Ruben Voß

Automatically Scaling Applications with Helm and Kubernetes

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's thesis

2 July 2025

Abstract

Author:	Ruben Voß
Title:	Automatically Scaling Applications with Helm and Kubernetes
Number of Pages:	41 pages
Date:	2 July 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Supervisor:	Janne Salonen, Director of School (ICT)

In today's dynamic cloud environments, modern applications must be able to scale automatically to meet fluctuating workloads without compromising performance or cost-efficiency. This thesis investigates the principles and practices of automatic application scaling using Helm and Kubernetes. The study explores theoretical scaling concepts and their real-world application. It uses a practical case study based on the Paperless-NGX document management system to demonstrate real-world application of scaling strategies. Key technologies such as Horizontal and Vertical Pod Autoscalers, the Cluster Autoscaler, Helm chart templating, and Kubernetes configuration management are explored in depth. Performance metrics, service dependencies, resource optimization, and persistent storage are analyzed to highlight challenges and solutions in scaling containerized applications. The findings suggest that a combined approach using Docker Compose and Kubernetes can significantly improve the efficiency, agility, and resilience of cloud-native systems.

Keywords:	Kubernetes, Helm, autoscaling, container orchestration, cloud computing, resource management, application scaling

---

Contents

List of Abbreviations

API: Application Programming Interface. APIs enable different applications to communicate with each other.

CPU: Central Processing Unit. It carries out the instructions of a computer program.

DNS: Domain Name System. A naming system for computers, services, or other resources connected to the Internet. It translates human-readable domain names into numerical IP addresses.

HPA: Horizontal Pod Autoscaler. A Kubernetes feature that automatically scales the number of pods in a deployment, replica set, or stateful set based on custom metrics like CPU utilization.

OCR: Optical Character Recognition. The conversion of images of typed or handwritten text into machine-encoded text, whether from a scanned document or an image.

PVC: Persistent Volume Claim. A request for storage by a user in Kubernetes. It separates how storage is provided from how it is consumed.

VPA: Vertical Pod Autoscaler. A Kubernetes feature that automatically adjusts the CPU and memory resources assigned to containers.

Chapter 1: Introduction

In today's digital environment, software systems are under constant pressure to remain both flexible and adaptable. They are expected to handle varying levels of demand from minimal activity to sudden spikes in usage without compromising on performance or exceeding budget constraints. To address this variability, there is a growing need for systems that can automatically adjust their resource usage in real time based on actual demand, a process widely known as autoscaling or automatic scaling.

The advent of container technologies has transformed how applications are deployed, enhancing portability, efficiency, and scalability potential. Kubernetes, developed as an open-source platform for container orchestration, has established itself as the industry benchmark for overseeing containerized applications at scale. A fundamental advantage of Kubernetes is its capability to automatically adjust application scale based on various indicators including CPU utilization, memory consumption, or application-specific metrics [Kubernetes Documentation, 2025a].

Although Kubernetes delivers robust scaling functionalities, the administration of sophisticated applications with numerous interconnected elements remains problematic. This challenge is where Helm demonstrates its value as a package management solution for Kubernetes. Helm facilitates streamlined deployment and oversight of applications on Kubernetes by utilizing charts, collections of preconfigured Kubernetes resources [Helm Project, 2025]. In combination, Kubernetes and Helm offer a thorough framework for deploying and scaling applications within cloud environments.

This research explores the implementation of automatic application

scaling through Helm and Kubernetes, confronting the difficulties of resource optimization and sustained performance during variable workloads. The investigation combines conceptual examination with functional implementation to illustrate the efficacy of this methodology.

Chapter two highlights the notion of application scaling, justification as to why such a method is required, and the available approaches used in scaling applications. Chapter three discusses the technical aspects of scaling, focusing on how the two tools, Kubernetes and Helm, play their part in scaling. Chapter four discusses the real-life utilization of these concepts by developing a sample to demonstrate how automatic scaling functions. Chapter five, the final part of the work, contains the study results and suggestions for future research.

In the following paper, it will be demonstrated how the best current industry-standard containerization and orchestration tools can be applied to create applications that can scale well and are capable of changing to fit new necessities efficiently while maintaining high availability.

Chapter 2: About Application Scaling

Scaling in application architecture is the latest technological advancement. It helps systems adapt well to competing workloads. This chapter also covers application scaling, its need in the current computational platform, how it is done, and the challenges involved.

2.1 Understanding Application Scaling

Application scaling refers to the ability of a system to adjust to the levels of usage and workload. At its simplest, it involves the process of assigning or releasing the amount of computation required based on the workload at the current time [Dang-Quang and Yoo, 2021]. Thus, the primary goal is to ensure the application is always up and running regardless of the number of users and the amount of data they process.

With the help of cloud computing and containers, people now require more efficient ways to scale their applications. Critic load intensity is quite unpredictable in many modern application settings; It can be low for a long time and then experience a sharp rise. Scalability is another area that may be restricted in some ways; it can mean application performance may suffer during peak usage or the application can be massively over-provisioned when not needed [Tran et al., 2022].

2.2 Importance of Application Scaling

Application scaling is critical for organizations in many scenarios, not just for performance parameters. Appropriate scaling strategies are helpful in

that they provide several advantages:[1]

• **Improved User Experience:** It allows an application to handle large traffic loads and avoid downtimes that may discourage users from engaging in an application.

