Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Kubedim: Self-Adaptive Service Degradation of Microservices-Based Systems

Author:
Kelvin Zhang

Supervisor:
Dr Antonio Filieri

Second Marker:
Dr Robert Chatley

June 2021

Abstract

Popular techniques to improve the resilience of applications written in the microservices architecture, such as load balancing and auto-scaling, are limited by the cost of scaling, performance bottlenecks in code, and time delays in responding to high load. Recent work in control theory has taken an orthogonal approach by responsively and uniformly reducing the availability of optional components in a system under load; however, this work has been limited in applicability to industry, both in the ease of configuration and the ability to meet business objectives.

We propose to address this issue with Kubedim, a self-adaptive reverse proxy which mediates access to application components based on system load to improve both system resilience and business objectives. Kubedim is designed to be easily integrated with Kubernetes, a common orchestration tool for the configuration and deployment of cloud applications. We implement two strategies to respond to high load: reducing availability of components non-uniformly with a model-based approach which accounts for side effects such as feature interactions and bottleneck transfers, and profiling users to reduce availability for low priority users using a declarative configuration.

We modified Sock Shop, an e-commerce reference application, to add optional components and improve the reproducibility of experiments. Experimental results indicate as much as a 22% increase in availability during high load compared with the uniform technique from prior work and a significant increase in items checked out from Sock Shop. We find that our approach is highly usable and applicable to developers in industry.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr Antonio Filieri, who has not only supported me throughout my project with his expertise and feedback, but also offered invaluable guidance and encouragement throughout the wider academic year.

I am also grateful for the support from my second marker, Dr Robert Chatley, whose feedback during the interim report and inputs on assessing usability in industry added a valuable perspective to the project.

Finally, I would like to thank my friends and family for their unwavering support throughout the years, particularly in providing the warmth and motivation which have been instrumental in enabling me to complete this project.

Contents

1 Introduction **5**
1.1 Motivation 5
1.2 Objectives and Challenges 6
1.3 Contributions 6

2 Background **8**
2.1 Cloud Architectures and Microservices 8
2.1.1 Microservices 8
2.1.2 Containers 9
2.1.3 Container Orchestration 9
2.1.4 Service Level Objectives and Service Quality Management 10
2.1.5 Circuit-Breaker Pattern and Brownout 11
2.2 Self-Adaptation and Control Theory 11
2.2.1 Self-Adaptive Software and Feedback Loops 11
2.2.2 Controllers 12
2.3 Modelling and Forecasting 13
2.3.1 Random Sampling 13
2.3.2 Time Series Analysis 14
2.3.3 Clustering 15
2.3.4 Dimensionality Reduction 15

3 Related Work **16**
3.1 Brownout and Control Theory in Adaptive Resource Management 16
3.2 Performance Modelling and Parametric Dependencies 17
3.3 Statistical Approaches to Resource Management 18
3.4 Reference Applications for Microservices 19

4 Kubedim **21**
4.1 Brownout Strategies 21
4.1.1 Baseline Dimming 21
4.1.2 Component Weightings 21
4.1.3 User Profiling 22
4.2 Deployment 22
4.3 Configuration 23
4.3.1 Component Weightings 23
4.3.2 User Profiling 24

5 Implementation **25**
5.1 High-level Architecture 25
5.2 Proxying and Brownout with Adaptive Control 26
5.2.1 Monitoring (Response Time Collection) 26

5.2.2 Analysis and Planning (PID Controller) 26
5.2.3 Execution (Reverse Proxy Actuation) 27
5.3 Component Weightings 28
5.3.1 Offline Training 28
5.3.2 Online Training 29
5.4 User Profiling 30
5.4.1 High-Level Implementation 31
5.4.2 Profiling Lifecycle 31

6 Sock Shop for Kubedim 33
6.1 Reproducibility 33
6.1.1 Carts Database 34
6.1.2 Session Storage 34
6.2 Optional Components 34
6.3 Deploying Kubedim 35

7 Evaluation 37
7.1 Kubernetes Setup 37
7.2 Load Testing Setup 38
7.2.1 User Behaviour Models 38
7.2.2 Load Shape 40
7.2.3 Data Collection 40
7.2.4 Deployment 40
7.3 High-level Methodology 41
7.4 Brownout Strategies 41
7.4.1 Baseline Dimming: Behaviour 41
7.4.2 Baseline Dimming: Improvement over Dimming Disabled 43
7.4.3 Component Weightings: Behaviour 44
7.4.4 Component Weightings: Improvement over Baseline Dimming 45
7.4.5 Component Weightings: Online Training Correctness 46
7.4.6 Component Weightings: Online Training Robustness 47
7.4.7 Profiling: Behaviour 48
7.4.8 Profiling: Improvement over Baseline Dimming 50
7.4.9 Profiling: Combining with Component Weightings 51
7.4.10 Comparing Component Weightings and Profiling 52
7.4.11 Conclusion 53
7.5 Developer Usability 54
7.5.1 Component Weightings 55

8 Conclusion and Future Work 58
8.1 Conclusion 58
8.2 Future Work 59
8.3 Ethical Considerations 60

A User Manual 65

B Sock Shop for Kubedim Configuration 66
B.1 Main Application Configuration 66
B.2 Kubedim Configuration 66
B.3 Offline Training Tool Configuration 66

C Saturation Experiments 67

D Brownout Strategy Experiments 69
D.1 Dimming Disabled . . . 69
D.2 Baseline Dimming . . . 69
D.3 Component Weightings Strategy . . . 70
D.4 Profiling Strategy . . . 71

E Developer Usability Experiments 72

Chapter 1

Introduction

1.1 Motivation

Microservices-oriented architecture has become common in industry for its versatility, but scalability is a key challenge to overcome [1]. Both proactive and reactive measures can be taken to ensure a system scales seamlessly to changes in traffic. However, when unexpectedly high loads occur and a system is unable to scale (perhaps due to vertical scalability limits or an under-investment in proactive or reactive measures), systems can fail unpredictably and quickly, with requests timing out for patiently waiting users, leading to a confusing and frustrating user experience.

More graceful service degradation can mitigate the impact of high load on user experience: the circuit breaker pattern [2] is commonly used by web servers and load balancers to reliably throttle requests under high load so that requests which cannot be fulfilled are confidently and quickly rejected with a relevant message displayed to the user. This graceful service degradation is particularly helpful as, over time, the technical environment is always evolving and changing (e.g., request mix, architecture layers, hardware configurations, features and their corresponding API routes being added/removed, configuration files, etc.), and unexpected reductions in system capacity can easily be introduced. Hence, even in a well-designed cloud application, the circuit breaker pattern can be a valuable fallback when all scalability measures have been exhausted.

The circuit breaker pattern is, however, not a perfect service degradation method: being indiscriminate to what type of requests are being throttled, availability is reduced for users regardless of how business-critical their requests are. However, in recent years, the concept of *brownout* has emerged in software engineering and control theory research [3], where an application can be split up into required and optional components, then a controller applies the circuit breaker pattern for a variable proportion of requests to optional components in response to changes in system load. Kotegov and Filieri [4] demonstrate an implementation of this pattern in a Kubernetes environment hosting a shopping cart reference application with an optional recommender system. This more discriminative method of applying the circuit breaker pattern is a step forward towards balancing the benefits of an effective fallback mechanism for systems experiencing high load with maintaining availability for business-critical user flows.

Open questions remain on how to bring brownout to applications in industry. First, both business objectives (service level objectives on user experience, revenue, etc.) and microservice architectures are more complex than use cases demonstrated in prior works. Second, reconciling these complex needs in a controller is an open area of investigation in control

theory [5]. With greater complexity, effective feature interaction analysis will be required: applying brownout changes the request mix of a system, which can shift bottlenecks between microservices and cause other side-effects, hence deciding which features would be most effective to apply brownout to is a challenging task for a more advanced brownout system.

1.2 Objectives and Challenges

In this project, we will create a brownout system which uses a knowledge-driven approach to make informed brownout decisions, meeting business objectives in a complex, real-world microservices environment. The two overall research objectives are to:

- Adapt existing control theory approaches to brownout, augmenting these approaches with previous work on systems modelling, so that optimal brownout decisions can be made at a selective component level.
- Create a workflow, from configuration to deployment, which is deemed usable by DevOps engineers in a business environment. Developers will need to be able to specify their business objectives during the configuration stage.

The first objective requires the brownout system to have knowledge about how system components contribute load to a system and interact with each other, as control theory alone is insufficient to select the most optimal features for brownout decisions to be made. We will investigate building a model which can be easily generated in order to provide valuable decisions to the controller based on monitoring data at runtime. To determine our model, we will need to understand what data will be relevant to decision-making (e.g., correlations in load between microservices, expected user flows, etc.), and the data available to us in the APIs we implement. We discuss possible modelling techniques in Section 3.2 and Section 3.3.

Beyond relevancy of data, designing the decision-making behaviour will be a significant challenge. First, the decision must not incur side-effects in the system: as microservices can depend on each other, applying brownout selectively can lead to bottleneck transfers and system instability if dependencies are not considered. Second, this requirement must be reconciled with business objectives, and conflicts must be resolved. Third, these requirements must be evaluated using a reference application which demonstrates sufficient complexity, which we consider in Section 3.4.

The second objective requires our system to be usable in a real-world environment. We will consider how the system will be configured, how easily the model can be trained, and how easily the brownout system can be integrated into existing systems. To achieve this, developers will need to be able to specify how microservices and endpoints correlate with their business goals, and this specification will need to be deployable on the microservices platform using available monitoring data and APIs. For example, a user ordering an item could be deemed as more important than a user browsing a catalogue, and this specification will need to be representable in our configuration, classifiable at training time and detectable at runtime.

1.3 Contributions

We summarise our contributions as follows:

- We implement a brownout-enabled reverse proxy which can easily be integrated into Kubernetes, a popular microservices orchestration platform, being mindful of our

second objective of developer usability. We introduce this system and describe its configuration and deployment workflows in Chapter 4, before detailing its implementation in Chapter 5.

- In addition to implementing prior brownout work by Kotegov and Filieri [6] as a baseline, we introduce two knowledge-based brownout strategies, introduced in Section 4.1, which meet our first objective. These strategies take into account side effects from feature interactions, as well as the impact of brownout on user experience and business objectives.
- We make significant changes to a shopping cart reference application, Sock Shop [7], to add optional components and to improve the reproducibility of our experiments. This is described in Chapter 6.
- We evaluate our work in Chapter 7 using a reproducible load testing setup, showing as much as a 28% in component availability during high load compared to prior work, as well as a significant increase in the number of items checked out from Sock Shop, meeting our first objective. We also find that our work has a high level of usability for developers due to the ease of configuring selective strategies, meeting our second objective.
- Our work is open sourced[1] and is accompanied by a user manual, shown in Appendix A. Our system's brownout logic is automatically deployed from a declarative configuration file with no control theory knowledge required from the developer. Hence, we reduce the knowledge barrier to the evaluation, adoption and improvement of our work by the wider developer community.

[1] https://github.com/kcz17

Chapter 2

Background

In this chapter, we introduce concepts which are required to understand this work. Sections 2.1 and 2.2 introduce microservices, resource management in the cloud and the concept of brownout in the context of control theory, the basis of our work. Then, Section 2.3 introduces statistical techniques which will be relevant to our work.

2.1 Cloud Architectures and Microservices

2.1.1 Microservices

Monolithic architectures consist of separate services developed within a single codebase and deployed as a single component. Monolithic applications exhibit several limitations at scale [8]: scalability is limited as all services must be scaled even if only one service is under high load; complex workflows are required for developers to work on the same codebase; and the deployed application is typically a single point of failure.

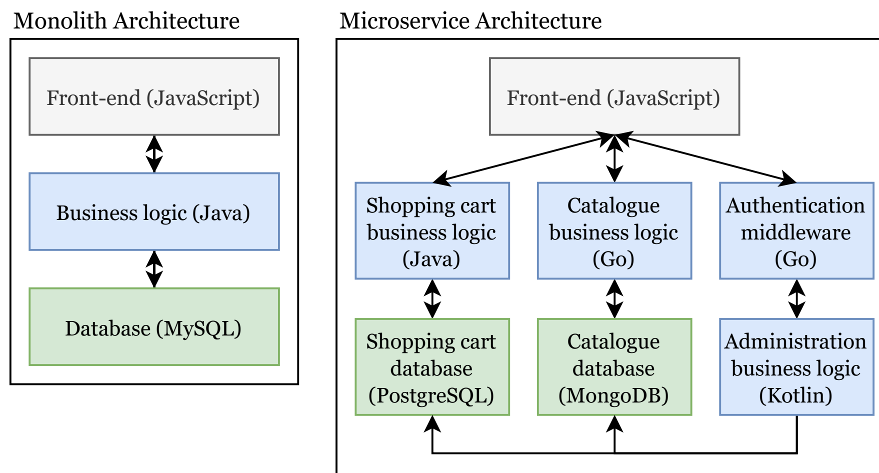


Figure 2.1: Comparison of architectures in an example e-commerce deployment [9].

In contrast, microservices are an architectural approach where services are split up into separate components which are independently deployable, scalable and designed to resist failure, as shown in Figure 2.1 [10]. This architecture is supported by automation and decentralisation of infrastructure so that services can be built with different sub-components and tooling, then automatically deployed and managed.

2.1.2 Containers

Lightweight containers are commonly used to deploy microservices [10]. Pahl et al. [11] describe containers as holding "packaged, self-contained, ready-to-deploy parts of applications, and if necessary, middleware and business logic (in binaries and libraries) to run the applications". Microservices are well-encapsulated by containers: each of the different languages and technologies (and their separate versions) which individual microservices are built with can be deployed as individual containers in isolation. Then, container engines such as Docker handle configuration and deployment of containers, as well as allowing containers to communicate with each other.

Container engines provide the virtualisation layer that containers are deployed onto, so that multiple containers can run on the same physical system at one time [11]. With cloud computing also using virtualisation in order to deploy virtual servers under physical systems, microservices architectures typically leverage the elasticity, availability and other features of cloud computing environments in hosting their containers.

2.1.3 Container Orchestration

In real-world environments, container engines alone are typically insufficient to deploy cloud applications due to the complexity required in managing a deployment spanning dozens of networked containers. Orchestrators coordinate the deployment and management of containers according to developer-specified policies, therefore abstracting away this complexity.

Our work focuses on Kubernetes, an orchestrator which provides features tailored towards cloud applications [12], including: *cluster management*, the management of several instances of containers deployed across a cluster of physical hosts; *service discovery*, keeping track of network addresses of instances; and resilience and scalability features such as efficient clustering and automatic scaling of containers, and load balancing network traffic.

We present below concepts and definitions used in Kubernetes relevant to our work.

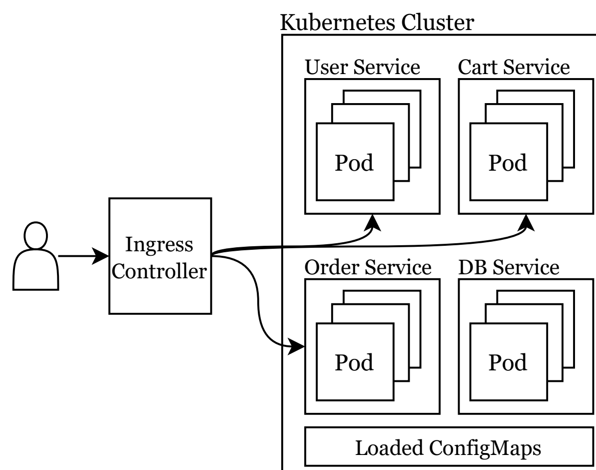


Figure 2.2: Visualisation of a Kubernetes cluster with an ingress controller [13].

Pods. Pods are the smallest unit of deployment, typically wrapping a single container.

Nodes. Nodes are virtual or physical machines which pods are run on. There are typically multiple pods running on the same node, and one or more nodes in a cluster. Kubernetes

is responsible for the scheduling of pods onto nodes based on pod requirements and node capacities.

Deployments. Pods can directly be created. However, a more common pattern is to create a deployment, which allows developers to declaratively specify how they want one or more pods to be deployed, including whether pods should be replicated.

Services. An application running on a set of pods can be exposed to the network. Services abstract replicas of pods by managing a unique DNS hostname assignment so that, while each pod in a service is assigned an internal IP address, external clients only need to work with the specified hostname. The service then decides how client requests to the hostname are routed to specific pods.

Ingress Controllers. An ingress resource defines an entry point between the services on a cluster and external clients, as shown in Figure 2.2. Each ingress resource is managed by an ingress controller, which applies routing rules, typically with load balancing. Ingress controllers are implemented as third party integrations with load balancing and reverse proxy platforms like NGINX Reverse Proxy and HAProxy.

ConfigMaps. A common pattern in software engineering is to avoid hardcoding configuration values in code by reading configuration values from environment variables or by loading a configuration file. Kubernetes provides the ConfigMap object as a method to declaratively set configuration values and specify how they will be made available to a running container (e.g., by mounting a volume containing a YAML file or by setting environment variables).

2.1.4 Service Level Objectives and Service Quality Management

Service level objectives (SLOs) are goals which application providers aim to reach based on quantifiable quality of service (QoS) metrics (e.g., response time) [14]. Cloud applications are often designed to meet SLOs: for example, an online shopping platform may aim to achieve sub-one second page load times at the 99.9th percentile, as higher load times turn away potential customers who are browsing the platform.

Application performance can be degraded due to factors such as sudden outages of physical hosts and high system load, leading to SLO violations. For example, *flash crowds* – surges in traffic due to a sudden increase in users – can quickly lead to bottlenecks in nodes' CPUs or network connections, leading to service unavailability. We introduce below common techniques in cloud environments used to maintain a high quality of service.

Auto-Scaling. Auto-scaling is a load management technique which leverages the elasticity of cloud environments, where replicas of resources can quickly be scaled up and down in order to respond to changes in workloads [15]. Auto-scaling is particularly useful against flash crowds, where resources are provisioned to meet service demand, then scaled back down to prevent the monetary expense of over-provisioning.

Self-Healing. Container orchestrators like Kubernetes can monitor the status of containers in a cluster and perform automated recovery if containers become unavailable. This is particularly useful in cases of sudden system failure: if Kubernetes detects a node failure in a multi-node cluster, pods can be re-created under other nodes.

Load Balancing. Load balancers ensure that load is distributed across replicas so that no single replica is overwhelmed by a majority of requests. Network requests first arrive at a load balancer before being routed and served by a replica. Load balancers can also be configured with additional rules, such as assigning priorities to particular requests.

Auto-scaling and self-healing are not perfect solutions: the number of maximum nodes to scale to in auto-scaling must be bounded to prevent the deployment cost from exceeding company budgets. In both techniques, it may take several seconds to minutes to create a new pod, during which the performance degradation would not be resolved. A deployment may have also a static allocation of resources and not use auto-scaling or self-healing, in which high load cannot be mitigated by managing resources.

2.1.5 Circuit-Breaker Pattern and Brownout

High load scenarios lead to negative impacts on user experience and the potential for complete system outages. For users of an application under high load, they either wait on a blank page for tens of seconds before it slowly loads, or grow impatient, leading to a high attrition rate. In some cases, a *retry storm* can occur, where the client-side, or services which consume other services, naively continue retrying their requests, leading to an unintentional denial of service attack. Two approaches, the circuit-breaker pattern and brownout, which are orthogonal to auto-scaling and other resource management techniques, attempt to mitigate these impacts.

The circuit-breaker pattern overcomes the above high load limitations by having the application accept requests up to a certain rate limit, then throttling and immediately rejecting requests above this limit. For users whose requests are rejected, they receive an error message immediately, an improvement in experience over patiently waiting for a page which may never load.

Brownout is an evolving technique inspired by the circuit-breaker pattern which leverages the fact that some components of a system are more important, perhaps in business value or user experience, than other components [3]. In brownout, the required and optional components of a cloud application are first determined. Then, during runtime, a system reacts to high load by disabling optional components: the higher the load, the less likely optional components are enabled. This technique is typically realised using control theory at the network ingress or load balancing layer [4].

2.2 Self-Adaptation and Control Theory

Filieri et al. [16] introduce the concept of self-adaptation at the intersection of software and control engineering. In this section, we explain the high-level design of self-adaptive systems from a software engineering perspective and then describe the implementation of such systems from a control theory perspective.

2.2.1 Self-Adaptive Software and Feedback Loops

Self-adaptive software is realised by a closed-loop system with a feedback loop aiming to adjust itself to changes during its operation in order to ensure that given goals are not violated [17]. This closed loop system is managed by the *adaptation manager*, which is responsible for handling this feedback loop.

In software engineering, a common feedback loop design is the Monitor-Analyse-Plan-Execute (MAPE) loop, optionally with a shared knowledge base (Figure 2.3). The *monitor* phase collects and correlates data from the system and its environment; the *analyse* phase analyses the monitoring data along with historical data; the *plan* phase determines what changes are need and the optimal way to effect the changes; the *execute* phase applies the determined actions. All steps may consult a shared knowledge base, leading to the MAPE-K acronym.

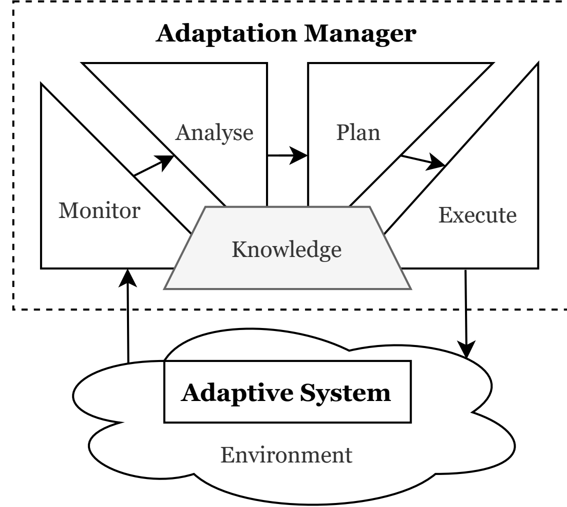


Figure 2.3: A self-adaptive system with a MAPE-K loop. Adapted from Filieri et al. [16].

In control theory, a self-adaptive system is realised by a control loop, consisting of a plant (i.e., the adaptive system) and controller, as shown in Figure 2.4. Intuitively, given a discretised time step k, the controller uses the error $e(k)$ between the plant output $y(k)$ and the plant goals $\overline{y}(k)$ in order to decide how to change the parameters (or *knobs*) $u(k)$ of the system. Control loops may also have to deal with *disturbances*, denoted by $d(k)$, which are external factors that may affect the output.

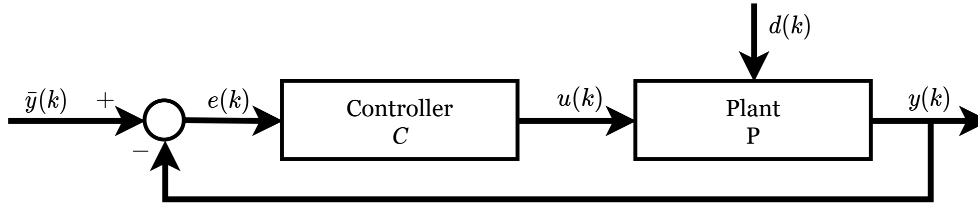


Figure 2.4: A control loop in a self-adaptive system. Adapted from Filieri et al. [16].

