for making large-scale simulation more feasible. Additionally, the tools we developed for extracting real-world statistical patterns from traces create a foundation for more realistic, data-driven simulation environments. These contributions are significant in a field where access to real-world production traces is extremely limited.

Importantly, the ideas we explored and validated throughout this project are being incorporated into a grant proposal written by our liaison titled "Generating Synthetic Traces for Autonomous Kubernetes Agents via Contraction Hierarchies". Our work throughout this year directly supports the goals of this proposal.

Chapter 7

Acknowledgments

We would like to extend our heartfelt thanks to our liaison, Dr David Morrison, whose unwavering support, guidance, and mentorship made a profound difference throughout this project. From navigating complex challenges to offering thoughtful advice at every step, he was an invaluable part of our research process. We are deeply grateful for his patience, insight, and encouragement—we truly could not have done this without him.

We would also like to extend our thanks to our advisor, Professor Beth Trushkowsky, for her steady guidance and support throughout this project. She played a key role in shaping our meeting dynamics, keeping us on track, and ensuring our work remained focused and purposeful. Her detailed reviews and thoughtful critiques of our methods challenged us to grow and refine our approach at every stage.

Finally, thank you to the clinic program for putting this whole program together.

Appendix A

Appendix

A.1 Scalable DSB Script

One of our goals was to generate synthetic trace data. To help us evaluate whether our generated trace data was legitimate, we wanted to use DSB to generate example trace data from DSB. By deploying a service from DSB multiple times, we were hoping to use it as a benchmark to compare against our synthetically generated traces from sk-gen using contraction hierarchies. We created a script to deploy DSB on AWS so that we could run DSB multiple times to easily generate new trace data. During the creation of this DSB script, we also developed a toolchain to automatically build DSB and a running instance of SimKube into an AMI image using Packer and automatically deploy it in parallel on AWS using Terraform. The script runs a working version of SimKube and DSB. But to be useful for deploying multiple DSB instances, the script would need to be updated to add options for export, non-declarative running of instances, and a way of specifying how long the user would like to run the benchmark.

A.2 Overview of the Codebase

Comprehensive documentation — including detailed function declarations, a description of the code file structure, and setup instructions for running the sk-gen code with the contraction hierarchy implementation—is provided in our code submission.

The sk-gen code uses cargo doc for documentation and provides function declarations, motivations for any design decisions, and in-line comments. The READ-ME contains setup instructions and an overall outline of the project code structure. We also have a repository that contains all the scripts used for analyzing the Alibaba data, translating data into SimKube format, and running DSB.