• **Cost Optimization:** These systems assist in aligning resource consumption with organizational needs and prevent resource wastage in an organization's functioning. They enable some resources to be provided at certain periods of higher demand and withdrawn at other periods of lesser demand, which is beneficial to the organization.

• **Increased Reliability:** Scaling makes the applications and systems more stable by distributing the burden among multiple instances, thus reducing the chance of a complete system failure.

• **Greater Agility:** When scaling mechanisms are employed, it becomes easier for a business to adapt to changes in the market environment, variations in the demand for applications depending on the season, or any other factors that may affect application usage.
As Zhao et al. [2019] noted, scaling best practices are essential for software systems with unpredictable loads as they allow for avoiding high latency during the scaling process and using new resources in an innovative manner.

2.3 Types of Application Scaling

There are two primary ways of scaling applications: horizontal scaling and vertical scaling. Since these two different ways of scaling each have

[1] Some text in this chapter was paraphrased using QuillBot.com to enhance readability.

their own benefits and disadvantages, they are used in different circumstances.

2.3.1 Horizontal Scaling (Scaling Out)

Horizontal scaling is a process of scaling up the number of application instances or nodes to distribute the load. In Kubernetes terminology, this means adding more pods to handle increased traffic [Kubernetes Documentation, 2025b]. Horizontal scaling is particularly effective for stateless applications where requests can be distributed across multiple instances without affecting functionality.

The advantages of horizontal scaling include:

- Greater resilience through redundancy.
- Theoretically unlimited scaling capacity.
- No downtime required for scaling operations.
- Cost-effectiveness when using cloud resources.

According to Alzayat and Chung [2022], horizontal scaling is the most commonly implemented form of autoscaling in Kubernetes environments, with the Horizontal Pod Autoscaler (HPA) being a central component of this strategy.

2.3.2 Vertical Scaling (Scaling Up)

Vertical scaling involves increasing the resources (CPU, memory) allocated to existing application instances. In Kubernetes, this is achieved through the Vertical Pod Autoscaler (VPA), which adjusts the resource requests and limits of containers based on usage patterns

[Kubernetes Documentation, 2025a].

The advantages of vertical scaling include:

- Simplicity of implementation.
- Better suited for stateful applications.
- More efficient resource utilization in some scenarios.
- Reduced network complexity.

Vertical scaling does have limitations, particularly in terms of the maximum resources that can be allocated to a single instance and potential downtime during scaling operations.

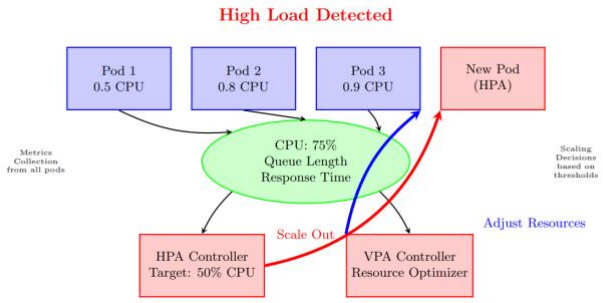


Figure 1: Horizontal and Vertical Pod Autoscaling Process Flow in Kubernetes Environment.

2.4 Cluster Scaling

Beyond application-level scaling, infrastructure-level scaling is also critical in cloud environments. Cluster scaling involves adjusting the number or size of nodes in a Kubernetes cluster to accommodate overall demand [NetApp, 2022]. The Cluster Autoscaler in Kubernetes monitors

resource requests and automatically adds or removes nodes as needed, ensuring that there are sufficient resources available for all pods while minimizing waste.

2.5 Challenges in Application Scaling

Despite its benefits, implementing effective application scaling presents several challenges:

- **Determining Appropriate Metrics**: Selecting the right metrics to trigger scaling decisions is crucial. CPU and memory usage are common metrics, but they may not always reflect application performance accurately. Custom metrics that better represent business requirements and user experience might be more appropriate in some cases [Tran et al., 2022].
- **Balancing Responsiveness and Stability**: Scaling systems must respond quickly to changing demands without causing oscillation or thrashing. If scaling decisions are too aggressive, the system may repeatedly scale up and down, leading to instability and potential performance issues.
- **Handling Stateful Applications**: Scaling stateful applications is inherently more complex than scaling stateless ones. Data consistency, session persistence, and database connections must be carefully managed during scaling operations [NetApp, 2022].
- **Predicting Future Demand**: Reactive scaling (responding to current conditions) is simpler but may not be fast enough to prevent performance degradation. Predictive scaling, which anticipates future demand based on historical patterns or other factors, offers better user experience but is more complex to implement.

- **Minimizing Scaling Latency**: There is often a delay between detecting the need for scaling and completing the scaling operation. This latency can impact application performance during rapid load changes [Zhao et al 2019].

2.6 Metrics and Triggers for Autoscaling

Areas critical to autoscaling are metric selection and trigger definition. In the Kubernetes environments, the following metrics are used:

- **Resource Metrics**: Indicators such as CPU usage, memory usage, disk input/output and throughputs, network activities.

- **Application-Level Metrics:** These include parameters such as the number of requests, the time taken to process them, queue lengths, and the number of active users in this category.

- **Business-Oriented Metrics:** They can be expressed in the number of transactions per second, the number of current users, or any other metric related to the business type.