Filieri et al. [18] describe a methodical approach of implementing a control loop. First, the system's goals must be identified, for example, by choosing a target *setpoint* (a reference value) or a quantity to minimise or maximise. Then, the knobs which change software behaviour, such as resource allocation to the plant, are identified. Next, a model of the plant needs to be devised, capturing the relationship between the goals and the knobs. Then, the controller is designed, a complex task with different approaches and controller types (e.g., time-based, knowledge-based, etc.). Finally, the controller is implemented and the system is tested and validated.

2.2.2 Controllers

PID Controllers. PID controllers are a popular, time-based control technique. They do not require an explicit model: instead, they can be tuned based on experimentation and heuristics [16]. In this technique, the control input is computed according to

$$u(k) = u(k-1) + \underbrace{K\Delta e(k)}_{\text{proportional}} + \underbrace{\frac{Kh}{T_i}e(k)}_{\text{integral}} + \underbrace{\frac{KT_d}{h}[\Delta e(k) - \Delta e(k-1)]}_{\text{derivative}} \tag{2.1}$$

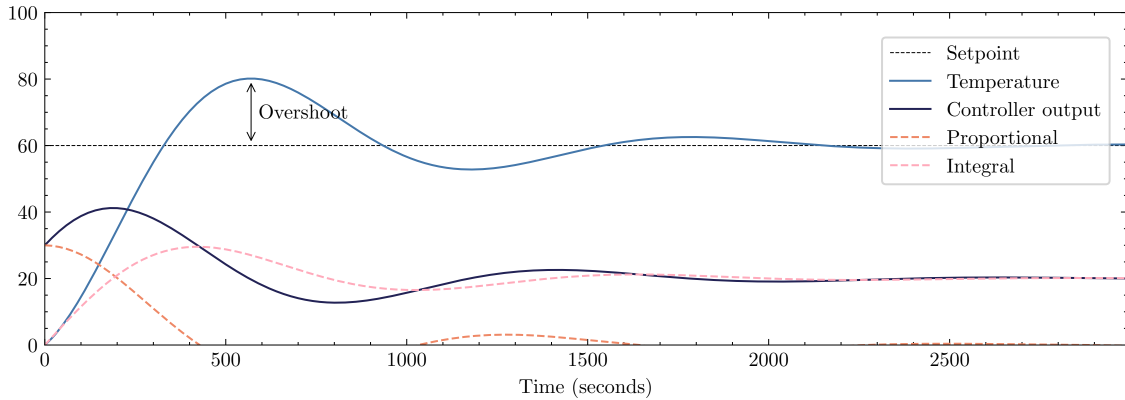


Figure 2.5: Example of a PI controller in a water boiler simulation.

where h is the sampling time of the controller, and K, T_i and T_d are the parameters of the controller. There are three main components of the controller:

- The proportional term is proportional to the error between the setpoint and actual value, generating a corrective response in the presence of an error. If the gain factor K is too large, the actual value may oscillate.
- The integral term integrates past values of the error, accelerating convergence towards the setpoint and mitigating steady-state error from purely using the proportional term, but can cause the actual value to overshoot the setpoint.
- The derivative term uses the slope of the error over time to improve settling time and reduce overshoot.

The integral and derivative terms may each be excluded if the other terms are adequate in causing the controller to meet the setpoint with minimal overshoot and oscillation. Figure 2.5 shows a simulation with only the proportional and integral terms set and derivative term excluded.

Knowledge-Based Controllers. Conventional controllers require a correct design and specific mathematical processes which may not be feasible to easily approach or generalise in the software engineering domain for complex systems such as cloud applications. Knowledge-based controllers allow a higher level of control by augmenting knowledge about the system with monitoring information using a reasoning engine.

2.3 Modelling and Forecasting

In this section, we introduce statistical techniques for modelling various forms of data relevant to our work.

2.3.1 Random Sampling

Halton Sequence. Our work will use random number sampling in order to find deterministic patterns without having to search an entire sample space. The Halton sequence [19] provides a much more evenly distributed, random-like set of points when compared against pseudorandomly generated numbers. This discrepancy in evenness can be seen in Figure 2.6.

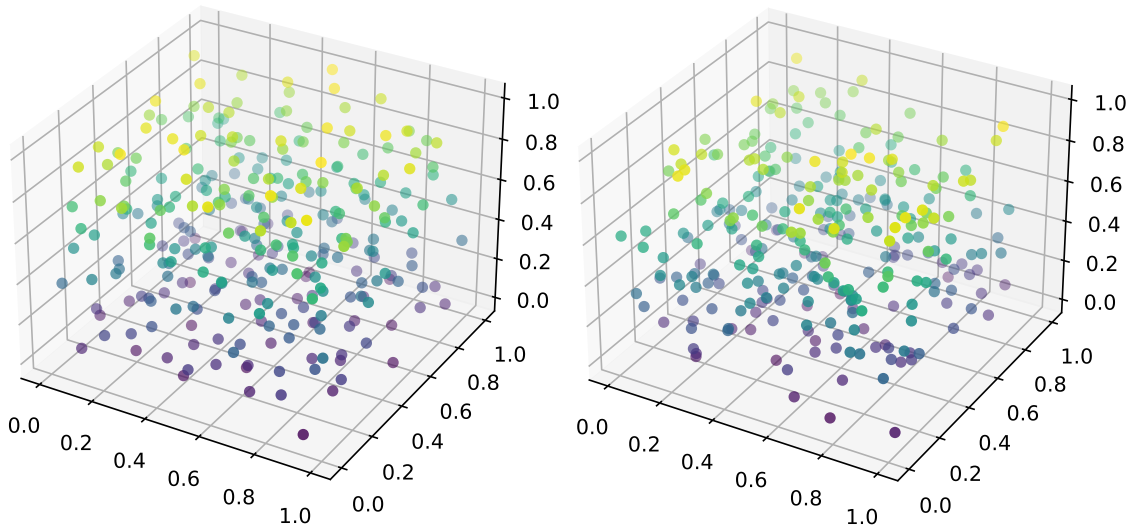


Figure 2.6: 300 points of a $[0,1]^3$ Halton sequence compared with a pseudorandom source.

2.3.2 Time Series Analysis

A time series is a series of data points collected over time. Metrics collected over time in a cloud environment can be represented as a time series, then analysed using statistical techniques. We describe techniques relevant to our work below.

Least Squares Estimation. Given multiple predictor values (e.g., a set of parameters observed over time), least squares estimation fits coefficients for these parameters to minimise the sum of squared errors for a prediction [20]. Coefficients $\beta_0, \beta_1, \ldots, \beta_k$ for parameters $x_0, x_1, \ldots, x_k$ are chosen to minimise

$$\sum_{t=1}^{T} \varepsilon_t^2 = \sum_{t=1}^{T} (y_t - \beta_0 - \beta_1 x_{1,t} - \beta_2 x_{2,t} - \cdots - \beta_k x_{k,t})^2 \tag{2.2}$$

where the subscript t denotes values in a time series indexed between $t = 1$ and T.

Auto-Regressive Modelling. Auto-regressive models forecast a variable based on a linear combination of the past values of the variable. Auto-regressive modelling can capture patterns in a time series, such as how a spike in CPU usage is influenced by previous spikes. An auto-regressive model of order p, $AR(p)$, is defined as

$$X_t = c + \sum_{p} \varphi_i X_{t-i} + \epsilon_t \tag{2.3}$$

where $\varphi_1, \varphi_2, \ldots, \varphi_p$ are the parameters of the model, c is a content and ϵ_t is white noise [21].

ARIMA. ARIMA is another approach to time series forecasting, where auto-regressive modelling is combined with *moving average* modelling (similar to $AR(p)$, but uses past forecast errors instead of past forecast values).

Granger Causality Testing. Granger causality tests can determine whether a time series is useful for predicting another time series. Giles [22] states an informal definition, where in the case of two time-series variables X and Y, “X is said to Granger-cause Y if Y can be better predicted using the histories of both X and Y than it can by using the

history of Y alone". To perform the test, X is compared with time-lagged versions of Y using an F-test, and the null hypothesis that X does *not* Granger-cause Y is rejected if the p-value is below a critical value [23].

Kolmogorov-Smirnov Testing. A two-sample Kolmogorov-Smirnov test (K-S test) can compare the similarity of two sampled continuous probability distributions [24]. In queuing theory, response times are typically modelled using exponential distributions [25]; the K-S test can be used here by obtaining a distribution of response times from captured data.

2.3.3 Clustering

Cluster analysis involves grouping together similar data points, commonly used for classification purposes. We describe k-means and k-shape clustering below, both of which scale linearly with the number of metrics.

K-means Clustering. The k-means algorithm divides data into k clusters by initially arbitrarily selecting k centroids, then iteratively re-computing centroid positions to decrease intra-cluster distances and inter-cluster distances until convergence [21]. The Euclidean distance is often used as the distance metric.

K-shape Clustering. K-shape clustering is a recent clustering algorithm applicable to time series which uses a novel distance metric called *shape-based distance* [23]. This metric considers the shape of two time series, with the advantage that it can detect similarities even if one lags behind another in the time dimension. The algorithm works by initially assigning time series to clusters randomly, then iteratively computing new centroids using shape-based distance until convergence.

2.3.4 Dimensionality Reduction

Principal Component Analysis. Principal component analysis (PCA) is a technique for reducing the dimensions of a dataset which, given a high-dimensional dataset, outputs a set of linearly uncorrelated principal components. Principal components can then be analysed to discover variables which strongly vary together for each principal component. Hence, PCA can be used to discover significant parameters in datasets by capturing a large number of parameters and their effects.

Chapter 3

Related Work

In this chapter, we describe and discuss work related to our aims. In Section 3.1, we describe earlier approaches for the adaptive resource management of cloud applications using control theory, the basis for our project. Then, in Section 3.2, we describe approaches to modelling workloads by analysing parametric dependencies and creating robust performance models. In Section 3.3, we discuss modelling dependencies between cloud workloads using statistical techniques. Finally, in Section 3.4, we discuss reference applications available for use in our evaluation.

3.1 Brownout and Control Theory in Adaptive Resource Management

Klein et al. [3] introduce brownout, a self-adaptive technique for cloud applications to be more resilient against unpredictable runtime variations such as *flash crowds* and unexpected hardware failures. In brownout, optional components are identified, and a controller varies the proportion of user requests where the optional components will not be processed. This proportion is dynamically varied by the controller in response to an indicator of system load (i.e., the maximum response time captured during the last control loop).

In evaluating the brownout technique, Klein et al. [3] demonstrate the potential of brownout to improve business objectives: using a demo e-commerce application augmented with an optional recommendation system, the authors demonstrate that brownout compliance leads to lower latency under high load, correlated with better user experience and higher revenue. However, instrumenting brownout by changing feature code conflates the responsibilities of feature teams and reliability engineering teams, limiting the scalability of this approach in a real-world environment. Additionally, in this technique, qualitative business effects are secondary to reducing overall maximum response time. In contrast, we will allow the business impact of dimming particular components to be directly involved in the decision process and instrument dimming through more developer-friendly approaches.

Dürango et al. [26] and Klein et al.'s later work [27] focus on the applicability of brownout in real-world scalable systems scenarios by investigating load balancing techniques which maximise optional components served while keeping response times low, further improving user experience. The authors' architectures both consist of a load balancer distributing requests to application replicas, where each replica has an independent brownout controller. Dürango et al. first find that a common, non-brownout aware algorithm, Shortest Queue First (SQF), is adequate in meeting these goals, then introduce novel algorithms

which allow the load balancer to make decisions which take into account dimming levels. Klein et al. then improve the load balancer performance by making the brownout-aware algorithms fully event-driven, where controller parameters update per-request instead of periodically. In contrast to modifying feature code, these works showcase contributions at an architecture layer which is more natural for DevOps engineer to be working on; our work will continue investigating brownout at this layer.

Xu and Buyya [28] propose a taxonomy of brownout approaches to adaptive resource management. The authors characterise approaches to brownout by comparing their features in each phase of the MAPE-K loop. The authors then propose a model for brownout-enabled cloud computing systems, with three requirements: (1) applications must be composed of mandatory and optional services, which must be identified beforehand; (2) a controller based on the MAPE-K model must be used to control brownout; and (3) the knowledge pool of the MAPE-K model balances tradeoffs between desired metrics. The authors then identify future research directions in areas including developer usability, applications of brownout to different metrics such as energy efficiency, and augmentation of brownout with other resource management techniques. The proposed model provides a basis for the architecture of this project.

Kotegov and Filieri [4] overcome Klein et al.'s limitations of requiring codebase changes by performing brownout and resource auto-scaling at the orchestrator level. In keeping to Xu and Buyya's [28] model, the authors implement a custom Kubernetes ingress controller behind a HAProxy load balancer, where load balancing rules are adaptively set using a PI controller to dim components, bringing the average response time close to a configured setpoint. Unlike Dürango et al. and Klein et al.'s load balancing work, Kotegov and Filieri shift the entire brownout process to the load balancing layer, requiring no feature code changes. Additionally, the authors take an alternative approach of working with replicas by allowing the controller to scale the number of replicas in response to load, though a limitation with this auto-scaling is that changes are not precisely or immediately reflected: Kubernetes runs its own, conflicting heuristics to decide whether the new scale targets should be met. Overall, this orchestrator-level approach is more developer friendly and our work will build upon this approach by allowing individual components to be dimmed at different rates depending on developer-specified parameters.

3.2 Performance Modelling and Parametric Dependencies

A significant challenge to overcome with our work, where real-world systems usually have more than one optional component, is that dimming components at different rates can reduce performance due to interactions between components, either from code dependencies or changes in user behaviour. In this section, we investigate a field which seeks to more formally understand these interactions.

Eismann et al. [29] introduce a method of accurately modelling the performance of a system by generating relationships called *parametric dependencies* between system components and their performance properties, based on component inputs and correlations between these inputs. The authors model on data captured from run-time monitoring, instead of modelling at design-time, in order to capture the variation of performance based on different workload mixes. System modelling for resolution of these parametric dependencies is implemented using the Descartes Modelling Language (DML). This work can be useful in the context of brownout, where knowing the impact of dimming a particular component on the response time of its dependencies can inform better dimming decisions. However, the authors' technique is not developer-friendly or scalable to large systems: DML knowledge

is required and a precise model of a system's dependencies needs to be created beforehand, which is time-consuming for large systems.

Ackermann et al. [30] describe a machine learning approach to characterise parametric dependencies given a set of identified potential dependencies and run-time monitoring data. The authors' technique first generates a decision tree to choose a machine learning algorithm based on characteristics of the available data, and then applies the algorithm on the data to generate the resulting parametric dependencies. This technique improves on the usability limitations of Eismann et al. [29] as a precise model of dependencies is not needed beforehand. Additionally, the authors evaluate on the same model used by Eismann et al., achieving acceptable results. However, potential dependencies still need to be identified beforehand and implementations require experience with monitoring frameworks (Ackermann et al. suggest the Kieker framework), again limiting real-world usability.

Ackermann et al. [30] also observe that automatically identifying parametric dependencies involves static code analysis which is unsuitable for modelling distributed service-oriented architecture. However, more recent research by Grohmann et al. [31] proposes using machine learning techniques to automatically detect parametric dependencies from monitoring data. The authors use the Kieker framework configured with custom metric collection probes to monitor data, then choose and apply a feature selection algorithm to identify the dependencies. This work removes the need for developers to manually identify parametric dependencies before the dependencies can be automatically characterised; however, specialised monitoring tools are still required to collect the data needed for this technique, hence we will investigate a trade-off between configuration and automatic detection of parametric dependencies.

Bauer et al. [32] present *Chamulteon*, which demonstrates the application of performance modelling to the adaptive resource management of microservices-oriented architecture. Chamulteon allows the coordinated scaling of multiple services, which contrasts against previous works which are prone to bottleneck shifting, where services are each placed behind independent scalers. The auto-scaler consists of a reactive and proactive component, both of which rely on the utilisation metric from queueing theory. The proactive component is particularly novel: a performance model implemented in DML is used for forecasting and estimation. The authors evaluate Chamulteon on a Docker deployment, showing accurate auto-scaling without bottleneck shifting. With performance modelling demonstrated in auto-scaling, further work could involve bringing this technique to other adaptive resource management areas such as brownout, though the requirement for DML knowledge is still a limiting factor.

3.3 Statistical Approaches to Resource Management

In contrast to the previous section, this section details techniques to understand and predict system behaviour which do not require specialised performance modelling knowledge, motivating developer-friendly approaches in our work.

Shah et al. [33] introduce a machine learning approach to extracting dependencies between microservices. The authors take a novel approach of training a long-short term model (LSTM) to learn the relationships between monitored time series data of a cloud application (i.e,. CPU, memory, network receive and network transfer for each microservice), then inspect properties of neurons in order to determine the most significant predictors. For example, in a setup consisting of an application service and database service, the authors find that application network receive, transmit and CPU are strong predictors for the

database transaction. The authors also determine that their approach leads to greater accuracy than other time series analysis techniques like ARIMA and Granger causality tests. The key limitation is that training is computationally expensive, requiring powerful hardware, hence, our work will focus on traditional statistical techniques.

Apte et al. [21] present a scalable, accurate workflow for discovering dependencies between virtual machines by performing time series analysis on CPU utilisation. The workflow consists of three steps: first, the authors monitor CPU utilisation for each virtual machine over a sampling period of around one second; then, auto-regressive modelling is applied to each time series; finally, k-means clustering is used to group virtual machines by distances between their AR models. The authors show that they are able to identify dependencies with a 97.15% overall accuracy rate. Though this paper focuses on virtual machines, we are able to easily apply the same techniques to identify correlations in a Kubernetes environment as the orchestrator makes inter-pod metrics available, and the authors' workflow consists of well-known, straightforward techniques.

Thalheim et al [23]. present *Sieve*, a general purpose microservices metrics analysis framework. Sieve analyses the call graph and time series of metrics for each microservice in a deployment, applies k-shape clustering, then applies Granger causality tests to establish a dependency graph to determine causality across microservices. The authors demonstrate the application of Sieve on real-world examples for auto-scaling and root cause analysis, with a focus on accuracy and developer usability, also sharing their code on GitHub[1]. Compared to Apte et al. [21], Thalheim et al. more convincingly justify their choices of statistical methods in the context of time series analysis and microservices architecture. Hence, the authors present strong candidates for statistical approaches to use in our work.

Kim et al. [34] present a technique which uses principal component analysis and factor analysis to schedule workloads onto a scientific computing cloud environment, optimising the deployment of a task given a choice of target cloud sites. The authors find principal components based on historical data for combinations of task input parameters, cloud site resource metrics and execution times, then schedule new tasks onto optimal sites based on these principal components. Our work could also use PCA for factor analysis of our service parameters, however, as stated by Thalheim et al. [23], the results of PCA are not easily interpretable compared by developers to other clustering techniques.

3.4 Reference Applications for Microservices

There are several reference applications for microservices which we can use to evaluate our work. To motivate our evaluation framework, we present and discuss four candidate reference applications below.

JPetStore-Kubernetes. Kotegov and Filieri [4] use JPetStore-Kubernetes[2], a containerised version of a monolithic Java web app orchestrated with Kubernetes. This application is composed of three services: a Java web application, a Go image search microservice and a MySQL database. This choice was reasonable for Kotegov and Filieri's use case of dimming the entire application. However, our use case involves dimming at a service level. Hence, the architecture is not decomposed enough to be used in our project.

Sock Shop. Sock Shop[3] is a demo shopping cart where features (cart, catalogue, etc.) are split into no more than ten separate Kubernetes services. Services are well-composed

[1] https://github.com/sieve-microservices/

[2] https://github.com/IBM-Cloud/jpetstore-kubernetes

[3] https://github.com/microservices-demo/microservices-demo

enough for our use case of dimming at a service level. This application has been widely used in the past: Gias et al. [35], for example, use Sock Shop to demonstrate model-driven auto-scaling for microservices. Additionally, Sock Shop demonstrates polyglot architecture well: services use different languages and frameworks, which help motivate requirements for our implementation. However, one limitation is that the load testing tool for this application must be able to handle conditional runtime logic: previous work with Sock Shop has shown that there are minor bugs which will need to be mitigated, such as the checkout process only working if the cart total is less than a certain value.

TeaStore Von Kistowski et al. [36] present TeaStore[4], a reference application designed for benchmarking, modelling and resource management, deployable on Docker and Kubernetes. TeaStore is composed of six services. Like JPetStore-Kubernetes, TeaStore encapsulates its web application in one service. The other services are for backend services for authentication and content provisioning (e.g., provisioning images and generating recommendations).

TeaStore is unique in its testing and benchmarking tooling. Designed for performance modelling, the application is pre-instrumented to allow the Kieker monitoring framework to automatically generate logs, including at a method call parameter level. Additionally, the authors have tested TeaStore with two load generators, LIMBO and JMeter, and profiles for these tools are publicly available[5] in their online documentation. In comparing TeaStore to Sock Shop, the authors identify that TeaStore meets the same requirements for what a microservice benchmark should have; however, TeaStore also has the advantage of meeting these additional research requirements of benchmarking.

Train Ticket. Zhou et al. [37] present Train Ticket[6], a ticket booking system designed to bridge typical benchmark microservice applications with applications found in industry, with greater complexity, better testing and better conformance to microservices design principles compared to existing benchmark applications. The system contains over forty microservices, with up to five layers of dependencies between microservices.

Train Ticket has a unique design of splitting up microservices for exploring, reserving and ordering train tickets into two classes: high-speed trains and normal-speed trains. The authors use this design to demonstrate the potential for resources for these two classes of microservices to be independently adjusted based on user demand. This design can be particularly useful for our goals of a discriminate dimming approach. However, we are unable to use this for our project: during preliminary testing, we have experienced bugs in its user interface, as well as some text being displayed in the authors' native language, which restricted our usability of this application.

[4] https://github.com/DescartesResearch/TeaStore
[5] https://github.com/DescartesResearch/TeaStore/wiki/Testing-and-Benchmarking
[6] https://github.com/FudanSELab/train-ticket

Chapter 4

Kubedim

This project presents **Kubedim**, a self-adaptive, brownout-enabled reverse proxy which applies brownout strategies on optional components of a cloud application in order to meet both system stability and business objectives. Like with Kotegov and Filieri [6], Kubedim is designed to be deployed within a Kubernetes cluster in a developer-friendly manner. With Kubernetes' popularity in production deployments of microservices-based applications [38], the choice of Kubernetes contributes to the goal of Kubedim's applicability to real-world use cases. This section presents an overview of Kubedim, with further implementation details discussed in the sections which follow.

4.1 Brownout Strategies

From Klein et al. [3] to Kotegov and Filieri [6], all previous implementations of brownout have uniformly dimmed optional components by the same parameter while a system is under load. In contrast, Kubedim implements Kotegov and Filieri's uniform dimming strategy as a baseline strategy, then introduces two non-uniform brownout strategies: non-uniform dimming based on *component weightings*, and non-uniform dimming based on *user profiling*.

4.1.1 Baseline Dimming

Kubedim's most basic brownout strategy, as with prior related work, is to dim optional components uniformly proportional to system load. In Kubedim's implementation of this brownout strategy, brownout starts when the response time at a given percentile exceeds a given setpoint (e.g., 3s at the 95th percentile). As with Kotegov and Filieri [6], the dimming parameter is a percentage between 0% and 100% which increases as system load increases, and represents the probability that the request for an optional component will be dimmed. If the dimming parameter is 0%, the optional component will never be dimmed; if the dimming parameter is 100%, the optional component will always be dimmed.

