

miniK8s: A Pedagogical Cloud-Native System

Dong Du*

dd_nirvana@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Haibo Chen

haibochen@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Mingyu Wu*

mingyuwu@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Binyu Zang

byzang@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Abstract

The rapid adoption of cloud-native technologies, particularly containerization and orchestration with systems like Kubernetes, necessitates their effective integration into undergraduate computer science curricula. However, the complexity of production-grade cloud-native systems present a steep learning curve. Traditional pedagogical approaches often involve either oversimplified toy projects built from scratch, which lack real-world relevance, or the direct use of complex cloud systems, which can obscure fundamental concepts. To overcome these barriers, we introduce miniK8s, a lightweight, Kubernetes-like platform designed for an undergraduate cloud computing course. miniK8s distinguishes itself by promoting the pedagogical vision of teaching students to build a substantial and realistic system by *integrating existing, robust open-source components* with a hand-written, simplified cornerstone component. With miniK8s, students can both learn the key concepts inside the cloud architecture (e.g., resource scaling) and build practical systems with open-source building blocks. We have successfully used miniK8s as a project in a cloud computing course for hundreds of undergraduate students, which can significantly enhance students' understanding of how real-world cloud-native systems work and their ability to build a complex system.

CCS Concepts

• **Social and professional topics** → **Computing education**; • **Software and its engineering** → *Cloud computing*.

Keywords

Cloud Computing Education, Kubernetes, Container Orchestration, Systems Integration, Project-Based Learning, Computer Science Education

ACM Reference Format:

Dong Du, Mingyu Wu, Haibo Chen, and Binyu Zang. 2026. miniK8s: A Pedagogical Cloud-Native System. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2026)*, February

*Both authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution 4.0 International License. *SIGCSE TS 2026, St. Louis, MO, USA*

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2256-1/2026/02
<https://doi.org/10.1145/3770762.3772592>

18–21, 2026, St. Louis, MO, USA. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3770762.3772592>

1 Introduction

Cloud computing and its infrastructure are pivotal in modern software infrastructure [13]. Integrating those topics into undergraduate curricula is essential, but teaching them is challenging, mainly due to their complexity: a cloud system could contain hundreds of diverse components, running distributed cloud-native applications. For example, Cloud Native Computing Foundation (CNCF), a well-known hub of cloud-native computing [2], already contains 210 projects covering various areas like orchestration, network, storage, observability, etc. Direct engagement with a full-fledged cloud system can overwhelm students and obscure core principles [9].

A typical approach to teaching over-complicated computer systems in practice is to create a *mini* one (or known as *reinventing the wheel*). In other courses like operating systems or distributed systems, we already have such mini projects, such as a Unix-like simplified kernel [1] or a MapReduce-like distributed computing system [3]. However, this approach is usually over-simplified and lacks real-world relevance, making it difficult for students to both understand the internal details of cloud infrastructure and integrate existing, powerful open-source components—a skill paramount in contemporary software engineering.

To address this challenge, we propose a solution named *reinvent the car*, which only minimizes a key component inside the cloud instead of the whole system. We find that the container orchestration platforms, such as Kubernetes (k8s) [6, 13], is a promising candidate for the key component. Thanks to the modularized nature of cloud systems, cloud applications and services have already been containerized, providing orchestration platforms with unified abstractions. In addition to Kubernetes, people have also designed more lightweight systems such as K3S [4] and KubeEdge [5], which still serve as general-purpose orchestration platforms for container-based applications but with reduced functionalities to suit resource-constrained execution scenarios.

To this end, this paper introduces miniK8s, an educational project for teaching cloud infrastructures in practice. The core vision behind miniK8s is to teach cloud systems by having students construct a functional, Kubernetes-like mini-orchestration platform. During implementation, the students' primary task is to implement the orchestration logic for fundamental k8s abstractions (Pods, Services, etc.), handle standard YAML/JSON configurations from applications,

and respond to user commands from a kubectl-like CLI. Meanwhile, they are also required to integrate the hand-built orchestration platform with various robust, widely-used open-source systems (i.e., building a *car* with existing *wheels*), such as Docker [27] or Containerd [16] for the container runtime, and etcd [18, 30] or ZooKeeper [20] for distributed state management. This helps students to understand the details of key concepts inside the cloud infrastructure (e.g., resource management and scheduling) while fostering an understanding of system composition and training students in practical software engineering skills.