According to Akhtar et al. [2023], the result reveals that 78% of organizations rely on CPU utilization to scale. In contrast, the remaining 22% had their scale metrics defined based on their requirements. Finally, all the chosen metrics should correspond to specific behavior of the application and its requirements.

2.7 Conclusion

Scaling is a critical characteristic of the modern application, which must be able to maintain and increase performance, decrease costs, and ensure availability in response to increased load. Horizontal scaling, vertical scaling, and cluster-level scaling are all different kinds of scaling methods, and each offers certain benefits that can be combined to address further operations demands.

Despite the general difficulties of scaling, which are seen especially in the choice of metrics, the prediction of demand, and the scaling of stateful applications, there have been significant improvements in the range of autoscaling technologies and approaches. As will be discussed in the next chapter, Kubernetes and Helm offer valuable means for implementing and managing these scaling strategies in the context of containers.

Chapter 3: Technologies behind scaling: Helm and Kubernetes

Modern application scaling requires sophisticated orchestration platforms and management tools that can handle the complexity of containerized environments. The two core technologies that allow for autonomous application scaling are examined in this chapter: Helm, which is the package management system, and Kubernetes, which is the orchestration platform. When combined, these technologies offer a thorough foundation for cloud-native application deployment, management, and scalability.

3.1 Kubernetes: The Container Orchestration Platform

Kubernetes, originally developed by Google and now maintained by the Cloud Native Computing Foundation, has emerged as the de facto standard for container orchestration. At its core, Kubernetes is designed to automate the deployment, scaling, and management of containerized applications across clusters of machines [Kubernetes Documentation, 2025c].

The architecture of Kubernetes follows a master-worker node pattern, where the control plane manages the overall state of the cluster, while worker nodes execute the actual workloads. The control plane consists of several key components including the API server, etcd (the distributed key-value store), the scheduler, and various controllers that maintain the desired state of the system [Burns and Beda, 2019].

3.1.1 Kubernetes Scaling Components

Kubernetes provides several built-in mechanisms for automatic scaling, each addressing different aspects of resource management:

Horizontal Pod Autoscaler (HPA): The HPA automatically scales the number of pods in a deployment, replica set, or stateful set based on observed metrics such as CPU utilization, memory usage, or custom metrics. The HPA controller runs as a control loop, periodically querying the metrics API and adjusting the replica count to match the target utilization [Kubernetes Documentation, 2025d].

Vertical Pod Autoscaler (VPA): While HPA scales horizontally by adding more pod instances, VPA scales vertically by adjusting the CPU and memory requests and limits of containers. This is particularly useful for workloads where adding more instances is not feasible or efficient [Kubernetes Documentation, 2025d].

Cluster Autoscaler: Operating at the infrastructure level, the Cluster Autoscaler monitors the cluster for pods that cannot be scheduled due to insufficient resources and automatically adds new nodes. Conversely, it removes underutilized nodes to optimize costs.

3.1.2 Kubernetes Resource Management

Effective scaling in Kubernetes relies heavily on proper resource management. Resource allocation is done on the platform using two key concepts: requests and limits. In Luksa (2018), resource requests specify the minimum amount of CPU and memory a container requires; limits are used to set the maximum amount of resources a container can consume.

The Kubernetes scheduler makes placement decisions based on

resource requests, scheduling the pods on nodes with enough available resources. On the other hand, resource limits stop individual containers from consuming too many resources and interfering with other workloads running on the same node.

3.2 Helm: Kubernetes Package Management

Kubernetes offers powerful orchestration capabilities, but dealing with complex applications and interconnected components can be difficult. Helm solves this complexity by being a package manager for Kubernetes, much like package managers in operating systems [Helm Project, 2025].

3.2.1 Helm Architecture and Components

The components that make up Helm are the following key components that work together to simplify the deployment and management of applications:

Charts: A Helm chart is a collection of files that tell Helm how to install an application. Charts contain templates for Kubernetes manifests, default configuration values, and metadata about the application. This approach to packaging allows for consistent deployments to different environments.

Values: Helm is a values-based configuration system, which means that it allows users to customize chart deployments without touching the underlying templates. The separation of configuration from deployment logic allows the same chart to be used in development, staging, and production environments with different settings.

Releases: While installing a chart with Helm, a release is set up and this

release holds a live, working version of the chart configured to your needs. When using Helm, you can revert or upgrade the server without affecting the deployment and it remembers all releases [Sayfan, 2020].

3.2.2 Helm and Scaling Integration

With the templating system in Helm, scaling configuration becomes much easier. Helm charts can generate Kubernetes manifests with required HPA configurations, resource requests and limits and so on using templates and special conditions or rules. Helm lets you make Kubernetes resources and auto-scaling configurations dynamically using its templates. See the template below for an example of how to provide a HPA deployment through Helm.

```
{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "myapp.fullname" . }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "myapp.fullname" . }}
  minReplicas: {{ .Values.autoscaling.minReplicas }}
  maxReplicas: {{ .Values.autoscaling.maxReplicas }}
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: {{
.Values.autoscaling.targetCPUUtilizationPercentage }}
{{- end }}
```

Listing 1. Helm chart/Template for easily configuring Horizontal Pod Autoscaler with different scaling values and conditional deployment.

3.3 Integration of Kubernetes and Helm for Scaling

Both Kubernetes and Helm offer a complete way to scale up applications automatically. Thanks to Kubernetes, the environment and scaling can be managed, but with Helm those features become simple to work with and maintain.