4.1.2 Component Weightings

Dimming by component weightings addresses two key considerations. First, Bauer et al. [32] recognise that system components tend to contribute towards load unevenly, so dimming can be more effective if components are dimmed in proportion to their individual contributions to overall load. Second, optional components such as advertising and recommender systems may impact business objectives (e.g., advertisements and recommen-

dations), so ensuring that optional components are not unnecessarily dimmed may lead to an improvement in business objectives.

This strategy is realised by assigning weightings to each optional component, represented by a probability between 0 and 1. The dimming parameter from baseline dimming is then weighted for each optional component by multiplying the optional component's weighting. Hence, a component which contributes more to load would be assigned a higher weighting than a component which contributes less to load.

Kubedim also provides tooling to train this component weighting model offline, ensuring that beneficial weightings can be chosen which do not cause side effects such as bottleneck transfers. Where non-optimal weightings are given, Kubedim also provides an option to perform online training, improving component weightings live in production. This ecosystem is explained in further detail in Section 4.3.

4.1.3 User Profiling

Dimming by user profiling is motivated by two considerations relating to business objectives and user experience. First, if users can have their tendency to meet business objectives identified by the dimmer, then optional components can be tailored towards higher-priority users, resulting in a positive impact to business objectives. Second, both the baseline dimming and component weightings strategies can reduce user experience due to optional components intermittently and inconsistently disappearing under high load, motivating a strategy which can provide a better user experience through stronger consistency.

This strategy consists of profiling users based on their session request history, then dimming a number of users for their entire session proportional to system load. User profiling is performed asynchronously by applying a set of rules specified by the developer based on the users' session request histories. Then, optional components for a number of low priority users proportional to system load are dimmed for the entire duration of the users' sessions while optional components always display for high priority users, leading to more consistent and predictable application behaviour.

4.2 Deployment

Kubedim is deployed as a pod within Kubernetes as a reverse proxy which inspects HTTP requests, forwarding requests to a backend pod when requests are not dimmed. Unlike Klein et al. [27], orchestrating brownout at the reverse proxy level means that application-level code does not have to be modified, increasing ease of developer usage. Figure 4.1 shows the structure of a typical deployment, where existing pods are not modified. Deployment instructions are provided in Appendix A.

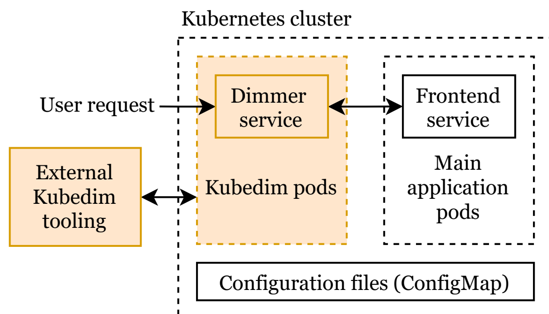


Figure 4.1: Structure of a typical Kubedim deployment.

There are two reasons why Kubedim is implemented as a reverse proxy in a pod, instead of following Kotegov and Filieri's [6] implementation of extending a third party ingress controller. First, production deployments likely already use ingress controllers for load balancing and traffic shaping, so implementing at the ingress controller layer would decrease the ability for Kubedim to be quickly installed. Second, existing ingress controllers possess limited extensibility for our advanced strategies: HAProxy and NGINX Reverse Proxy are both declaratively configured around a limited set of rules, for example. Hence, implementing Kubedim as an independent reverse proxy allows for easier installation by users and implementation of more advanced features.

In order to manage Kubedim, administration endpoints are exposed on a port, allowing system administrators to perform actions such as changing the brownout strategy and overriding component weightings.

4.3 Configuration

Configuration for Kubedim is specified in YAML, the same configuration language used to configure Kubernetes deployments, and is propagated to the dimmer pod via a ConfigMap. A manual is provided, which can be found in Appendix A. As Kubedim is an application layer reverse proxy, optional components are identified by the HTTP methods and paths used to access the components. As a result, configuring Kubedim for baseline dimming is straightforward, where only connection information and optional components need to be specified.

```yaml
dimmableComponents:
  - path: "recommender"
    method:
      shouldMatchAll: true
    probability: 0.5
  - path: "news"
    method:
      shouldMatchAll: true
    probability: 0.5
  - path: "cart"
    method:
      method: "GET"
    probability: 0.9
```

Listing 1: Excerpt of a Kubedim configuration file specifying optional components. Component weightings are highlighted.

Listing 1 shows an example of how optional components can be specified, along with advanced filtering options for cases such as a request path being used for both optional and required components of a system.

The more advanced strategies, component weightings and user profiling, require further configuration detailed as follows.

4.3.1 Component Weightings

Component weightings can be specified by providing probabilities for each path. However, it can be difficult for developers to make educated guesses for the exact weightings to use: dimming unevenly can cause side effects if optional components interact with each other or cause user behaviour to change. As a result, Kubedim provides an *offline training* tool,

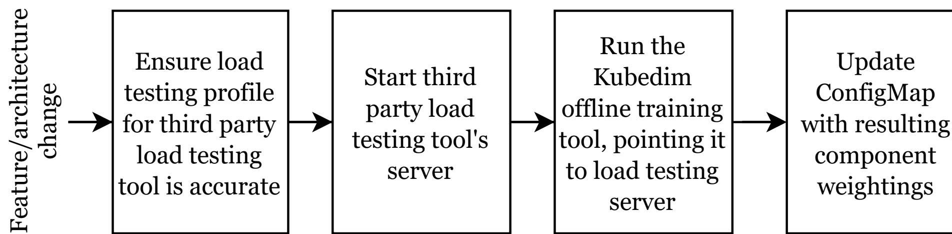


Figure 4.2: Typical developer workflow for updating component weightings.

designed to be used in non-production (*offline*) environments, to model the contributions of different weightings to overall system load and therefore predict the optimal weightings. Modelling a black-box system requires a large amount of data, hence the tool takes on the order of hours to run to gain sufficient data. The resulting weightings are then pasted into the ConfigMap. A typical workflow is shown in Figure 4.2 and an example configuration is highlighted in Listing 1.

Kubedim also provides a way to improve component weightings in a production (*online*) environment in order to overcome sub-optimal weightings which result either from developers making guesses about weightings or from prediction errors in offline training. Prediction errors usually originate from differences between simulated user behaviour during offline training and actual user behaviour, or from noise in the model. This *online training* tool works in a manner similar to A/B testing [39], continuously assigning a set of new candidate weightings to a small percentage of user sessions, in order to determine whether candidate weightings perform better than the provided weightings. Online training can be activated via the administration endpoint.

4.3.2 User Profiling

```yaml
rules:
  - description: "User has checked out items in past"
    path: "/cart"
    method:
      shouldMatchAll: false
      method: "POST"
    occurrences: 1
    result: "high"
  - description: "User is browsing and unlikely to buy"
    path: "/category.html"
    method:
      shouldMatchAll: true
    occurrences: 5
    result: "low"
```

Listing 2: Excerpt of a Kubedim configuration file specifying profiler rules.

User profiling requires two configuration steps: pointing Kubedim to a storage driver for logging users' requests and specifying a list of rules for users to be profiled with. Rules are provided by the developer and are evaluated in sequential order, returning an *unknown* priority if no rules are met. Listing 2 above shows an example of a rules configuration.

Chapter 5

Implementation

Kubedim is implemented in Go due to its prevalence in modern microservices architectures [40], its use as the main language for the Kubernetes codebase and its higher performance in cloud environment benchmarks against other languages such as Java and Python [41]. In this section, we detail implementation specifics and discuss design decisions.

5.1 High-level Architecture

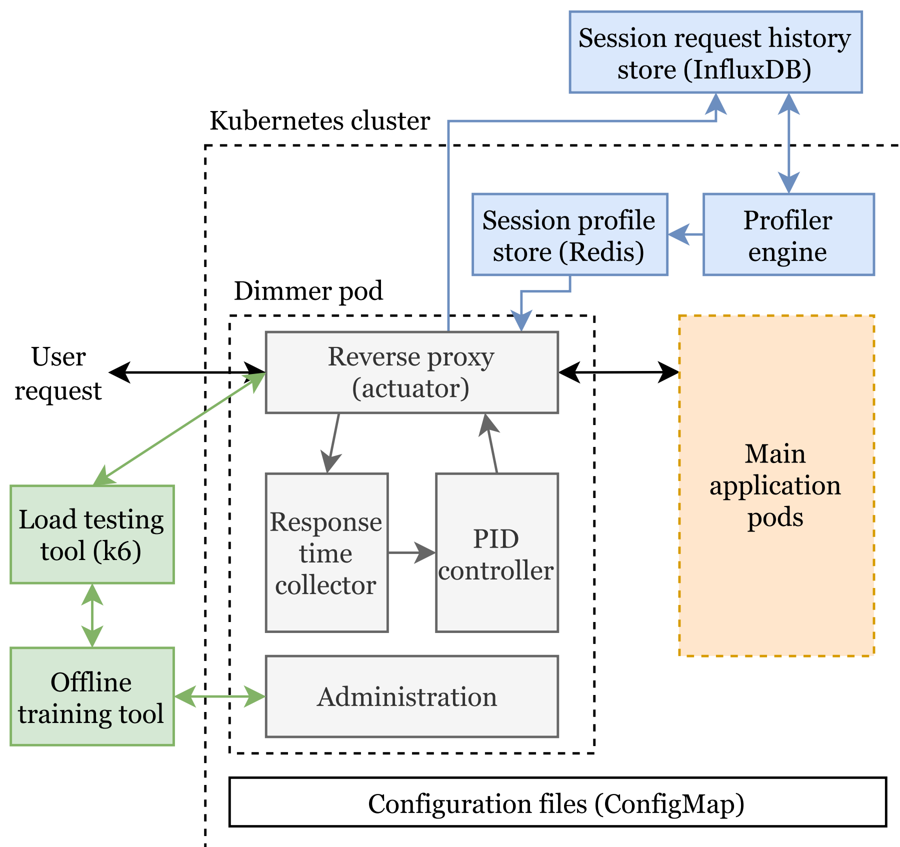


Figure 5.1: The Kubedim architecture.

The architecture for Kubedim is shown in Figure 5.1. There are four main segments, as follows:

- Pods for the original application are not modified.

- The **dimmer** is a single Go binary, deployed as a single pod. At the most basic level, it acts as a reverse proxy, being placed between the external user and all the pods of the original application. However, the dimmer is also responsible for collecting and processing request data for the purpose of making brownout decisions. An administration endpoint is also exposed, allowing dimmer settings to be changed in realtime.

- The **offline training** tool is a Go binary designed to be run externally as part of the weighted components training process. The tool interfaces with the dimmer administration endpoint and a load testing tool in order to capture data to train a model representing component weightings.

- The **profiler** is composed of two pods within the cluster and an external database. Request details for every session are sent to the database. The request history database is processed asynchronously at intervals by the profiler pod and profiled sessions are stored in a key-value store pod for retrieval by the dimmer.

5.2 Proxying and Brownout with Adaptive Control

The dimmer consists of a lightweight HTTP reverse proxy augmented with hooks to capture and modify request and response data for brownout. As with prior work [3, 6], we have chosen to follow a MAPE-K loop-based design, centred around a PID controller which updates every second. This design increases the ability for developers to easily understand brownout behaviour and does not have the complex tooling and expert knowledge requirements of traditional performance modelling or machine learning approaches.

5.2.1 Monitoring (Response Time Collection)

The metric used by Kubedim to quantify system load for brownout actuation is response time at a specified percentile (50th, 75th and 95th are available). We specify a default of the 95th percentile, a commonly seen percentile in SLAs, as five percent of users experiencing high response times in a medium-sized application may lead to discernible business impact. Large enterprise companies like Amazon may have even stricter SLAs (i.e., at the 99.9th percentile [42]), however, these SLAs would be too sensitive for medium-sized applications.

Monitoring is implemented so that response time instrumentation adds minimal overhead to each request. Capturing the response times of all requests over one second for querying by the PID controller would require complex, time-indexed data structures or an external time series database. Instead, like with HAProxy and other systems [6], a heuristic is used where a fixed window of the n latest response times is stored. n is set to 100 by default, a trade-off between a lower number, which would cause aggregations to be more sensitive to changes in response time, and a higher number, where the window would capture a wider distribution of response times. The fixed window is implemented locally in the same process as the dimmer and the duration for each proxied request to complete is captured and appended to this window.

5.2.2 Analysis and Planning (PID Controller)

Like with prior work [3, 6], we use a PID controller to provide a white-box, predictable approach to responding to changes in response time. Though a PID controller needs to be tuned, developer usability is retained as tuning a controller successfully does not require a formal understanding of control theory and expected behaviour can still be achieved with

sub-optimal tuning parameters [43]. Based on our experience throughout this project, we set default parameters which should not need to be overridden for most applications. When necessary, the controller's tuning can be customised by the developers to achieve specific dimming properties or optimise the controller on specific features of the application.

At its foundation, Kubedim uses a standard PID controller which takes in the response time at the specified percentile in seconds as input and outputs a dimming strength as a percentage between 0% and 100%. The period of each cycle is set to 1 second by default, long enough to capture a useful a sufficient distribution of response times and short enough for actuation to be responsive.

The setpoint is set to a default of 3 seconds, following Google's research into page abandonment for sites which take longer than 3 seconds to load [44]. Hence, in the default configuration, for 95th percentile response times below 3 seconds, the controller will output 0%; for greater response times than 3 seconds, the controller will output a non-zero percentage proportional to the difference from the setpoint.

For the default tuning, we set the proportional gain to 2, integral gain to 0.2 and derivative gain to 0. Typically, tuning would involve a systematic method such as the Ziegler-Nichols method [16], however, we found tuning based on our understanding of the system to be sufficient. These parameters are justified as follows:

- The proportional gain is set to a small number so that as the response time begins to violate the setpoint, the system has a chance to stabilise with only a small percentage of requests being rejected. Systems under high load which do not stabilise at this point typically quickly incur tail response times reaching tens of seconds, to which the proportional gain responsively yields an output in the tens of percents.
- The integral gain is set to ensure the dimming percentage still increases if the response time remains constant above the setpoint, and to ensure the dimming percentage is sustained when the response time is brought down to the setpoint. A small value of 0.2 is used as larger values cause the integral gain to be more sensitive, leading to overshoot and oscillation.
- The derivative gain is set to zero as derivative gain is semantically a prediction of the near future evolution of the input signals, which we do not need to rely on to achieve expected results. Most PID controllers, in fact, do not use the derivative term [45].

Finally, the controller is implemented with two additional features:

- Exponential smoothing is performed on the input with a smoothing factor of 0.9. During experimentation, we found that exponential smoothing reduced oscillations in the dimming percentage resulting from noise in the input response times.
- Anti-windup is implemented to prevent excessive overshooting in dimming percentage due to accumulation of integral action over longer times when the target response time is not achieved, or cannot be achieved, with the available resources [45].

5.2.3 Execution (Reverse Proxy Actuation)

Like with Kotegov and Filieri [6], when a request arrives at the dimmer, the reverse proxy either forwards the request to the actual application and returns the response to the user, or it immediately returns the HTTP *427 Too Many Requests* error code to the user. However, instead of setting a rate limit proportional to the dimming percentage, we opt for a more direct, probabilistic approach which allows for more control, particularly with advanced brownout strategies.

The probabilistic approach for baseline dimming is simple: the probability that a request is dimmed is equal to the dimming percentage. If a request method and path represent an optional component, then a randomly sampled number between 0 and 1 is compared against the dimming percentage. If the sampled number is less than the dimming percentage, the request is dimmed with *427 Too Many Requests*. Otherwise, the request is not dimmed and is proxied as normal.

More advanced brownout strategies build upon this probabilistic approach, explained in the following sections.

5.3 Component Weightings

Dimming by component weightings is actuated by modifying the probability check in baseline dimming. Instead of directly comparing a sampled number against the dimming percentage, the sampled number is compared against the weighted dimming percentage, *Dimming Percentage* × *Weighting for Optional Component*. As the weighting for an optional component is between 0 and 1, a weighting which is less than 1 will cause a component to be less likely to be dimmed. Likewise, if all weightings are set to 1, the system behaves equivalently to baseline dimming.

5.3.1 Offline Training

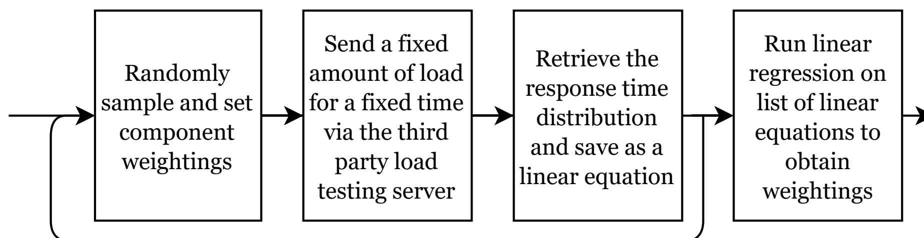


Figure 5.2: The offline training loop.

As it is difficult for developers to make educated guesses about what component weightings to choose, Kubedim provides a tool for predicting the weightings on a non-production environment.

The offline training tool uses linear regression to perform training. In this model, the independent variables are the component weightings and the dependent variable is the response time at the developer's chosen percentile. Training a linear regression model on the effects of different component weightings on the response time of an application outputs a linear equation, where the coefficients describe the effect of dimming a component on the overall application response time. For example, `response time = 7.5 + news*-1.5 + cart*-15.3` implies dimming the cart endpoint more strongly than the news endpoint will have a greater impact on reducing the overall application response time.

This *a posteriori* approach is a simpler, more effective approach than the *a priori* related work [30, 32] which require expert knowledge combined with advanced modelling theory. By sampling a large number of combinations of component weightings, the regression model is able to fit around side effects, such as cases where dimming components unevenly instead causes an increase in response time. Hence, the need for the developer to understand dependency analysis and complex performance modelling techniques to prevent side effects is obviated; instead, they must ensure observed response times reflect real-world system behaviour, a task explained below.

In practice, as shown in Figure 4.2, the developer runs the training tool in conjunction with a load testing tool with a load profile configured to simulate real-world usage. Given the use of user simulations in system testing and chaos engineering [46], it is a reasonable expectation for developers to understand high-level user behaviour and be able to create accurate load testing profiles. Currently, only the k6 load testing tool is supported; however, the training tool has been implemented with the adapter pattern for future compatibility with other load testing tools.

Figure 5.2 shows the main steps of the offline training tool. In each iteration, the following steps are performed:

1. The tool sets a randomly sampled set of component weightings via the non-production environment's dimmer administration endpoint. The Halton sequence is used for sampling a more evenly-covered distribution of points [19].
2. The offline training tool interfaces with the load testing tool to send a fixed level of load for a fixed amount of time.
3. The offline training tool retrieves the response time distribution from the administration endpoint.

After all iterations are complete, all response time distributions are fit to a linear regression model and the resulting coefficients are normalised into probabilities between 0 and 1 ($p_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$).

5.3.2 Online Training

Load testing profiles will never perfectly simulate real-world usage and it is infeasible for developers to run offline training for every feature change, given comprehensive offline training can take on the order of hours to gather enough data. As a result, online training follows principles of A/B testing in order to improve the given component weightings. Here, users are divided into a control group, which experiences the developer's given component weightings, or a candidate group, which experiences a randomly sampled set of component weightings.

Online training seeks to balance strict, high-value component weightings which decrease system load, with relaxed, low-value component weightings which increase component availability and therefore improve business objectives. Figure 5.3 shows this overall behaviour of the online training loop.

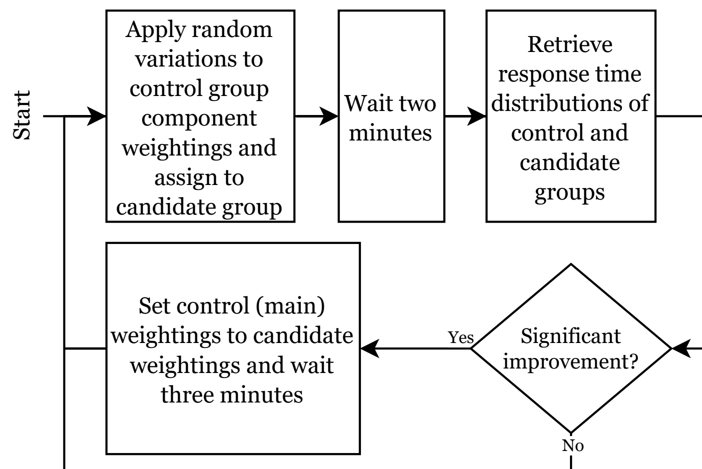


Figure 5.3: High-level behaviour of the online training loop.

When enabled through the administration endpoint, the online training service labels new sessions as belonging to either the control group or candidate group, which is persisted by setting a cookie. By default, 5% of requests are assigned to the candidate group, a trade-off between ensuring only a low number of users experience potentially non-optimal component weightings and ensuring enough response times are captured to make a comparison against the control group.

The online training loop runs every three minutes – a trade-off between being responsive to changes in load profiles and capturing enough data in both groups. During each iteration, the following steps are made:

- New candidate component weightings are randomly sampled for only component per iteration, with the remaining components using their respective control group weightings.

- Response time distributions are captured for both control and candidate groups over three minutes.

- If the sampled weighting has decreased in the candidate group, the candidate group 95th percentile response time is within 3% of the control group response time, and there is an insignificant difference in response time distributions (via a Kolmogorov-Smirnov test at the 95th percentile), the control group weightings are replaced with the candidate group weightings. This occurs if the system stays stable when component weightings are relaxed, therefore increasing availability of the optional component under load.

- If the sampled weighting has increased in the candidate group, the candidate 95th percentile response time is lower than the control group response time, and there is a significant difference in response time distributions (via a K-S test at the 99th percentile), the control group weightings are replaced by the candidate group weightings. This occurs if the system becomes more responsive when component weightings are more strict, therefore increasing stability of the system under load.

- If control group weightings have been replaced by the candidate group, online training pauses for a three minute cool-down period in order for the system to adjust to its new component weightings.

Our percentages and percentiles for the above statistical tests have been set based on experimentation to reduce both false positives and false negatives.

5.4 User Profiling

Profiling is implemented as a rules-based system, leveraging the developer's basic knowledge about their system and user flows in order to classify users. This contrasts against more advanced user profiling methods such as clustering approaches, where advanced instrumentation and analytics pipelines are required to gather data for training and classification. Profiling works by processing rules on a list of requests for every session. Once the priority of a session is determined, a long-lived decision to dim all optional components for the session is made (in contrast to dimming per-request), providing a more consistent experience for the user.

5.4.1 High-Level Implementation

Profiling sessions and logging session data within the dimmer would add too much overhead. Instead, we split up Kubedim's profiler into three components, as shown in Figure 5.1:

- A Redis key-value database stores the resulting priorities of profiled sessions. As an in-memory key value store, the dimmer reverse proxy can query the store with low overhead.
- Session request histories are stored in an external database. In its current implementation, Kubedim can be changed to support any popular database engine. However, a time series database, InfluxDB, is specifically chosen as future implementations of Kubedim may support rules which involve the time axis. InfluxDB also allows data to be persisted asynchronously, reducing the overhead of recording requests per session in the reverse proxy.
- The profiler engine asynchronously processes sessions which do not have priorities assigned at regular intervals according to developer-specified rules. This interval-based approach reduces overall system load as profiling is not a time-sensitive action.