The key contributions of this work include detailing the design and pedagogical rationale of miniK8s with its emphasis on system integration, describing its architecture and support for core k8s concepts, outlining its project structure within the course, and discussing the pedagogical benefits, particularly how miniK8s cultivates a deeper understanding of cloud-native architectures and crucial system integration skills distinct from those developed in traditional systems courses. We have integrated miniK8s as a semester-long project into a cloud computing course in Shanghai Jiao Tong University for the last four years. According to feedback from over 130 students, miniK8s significantly enhances their understanding of cloud architectures and skills in building real-world complicated systems.

All resources related to miniK8s, including the lectures, tutorials, documents, etc., will be open-source and available online at <https://github.com/SJTU-IPADS/Cloud-Operating-Systems>. We hope the miniK8s project can help other educators.

2 Background and Related Work

Cloud computing has become an important subject of study due to its wide range of applications [12]. Since nowadays cloud platforms have become large-scale and complicated systems, how to provide adequate projects for students remains a challenge. Many existing cloud-related courses provide projects aiming at application development where students learn to use the basic units (e.g., virtual machines) and services (e.g., cloud storage) in the cloud [11, 25, 32, 33]. Those projects are usually based on cloud systems and do not help students to unveil the cloud architecture.

A common approach to designing course projects for complicated computer systems is *reinventing the wheel*. That is, students are instructed to implement a simplified but complete one (“wheel”) so as to learn the details under the hood. This approach has been used in other courses teaching complicated systems. For example, operating systems courses [1, 8] provide a simplified kernel as the course project, and distributed systems courses have assignments on building small-scale MapReduce clusters [3]. However, with a simplified architecture, those projects usually show poor compatibility with existing open-source components, making it difficult for students to build larger applications. Therefore, this approach is less welcome in learning the cloud architecture, which is composed of an ocean of open-source services and frameworks and hinders students from understanding the cloud system’s actual composition.

To summarize, to learn the cloud architecture, a course project needs to (1) unveil the details behind the abstractions provided by the cloud and (2) remain compatible so as to integrate existing

services and scaffolds and build larger and meaningful applications. To satisfy both goals, we propose a pedagogical project named miniK8s and share the experiences when using it in our classes. Other courses in system software are also exploring similar pedagogical exercises to achieve both goals [29], but miniK8s is the first pedagogical project in the cloud area to our best knowledge.

3 miniK8s Design

The miniK8s project is a capstone experience in an undergraduate cloud computing course, challenging students to build a mini (but useful) container orchestration system for managing applications across multiple machines.

3.1 Learning Objectives

The Washington Accord [7] is an agreement for accreditation or recognition of tertiary-level engineering qualifications, focusing on academic programmes which deal with the practice of engineering at the professional level.

Our cloud computing curriculum and the miniK8s project directly align with Washington Accord graduate attributes [7] for professional engineering education, emphasizing:

- (1) **Knowledge of cloud systems:** Demonstrate systematic understanding of cloud-native architectures through hands-on implementation, including containerization, networking, scheduling, service coordination, failure resilience patterns, cloud-native applications, and others.
- (2) **Complex problem formulation & resolution:** Identify, analyze, and model non-trivial cloud engineering challenges, from networking anomalies to performance bottlenecks, using first principles from computer science and empirical observations.
- (3) **Modern tool proficiency:** Effectively leverage open-source ecosystems (Docker, Prometheus, etc.) and industrial-grade practices (CI/CD pipelines, infrastructure-as-code) to design, debug, and optimize cloud infrastructure components.
- (4) **Collaborative engineering practice:** Execute both individual technical leadership and team-based development roles, demonstrating professional communication across code reviews, architecture documents, and upstream open-source contributions.

Through miniK8s’s scaffolded challenges, students transition from theoretical concepts to validated competencies measurable against international engineering standards.

3.2 Philosophy: Reinvent the Car, Not the Wheel

The central philosophy is to empower students to build a practical cloud system with both hand-built key components and existing open-source building blocks. In contrast to the traditional *reinventing the wheel* philosophy, we refer to it as *reinventing the car*. This philosophy stems from the observation that modern system software, particularly in the cloud, heavily relies on composing systems from specialized and usually open-sourced building blocks (or *wheels*). Therefore, most components in the cloud have been standardized, thanks to the virtualization technology (e.g., virtual

machines and containers) and unified interfaces (e.g., container network interfaces for network-related customization), which open the door for building a *car* to learn the overall cloud architecture.