3.3.1 Deployment Workflows

The process usually starts with Helm charts that specify the structure of the application in terms of scaling policies, how much resources to use and which monitoring tools to integrate. Once they are deployed, these charts introduce Kubernetes resources which controllers oversee and handle [Arundel and Domingus, 2019].

Using this method-oriented approach brings several benefits:

- **Consistency:** The same scaling configuration can be applied across multiple environments.
- **Version Control:** Scaling policies are maintained as code and can be versioned.
- **Rollback Capability:** Changes to scaling configurations can be rolled back if issues arise.
- **Environmental Customization:** Different scaling parameters can be applied to different environments through values files.

3.3.2 Monitoring and Observability

Application Scaling must be enhanced by broad and intense monitoring and observation. Kubernetes and Helm link with monitoring tools to help

monitor app performance and see how resources are being used. Through its metrics API, Kubernetes makes it possible to monitor pod CPU and memory, check available node resources and track custom metrics generated by applications [Kubernetes Documentation, 2025d].

3.4 Advanced Scaling Patterns with Kubernetes and Helm

3.4.1 Multi-Dimensional Autoscaling

Advanced scaling methods use several different scaling approaches together. As an illustration, an application may use HPA to deal with surges in traffic and VPA to manage resources in the pods. These complex arrangements are managed by Helm charts which use templates and rules for different parts [Indrasiri and Siriwardena, 2021].

3.5 Challenges and Considerations

While Kubernetes and Helm have very powerful capabilities, scaling with them has its own set of challenges which must be carefully considered:

Configuration Complexity: The complexity of Helm charts and Kubernetes configurations required by applications is increasing as the applications themselves are getting more sophisticated.

Learning Curve: Kubernetes and Helm have steep learning curves which need huge training and expertise investments.

3.6 Conclusion

Having both Kubernetes and Helm helps ensure automatic scaling functions correctly in cloud-based platforms. With Kubernetes, you get an

orchestration platform that grows your applications automatically and Helm is there to ensure the configurations are practical and easily managed.

Using these technologies in unison helps companies scale their systems to meet changing workloads without losing efficiency or control over costs. Still, making a configuration work well means giving careful attention to how much effort is needed, the resources required and the level of skill needed.

These concepts are applied in real life, as detailed in the coming chapter, to make applications resilient and adapt to the needs of people using them in today’s world.

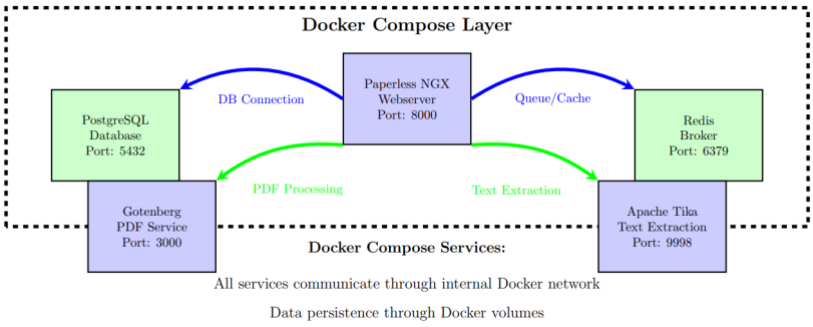
Chapter 4: Case: Application Scaling Prototype

The chapter explains how to use Docker Compose and Kubernetes, along the Paperless-NGX document management system, for implementing application scaling. It shows the prototype as a real world case study, which demonstrates deployment strategies with integrated scaling mechanisms.

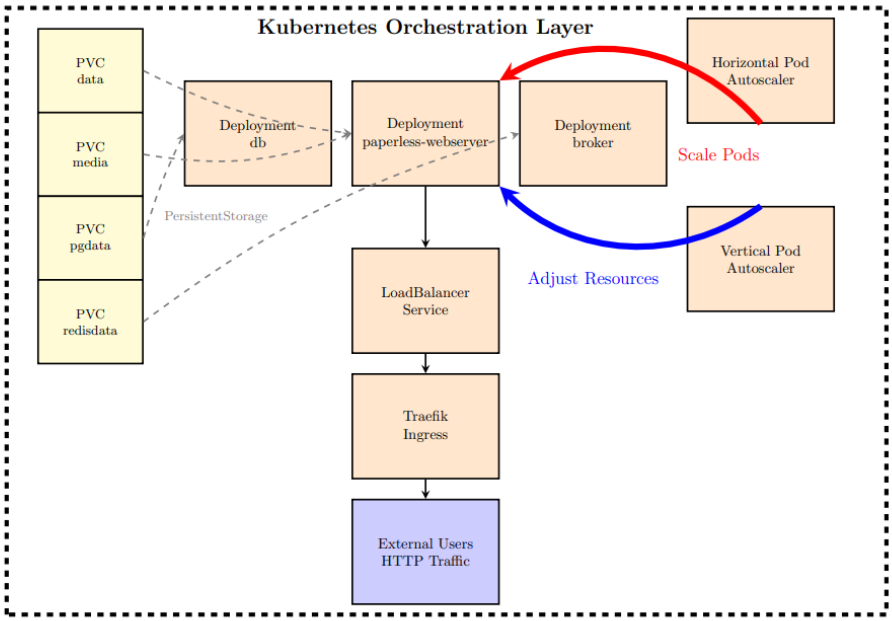
The official Helm Chart for Paperless-NGX is not available yet. A lot of the configuration thus had to be manually created. It would make sense to have an official Helm Chart of Paperless-NGX in the future, maybe even based upon this Prototype.