5.4.2 Profiling Lifecycle

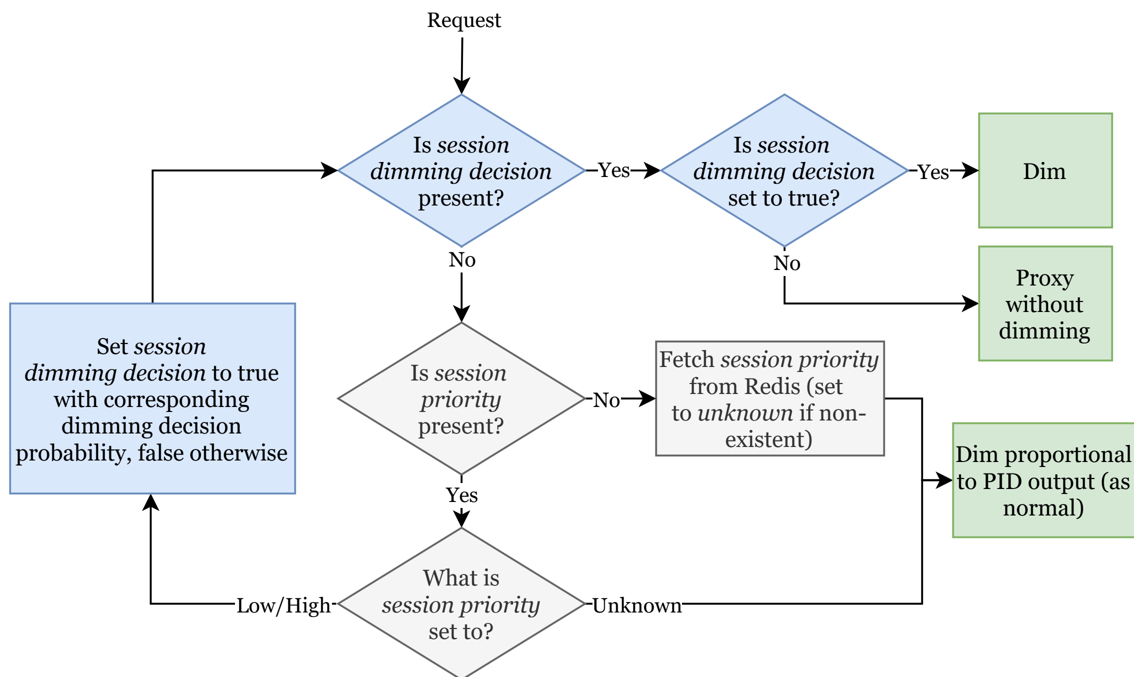


Figure 5.4: Decision tree showing actions taken by the dimmer's actuation module.

There are two phases in profiling in the lifecycle of a request, as shown in Figure 5.4.

Profiling Session History

The first phase involves profiling session history. The developer specifies the name of the cookie which identifies a session in their original application. All requests with a session cookie present are saved to InfluxDB. Session history in InfluxDB is fetched and evaluated by the profiler in fixed intervals according to developer-specified rules, setting a *session priority* cookie with the resulting priority.

In our current implementation of rules processing, we take a simple yet effective approach of allowing developers to specify in each rule that a session should be assigned a specified priority if the user has visited an endpoint more than a specified number of times. An example is shown in Listing 2. This allows priorities to be assigned with conditional statements based on developers' knowledge of user behaviour. For example, in a shopping cart application, a user may be considered high priority if they visit a shopping cart page often, whereas they may be considered low priority if they are only looking at past order details.

The session priority cookie can be set to one of three values, as follows:

- If a rule matches a session's history, the session priority cookie will be set to the resulting priority (either *high* or *low*) with a default expiry of 2 hours. This long expiry is set to reduce profiler workload, justifiable as users are unlikely to change behaviour during a session.
- If no rules are matched, either due to there being an insufficient number of requests for the session or due to rules being non-exhaustive, the session priority cookie will be set to *unknown*. The expiry for unknown sessions is set to a default of 2 minutes, allowing the user to make more requests before the profiler next processes the session.

Dimming Decision

The second phase is responsible for the actuation of dimming. In order to actuate a long-lived decision to dim all optional components for a session based on its profiled priority, a *dimming decision cookie* containing a true boolean value is set which is sent along with all requests for that session.

The dimming decision cookie is set by the actuator depending on presence and contents of the session priority cookie. The cookie is set depending on two configuration parameters, the probability of making a dimming decisions given a request is low priority, P_L, and the probability given the request is high priority, P_H, which are set by default to $P_L = 0.05$ and $P_H = 0.95$. The cookie has a default expiry time of two minutes, allowing for a trade-off between responding to changes in the PID controller output and giving users consistently dimmed user experiences.

If the dimming decision cookie is not present when a request arrives at the dimmer, the actuator makes one of two decisions:

- If the session priority is set to *unknown*, the dimmer falls back to baseline dimming without setting a dimming decision cookie.
- If the session priority is set to *low* or *high*, the probability that the resulting dimming decision is set to *true* is

$$P(Dimming\ Decision = true) = \begin{cases} PID\ Output \cdot \frac{P_L \cdot N_L}{P_L(N_L+1)+P_H(N_H+1)} & Priority = Low \\ PID\ Output \cdot \frac{P_H \cdot N_H}{P_L(N_L+1)+P_H(N_H+1)} & Priority = High \end{cases}$$

  where N_L and N_H are the number of recent low priority and high priority sessions respectively.

Dimming decisions are made with the PID output in mind to ensure that sessions are dimmed in proportion to system load. The fractional part of the above equation exists to handle the edge case where most or all of the requests are only of one priority type. For example, if the system is under heavy load, but all requests are of high priority, we want to dim a large number of high priority requests instead of only dimming the default of 5% of requests.

Chapter 6

Sock Shop for Kubedim

We present **Sock Shop for Kubedim**, a modified version of Sock Shop with two areas of significant changes: first, making changes to improve reproducibility of experiments and second, the addition of optional components for testing brownout strategies. Figure 6.1 shows the architecture of this the modified system. In this chapter, we describe the changes we have made and discuss the deployment of Kubedim on our variant of Sock Shop.

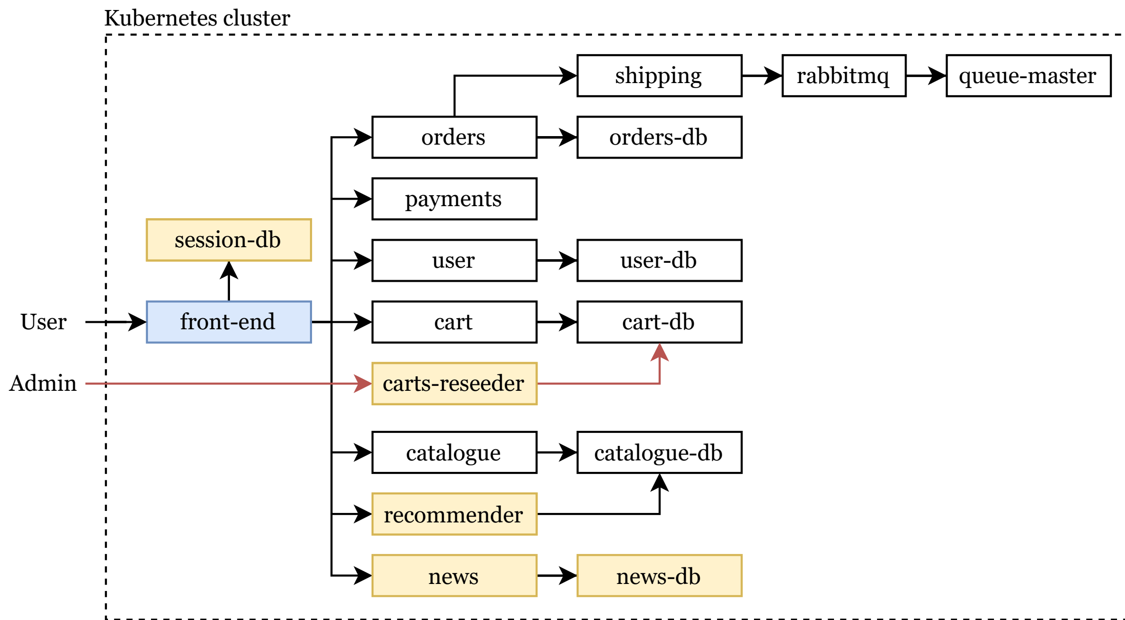


Figure 6.1: Architecture diagram of modified Sock Shop. Orange represents new pods; blue represents modified pods.

6.1 Reproducibility

Our initial experimentation with an unmodified Sock Shop deployment yielded inconsistent results which could not be reproduced. An investigation yielded that request response times were linearly correlated with the size of the carts database and the size of the front-end sessions store, and that both data stores grew rapidly during each load test. In achieving more reproducible results for Sock Shop, we have built tooling to manage the number of entries in the carts database, and fixed various bugs with session storage.

6.1.1 Carts Database

The `carts-db` microservice is a MongoDB database which is responsible for storing the items in a cart for each session. Each row in the carts collection is an empty cart, with items rows nested underneath. A new empty cart row is created every time Sock Shop requests the number of items in a cart for a new session.

The main source of irreproducibility of Sock Shop experiments is that the response time of all `carts-db` requests, be it querying, insertion or deletion, is directly proportional to the number of entries in the database. For example, if there are no entries in the database, queries can take single-digit milliseconds, whereas if there are 200,000 entries in the database, queries can take hundreds of milliseconds.

To ensure reproducible experiments, there are two requirements: first, we must be able to pre-seed the database with a set number of rows, and we must configure our experiments to ensure that the growth in the carts database over the course of these experiments only causes a negligible increase in query response time. For example, the difference in query time for a database with 200,000 rows and 202,000 rows is negligible.

We achieve these reproducibility requirements by implementing a new microservice, `carts-reseeder`, which allows us to view the number of rows in the carts database, delete all rows in the database, and seed a specified number of rows with random GUIDs.

We did not decide to do the alternative of rewriting Sock Shop's cart logic from scratch to be efficient. First, Kubedim is designed to mitigate bottlenecks and scalability issues such as these effectively. Second, the applicability of Kubedim is not prevented by the current `carts-db` logic: only experimental reproducibility is affected. Likewise, there are many more improvements we could make to Sock Shop, given it is a reference application as opposed a production-grade system, but this would not be an aim of this project.

6.1.2 Session Storage

The `front-end` microservice is the main endpoint of a typical Sock Shop cluster, containing a Node.js server which serves static files and connects external API endpoints with internal microservices responsible for serving those endpoints. This microservice is also responsible for maintaining session cookies, a source of various bugs which we fixed in order to improve reproducibility of our experiments.

We first identified that the service was using a session handling library using a local, in-memory store designed only for development usage. This caused front-end response times to grow proportionally to the number of sessions, which we resolved by deploying a Redis pod and switched to a production-grade Redis session driver, eliminating this issue.

Next, we identified that new sessions were being created with every request, leading to a new row being created in `carts-db` for every request, causing `carts-db` queries to slow down very quickly during the course of our experiments. We identified and fixed two causes of this. First, the Node.js server used separate cookie decoder and session management libraries with mismatched secrets, causing session cookies set by the latter library to not be recognised. Second, cookies were not persisted due to use of `res.end` instead of `res.send` when returning requests in some endpoints.

6.2 Optional Components

In order to experiment with brownout strategies, we modified the `front-end` microservice to introduce three optional components: `cart`, `news` and `recommender`.

The Sock Shop user interface contains a cart button in the top-right corner of every page, which also displays the number of items in the cart. This number is fetched with a call to `GET /cart` on each page load, which causes a query to `carts-db` in the microservices call stack. Recognising the contribution of `carts-db` to overall system load, we make this numeric indicator an optional component, allowing load on `carts-db` to be alleviated.

Next, we add a news page to the `front-end` microservice (with endpoints `GET /news.html` and `GET /news`) and a corresponding `news` microservice, representing updates for Sock Shop's company (e.g., delivery and stock updates). `news` is a Go microservice which, in order to potentially find interesting results in brownout strategy experimentation, has an artificially induced wait time sampled from a normal distribution with $\mu = 1, \sigma = 1$. We only mark `/news` as the optional component as it would be redundant to mark the static page too.

Finally, we add a recommender microservice callable via `GET /recommender`, which simulates a system which recommends a single sock item. The `front-end` microservice is modified to display recommendations in various catalogue and item pages. Like the news component, we implement `recommender` in Go with an artificially induced wait time, sampled from a normal distribution with $\mu = 2, \sigma = 2$. The greater normal distribution parameters reflect the increased time and variation it takes for a recommendation system to fetch appropriate recommendations.

6.3 Deploying Kubedim

In order to deploy our version of Kubedim on our version of Sock Shop, we first follow the user manual described in Appendix A, resulting in the configuration file described in Appendix B.

```yaml
# ...
dimmableComponents:
  - path: "recommender"
    method:
      shouldMatchAll: true
    probability: 0.0
  - path: "news"
    method:
      shouldMatchAll: true
    probability: 0.0366
  - path: "cart"
    method:
      method: "GET"
    exclusions:
      - method: "GET"
        substring: "basket.html"
    probability: 1.0
# ...
profiler:
  sessionCookie: "md.sid"
  # ...
```

Listing 3: Excerpt of configuration values relevant to Sock Shop.

Listing 3 shows the most relevant part of our configuration, with the following features:

- Under `dimmableComponents`, we specify the paths of the three optional components to be dimmed, along with whether dimming should only occur for a given HTTP method or for all methods.
- With the `cart` endpoint, we do not want to dim GET requests which originate from the basket page during the checkout process, so we also manually specify an exclusion.
- We specify component weightings from offline training for each component, detailed further in Section 7.4.3.
- We specify the key of the cookie which Sock Shop uses to identify sessions, `md.sid`.

Kubedim is then deployed by running `kubectl apply` on the resulting manifest files.

Chapter 7

Evaluation

In this chapter, we primarily focus on the evaluation of Kubedim's brownout strategies, both in their ability to respond to load and their ability to meet typical business objectives. Then, in Section 7.5, we evaluate our additional aim of investigating the developer usability of Kubedim.

7.1 Kubernetes Setup

To evaluate brownout strategies, we first consider whether a system with baseline dimming shows an improvement over a system without dimming. Then, we compare our three brownout strategies with each other. To do so, we need a reference application to deploy and test Kubedim.

We choose Sock Shop for Kubedim, introduced in the previous chapter, as our reference application. Sock Shop is deployed on a physical Kubernetes cluster consisting of one control plane node and five pod-deployable nodes, with all with their specifications described in Listing 4. An external InfluxDB data store used for profiling data is deployed on the same network under a virtual machine, with specifications described in Listing 5.

```
OS: Ubuntu 18.04.5 LTS
Kernel: 4.15.0-140-generic x86_64
CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (8 cores)
GPU: NVIDIA GeForce 630 OEM
Memory: 15944 MiB
Kubelet version: v1.20.5
```

Listing 4: Specifications for all nodes in the Kubernetes cluster.

```
OS: Ubuntu 20.04.2 LTS
Kernel: 5.4.0-72-generic x86_64
CPU: AMD EPYC IBPB SSBD (4 vCPU cores), backed by 2 x AMD EPYC 7401 (24 cores)
Memory: 8192 MB
Network data-transfer rate: 10 GbE
InfluxDB version: 2.0.4
```

Listing 5: Specifications of the InfluxDB profiling virtual machine.

7.2 Load Testing Setup

We choose k6 as our load testing tool for sending load to our reference application. Unlike prior work like which use Apache JMeter [6] or Locust [35], we use k6 due to its configurability and ability to orchestrate load through its JavaScript and Go APIs.

In order to most accurately represent users, we have created user behaviour graphs with probabilistic transitions between states, along with a highly-customised k6 script which simulates these user behaviour graphs and collects data relevant to Sock Shop's business objectives.

7.2.1 User Behaviour Models

We have chosen to implement three user behaviour models: *buying*, *browsing* and *news*. Users of the *buying* model are high priority users, likely to check out items. Users of the *browsing* model are lower priority, browsing the catalogue without checking out an item. Finally, users of the *news* model are the lowest priority users, who only visit Sock Shop's news endpoint to discover delivery updates and blog entries.

Scheduling and Attrition

We implement a scheduler where each page is represented by a state in the user behaviour diagrams below. The scheduler is responsible for managing the transition to the next state, as well as managing stochastic behaviour for our scheduler in order to more accurately model user behaviour.

All page visits involve two actions in order to closely mirror browser behaviour. First, the entire HTML page is fetched. Then, API endpoints are called in parallel using batched HTTP requests, mirroring how Sock Shop fetches API endpoints asynchronously on page load.

Stochastic behaviour consists of think time and attrition. First, the scheduler induces a think time between each page visit, sampling the time from the uniform distribution with range $[2, 7]$. Second, in order to model user impatience, the scheduler uses the maximum response time of all non-optional components requested while visiting page. Attrition is sampled based on a linear line starting from ~10% chance of attrition at 3 seconds, rising to ~30% chance of attrition at 10s [44].

Buying

Figure 7.1 shows the user behaviour graph for a user which is likely to buy an item. The key features of this model are as follows:

- Users who are buying items will immediately visit the catalogue. Users then visit an item and return back to the catalogue.
- The probability that a user will continue visiting more items after returning to the catalogue is 0.8.
- If the optional `recommender` component is enabled and a recommendation exists in the catalogue, the user will visit this recommended item first. To simplify our chain of decision processes, once a user visits a recommended item, they will not visit further recommendations.
- When a user is visiting an item, the probability that the user will add the item to their cart is 0.8.

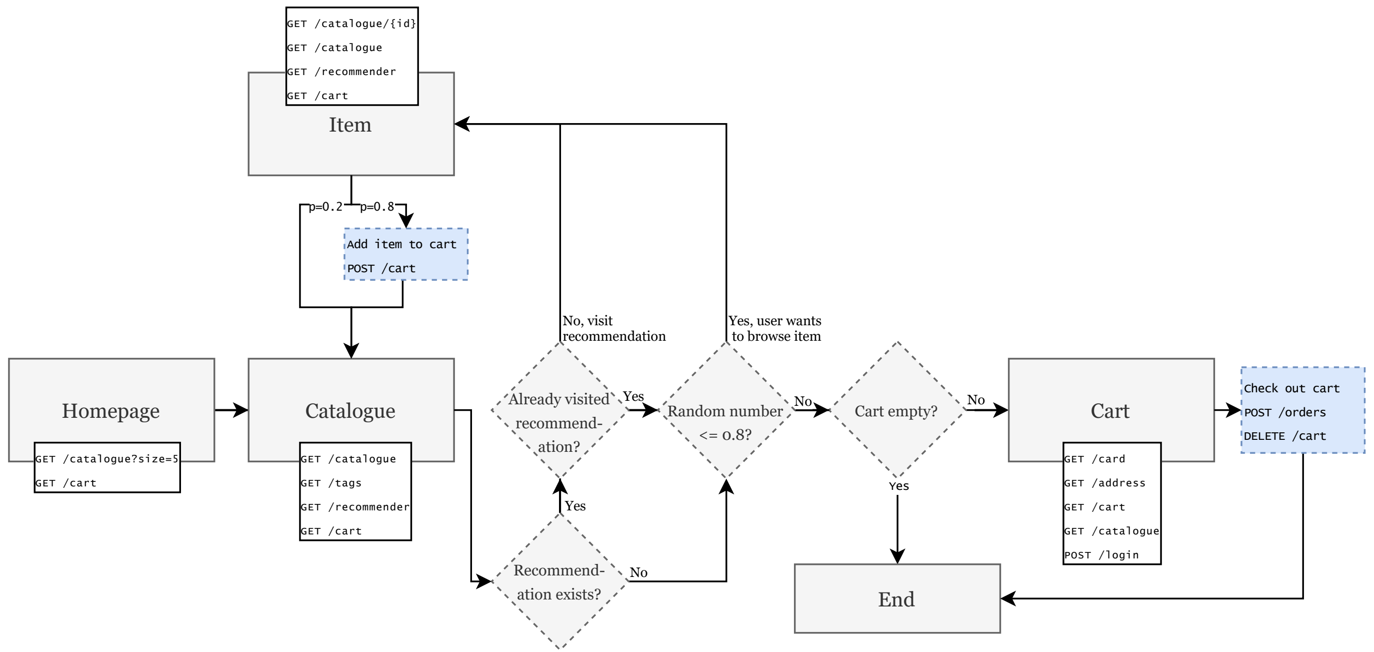


Figure 7.1: User behaviour diagram for the buying load testing profile.

0.8 is chosen for our probabilities so that users are likely to perform these actions more than once, but will be very unlikely to perform these actions more than ten times. This is motivated by real-world shopping only involving a single-digit number of items.

Browsing and News

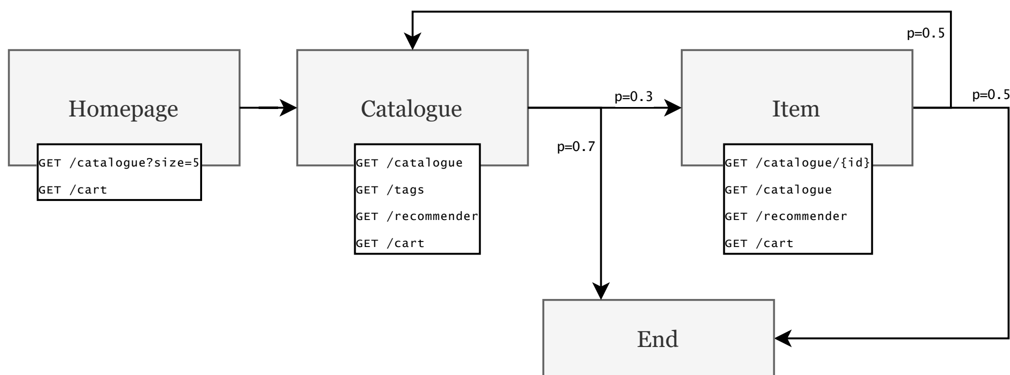


Figure 7.2: User behaviour diagram for the browsing load testing profile.

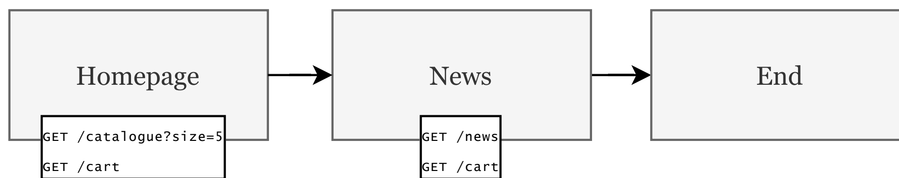


Figure 7.3: User behaviour diagram for the news load testing profile.

The *browsing* and *news* profile are lower priority user types. Figure 7.2 shows the user behaviour graph for a user which is only browsing for items. Probabilistic transitions are added in order to model disinterest in items and to add variations in user behaviour. Likewise, Figure 7.3 shows the user behaviour graph for a user browsing for news. The

main difference, however, is that the news endpoint is optional, given that a page dedicated to delivery and company updates does not provide short-term income to the store.

7.2.2 Load Shape

The default load testing shape in our experiments is the **constant load** shape. At the start of the experiment, we ramp up from 0 to 280 users over 10 seconds. At 10 seconds from start, we sustain the number of users at 280, where each user immediately restarts from their start state once their behaviour loop ends. After 30 minutes and 10 seconds from start, we end the experiment. These parameters are justified as follows:

- Appendix C shows that Sock Shop with dimming disabled saturates after 290 users, at which point the system becomes unstable and control group data cannot be captured. Therefore, we use a slightly lower number of users.
- We run each iteration under load for 30 minutes (from an originally planned duration of 5 minutes) to ensure that dimming behaviour is reliable. In particular, Kotegov and Filieri [6] have observed unexpected behaviour over tens of minutes due to Kubernetes' complex pod scheduling algorithms [47].