Among all components, we find the orchestration framework, such as Kubernetes (known as k8s), is a promising candidate. First, k8s manages applications in containers, a widely accepted abstraction in the cloud environment, making it simple to support various building blocks. Second, the functionalities of k8s, including resource scheduling, application deployment, scaling, and fault tolerance, reflect the key advantages provided by a cloud system. Third, we have already seen simplified versions of k8s in practice [4, 5], which reduce the functionalities but retain the compatibility with most applications. Therefore, miniK8s is designed as a drop-in support for k8s to help students learn the cloud architecture.

Figure 1 materializes our “reinventing the car” idea, using miniK8s as an example. In the cloud environment, miniK8s serves as the body, while other building blocks can be borrowed to build the whole car. Those “wheels” include Docker or Containerd (containers), gRPC (remote procedure call), Prometheus [34] (monitoring), OpenFaaS [23] (Function-as-a-Service [FaaS] computing), and so on. With the car, students can further build more complicated applications, such as microservices, serverless applications, and AI-assisted apps.

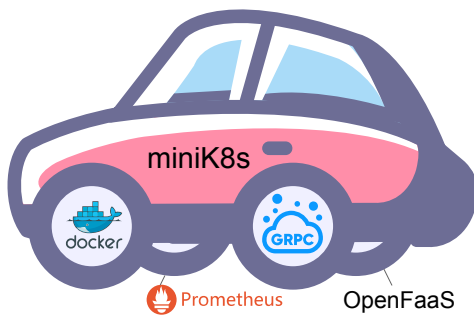


Figure 1: The reinventing the car philosophy, exemplified by miniK8s

miniK8s also aims for conceptual compatibility with Kubernetes. Students design their system to accept YAML/JSON manifests similar to those used by Kubernetes and develop a CLI tool that mimics basic `kubectl` commands, reinforcing industry-standard practices. This experiential learning of system complexity allows students to grapple with challenges like inter-component communication, API considerations, error handling across boundaries, and consistency in distributed systems. This approach differentiates miniK8s from projects involving isolated, from-scratch systems, aiming instead to equip students with the ability to understand, adapt, and contribute to complex, composed systems prevalent in their future careers.

3.3 Basic Design: Deconstructing K8s

Although simplified, a container orchestration framework is still non-trivial to implement. Therefore, we further break it into units and provide a guide for students to build one from scratch. The detailed roadmap is shown in Table 1, where the students are required to implement all those functionalities. Before diving into the

implementation of miniK8s, we also provide a warmup exercise. This exercise requires students to deploy a k8s cluster and use its essential abstraction, enhancing their understanding of k8s at a user’s perspective.

To implement miniK8s, students are recommended to support *Pod*, the fundamental abstraction in k8s. A pod is actually a collection of containers treated as a deployment and scheduling unit by k8s. Supporting it mainly requires using the existing container runtime (e.g., Containerd) to manage containers. Meanwhile, students also need to use a declarative programming paradigm to describe pods in a YAML format and transform those descriptions into running pods.

The second step is to support *Service*, an abstraction that can be accessed by end users through an outward-facing endpoint. This step involves implementing network-related functionalities for miniK8s, which typically involves pod-to-pod communication and internal IP assignment (through CNI). Since a service can actually be supported by multiple pods, network traffic forwarding and load balancing are also required. Lastly, a DNS-based naming layer is expected for end users to discover and access exposed services.

The next abstraction is *ReplicaSet* containing a fixed number of replicated pods. By controlling the number, *ReplicaSet* can be used to support resource scaling. Atop *ReplicaSet*, we further require students to implement an automatic resource scaling mechanism, which determines whether to scale up and down according to its monitoring of system resources (CPU, memory, I/O, etc.).

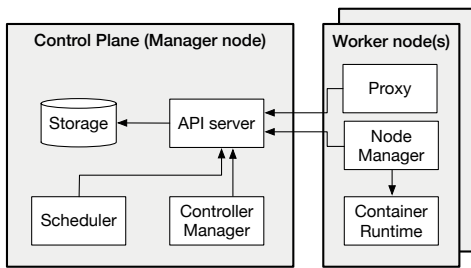
Afterwards, we scale miniK8s to multiple machines. This part includes an extra abstraction: *Node*, which is used by miniK8s to manage each machine. Meanwhile, miniK8s should include a *scheduler* to schedule pods across all nodes. The final task is fault tolerance, where we look back to enhance the reliability of miniK8s’s control plane and data plane.