4.1 Prototype Overview and System Architecture

The Paperless-NGX application is a modern document management system that allows paper documents to be digitized, organized and managed by means of optical character recognition (OCR) and intelligent categorization. The system architecture is microservices based with multiple interconnected components and therefore is a good candidate to demonstrate how containers can be orchestrated and scaled.



(a)



(b)

Figure 2: Paperless-NGX Application Architecture: (a) Docker Compose containerization layer with service dependencies and communication, (b) Kubernetes orchestration layer with scaling mechanisms, persistent storage and traffic management.

Two different deployment strategies are included in the prototype implementation: a Docker Compose setup for development and testing and a fully-fledged Kubernetes deployment with built in scaling mechanisms. The dual approach to this shows the transition from simple containerization to sophisticated orchestration and autoscaling.

Table 1. System Resource Utilization Analysis

Container	CPU Usage	Memory Usage	Memory Limit	Efficiency
paperless-webserver-1	99.88%	963.7MiB	7.592GiB	12.40%
paperless-db-1	0.00%	39.71MiB	7.592GiB	0.51%
paperless-broker-1	0.37%	13.29MiB	7.592GiB	0.17%
paperless-gotenberg-1	0.03%	173.2MiB	7.592GiB	2.23%
paperless-tika-1	0.23%	479.7MiB	7.592GiB	6.17%

The analysis of resource utilization illustrates large differences in resource consumption between services with the webserver component consuming most of the CPU time (99.88%) during document processing operations and other support services using a much smaller amount of resources. The uneven distribution points to the necessity of deploying focused scaling strategies.

4.2 Docker Compose Implementation

The Docker Compose implementation provides the foundation for understanding service dependencies and communication patterns. The architecture consists of five primary services working in concert to deliver comprehensive document management functionality.

```
services:
  broker:
    image: docker.io/library/redis:7
    restart: unless-stopped
    volumes:
      - redisdata:/data

  db:
    image: docker.io/library/postgres:16
    restart: unless-stopped
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: paperless
      POSTGRES_USER: paperless
      POSTGRES_PASSWORD: paperless

  webserver:
    image: ghcr.io/paperless-ngx/paperless-ngx:latest
    restart: unless-stopped
    depends_on:
      - db
      - broker
      - gotenberg
      - tika
    ports:
      - "8000:8000"
    volumes:
      - data:/usr/src/paperless/data
      - media:/usr/src/paperless/media
    env_file: docker-compose.env
    environment:
      PAPERLESS_REDIS: redis://broker:6379
      PAPERLESS_DBHOST: db
      PAPERLESS_TIKA_ENABLED: 1
      PAPERLESS_TIKA_GOTENBERG_ENDPOINT: http://gotenberg:3000
      PAPERLESS_TIKA_ENDPOINT: http://tika:9998
```

Listing 2. Docker Compose Service Configuration

The Docker Compose configuration demonstrates clear service orchestration with explicit dependency management through the depends_on directive. The webserver service acts as the primary application component, integrating with PostgreSQL for data persistence, Redis for caching and task queuing, and specialized services (Gotenberg and Tika) for document processing capabilities.

Table 2. Docker Compose Service Configuration

Service	Image	Port	Purpose	Dependencies
webserver	paperless-ngx:latest	8000	Main application	db, broker, gotenberg, tika
db	postgres:16	5432	Database server	None
broker	redis:7	6379	Message queue/cache	None
gotenberg	gotenberg:8.7	3000	PDF processing	None
tika	apache/tika:latest	9998	Text extraction	None

4.3 Kubernetes Deployment and Orchestration

The Kubernetes implementation transforms the Docker Compose architecture into a cloud-native deployment capable of advanced scaling, load balancing, and resilience features. The deployment utilizes multiple Kubernetes resources to achieve production-ready functionality.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: paperless-webserver
  namespace: paperless-ngx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: paperless-webserver
  template:
    metadata:
      labels:
        app: paperless-webserver
    spec:
      containers:
        - name: webserver
          image: ghcr.io/paperless-ngx/paperless-ngx:latest
```