The user distribution between profiles is as follows: 2/7 of users use the *buying* profile; 4/7 of users use the *browsing* profile; 1/7 of users use the *news* profile. In order to reduce side-effecting `carts-db` behaviour described in Section 6.1.1, session cookies and the association between virtual users and their profiles remain the same across behvaiour loops.

7.2.3 Data Collection

Our experiments require collecting data to evaluate the load mitigation effects of brownout, and the business objective effects of brownout.

As a direct indicator of how the system responds to brownout, we capture response times at the 50th, 75th and 95th percentile.

To evaluate the business objective effects of brownout with Sock Shop, we consider purchases to be the main indicator of success. First, we record the number of items checked out across an entire experiment. Next, as more items are usually checked out if the optional recommender component is displayed, we also record the number of recommended items checked out. Finally, as users leave the application due to long wait times, we also record the number of attrition events.

7.2.4 Deployment

The load testing tool runs on a virtual machine with specifications given in Listing 6.

```
OS: Ubuntu 20.04.2 LTS
Kernel: 5.4.0-72-generic x86_64
CPU: AMD EPYC IBPB SSBD (4 vCPU cores), backed by 2 x AMD EPYC 7401 (24 cores)
Memory: 4096 MB
Network data-transfer rate: 10 GbE
k6 version: v0.30.0 (2021-01-20T13:14:50+0000/2193de0, go1.15.6, linux/amd64)
```

Listing 6: Specifications of the k6 load testing virtual machine.

The load testing tool is configured to send load to the Kubernetes cluster described in Section 7.2. In order to reduce unexpected variations in response times, the tool is situated on the same internal network as the Kubernetes cluster.

7.3 High-level Methodology

Our load tests are automated using a script which interfaces with our Kubernetes cluster and load testing server. Before an experiment cycle, Sock Shop is pre-seeded with user accounts which are re-used between sessions, shifting the overhead of registrations away from the runtime of load tests. Experiments are repeated at least five times. However, we did not notice significant variance across multiple runs and therefore plot graphs for only one iteration in the experiments set out in this report. Scripts to reproduce the experiments (including on different Kubernetes clusters) are open-sourced[1]. Each load testing iteration is orchestrated as follows:

1. A request is sent to `carts-reseeder` to reseed `carts-db` with 200,000 rows. This number is chosen as a few thousand requests are sent over the course of an iteration and the increase in load as a result of these new rows in comparison to the existing rows is negligible.
2. A request is sent to the dimmer administration endpoint to reset the state of the PID controller.
3. A command is sent to k6 to orchestrate load using a given load shape, such as the constant load shape introduced in Section 7.2.2.
4. The collected data is saved for output at the end of the experiment.

7.4 Brownout Strategies

In this section, we evaluate the load mitigation and business objective effects of Kubedim by running load tests on our brownout strategies and comparing these strategies with each other.

7.4.1 Baseline Dimming: Behaviour

Aims. (A) To validate our adaptive control theory approach by ensuring our response time objective converges to its setpoint with minimal oscillations. (B) To ensure that our controller is reactive to changes in the 95th percentile response time, in particular, decreasing dimming once the server is no longer under high load.

Method. To investigate Aim A, we ran the constant load scenario described in Section 7.2.2 once with baseline dimming enabled, then once with no dimming at all, to provide comparable data. We plotted response time (and PID output when baseline dimming is enabled) against time and obtained statistics from the collected raw data for analysis. To investigate Aim B, we used a *flash crowd* load shape shown in Figure 7.4 with baseline dimming enabled only, and captured the same data as in Aim A. This load shape has more gradual changes in the number of users, simulating brief periods of high load.

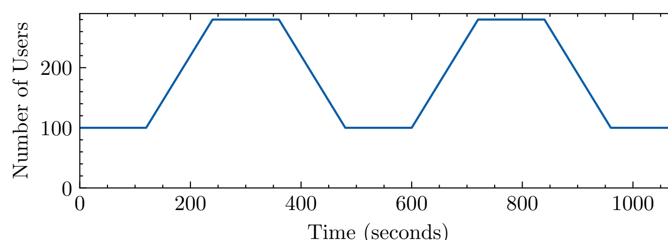


Figure 7.4: The flash crowd load shape, which sends load between 100 and 280 users.

[1] `https://github.com/kcz17/manifests` and `https://github.com/kcz17/experiments`

Results. Figure 7.5 shows the constant load test with dimming disabled, with raw data showing a response time of $\mu = 5.25s, \sigma = 1.15s$ between 200 and 1810 seconds. Figure 7.6 shows the constant load test with baseline dimming enabled, with raw data showing a response time of $\mu = 2.97s, \sigma = 0.743s$ and PID output of $\mu = 46.9\%, \sigma = 1.6\%$ between 200 and 1810 seconds. Figure 7.7 shows the flash crowd test for Aim B with baseline dimming enabled.

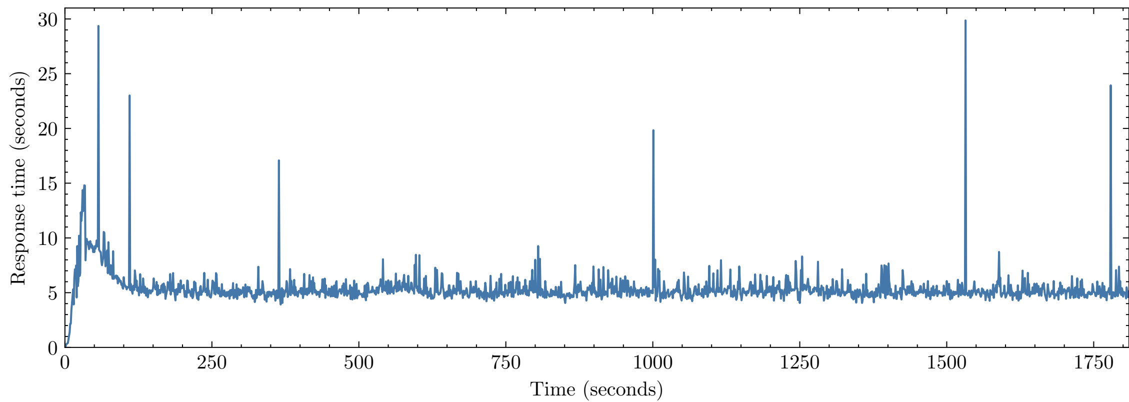


Figure 7.5: 95th percentile response time graph for the constant load scenario with dimming disabled.

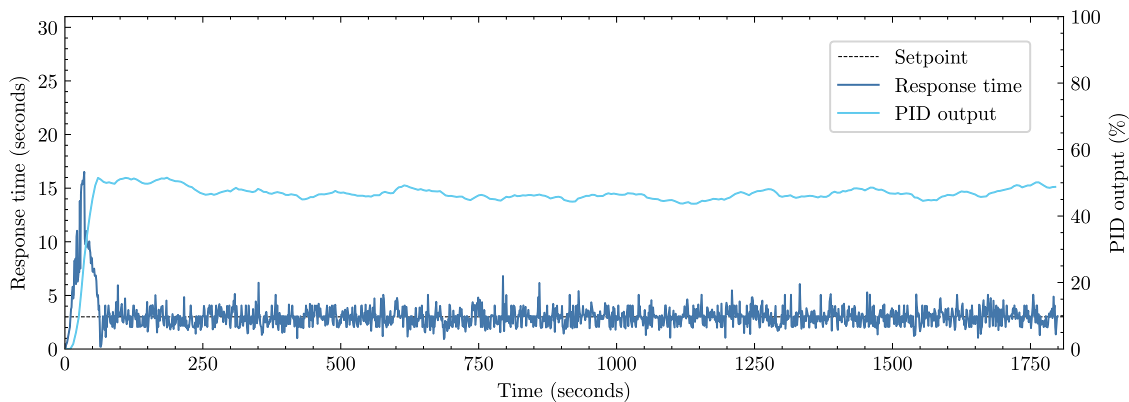


Figure 7.6: 95th percentile response time and PID controller output graph for the constant load scenario with baseline dimming enabled.

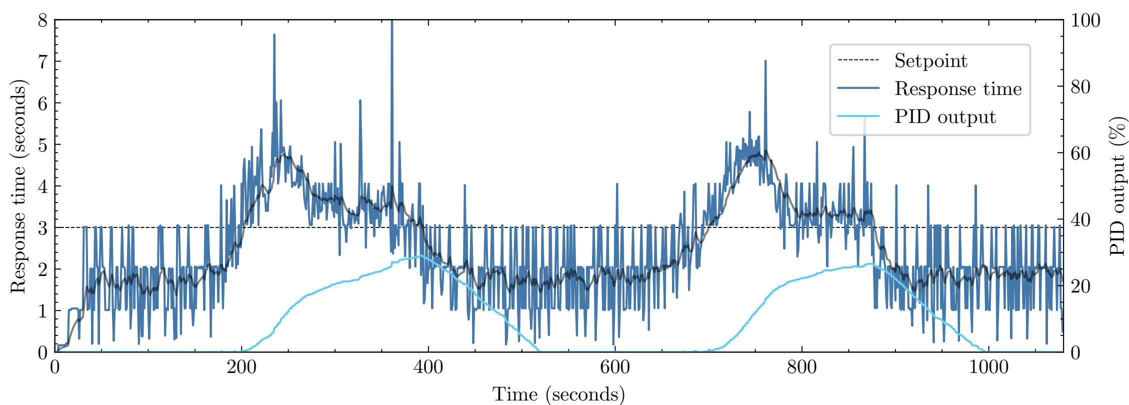


Figure 7.7: 95th percentile response time and PID controller graph for the flash crowd scenario with baseline dimming enabled. Trend line is displayed with exponential weighted average over 20 seconds for readability.

Discussion. Figure 7.6 shows that the PID output responsively increased when the system encountered load, successfully causing the 95th percentile response time to converge to the setpoint. Furthermore, we see that the PID output stabilised quickly with no overshoot and no significant oscillation. Hence, Aim A is met. Figure 7.7 shows that PID output gradually reduced over the course of two minutes once the 95th percentile response time returned below the setpoint, hence, Aim B is also met.

We see from our statistical results that dimming does introduce non-negligible noise. However, this variance of 0.743s is acceptable, particularly as the PID output would simply increase to mitigate frequent setpoint violations as a result of noise. Moreover, the load spikes shown in Figure 7.5 are completely removed in Figure 7.6 when baseline dimming is enabled, suggesting greater system stability.

We also note that there is no observable side-effecting behaviour that would affect the validity or reproducibility of our results. First, unlike the original Sock Shop, our variant did not have an noticeable increase in mean response time over time due to growth in `carts-db` data. Second, we did not observe any unexpected scheduling or load balancing behaviour from Kubernetes' orchestration cycles. This is expected as our news and recommender components have response times independent of the number of replicas, and our cart component is bottlenecked by `carts-db`, which cannot be replicated.

7.4.2 Baseline Dimming: Improvement over Dimming Disabled

Aims. To investigate the effect of baseline dimming on Sock Shop business objectives.

Hypotheses. (A) There is an increase in the number of items checked out when baseline dimming is enabled. (B) There is an increase in the number of recommendations checked out when baseline dimming is enabled.

Method. We ran the constant load scenario described in Section 7.2.2 for five repeats with no dimming at all (control group), then for five repeats with baseline dimming enabled (candidate group), collecting the number of items and recommendations checked out to perform Welch's t-tests for our two hypotheses.

Results. Table 7.1 shows the results of running the constant load with dimming disabled and Table 7.2 shows the results of running the same scenario with baseline dimming enabled.

Run	Items Checked Out		Attrition
	Total	of which Recommended	
1	14	7	2327
2	84	44	2205
3	17	14	2355
4	12	9	2343
5	70	42	2228
avg	39.4	23.2	2291.6
s	34.7	18.3	69.7

Table 7.1: Shopping metrics for five constant load scenario runs with dimming disabled.

Run	Items Checked Out		Attrition
	Total	of which Recommended	
1	2393	884	1173
2	2387	882	1228
3	2301	871	1233
4	2298	862	1237
5	2188	938	1290
avg	2313.4	887.4	1232.2
s	83.5	29.6	41.5

Table 7.2: Shopping metrics for five constant load scenario runs with baseline dimming enabled.

(A) The number of items checked out with baseline dimming enabled ($\mu = 2313.4, s = 887.4$) was significantly higher than that of dimming disabled ($\mu = 39.4, s = 34.7$), $t(5) = 56.2$, $p < .001$, one-tailed.

(B) The number of recommendations checked out with baseline dimming enabled ($\mu = 887.4, s = 29.6$) was significantly higher than that of the dimming disabled control group ($\mu = 23.2, s = 18.3$), $t(5) = 55.5$, $p < .001$, one-tailed.

Discussion. We see a significant increase in both items and recommendations checked out. This is expected as users encounter high response times and system instability when dimming is disabled, causing the vast majority of users to leave before purchasing an item. Dimming optional components under load brings the response time below our setpoint, causing fewer users to attrite.

7.4.3 Component Weightings: Behaviour

Aim. To investigate the effect of the component weightings strategy on response time and PID controller output.

Method. We obtained our component weightings by running the offline training tool for our three optional components for 300 iterations with 280 users. We then ran the constant load scenario described in Section 7.2.2 once with our obtained component weightings, collecting PID output and response time data.

Results. Figure 7.8 shows the results of running the constant load scenario with our resulting component weightings in Listing 7. Raw data shows a response time of $\mu = 2.99s, \sigma = 0.71s$ and PID output of $\mu = 36.6\%, \sigma = 1.7\%$ between 200 and 1810 seconds.

```
Regression formula:
    Predicted = 5.5879 + recommender*0.3293 + news*0.1449 + cart*-4.7103
Component weightings:
    [{Path: "recommender", Coefficient: 0},
    {Path: "news", Coefficient: 0.03660538384937473},
    {Path: "cart", Coefficient: 1}]
```

Listing 7: Output of offline training command with resulting component weightings.

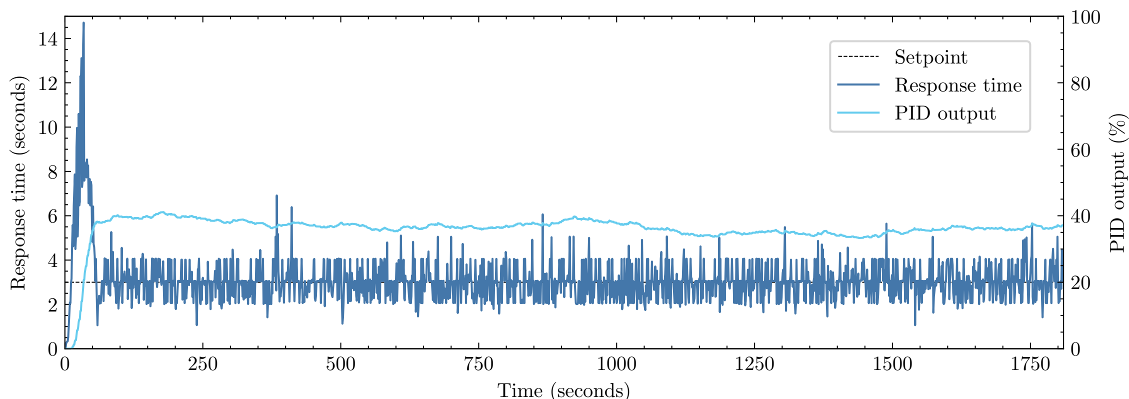


Figure 7.8: 95th percentile response time and PID controller graph for the constant load scenario with component weightings set.

Discussion. We see from the offline training output that the cart endpoint was correctly identified as the main contributor to system load in Sock Shop. The resulting graph in Figure 7.8 not only shows that the expected dimming behaviour was followed, but also

shows a significant decrease in PID output when compared against the baseline dimming behaviour graph in Figure 7.7 (46.9% to 36.6%). This is expected as the cart endpoint is dimmed more heavily than the other optional endpoints. Hence, if there is a clear bottleneck in system load which can be correctly identified by the offline training tool, then the component weightings strategy is a promising candidate as there will be lower overall dimming, resulting in users experiencing increased optional component availability.

7.4.4 Component Weightings: Improvement over Baseline Dimming

Aim and Hypothesis. To investigate whether dimming with component weightings leads to an significant improvement in both (A) the number of items and (B) recommendations checked out over baseline dimming.

Method. We run the constant load scenario described in Section 7.2.2, first for five repeats with baseline dimming enabled (control group), then for five repeats with dimming with the component weightings obtained in the previous section (candidate group), collecting the number of items and recommendations checked out to perform a Welch's t-test on the two groups.

Results. Figure 7.3 shows the results of running the constant load scenario with our component weightings in Listing 7. Due to an identical methodology step, we used our existing baseline dimming data from Table 7.2.

(A) The number of items checked out for dimming with component weightings ($\mu = 2698.4, s = 33.3$) was significantly higher than that of baseline dimming ($\mu = 2313.4, s = 83.5$), $t(7) = 9.58$, $p < .001$, one-tailed.

(B) The number of recommendations checked out for dimming with component weightings ($\mu = 1439.0, s = 20.7$) was significantly higher than that of baseline dimming ($\mu = 887.4, s = 29.6$), $t(5) = 34.1$, $p < .001$, one-tailed.

Run	Items Checked Out		Attrition
	Total	of which Recommended	
1	2688	1412	1027
2	2645	1432	992
3	2722	1437	989
4	2726	1469	1014
5	2711	1445	982
avg	2698.4	1439.0	1000.8
s	33.3	20.7	18.9

Table 7.3: Shopping metrics for five constant load scenario runs for dimming with component weightings.

Discussion. We see significant increases in both items and recommendations checked out with the component weightings strategy compared against baseline dimming. This is expected as baseline dimming is equivalent to using the component weightings strategy with all weightings set to 1, restricting user behaviour as optional components are dimmed more heavily. With the weighting for the `recommender` component relaxed, more recommendations are visited during user sessions, leading to these significant increases.

7.4.5 Component Weightings: Online Training Correctness

Aim. To investigate whether online training improves component weightings and whether they converge upon the optimal offline training weightings given in Listing 7.

Method. We ran a slightly modified version of the constant load scenario described in Section 7.2.2. The scenario ran once for a duration of 6 hours 10 seconds with 270 users. The number of users was slightly reduced from our usual number of 280 to prevent system saturation with sub-optimal component weightings. Dimming by component weightings was enabled, with initial weightings set to 0.5. We captured response time and PID output data, along with the control and candidate group component weightings.

Results. Figure 7.9 shows changes to component weightings over the course of the experiment. The `cart` component weighting decreased to around 0.8 over time, with a small period of decrease to around 0.3 around the 2000 second mark. The `recommender` weighting fluctuated between large and small values. The `news` component decreased over time, settling around 0.2.

Figure 7.10 shows the effect of our experiment on the response time and PID controller output. The response time and PID output mirror the changes in component weightings as expected: due to the `cart` component being a large contributor to system load, a decrease in its weighting increases PID output and vice-versa. Likewise, changes in the `recommender` and `news` components do not have visible effects on response time or PID output.

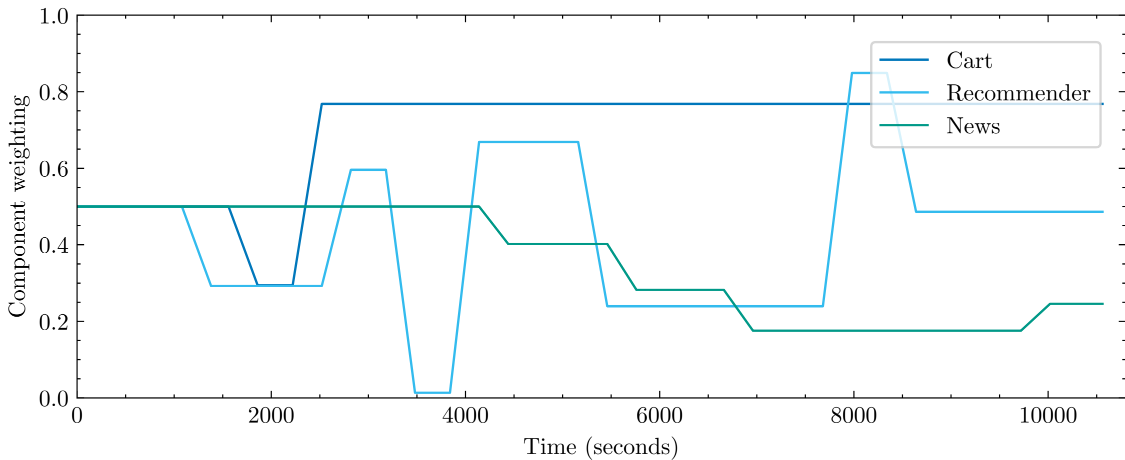


Figure 7.9: Control group component weightings for the online training correctness experiment.

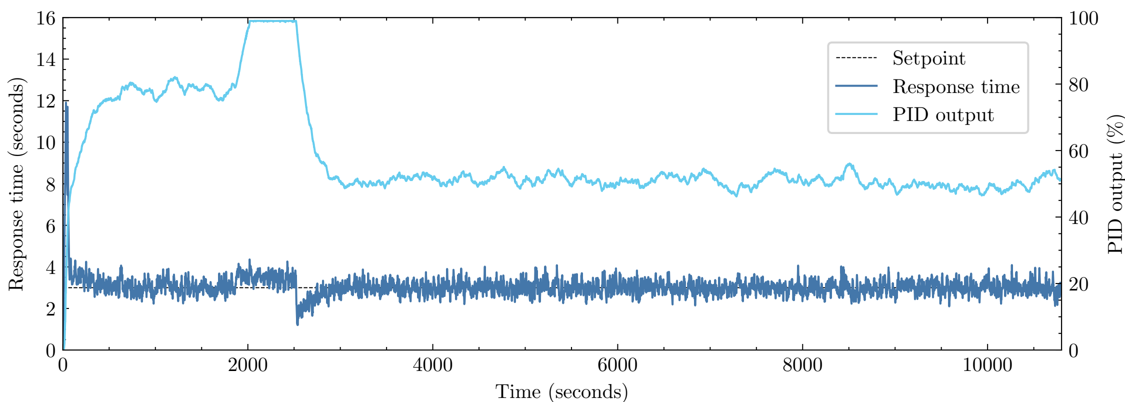


Figure 7.10: 95th percentile response time and PID controller graph for the online training correctness experiment.

Discussion. We see an overall improvement in weightings compared to baseline dimming without component weightings, hence the availability of components is increased with a minimal increase in PID output. We find that online training works in increasing the weighting of the `cart` component, which significantly reduces system load. Likewise, for `news`, which does not heavily contribute to load, online training successfully relaxes its weighting, making the optional component available to more users.

However, we see from fluctuations in the `recommender` component that online training is not a perfect solution. The reason for this observation is a combination of interactions between components, and noise in the control and candidate group response times. As the `recommender` component is made more available, the `cart` component is called more frequently over time due to heightened user interest in items, leading to marginally higher 95th percentile response times, and vice-versa. Since both groups use the same resources (such as `carts-db`), the control group response times are also affected, leading to additional noise. As the contribution to overall system load of making the `recommender` component more available is small but not negligible, statistical errors can occur which lead to the inability for weightings to settle over time.