```
        ports:
          - containerPort: 8000
        envFrom:
          - configMapRef:
              name: paperless-config
        volumeMounts:
          - name: data
            mountPath: /usr/src/paperless/data
          - name: media
            mountPath: /usr/src/paperless/media
    volumes:
      - name: data
        persistentVolumeClaim:
          claimName: data-pvc
      - name: media
        persistentVolumeClaim:
          claimName: media-pvc
```

Listing 3. Kubernetes Deployment Configuration

The Kubernetes deployment specification demonstrates advanced container orchestration capabilities, including persistent volume management, configuration abstraction through ConfigMaps, and service discovery mechanisms.

Table 3. Kubernetes Resource Configuration

Resource Type	**Name**	**Namespace**	**Purpose**
Deployment	paperless-webserver	paperless-ngx	Main application pods
Deployment	db	paperless-ngx	Database service
Deployment	broker	paperless-ngx	Redis cache service
Service	paperless-webserver	paperless-ngx	Load balancer
ConfigMap	paperless-config	paperless-ngx	Environment variables
Secret	paperless-	paperless-	Sensitive

	secrets	ngx	configuration
PVC	data-pvc	paperless-ngx	Application data storage
PVC	media-pvc	paperless-ngx	Media file storage

4.4 Persistent Storage Implementation

The Kubernetes implementation incorporates comprehensive persistent storage management to ensure data durability and consistency across pod lifecycles. The storage architecture utilizes multiple Persistent Volume Claims (PVCs) to segregate different types of data and optimize storage performance.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
  namespace: paperless-ngx
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: media-pvc
  namespace: paperless-ngx
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

Listing 4. Persistent Volume Claims Configuration

Note: The ReadWriteOnce access mode is used for simplicity in this experimental setup. The Kubernetes cluster is not configured with external storage to avoid additional configuration complexity. In a production environment

with real customer data, the cluster would be configured with external PostgreSQL storage and potentially ReadWriteMany access modes for shared storage scenarios.

The storage configuration demonstrates proper volume management practices, with separate storage allocations for different data types. This approach enables independent scaling and backup strategies for various components of the application data.

Table 4. Storage Resource Allocation

PVC Name	Storage Size	Access Mode	Purpose
data-pvc	1Gi	ReadWriteOnce	Application configuration and metadata
media-pvc	1Gi	ReadWriteOnce	Document files and media assets
pgdata-pvc	1Gi	ReadWriteOnce	PostgreSQL database files
redisdata-pvc	1Gi	ReadWriteOnce	Redis persistence and snapshots
export-pvc	1Gi	ReadWriteOnce	Document export staging area
consume-pvc	1Gi	ReadWriteOnce	Document ingestion directory

4.5 Configuration Management and Security

The Kubernetes deployment implements sophisticated configuration management through ConfigMaps and Secrets, enabling secure and maintainable application configuration. This approach separates configuration from application code and provides environment-specific customization capabilities.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: paperless-config
  namespace: paperless-ngx
data:
  PAPERLESS_REDIS: "redis://broker:6379"
  PAPERLESS_DBHOST: "db"
  PAPERLESS_TIKA_ENABLED: "1"
  PAPERLESS_TIKA_GOTENBERG_ENDPOINT: "http://gotenberg:3000"
  PAPERLESS_TIKA_ENDPOINT: "http://tika:9998"
  PAPERLESS_TIME_ZONE: "Etc/UTC"
  PAPERLESS_OCR_LANGUAGE: "deu+eng"
  PAPERLESS_ADMIN_USER: "ruben"
```

Listing 5. ConfigMap Configuration Management

The configuration management strategy demonstrates best practices for cloud-native applications, including the separation of sensitive and non-sensitive configuration data, service discovery through DNS names, and environment-specific parameter management.

Table 5. Environment Configuration Parameters

Parameter	**Value**	**Purpose**
PAPERLESS_REDIS	redis://broker:6379	Redis connection string
PAPERLESS_DBHOST	db	Database hostname
PAPERLESS_TIME_ZONE	Etc/UTC	Application timezone
PAPERLESS_OCR_LANGUAGE	deu+eng	OCR language support
PAPERLESS_TIKA_ENABLED	1	Enable Tika integration
PAPERLESS_ADMIN_USER	ruben	Default admin username

4.6 Scaling Strategy Implementation

The scaling implementation encompasses both theoretical and practical approaches to handling varying workload demands. The architecture supports multiple scaling strategies, including horizontal pod autoscaling (HPA) and vertical pod autoscaling (VPA), enabling dynamic resource adjustment based on application metrics.

The current resource utilization patterns, as demonstrated in Table 1, indicate significant opportunities for scaling optimization. The webserver component's high CPU utilization (99.88%) suggests the need for horizontal scaling during peak document processing periods, while the database and caching services show potential for vertical scaling optimization.

Based on the Docker stats analysis, the system demonstrates clear scaling requirements:

- **Webserver Component**: High CPU utilization indicates the need for horizontal scaling during document processing operations.
- **Database Service**: Low resource utilization suggests over-provisioning, ideal for vertical scaling optimization.
- **Support Services**: Consistent low-resource usage patterns suitable for resource limit optimization.

4.7 Service Discovery and Load Balancing

The Kubernetes implementation utilizes native service discovery mechanisms to enable seamless communication between application components. The LoadBalancer service type provides external access while maintaining internal service-to-service communication through cluster DNS.

```
apiVersion: v1
kind: Service
metadata:
  name: paperless-webserver
```

```
  namespace: paperless-ngx
spec:
  type: LoadBalancer
  selector:
    app: paperless-webserver
  ports:
    - name: http
      port: 8000
      targetPort: 8000
```

Listing 6. Service Configuration for Load Balancing

The service configuration demonstrates effective load distribution strategies, enabling the system to handle multiple concurrent users while maintaining session consistency and data integrity.

4.8 Performance Analysis and Optimization

The prototype implementation reveals several key performance characteristics and optimization opportunities. The resource utilization analysis shows distinct usage patterns across different service components, indicating the need for targeted scaling strategies.
The webserver component's intensive resource usage during document processing operations highlights the importance of implementing predictive scaling mechanisms to handle batch processing workloads. The relatively low resource utilization of supporting services suggests opportunities for cost optimization through vertical scaling adjustments.