Overall, online training works adequately: though the component weightings do not converge on those given in Listing 7, we still see an improvement where weightings for components which contribute more load are higher. Fluctuations and occasional errors do occur, however, these are expected as perfect online training would obviate the need for offline training.

7.4.6 Component Weightings: Online Training Robustness

Aim. To investigate whether online training is responsive to changes in workload mixes.

Method. Like in the correctness experiment, we run a slightly modified version of the constant load scenario described in Section 7.2.2 for 6 hours 10 seconds with 270 users and all initial weightings set to 0.5. To vary the workload, at 3 hours 10 seconds, we changed the load testing profile mix to a split of zero users on the *buying* profile, three-sevenths on *browsing* and four-seventh on *news*. We captured response time and PID output data, along with control and candidate group component weightings.

Results. Figure 7.11 shows changes to component weightings during the experiment with a vertical line representing the point of change in profile mixes. During the original profile mix, we see the `cart` and `news` components increased in weightings, while the `news` component stayed the same. During the later profile mix, we see that all component weightings ultimately experienced decreases of small magnitudes, though the `cart` component also experienced a brief increase in weightings. Figure 7.12 shows response time and PID output of the system.

Discussion. We find that online training is robust to changes in workloads. All weightings decreased in the second half of the experiment as the *buying* user profile, which places a high amount of load on the carts database, was no longer used. As the workload shifted towards the *browsing* and *news* profiles, the `cart` component was used less intensively, causing the 95th percentile response time to be lower and more stable. We are also confident that these observations are not due to error: although decreases in *recommender* and *news* workflows alone could be due to statistical error as discussed in the correctness experiment, seeing a decrease occur in `cart` weightings would not be plausible unless the decrease truly had no effect on the response time distribution.

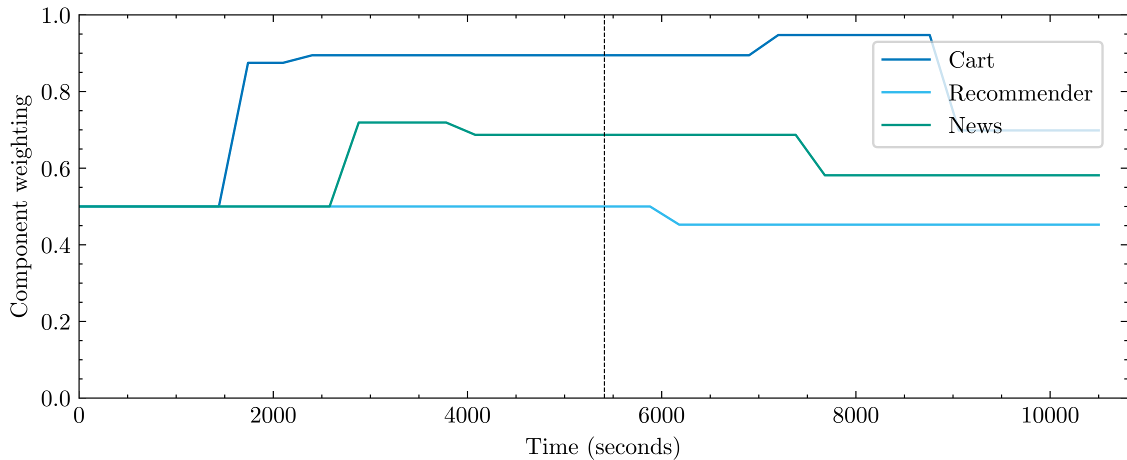


Figure 7.11: Control group component weightings for the online training robustness experiment.

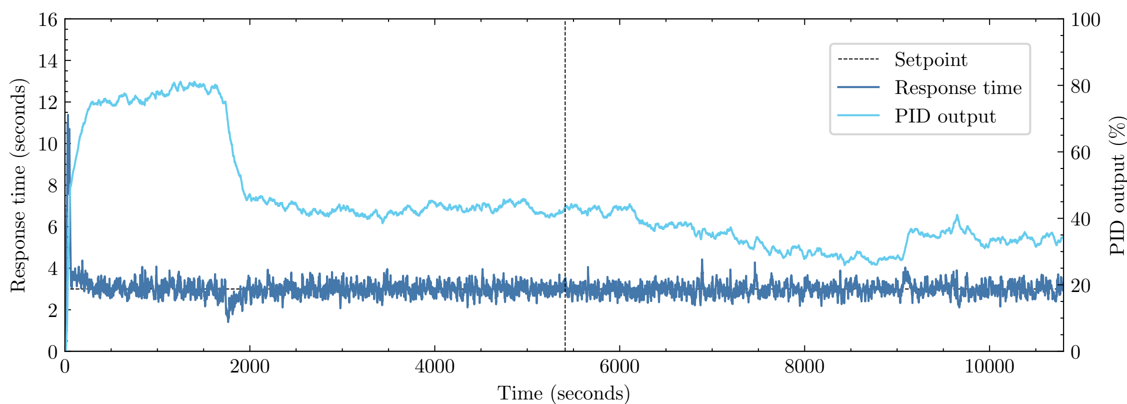


Figure 7.12: 95th percentile response time and PID controller graph for the online training robustness experiment.

7.4.7 Profiling: Behaviour

Aim. (A) To investigate the effect of the profiling strategy on response time and PID controller output. (B) To ensure that the edge case where the system must be dimmed when all requests are only of one priority is correctly handled.

Method. For Aim (A), we ran the constant load scenario as described in Section 7.2.2 once with profiling enabled and component weightings not set. For Aim (B), we ran this scenario again, however, we tested the high priority edge case by setting all users (a total of 160 instead of 280 to prevent saturation) to the *buying* profile, and we tested the low priority edge case by setting all users to the *browsing* profile. For both aims, we collected the response time, PID output, and priority assignments.

Results. Figure 7.13 shows the effect of running the constant load scenario with profiling enabled. Raw data shows a response time of $\mu = 2.97s, \sigma = 0.72s$ and PID output of $\mu = 37.6\%, \sigma = 2.3\%$ between 200 and 1810 seconds. Figure 7.14 shows profiles being assigned over time for the constant load scenario.

Figure 7.15 shows the effect of sending load with either all low priority or all high priority profiles on response times and PID output. Figure 7.16 shows profile assignments for these edge cases.

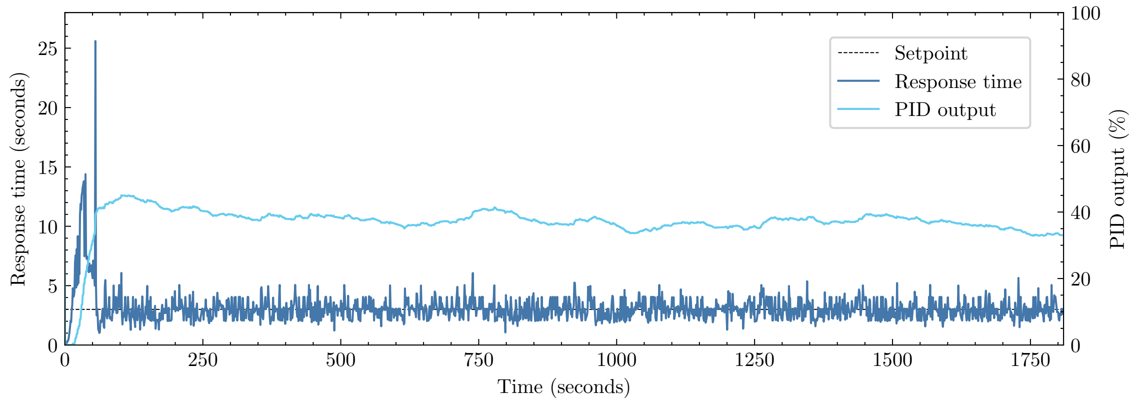


Figure 7.13: 95th percentile response time and PID controller graph for the constant load scenario, dimming with profiling, component weightings not set.

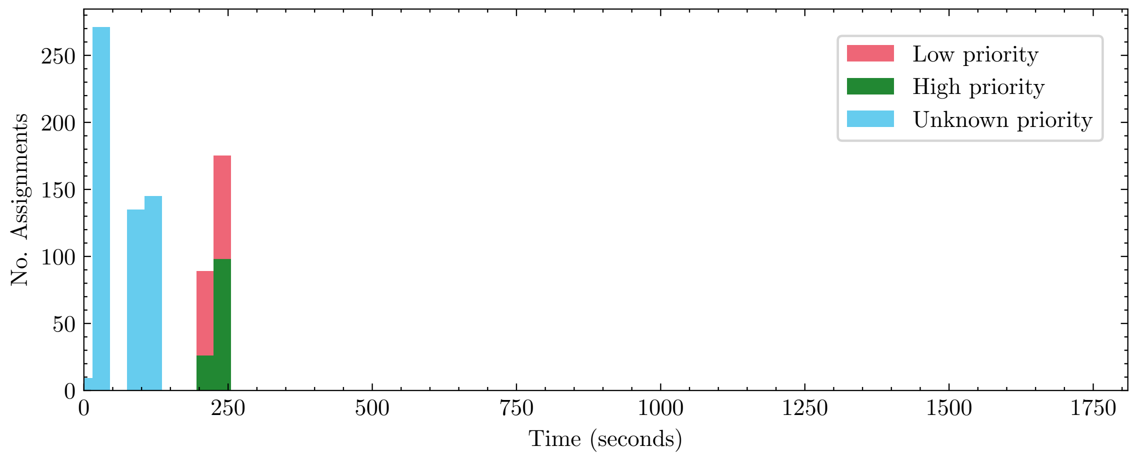


Figure 7.14: Profile assignments aggregated every 30 seconds for the constant load scenario, dimming with profiling, component weightings not set.

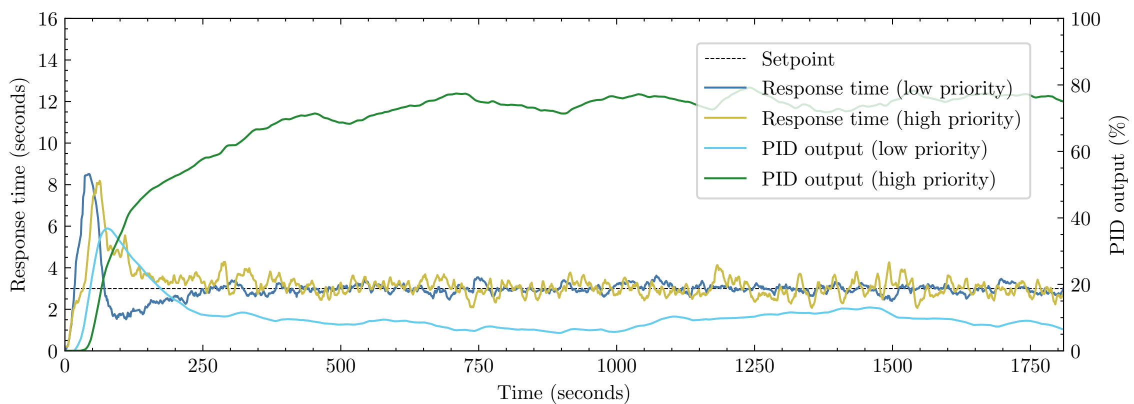
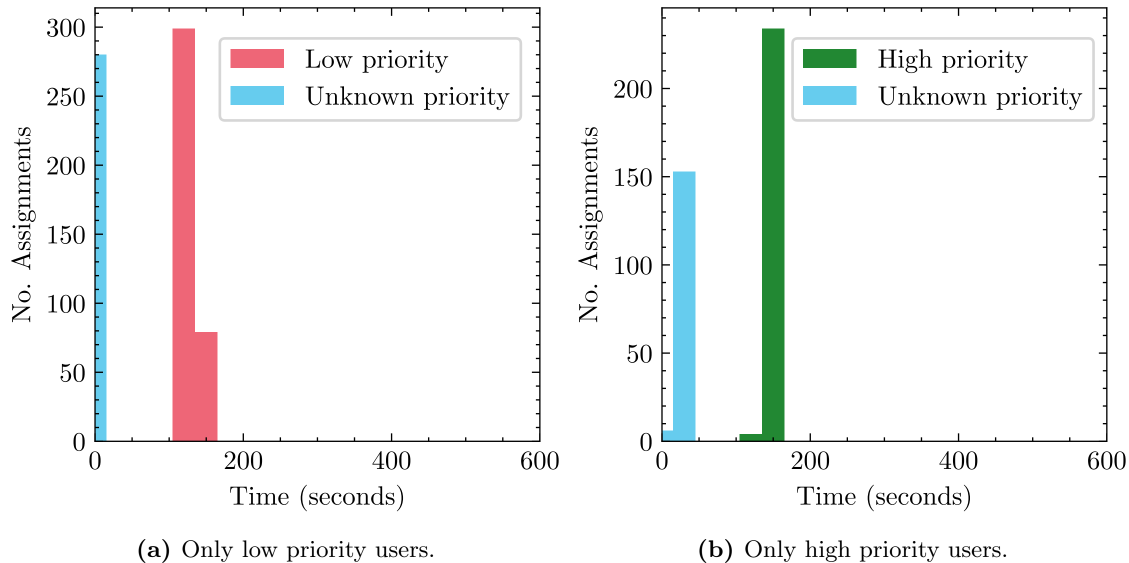


Figure 7.15: Profile assignments aggregated every 30 seconds for the constant load scenario for edge cases, dimming with profiling, component weightings not set. Smoothened with exponential weighted average over 20 seconds for readability.

Discussion. We see from the standard constant load run that the expected brownout behaviour is achieved. We note a slightly greater variance compared to baseline dimming as dimming decisions for profiled sessions last for two minutes, reducing the dimmer's responsiveness to the PID controller output. Profiling assignments also behaved as ex-



(a) Only low priority users.

(b) Only high priority users.

Figure 7.16: Profile assignments aggregated every 30 seconds for the constant load scenario edge cases, dimming with profiling, component weightings not set.

pected: unknown priorities are assigned at the start as the system has insufficient data for all sessions, and all sessions are successfully profiled as low or high priority once the unknown session assignment cookies expire. We note that the number of assignments was greater than the total number of users as duplicate session IDs were present in the profiling queue.

For the edge cases where all users are all either of low or high priority, dimming occurred as expected, validating that our dimming decision equation handles edge cases correctly.

7.4.8 Profiling: Improvement over Baseline Dimming

Aims and Hypothesis. To investigate whether dimming with profiling *without* component weightings set leads to an significant improvement in both (A) the number of items and (B) recommendations checked out over baseline dimming.

Method. We ran the constant load scenario described in Section 7.2.2 for five repeats with baseline dimming enabled (control group), then for five repeats with dimming with profiling enabled (candidate group), collecting the number of items and recommendations checked out to perform Welch's t-tests for our two hypotheses.

Results. Table 7.4 shows the results of running the constant load scenario for dimming with profiling. Due to an identical methodology step, we use our existing baseline dimming data from Table 7.2.

(A) The number of items checked out for dimming with profiling ($\mu = 2089.4.4, s = 89.9$) was not significantly higher than that of baseline dimming ($\mu = 2313.4, s = 83.5$), $t(6) = -3.85$, $p < .001$, one-tailed.

(B) The number of recommendations checked out for dimming with profiling ($\mu = 1056.0, s = 51.4$) was significantly higher than that of baseline dimming ($\mu = 887.4, s = 29.6$), $t(7) = 13.4$, $p < .001$, one-tailed.

Discussion. We observe a higher number of recommendations when dimming with profiling as opposed to baseline dimming. This is because once low and high priorities were assigned, an average of ~37% of all low priority users were always dimmed, while high pri-

Run	Items Checked Out		Attrition
	Total	of which Recommended	
1	2181	1091	1170
2	2042	1022	1238
3	2187	1121	1195
4	1986	993	1281
5	2051	1053	1209
avg	2089.4	1056.0	1218.6
s	89.9	51.4	42.7

Table 7.4: Shopping metrics for five constant load scenario runs, profile-based dimming, component weightings not set.

ority users were never dimmed. However, we observe a ~10% decrease in the total number of items checked out – a more important metric than the number of recommendations. One contributing reason for a lack of increase may be that users were assigned an unknown priority for the first three to four minutes of each experiment iteration, where the system behaved similarly to baseline dimming. However, this reason alone does not justify a decrease. We believe an additional factor was that the spike in 95th percentile response time to 25 seconds shown at 60 seconds in Figure 7.13 – not as prominent in other experiments – caused a significant number of users to attrite after a long wait. We remark that we adopted a specific profiling method for our experiments and different methods might perform differently.

7.4.9 Profiling: Combining with Component Weightings

Aim and Hypothesis. To investigate the effect of combining the component weightings strategy with profiling on response time and PID controller output, as well as the effect on business objectives.

Method. We ran the constant load scenario described in Section 7.2.2 with the profiling strategy and component weightings in Listing 7 set for five repeats, then did the same with profiling without component weightings set. We collected response time and PID output data, along with the number of items and recommendations checked out, in order to perform Welch t-tests.

Results. Table 7.5 and Figure 7.17 show the results of running the constant load scenario for profiling with component weightings set. Due to an identical methodology step, we used our existing profiling without component weightings data from Table 7.4.

There was no significant increase in the number of items checked out for profiling with component weightings set ($\mu = 2144.4, s = 51.6$) compared with profiling without component weightings set ($\mu = 2089.4, s = 89.9$), $t(6) = 1.187$, $p < 0.1$, one-tailed. On the other hand, the number of recommendations checked out for profiling with component weightings set ($\mu = 1146.4, s = 31.6$) was significantly higher than that of profiling without component weightings set ($\mu = 1056.0, s = 51.4$), $t(6) = 3.35$, $p = 0.01$.

Additionally, the PID output between 200 and 1810 seconds of the profiling with component weightings strategy ($\mu = 44.1\%, \sigma = 2.57\%, N = 1710$) in Figure 7.17 compared with that

of the profiling without component weightings strategy ($\mu = 37.6\%, \sigma = 2.26\%, N = 1710$) in Figure 7.13 showed a significant increase, $z = 78.5$, $p < .001$, one-tailed.

Run	Items Checked Out		Attrition
	Total	of which Recommended	
1	2192	1175	1169
2	2068	1100	1211
3	2128	1131	1221
4	2192	1174	1188
5	2142	1152	1194
avg	2144.4	1146.4	1196.6
s	51.6	31.6	20.3

Table 7.5: Shopping metrics for five constant load scenario runs, profile-based dimming, component weightings set.

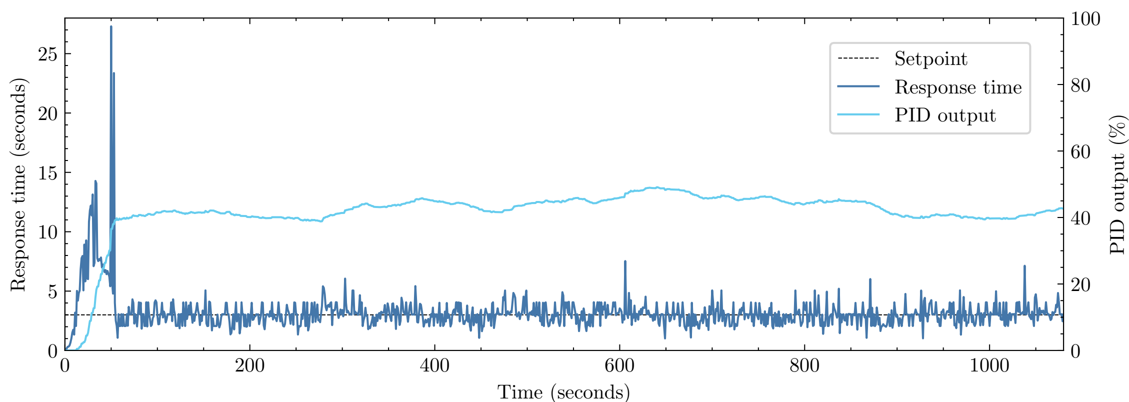


Figure 7.17: 95th percentile response time and PID controller graph for a single run of the constant load scenario, dimming with profiling, component weightings set.

Discussion. Despite the statistically significant increase in items checked out when combining profiling with component weightings, we conclude that profiling with component weightings set does not lead to an improvement over profiling without component weightings set. First, the absolute increase in recommendations checked out was small (<100) and the total number of items checked out, which did not show a significant increase, is the more important statistic. More importantly, there was a significant increase in PID output, leading to more optional components being dimmed for users, despite there not being a significant increase in the total number of items checked out. With heavier dimming leading to little increase in business objectives, combining profiling with component weightings does not appear to be a worthwhile strategy.

7.4.10 Comparing Component Weightings and Profiling

Aim. To investigate whether the component weightings and profiling strategies are equally effective.

Method. We used data for response time, PID output, and the number of items and recommendations checked out captured in the previous component weightings and profiling

strategy experiments to perform a two-tailed Welch t-test, informing the existence of a significant difference between the two strategies. For the profiling strategy, we used data for profiling without component weightings set, as we have demonstrated in the previous experiment that profiling is not improved when component weightings are set.

Results. The number of items checked out when dimming with component weightings ($\mu = 2698.4, s = 33.3$) compared with dimming with profiling ($\mu = 2089.4, s = 89.9$) showed a significant difference, $t(5) = 14.2, p < 0.001$, two-tailed. Likewise, the number of recommendations checked out when dimming with component weightings ($\mu = 1439.0, s = 20.7$) compared with dimming with profiling ($\mu = 1056.0, s = 51.4$) showed a significant difference, $t(5) = 15.4, p < 0.001$, two-tailed.

Additionally, the PID output between 200 and 1810 seconds of the component weightings strategy ($\mu = 36.6\%, \sigma = 1.7\%, N = 1710$) in Figure 7.8 compared with that of the profiling strategy ($\mu = 37.6\%, \sigma = 2.3\%, N = 1710$) in Figure 7.13 from the runs in our previous sections showed a significant difference, $z = 14.19$, $p < .001$, two-tailed.

Discussion. Our results show that profiling performed worse than component weightings. First, fewer items and recommendations were checked out during profiling. Second, there was a small increase in PID output during profiling, making optional components inaccessible for a larger number of users. Hence, component weightings can be viewed as a better strategy than profiling with our evaluation setup and specific profiling method.

However, our assessment only applies to testing with Sock Shop for Kubedim: in other environments, where there is no optional component that should be dimmed significantly more than others, profiling may be a stronger strategy, especially where clear user patterns and priorities can be established. Therefore, we consider component weightings and profiling to be alternatives, with the preferred strategy dependent on the similarity between component weightings and the ability to specify accurate profiling rules. Likewise, this assessment could change with more sophisticated profiling methods.

7.4.11 Conclusion

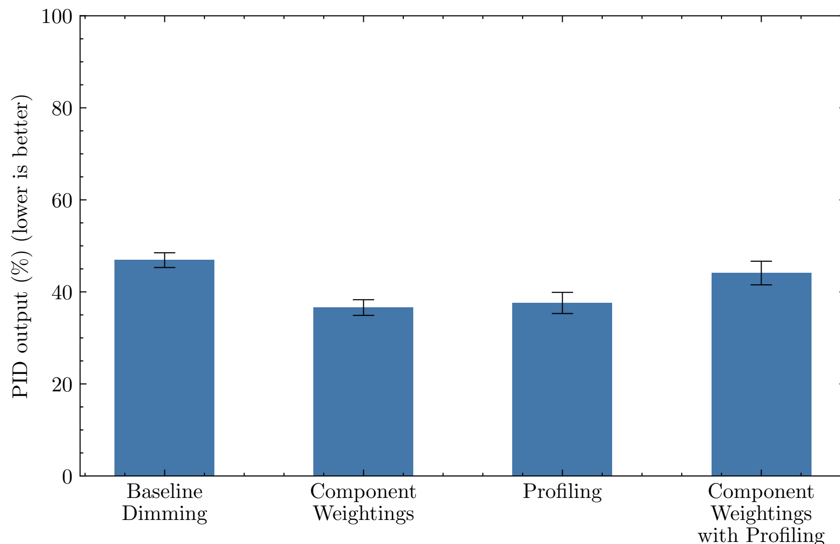


Figure 7.18: Comparison of PID output from the brownout strategy experiments.