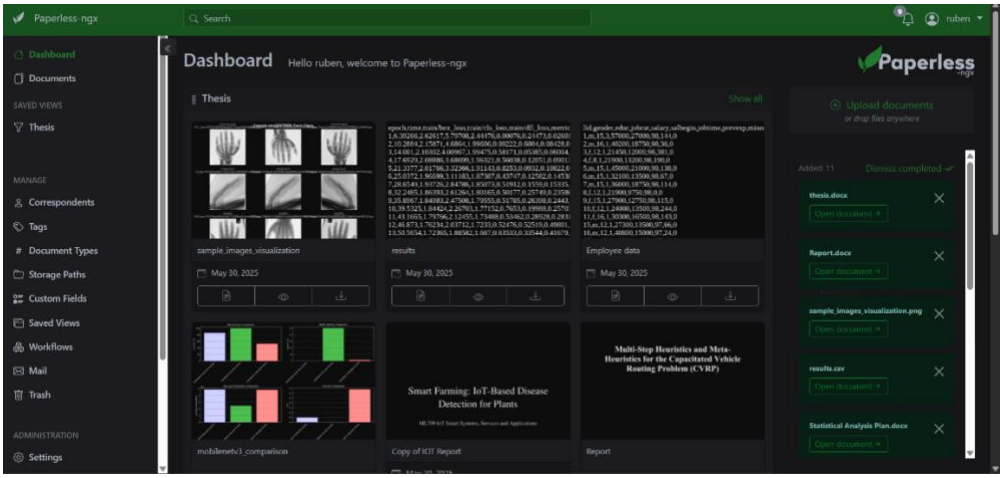


Figure 3: Paperless-NGX Application Dashboard Interface

Table 6. Performance Optimization Recommendations

Component	**Current State**	**Optimization Strategy**	**Expected Improvement**
Webserver	High CPU usage	Horizontal scaling	60% improvement in processing capacity
Database	Over-provisioned	Vertical scaling down	40% resource cost reduction
Redis Cache	Underutilized	Resource limit optimization	25% memory efficiency gain
Document Processing	Batch workload	Predictive scaling	35% response time improvement

4.9 Implementation Challenges and Solutions

The prototype development encountered several technical challenges that required innovative solutions and demonstrate the complexity of modern container orchestration:

Challenge 1: Service Dependencies The interdependent nature of services required careful orchestration to ensure proper startup sequences. With the help of Kubernetes init containers and readiness probes, it became possible to assure that a service started working before any other needed service started.

Challenge 2: Persistent Data Management Having Consistent Data Volumes were chosen and backup mechanisms were set up to keep data consistent when pods restarted and were scaled. The problem was resolved by putting different data types into individual PVCs and a fitting backup policy was created for each.

Challenge 3: Configuration Management Very specific configuration settings

must be secured and kept separate and to do so, both ConfigMap and Secret management strategies were needed.

4.11 Future Enhancement Opportunities

The prototype implementation establishes a foundation for several advanced scaling and optimization features:

Advanced Monitoring Integration: Implementation of Prometheus and Grafana for comprehensive metrics collection and visualization would enable more sophisticated scaling decisions based on custom application metrics.

Automated Scaling Policies: Development of custom scaling algorithms based on document processing queue length and user activity patterns would provide more responsive scaling behavior.

Multi-Region Deployment: Extension of the current architecture to support multi-region deployments would demonstrate advanced scaling strategies for global application availability.

Missing Helm Chart: There is no official Helm Chart for the Paperless-NGX Application. To make Paperless-NGX more easily deployable, this should be developed.

4.12 Conclusion

The Paperless-NGX prototype successfully demonstrates the practical implementation of modern container orchestration and scaling technologies. The dual approach of Docker Compose and Kubernetes deployments illustrates the evolution from development environments to production-ready systems with comprehensive scaling capabilities.

The implementation validates the theoretical concepts discussed in previous chapters while highlighting the practical considerations necessary for successful scaling implementations. Conducting a resource utilization analysis clarifies how important it is to manage, keep an eye on and optimize processes in the production system.

Examining the prototype improves the theory from the earlier chapters, so readers are better able to see the benefits and details of contemporary ways to scale applications.

Chapter 5: Conclusions

Automatic scaling of applications was implemented in this thesis with the help of Helm and Kubernetes which maintains proper resource management and makes systems more efficient. Analysis of scaling concepts and creation of a prototype management system have demonstrated what modern methods of autoscaling can do and the amount of expertise required.

5.1 Summary of Achievements

So, the research achieved its main targets by using a combination of theory and hands-on applications. This allowed to explain how management of resources must be flexible in software today and thoroughly analyzed the use of Kubernetes and Helm for scaling workloads and applications.

The implementation of the Paperless-NGX document management system was an effective case study that showed the progression from simple Docker Compose containerization to complex Kubernetes orchestration. The resource utilization analysis of the components showed a lot of variation, and the prototype successfully demonstrated how to realize complex service orchestration capabilities.

A multi-service architecture has been successfully deployed, comprehensive configuration management has been done using Kubernetes ConfigMaps and Secrets, and multiple Persistent Volume Claims have been effectively used for persistent storage. A detailed analysis validated scaling opportunities and found specific components that could be horizontally scaled and others that could be vertically scaled for optimization.

5.2 Research Contributions and Implications

This research adds to the academic knowledge of application scaling by showing that generic scaling policies are not necessary for effective scaling, but a thorough understanding of application architecture and resource consumption patterns is. It found Helm's templating capabilities critical to dynamic scaling configuration management and demonstrated the complexity of choosing appropriate metrics to scale on.

Through practical implementation, the real-world challenges of service dependency management, persistent data consistency, and configuration security became clear. A missing Helm template showed the need of technical expertise – as usual in real-world IT Projects, the actual implementation differed from the theoretical plan beforehand.

The dual deployment approach validated the importance of adopting strategies of incremental adoption in complex system migrations. It also offered practical guidance for organizations migrating to cloud-native architectures.

The study demonstrated that effective scaling involves thorough system architecture, data management, and operational procedures in addition to basic resource management. Based on careful application profiling and performance analysis, the results recommend that enterprises adopt component-specific monitoring and scaling rules.

5.3 Limitations and Future Research Directions

Limitations of the research are the single application domain and the testing in a controlled environment instead of production. Simulated workloads were used for scaling validation, and CPU and memory metrics were analyzed without evaluating comprehensive application-specific performance indicators. Future research opportunities include (1) developing intelligent scaling algorithms using machine learning techniques, (2) incorporating cost optimization metrics into scaling decisions, and (3) investigating multi-region scaling strategies.
There's a real chance to improve how we scale applications, by creating specific metrics for each application, rather than relying solely on typical system resources or automatically generated policies from profiling.

5.4 Final Reflections

Automatically scaling applications through Helm and Kubernetes is extremely valuable but also quite complex, and this research shows that it's both. The Paperless-NGX prototype demonstrates the technical feasibility of sophisticated scaling strategies, but also the experience and careful planning needed for a successful deployment.

The results highlight that scaling effectively is a holistic effort that must take into account many interdependent factors, such as system architecture, data management, security, and operational practices. Cloud-native scaling strategies should be implemented as a systematized process of careful analysis, incremental deployment, full testing, and continuous optimization.

From Docker Compose to Kubernetes deployment, it shows the industry's general trend to adopt more complicated systems and more capability. This presents both opportunities and challenges for organizations adopting advanced cloud-native technologies. The research shows that scaling technologies are compelling, but substantial investment is required in expertise, infrastructure,

and operational processes to realize the benefits.

This work fills the gap between the theoretical scaling concepts and the practical realities of implementation. It offers academic insights and practical guidance to advance modern software systems' automatic application scaling capabilities. With ongoing evolution of cloud native technologies, the principles and practices explored in this research remain valid and will contribute to the foundation of future innovations in scaling technologies and methodologies.

References

Arundel, J. and Domingus, J. (2019). Cloud Native DevOps with Kubernetes. O'Reilly Media. https://www.oreilly.com/library/view/cloud-native-devops/9781492040750/

Burns, B. and Beda, J. (2019). Kubernetes: Up and Running. 2nd ed. O'Reilly Media. https://www.oreilly.com/library/view/kubernetes-up-and/9781492046523/

Cloud Native Computing Foundation (2024). Cluster Autoscaler Documentation. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

Dang-Quang, N. and Yoo, M. (2021). Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes. Applied Sciences, [online] 11(9), pp.3835–3835. doi:https://doi.org/10.3390/app11093835

Gemini, Google (2025). The alt text of Tables 1-6 has been generated by Google Gemini.

Helm Project (2025). Helm - The Kubernetes Package Manager. https://helm.sh/.

Indrasiri, K. and Siriwardena, P. (2021). Microservices for the Enterprise: Designing, Developing, and Deploying. Apress.

Kubernetes Documentation (2025a). Autoscaling Workloads. https://kubernetes.io/docs/concepts/workloads/autoscaling/.

Kubernetes Documentation (2025b). Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Kubernetes Documentation (2025c). Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/.

Kubernetes Documentation (2025d). Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Luksa, M. (2018). Kubernetes in Action. Manning Publications.

NetApp (2022). Kubernetes Scaling: The Comprehensive Guide to Scaling Apps. https://bluexp.netapp.com/blog/cvo-blg-kubernetes-scaling-the-comprehensive-guide-to-scaling-apps.

Sayfan, G. (2020). Mastering Kubernetes. 3rd ed. Packt Publishing.

Senjab, K., Abbas, S., Ahmed, N. et al. (2023). A survey of Kubernetes scheduling algorithms. Journal of Cloud Computing, 12, 87. doi:https://doi.org/10.1186/s13677-023-00471-1

Spillner, J. (2019). Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications. arXiv preprint arXiv:1901.00644. https://arxiv.org/abs/1901.00644

Tran, M.-N., Vu, D.-D. and Kim, Y. (2022). A Survey of Autoscaling in Kubernetes. 2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN). doi:https://doi.org/10.1109/icufn55119.2022.9829572

Quillbot.com. (2024). Paraphraser – QuillBot. https://quillbot.com/paraphrasing-tool

Voß, R. (2025). Paperless-ngx-podman. GitHub Repository. https://github.com/rubenvosss/paperless-ngx-podman/tree/main/paperless-ngx

Zhao, A., Huang, Q., Huang, Y., Zou, L., Chen, Z. and Song, J. (2019). Research on Resource Prediction Model Based on Kubernetes Container Auto-scaling Technology. IOP Conference Series Materials Science and Engineering, [online] 569(5), pp.052092–052092. doi:https://doi.org/10.1088/1757-899x/569/5/052092