We see from our behavioural experiments that the baseline, component weightings and profiling brownout strategies all follow the expected adaptive control behaviour of responding

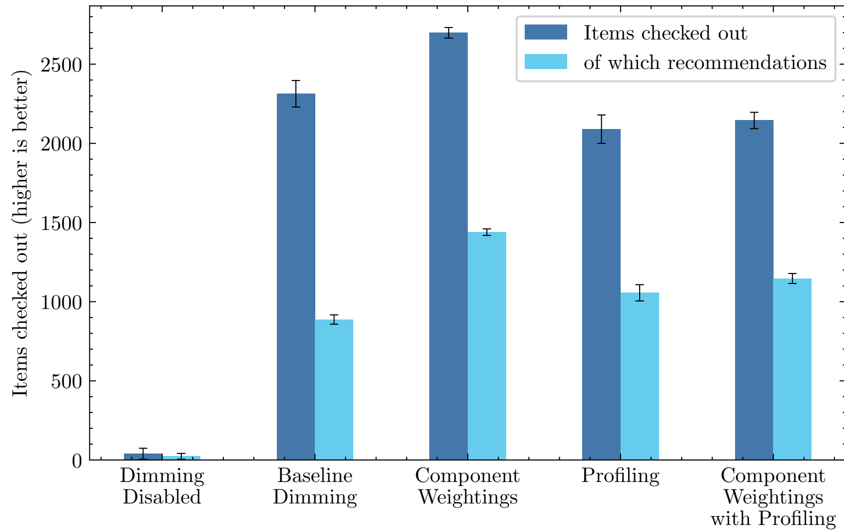


Figure 7.19: Comparison of number of items checked out from the brownout strategy experiments.

to response time setpoint violations with an acceptable settling time and minimal oscillations. Hence, all strategies significantly improve a system's ability to respond to high load compared to no dimming at all. As shown in Figure 7.18, the component weightings and profiling strategies improve upon baseline dimming by having a lower PID output, causing optional components to be displayed more frequently for users. We note that these positive results originate from optimal configurations of component weightings and profiling rules. Where component weightings are sub-optimal due to inability to exhaustively run offline training or changes in workload mixes, the online training mode can make appropriate adjustments to improve component weightings in production.

Using our e-commerce reference application, we have shown that all brownout strategies significantly improve business objectives compared to no dimming at all. These results are summarised in Figure 7.19. Likewise, we have shown that the component weightings strategy significantly improves upon business objectives compared to baseline dimming. However, in the instance of our reference application, profiling performs on-par with, but not better than, baseline dimming, likely due to the time taken for sessions to be profiled.

Finally, through combining and contrasting the component weightings and profiling strategies, we have found that combining the strategies is not effective due to a higher level of dimming despite no significant improvement in business objectives. Second, from contrasting component weightings against profiling, we have found that component weightings is the better strategy for Sock Shop; however, this result is not generalisable to wider applications: users of Kubedim should instead find the better strategy which works for them.

7.5 Developer Usability

In this section, we focus on evaluating the extent of which Kubedim meets the aim of being a drop-in solution with easy installation and configuration.

Our evaluation focuses on component weightings. For other aspects of Kubedim, from PID controller tunings to the expiry times of profiling cookies, we have set default parameter values which can reasonably apply to deployments outside of Sock Shop. Likewise, the configuration for our other brownout strategy, profiling, is rule-based and purely dependent

on developer knowledge, with instructions available in the user manual (Appendix A). In contrast, having accurate component weightings depends on running a training tool to capture black-box behaviour, so we will investigate the extent of which this strategy can be deployed with minimal time spent on configuration.

7.5.1 Component Weightings

Aim. To investigate the sensitivity of the PID output to changes in component weightings, thereby informing the extent of which using sub-optimal component weightings (e.g., due to an informed guess) would be effective.

Method. For each of the optimal components given in Listing 7, we fixed the value of the other two components to the optimal value, then varied the chosen component with a range from 0.0 to 1.0 inclusive with a step of 0.05. We ran the constant load scenario described in 7.2.2 for each combination, capping each iteration at 4 minutes 10 seconds instead of the usual 30 minutes 10 seconds to reduce runtime. We collected the 95th percentile response time and PID output for each iteration for qualitative analysis.

Results. Figure 7.20 shows the effect of running the above scenario with optimal component weightings, acting as a baseline for our analysis. Figures 7.21, 7.22 and 7.23 show the effects of varying weightings for the news, recommender and cart components respectively.

Figures 7.21 and 7.22 show that, regardless of weightings assigned to the news and recommender components, the PID output and response time remained at similar levels for these components.

On the other hand, Figure 7.23 shows that cart weighting assignments had a non-negligible effect on PID output. We observe that, with the optimal assignment for this component being 1.0, a small change in weighting to 0.95 did not show a significant difference in PID output. However, an assignment of 0.75 led to around a 20% increase in PID output. Finally, at the extreme end with an assignment of 0.0, the component became fully dimmed and the dimmer was unable to bring the response time to the setpoint, causing the system to saturate.

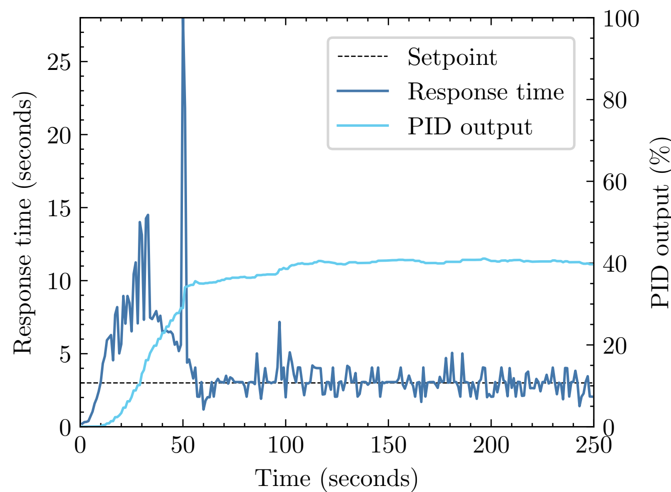


Figure 7.20: 95th percentile response time and PID output for the constant load scenario capped at 250 seconds, with component weightings from Listing 7 set.

Discussion. The results for varying the news and recommender component weightings are expected: both endpoints contributed negligible differences to the overall system load,

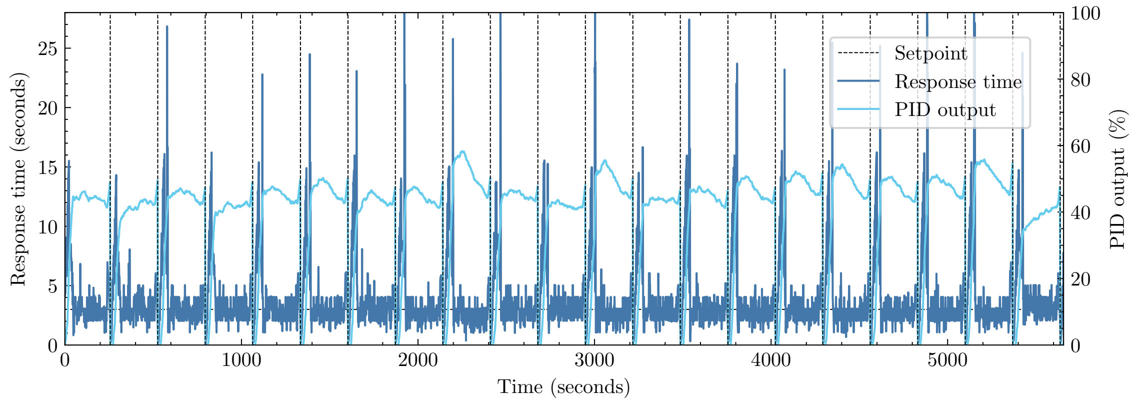


Figure 7.21: 95th percentile response time and PID output for the constant load scenario capped at 250 seconds, with optimal recommender and cart weightings, and variable news weightings.

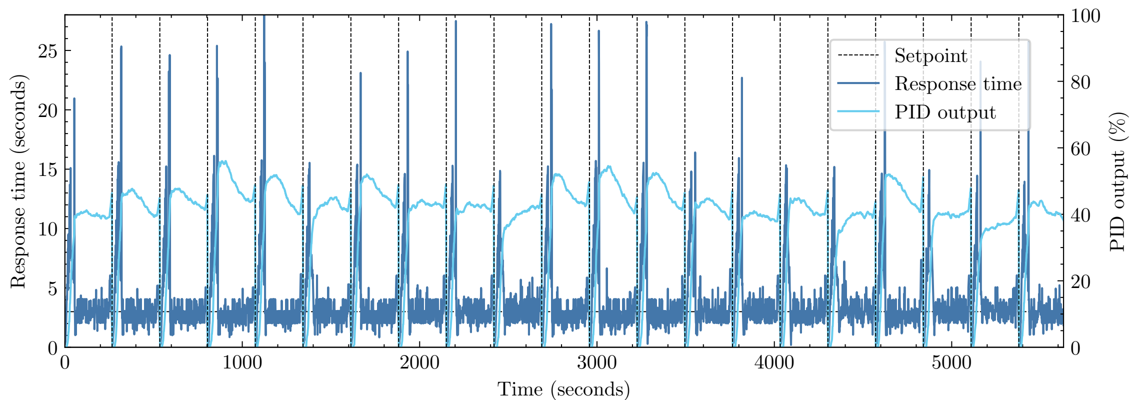


Figure 7.22: 95th percentile response time and PID output for the constant load scenario capped at 250 seconds, with optimal news and cart weightings, and variable recommender weightings.

so varying their weightings did not have a major effect on PID output and response time. Hence, low load components are not sensitive to changes in component weightings. In contrast, varying the cart component weightings led to significant changes in response time and PID output. This suggests that sensitivities of weightings correlate with components' contributions to system load.

This correlation relates to the usability of Kubedim as developers may be able to see the contribution of components to system load, as a core aim of microservices is that components are decoupled from each other [11]. If this aim is followed, then developers will be able to identify high load components with sub-optimal component weightings from metrics such as CPU usage and increase weightings appropriately.

Relying on the decoupling of microservices may not be a completely effective strategy in overcoming sub-optimal component weightings. We have seen from Eismann et al. [29] and Ackermann et al. [30] that in complex, real-world systems, the performance of components can depend on each other and change under different workload mixes, and user behaviour may change with component availability, so it may not be clear how components correlate with high load. In this case, we may be able to rely on our use of a PID controller to maintain resilience to load with sub-optimal weighting assignments. In Figure 7.23, we see that the PID output reliably increased with lower assignments, validating the controller's robustness to sub-optimal weightings.

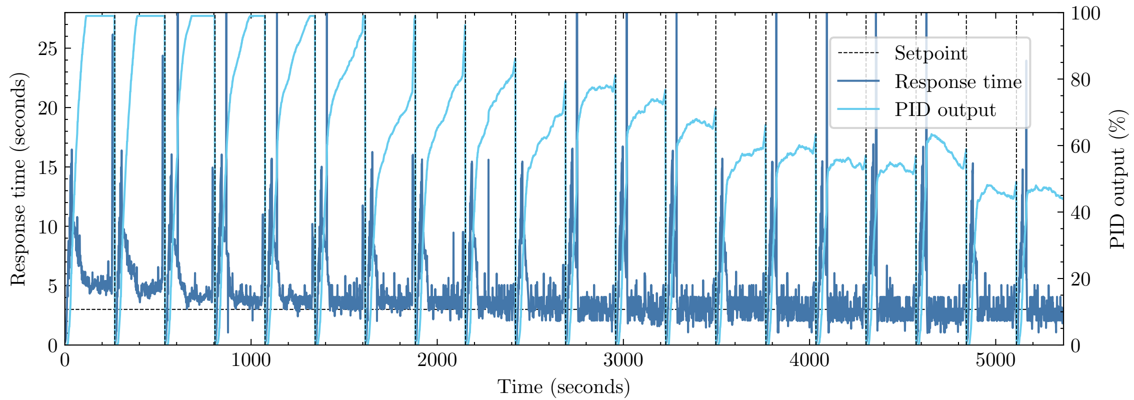


Figure 7.23: 95th percentile response time and PID output for the constant load scenario capped at 250 seconds, with optimal recommender and news weightings, and variable cart weightings.

In the worst case scenario where weighting assignments cause the system to saturate under high load, as seen with lower assignments in Figure 7.23, the ability for developers to diagnose the responsible components with sub-optimal assignments may be affected due to system unavailability. This behaviour can be mitigated by enabling online training, where less sub-optimal weightings may be discovered under medium load before the system saturates. Alternatively, developers can perform a full run of offline training over the course of several hours to obtain optimal weightings. In all cases, the increase in system stability will be better than no dimming at all.

Conclusion. We conclude that Kubedim has a high level of developer usability as more involved steps of performing offline training are only required in worst case scenarios. In some real-world use cases, sub-optimal component weightings can easily be detected as component weightings correlate with system load in architectures where microservices are well-decoupled from each other, hence educated guesses in weightings would suffice. In other cases where component interactions are much less trivial, sub-optimal component weightings can still suffice due to the robustness of the PID controller. Where weightings are non-optimal and the system is at risk of saturation even with full dimming, developers must either enable online training or performing a full run of the offline training tool in order to obtain adequately optimal component weightings. In all cases, Kubedim increases system stability compared to no dimming at all and provides tooling to handle all cases.

Our results give promising indications that Kubedim configurations are robust enough not to require developers to have a deep knowledge of control theory or performance modelling. However, we also remark that an empirical usability experiment with developers from industry would be an interesting direction to further assess Kubedim's usability.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

We have presented Kubedim, a self-adaptive reverse proxy enabling the orchestration of brownout techniques on optional components of a system deployed with Kubernetes. Like prior work, Kubedim operates on the load balancer level, allowing straightforward installation in a Kubernetes cluster without modification of existing source code.

In particular, Kubedim focuses on balancing improving the stability of a service with improving its business objectives. Beyond the baseline strategy of dimming all optional components uniformly as done by previous work such as Kotegov and Filieri [6], Kubedim introduces the following strategies which allow components to be dimmed non-uniformly, increasing the availability of optional components such as advertisements and recommendations:

- **Component Weightings:** This strategy allows developers to set probabilities that each optional component will be dimmed, where the baseline strategy is equivalent to this strategy with all probabilities set to 1. As optimal probability assignments can be non-trivial due to interactions between components as their availabilities are changed, Kubedim provides an *offline training* tool enabling a developer to automatically find optimal parameters by training a linear regression model on a non-production environment over a number of hours. Where these probability assignments become sub-optimal, either due to workload mixes changing or comprehensive offline training over the course of several hours being infeasible, we provide an *online training* mode to improve probability assignments in production based on A/B testing principles.
- **Profiling:** This strategy dims users based on priorities, where optional components are consistently available to high priority users and unavailable to low priority users. Users priorities are profiled by a set of developer-supplied rules based on their understanding of user flows. This consistency improves user experience and ensures component availability is tailored towards those who are likely to contribute more towards business objectives.

In our implementation, we have overcome the challenge of mitigating side effects such as bottleneck transfers due to interactions between components by taking a black-box linear regression approach during offline training for the component weightings strategy, and allowing weights to be continuously updated during online operation to reflect the impact of each endpoint in the current operation conditions and workload. This approach also increases developer usability, as highly theoretical performance modelling knowledge is not required. In order to achieve applicability to industry use-cases, we have also carefully

tuned parameters to reduce the amount of configuration needed, while Kubedim retains the flexibility for advanced users to adjust the tuning as desired.

In evaluating Kubedim, we have first modified Sock Shop, a popular e-commerce reference application by adding optional components and significantly improving the reproducibility of load testing experiments. We have open-sourced these changes, allowing the research and developer communities to experiment with it and reproduce our work. Using this reference application, we have shown that our strategies show significant improvements in resilience and business objectives compared to a system without dimming. Likewise, we have shown that component weightings performs significantly better than baseline dimming in both objectives, and that profiling performs on-par with baseline dimming. We conclude that dimming by component weightings is most effective and usable by developers for systems which allow for feasible offline training by having a small number of optional components, predictable workload mixes and some components which contribute significantly more load than others. In lieu of these features, developers may achieve better results with the profiling strategy, although this strategy did not prove to be significantly better than baseline dimming with Sock Shop.

We have also demonstrated positive preliminary findings about the developer usability of Kubedim, concluding that Kubedim has a high level of usability due to the ease of which strategies can be configured. In particular, we have concluded that the component weightings strategy does not necessarily require hours-long running of offline training to obtain adequate results for systems with a low level of feature interactions. However, where sub-optimal component weightings are inadequate, we provide both offline and online training which can improve weightings with a small number of additional steps.

8.2 Future Work

Limitations of Kubedim which would have been resolved with more time, along with future extensions, are discussed as follows:

- **Evaluating on further reference applications.** A key limitation with our evaluation is that, due to the time required to set up a reference application and and ensure reproducibility of load test experiments, we have only tested Kubedim on Sock Shop. While Sock Shop is a benchmark widely used by the performance engineering community, and it has been developed by several parties to be representative of an industrial deployment, testing Kubedim on more reference applications would enable a more thorough evaluation of brownout strategies, particularly as different applications have different subtle feature interactions, and allow a better understanding of whether a component weightings or profiling strategy would be more suited to an arbitrary application.
- **Forecasting and heuristics in online training.** Online training currently uses heuristics based on A/B testing principles. With more time, we could investigate alternative approaches, such as involving ARIMA to perform workload forecasting or k-means clustering to allow adjustment of component weightings to multiple different workload mixes, or implementing online optimisation methods for faster convergence to near-optimal component weightings.
- **Investigating combination with orthogonal techniques.** Following investigations by Kotegov and Filieri [6] and Dürango et al. [26] in combining brownout with auto-scaling and load balancing algorithms respectively, we would like to understand how Kubedim behaves with similar orthogonal techniques, particularly with auto-scaling and load balancing technologies with popular cloud Kubernetes providers

such as Google Cloud Platform and Amazon Web Services, in order to increase the industrial applicability of our work.

- **Investigating alternative profiling techniques.** Though our profiling technique has been designed to rely on developer knowledge without any need for model training, we would like to investigate means to make the profiling strategy more performant, reducing the time taken to profile a session and reducing the reliance on accurate developer-specified rules to successfully classify a large proportion of users. Likewise, different profiling strategies may prove more effective than ours, in particular on broader user populations and richer profiling facets. For example, we may be able to delegate profiling to the client-side through Google's recent Federated Learning of Cohorts proposal [48]. Alternatively, we could also investigate lowering the barrier to entry of creating analytics pipelines and machine learning models for more automated and adaptable user profiling.

8.3 Ethical Considerations

In this section, we consider implications for data use and software licensing. We do not believe there are other significant ethical considerations to discuss.

The primary ethical consideration is the possibility of requiring personal or sensitive data to be collected, both with our implementation and with proposed future work for the profiling strategy. Profiling users involves keeping a log of actions they have taken, such as page visits in our implementation. These actions may associate with more sensitive topics, such as the purchase of medicine in an e-commerce environment. Though we store an association between session IDs and page visits instead of storing user information, session IDs can be linked against users, de-anonymising them and causing privacy concerns.

User behaviour logging is a common issue in website analytics, where mitigations involve requiring users to consent and opt-in to tracking, encrypting data at rest and setting appropriate data retention policies, in line with legislation such as GDPR. Even if profiling solely for purposes of improving service availability may be considered to lie under the "strictly necessary" exemption and opt-in consent may not be required under the GDPR and related regulations [49], developers using Kubedim have an ethical responsibility to clearly inform users of this logging.

One potential mitigation for this ethical concern is to perform profiling on the client-side. From Google's Federated Learning of Cohorts proposal [48] to advancements in on-device machine learning [50], future work could shift profiling to the user's device, passing only the result to Kubedim.

We also identify a minor ethical consideration in using and publishing open source work. First, we have licensed our codebase under the MIT License in order to allow others to modify and distribute our work. Likewise, we have ensured that our codebase has used libraries with permissive open source licenses. When modifying and redistributing work, as is the case with our modifications to Sock Shop, we have ensured compliance with the appropriate licenses.

Bibliography

[1] Fowler SJ. Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization. In: Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization. 1st ed. O'Reilly Media, Inc.; 2017. Available from: https://www.oreilly.com/library/view/production-ready-microservices/9781491965962/ch04.html.

[2] Richardson C. Circuit Breaker [Internet]; 2017. [Accessed 17th January 2021]. Available from: https://microservices.io/patterns/reliability/circuit-breaker.html.

[3] Klein C, Maggio M, Arzén KE, Hernández-Rodriguez F. Brownout: Building more robust cloud applications. In: Proceedings - International Conference on Software Engineering. 1; 2014. p. 700–711.

[4] Kotegov I, Filieri A. Towards coordinated autoscaling and application brownout at the orchestrator level. In: Communications in Computer and Information Science. vol. 1269 CCIS. Springer Science and Business Media Deutschland GmbH; 2020. p. 269–274. Available from: https://doi.org/10.1007/978-3-030-59155-7_21.

[5] Filieri A, Hoffmann H, Maggio M. Automated multi-objective control for self-adaptive software design. 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings. 2015:13–24.

[6] Kotegov I. Model-free Control for Adaptive Software: The Brownout. Imperial College London; 2020.

[7] Weaveworks, Inc. Microservices Demo: Sock Shop [Internet]; 2021. [Accessed 14th June 2021]. Available from: https://microservices-demo.github.io/.

[8] Villamizar M, Garcés O, Castro H, Verano M, Salamanca L, Gil S. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. 10th Computing Colombian Conference. 2015:583–590.

[9] Richardson C. Microservice Architecture pattern [Internet]; 2017. [Accessed 25th May 2021]. Available from: https://microservices.io/patterns/microservices.html.

[10] Zimmermann O. Microservices tenets: Agile approach to service development and deployment; 2017. 3-4.

[11] Pahl C, Brogi A, Soldani J, Jamshidi P. Cloud Container Technologies: a State-of-the-Art Review. IEEE Transactions on Cloud Computing. 2017;7(3):677–692.

[12] Shah J, Dubaria D. Building modern clouds: Using docker, kubernetes google cloud platform. 2019 IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC 2019. 2019:184–189.

[13] Rawdat A. Announcing NGINX Ingress Controller for Kubernetes Release 1.8.0 - NGINX [Internet]; 2020. [Accessed 25th May 2021]. Available from: https://www.nginx.com/blog/announcing-nginx-ingress-controller-for-kubernetes-release-1-8-0/.

[14] Ding J, Cao R, Saravanan I, Morris N, Stewart C. Characterizing Service Level Objectives for Cloud Services: Realities and Myths. In: 2019 IEEE International Conference on Autonomic Computing (ICAC); 2019. p. 200–206.

[15] Mayr A, Putz P, Gerber A, Wallerstorfer D. Cloud-Native Evolution. In: Cloud-Native Evolution. O'Reilly Media, Inc.; 2017. Available from: https://www.oreilly.com/library/view/cloud-native-evolution/9781492048794/ch04.html.

[16] Filieri A, Maggio M, Angelopoulos K, D'Ippolito N, Gerostathopoulos I, Hempel AB, et al. Software Engineering Meets Control Theory. In: Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015; 2015. p. 71–82.

[17] Salehie M, Tahvildari L. Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems. 2009;4(2).

[18] Filieri A, Maggio M, Angelopoulos K, D'Ippolito N, Gerostathopoulos I, Hempel AB, et al.. Control strategies for self-adaptive software systems; 2017.

[19] Owen AB. A randomized Halton algorithm in R; 2017. Available from: http://arxiv.org/abs/1706.02808.

[20] Hyndman RJ, Athanasopoulos G. Least squares estimation. In: Forecasting: principles and practice. 2nd ed. Melbourne, Australia: OTexts; 2018. Available from: https://otexts.com/fpp2/least-squares.html.

[21] Apte R, Hu L, Schwan K, Ghosh A. Look Who's Talking: Discovering Dependencies between Virtual Machines Using CPU Utilization. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. HotCloud'10. USA: USENIX Association; 2010. p. 17.

[22] Giles D. Testing for Granger Causality [Internet]; 2011. [Accessed 12th January 2021]. Available from: https://davegiles.blogspot.com/2011/04/testing-for-granger-causality.html.

[23] Thalheim J, Rodrigues A, Akkus IE, Bhatotia P, Chen R, Viswanath B, et al. Sieve: Actionable insights from monitored metrics in distributed systems. Middleware 2017 - Proceedings of the 2017 International Middleware Conference. 2017:14–27.

[24] Crualaoich DO. The Kolmogorov-Smirnov Test — Kolmogorov-Smirnov [Internet]; 2016. [Accessed 25th May 2021]. Available from: https://daithiocrualaoich.github.io/kolmogorov_smirnov/.

[25] Vilaplana J, Solsona F, Teixidó I, Mateo J, Abella F, Rius J. A queuing theory model for cloud computing. Journal of Supercomputing. 2014 4;69(1):492–507.

[26] Dürango J, Dellkrantz M, Maggio M, Klein C, Papadopoulos AV, Hernández-Rodriguez F, et al. Control-theoretical load-balancing for cloud applications with brownout. In: 53rd IEEE Conference on Decision and Control; 2014. p. 5320–5327.

[27] Klein C, Papadopoulos AV, Dellkrantz M, Dürango J, Maggio M, Årzén KE, et al. Improving Cloud Service Resilience Using Brownout-Aware Load-Balancing. In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems; 2014. p. 31–40.

[28] Xu M, Buyya R. Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions. ACM Computing Surveys. 2019;52(1). Available from: https://doi.org/10.1145/3234151.

[29] Eismann S, Walter J, Von Kistowski J, Kounev S. Modeling of Parametric Dependencies for Performance Prediction of Component-Based Software Systems at Run-Time. Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018. 2018:135–144.

[30] Ackermann V, Eismann S, Grohmann J, Kounev S. Black-box learning of parametric dependencies for performance models. In: CEUR Workshop Proceedings. vol. 2245; 2018. p. 78–86. Available from: https://github.com/Olifee/automatic_dependency_characterization.

[31] Grohmann J, Eismann S, Elflein S, Kistowski JV, Kounev S, Mazkatli M. Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques. In: 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS); 2019. p. 309–322.

[32] Bauer A, Lesch V, Versluis L, Ilyushkin A, Herbst N, Kounev S. Chamulteon: Coordinated Auto-Scaling of Micro-Services. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS); 2019. p. 2015–2025.

[33] Shah SY, Yuan Z, Lu S, Zerfos P. Dependency analysis of cloud applications for performance monitoring using recurrent neural networks. 2018:1534–1543.

[34] Kim S, Kim JS, Hwang S, Kim Y. An allocation and provisioning model of science cloud for high throughput computing applications. ACM International Conference Proceeding Series. 2013.

[35] Gias AU, Casale G, Woodside M. ATOM: Model-driven autoscaling for microservices. In: Proceedings - International Conference on Distributed Computing Systems. vol. 2019-July. Institute of Electrical and Electronics Engineers Inc.; 2019. p. 1994–2004.

[36] Von Kistowski J, Eismann S, Schmitt N, Bauer A, Grohmann J, Kounev S. TeaStore: A micro-service reference application for benchmarking, modeling and resource management research. Proceedings - 26th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2018. 2018:223–236.

[37] Zhou X, Peng X, Xie T, Sun J, Xu C, Ji C, et al. Benchmarking microservice systems for software engineering research. Proceedings - International Conference on Software Engineering. 2018:323–324.

[38] Verreydt S, Beni H, Truyen E, Lagaisse B, Joosen W. Leveraging Kubernetes for adaptive and cost-efficient resource management. 2019. Available from: https://doi.org/10.1145/3366615.3368357.

[39] Kohavi R, Longbotham R. Online Controlled Experiments and A/B Testing. In: Encyclopedia of Machine Learning and Data Mining. Springer US; 2017. p. 922–929. Available from: https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7687-1_891.

[40] Andrawos M, Helmich M. Cloud Native Programming with Golang: Develop Microservice-Based High Performance Web Apps for the Cloud with Go. Packt Publishing; 2017.

[41] Bezrąk K, Przyłucki S. Impact of the cloud application programming language on the performance of its implementation in selected serverless environments; 2020. Available from: https://ph.pollub.pl/index.php/jcsi/article/view/1572.

[42] DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, et al. Dynamo: Amazon's highly available key-value store. In: Operating Systems Review (ACM). New York, New York, USA: ACM Press; 2007. p. 205–220. Available from: http://portal.acm.org/citation.cfm?doid=1294261.1294281.

[43] Wescott T. PID Without a PhD. Embeded Systems Programming. 2000:86–108.

[44] ThinkWithGoogle. Mobile site load time statistics - Think with Google [Internet]; 2018. [Accessed 3rd March 2021]. Available from: https://www.thinkwithgoogle.com/data/mobile-site-load-time-statistics/.

[45] Åström KJ, Murray RM. Feedback Systems: An Introduction for Scientists and Engineers; 2009. Available from: http://press.princeton.edu/titles/8701.html.

[46] Treat T. Guidelines for Chaos Engineering, Part 2 [Internet]; 2020. [Accessed 25th May 2021]. Available from: https://blog.realkinetic.com/guidelines-for-chaos-engineering-part-2-ad0be582ff12.

[47] Yuan X. A Brief Analysis on the Implementation of the Kubernetes Scheduler - Alibaba Cloud Community [Internet];. [Accessed 25th May 2021]. Available from: https://www.alibabacloud.com/blog/a-brief-analysis-on-the-implementation-of-the-kubernetes-scheduler_595083.

[48] Bindra C. Building a privacy-first future for web advertising [Internet]; 2021. [Accessed 25th May 2021]. Available from: https://blog.google/products/ads-commerce/2021-01-privacy-sandbox/.

[49] ICO. What are the rules on cookies and similar technologies? | ICO [Internet]; 2021. [Accessed 1st June 2021]. Available from: https://ico.org.uk/for-organisations/guide-to-pecr/guidance-on-the-use-of-cookies-and-similar-technologies/what-are-the-rules-on-cookies-and-similar-technologies/.

[50] Dhar S, Guo J, Liu J, Tripathi S, Kurup U, Shah M. On-Device Machine Learning: An Algorithms and Learning Theory Perspective. 2019 11. Available from: http://arxiv.org/abs/1911.00623.

Appendix A

User Manual

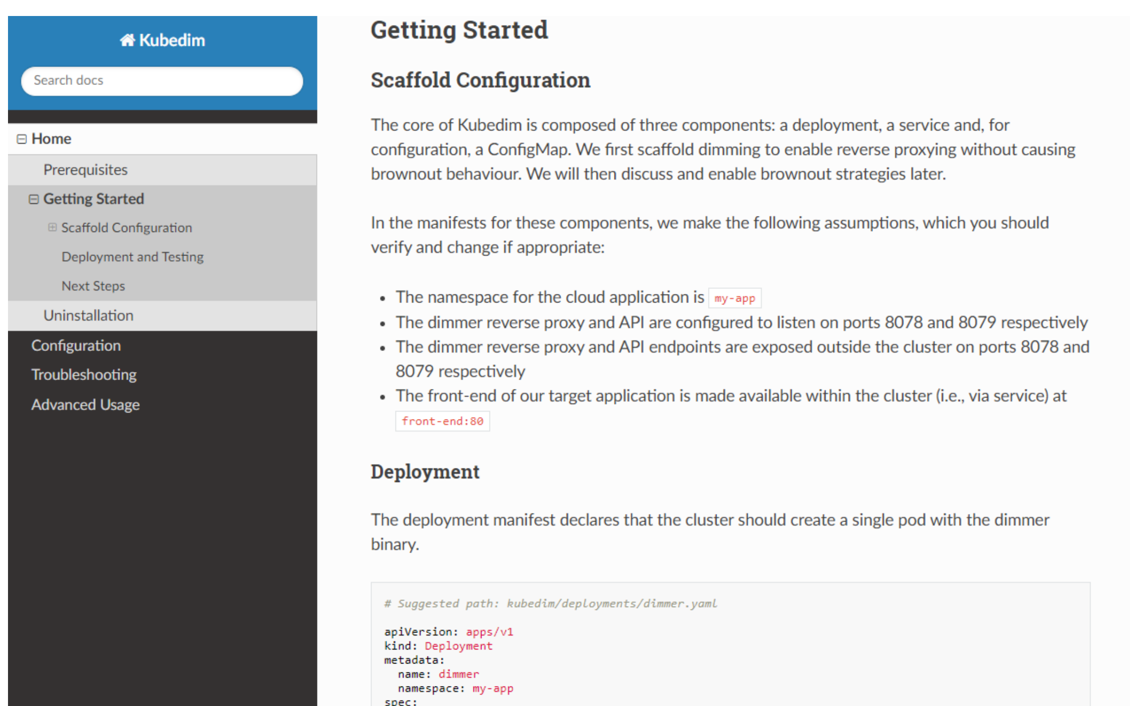


Figure A.1: A screenshot of an excerpt of the user manual.

Supplementary Data File

Description: As shown in Figure A.1, Kubedim includes a user manual instructing installation and usage for a developer audience.

Uploaded file path: `docs/`

Additional notes: A live version can be found online at `https://kubedim.readthedocs.io/`.

Appendix B

Sock Shop for Kubedim Configuration

B.1 Main Application Configuration

Supplementary Data File

Description: Kubernetes YAML configuration files for the namespace, deployments and services used to deploy Sock Shop for Kubedim.

Uploaded file path: `manifests/sockshop/`

Additional notes: A live version can be found online at `https://github.com/kcz17/manifests/tree/main/sockshop`.

B.2 Kubedim Configuration

Supplementary Data File

Description: Kubernetes YAML configuration files used to deploy Kubedim for the Sock Shop application.

Uploaded file path: `manifests/kubedim/`

Additional notes: A live version can be found online at `https://github.com/kcz17/manifests/tree/main/kubedim`.

B.3 Offline Training Tool Configuration

Supplementary Data File

Description: Configuration file for the offline training tool.

Uploaded file path: `train/config.yaml`

Additional notes: A live version can be found online at `https://github.com/kcz17/train/blob/master/config.yaml`.

Appendix C

Saturation Experiments

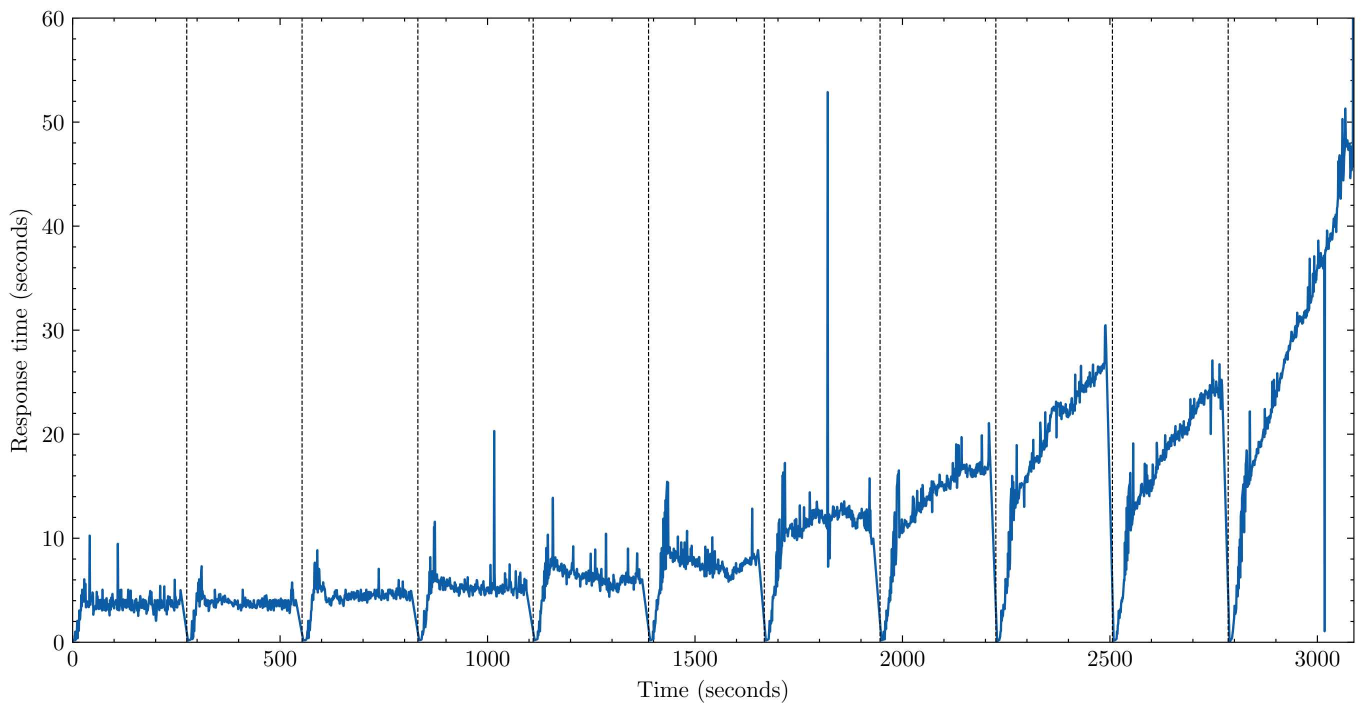


Figure C.1: Response time graph of repeated constant load experiments with dimming disabled, with the number of users set from 230 to 330 inclusive with intervals of 10.

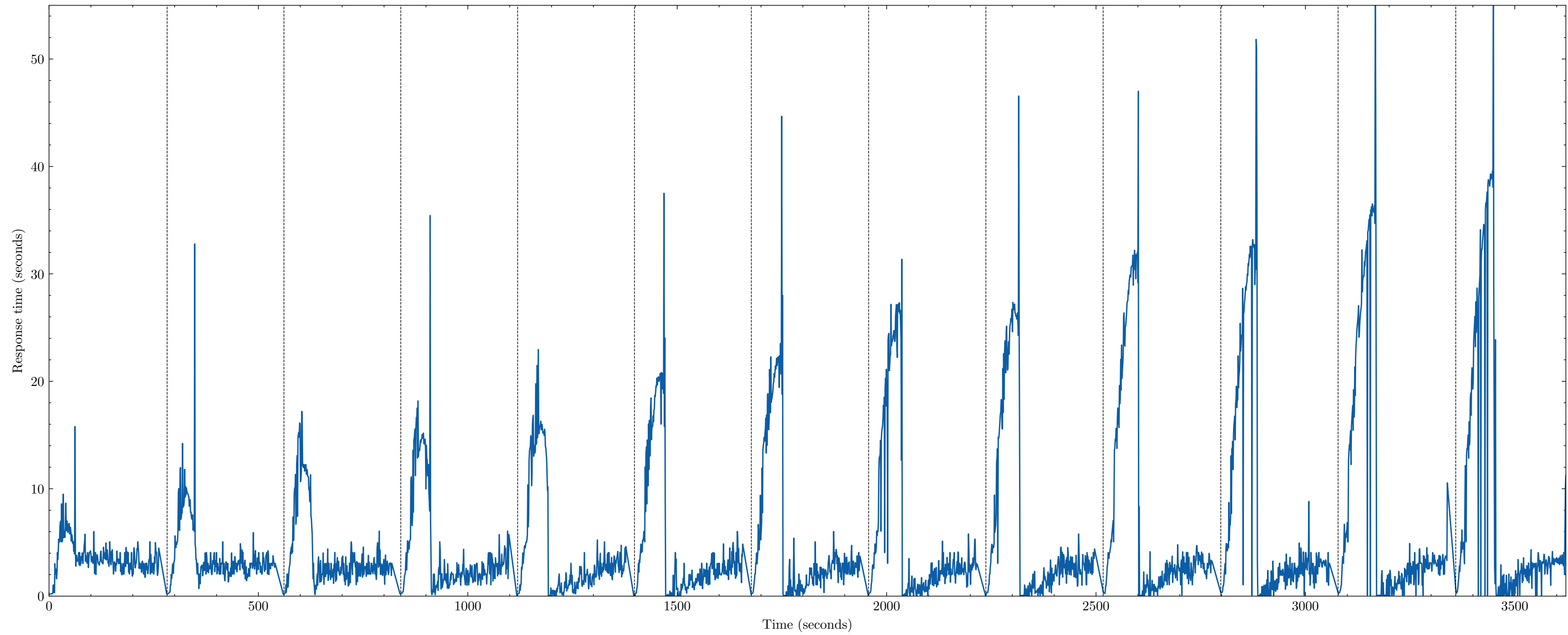


Figure C.2: Response time graph of repeated constant load experiments with baseline dimming, with the number of users set from 260 to 500 inclusive with intervals of 20.

Appendix D

Brownout Strategy Experiments

D.1 Dimming Disabled

Supplementary Data File

Description: 95th percentile response time data for the constant load scenario with dimming disabled, used for Figure 7.5.

Uploaded file paths: `experiments/notebooks/data/constant_load_disabled_p95_2021-05-13-17-04_chronograf_data.csv`

Supplementary Data File

Description: 50th, 75th and 95th percentile response time data for the constant load scenario with dimming disabled with five repeats, used for Table 7.1.

Uploaded file path: `experiments/notebooks/data/constant_load_repeats_dimming_disabled_response_time_2021-05-10-21-38_chronograf_data.csv`

D.2 Baseline Dimming

Supplementary Data Files

Description: PID output, and 50th, 75th and 95th percentile response time data for the constant load scenario with baseline dimming enabled, used for Figure 7.6.

Uploaded file paths:

- `experiments/notebooks/data/constant_load_pid_chronograf_data.csv`
- `experiments/notebooks/data/constant_load_p{50|75|95}_2021-05-10-12-14_chronograf_data.csv`

Supplementary Data Files

Description: PID output, and 50th, 75th and 95th percentile response time data for the flash crowd scenario with baseline dimming enabled, used for Figure 7.7.

Uploaded file path: `experiments/notebooks/data/flash_crowd_2021-05-13-19-27_chronograf_data.csv`

Supplementary Data Files

Description: PID output, and 50th, 75th and 95th percentile response time data for the constant load scenario with baseline dimming enabled for five repeats, used for Table 7.2.

Uploaded file paths:

- `experiments/notebooks/data/constant_load_repeats_baseline_repeats_pid_output_2021-05-10-21-43_chronograf_data.csv`
- `experiments/notebooks/data/constant_load_repeats_baseline_repeats_response_time_2021-05-10-21-42_chronograf_data.csv`

D.3 Component Weightings Strategy

Supplementary Data Files

Description: PID output, and 50th, 75th and 95th percentile response time data for the constant load scenario with dimming with component weightings enabled for five repeats, used for Table 7.3.

Uploaded file paths:

- `experiments/notebooks/data/constant_load_repeats_weightings_repeats_pid_output_2021-05-11-15-07_chronograf_data.csv`
- `experiments/notebooks/data/constant_load_repeats_weightings_repeats_response_times_2021-05-11-15-06_chronograf_data.csv`

Supplementary Data Files

Description: Probability assignments, PID output, and 50th, 75th and 95th percentile response time data for the online training correctness experiment in Section 7.4.5.

Uploaded file paths:

- `experiments/notebooks/data/online_training_correctness_probabilities_2021-05-27-12-16_chronograf_data.csv`
- `experiments/notebooks/data/online_training_correctness_pid_and_response_times_2021-05-27-12-15_chronograf_data.csv`

Supplementary Data Files

Description: Probability assignments, PID output, and 50th, 75th and 95th percentile response time data for the online training robustness experiment in Section 7.4.6.

Uploaded file paths:

- `experiments/notebooks/data/online_training_robustness_probabilities_2021-05-27-16-38_chronograf_data.csv`
- `experiments/notebooks/data/online_training_robustness_pid_and_response_times_2021-05-27-16-38_chronograf_data.csv`

D.4 Profiling Strategy

Supplementary Data Files

Description: Profiling assignments, PID output, and 50th, 75th and 95th percentile response time data for the profiling without component weightings experiments in Section 7.4.7 and 7.4.8.

Uploaded file paths:

- `experiments/notebooks/data/constant_load_repeats_profiling_profile_assignments_2021-05-12-15-32_chronograf_data.csv`
- `experiments/notebooks/data/constant_load_repeats_profiling_pid_output_2021-05-12-15-29_chronograf_data.csv`
- `experiments/notebooks/data/constant_load_repeats_profiling_response_times_2021-05-12-15-30_chronograf_data.csv`

Supplementary Data Files

Description: Profiling assignments, PID output, and 50th, 75th and 95th percentile response time data for the profiling edge case experiments in 7.4.7.

Uploaded file paths:

- `experiments/notebooks/data/profiling_all_low_edge_case_priority_assignments_2021-05-17-14-12_chronograf_data.csv`
- `experiments/notebooks/data/profiling_all_low_edge_case_priority_assignments_2021-05-17-14-12_chronograf_data.csv`
- `experiments/notebooks/data/profiling_all_high_edge_case_priority_assignments_2021-05-17-17-20_chronograf_data.csv`
- `experiments/notebooks/data/profiling_all_high_edge_case_response_time_and_pid_output_2021-05-17-17-20_chronograf_data.csv`

Supplementary Data Files

Description: Profiling assignments, PID output, and 50th, 75th and 95th percentile response time data for the experiment combining profiling with component weightings in Section 7.4.9.

Uploaded file paths:

- `experiments/notebooks/data/constant_load_repeats_profiling_with_component_weightings_priorities_2021-05-14-17-12_chronograf_data.csv`
- `experiments/notebooks/data/constant_load_repeats_profiling_with_component_weightings_2021-05-14-17-11_chronograf_data.csv`

Appendix E

Developer Usability Experiments

Supplementary Data Files

Description: PID output and 95th percentile response time data for the developer usability experiments in Section 7.5.

Uploaded file paths:

- `experiments/notebooks/data/developer_usability_baseline_2021-05-19-14-13_chronograf_data.csv`
- `experiments/notebooks/data/developer_usability_cart_2021-05-19-14-29_chronograf_data.csv`
- `experiments/notebooks/data/developer_usability_news_2021-05-19-14-21_chronograf_data.csv`
- `experiments/notebooks/data/developer_usability_recommender_2021-05-19-14-14_chronograf_data.csv`