Although the roadmap does not restrict the architecture of miniK8s, we suggest students to follow the architecture of k8s. Figure 2 shows a recommended architecture that we teach in the classes: it features a control plane and multiple worker nodes, mirroring k8s. The key designs of k8s are also introduced in classes to help students implement miniK8s (although not compulsory):

- **Object-based state management.** Everything (including abstractions introduced before) in k8s is an object (referred to as *API objects*). Therefore, it has a component to manage all objects (the API server) in a persistent storage. Other components can contact the API server to retrieve and modify objects.
- **Controller-based state reconciliation.** K8s introduces controllers to monitor the clusters’ states (stored in objects). When their state are not equal to a pre-declared desired state, the controller continues reconciling towards the desired state. An example is *ReplicaSet*: if the number of pods is smaller than a number in the preconfigured YAML file, the related controller needs to create more pods until reaching the same number. In K8s, all those controllers are managed by a controller manager.

Table 1: miniK8s Lab Activities. *Objectives show how each lab is related to the learning objectives in §3.1.*

Activity	Brief Description	Core miniK8s Features	Objectives (§3.1)
#0 Warmup	Deploy a k8s cluster and use it.	N/A	(1)
#1 Pod Support	Manage Pods using YAML and kubectl.	Pod, CLI, container runtime integration.	(1) and (3)
#2 Service Exposure	Deploy multiple Pod replicas and expose them as an endpoint to users.	Service, Deployment, load balancing, CNI, DNS.	(1)–(3)
#3 Scalable Deployment	Use a ReplicaSet to manage Pod replicas; scale up/down.	ReplicaSet definition, state reconciliation.	(1)–(3)
#4 Multi-machine Support	Use Nodes to manage multiple machines; support multi-machine scheduling.	Node definition, resource scheduling.	All
#5 Fault Tolerance	Enhance miniK8s’s fault tolerance.	Control/data plane failure detection and recovery.	All
#O1 App: microservice	Support microservice on miniK8s.	Cloud-native apps.	All
#O2 App: FaaS	Support FaaS on miniK8s.	Cloud-native apps.	All

**Figure 2: The recommended architecture of miniK8s**

3.4 Extended Design: Cloud-Native Applications

After implementing the basic abstractions, students are required to implement a kind of cloud-native application they prefer: microservice and function-as-a-service (FaaS). In addition to developing those applications, miniK8s also needs to extend to support more high-level features.

Microservices. The microservices architecture divides applications into loosely-coupled, fine-grained components (namely micro-services) and thus become suitable for cloud deployment. This architecture has been commonly used in mainstream cloud providers [14, 15]. Microservices necessitate inter-service communication, so we require students to enhance miniK8s to provide advanced service-related features. The recommended architecture is the sidecar model, which is used by mainstream service mesh systems [24]. Students are required to first inject proxies into pods to manage all network traffic. Afterwards, they can implement more complicated features such as service discovery, rolling updates, and canary releases. All those features can be demonstrated by a practical microservice application, where we encourage students to refer to existing benchmarks [19] to enhance its practicality.

FaaS. FaaS [22] is a new paradigm in cloud computing that allows users to upload fine-grained code snippets (namely functions) for automatic deployment and resource scaling (also known as serverless computing [10, 21]). To support FaaS, students are required to provide function-based programming support, including functions, DAG-based workflows (consisting of multiple functions), and triggers to initiate functions. In the underlying layer, they need to support *scale-to-zero*, which is an extension to prior scaling mechanisms and removes all related containers when facing

no requests. Similar to microservices, students are also asked to implement practical FaaS-based applications [17, 26, 35].

In addition to cloud-native applications, we also prepare other extensions to miniK8s. Those extensions are not necessary to support applications above, so we leave them as optional features. We have incorporated the following extensions into our assignments as optional features, and we plan to add more in the future.

- **Persistent storage:** support the persistent volume (PV) and persistent volume claim (PVC) abstraction so that pods in miniK8s can request persistent storage resources.
- **GPU:** support applications running on GPUs. With this extension, miniK8s can manage simple CUDA-based applications.
- **Security:** support configuring *security contexts* for containers inside a pod for security enhancement.
- **Observability:** support logging and monitoring with Prometheus to improve applications’ observability.

3.5 Integration into a Cloud OS Course

miniK8s is now integrated into a course named *Cloud Operating Systems: Design and Practice*, which aims to teach the cloud architecture and how they are used to support different types of cloud-native applications. The contents in miniK8s nearly cover all topics in its classes. Due to the complexity, miniK8s is designed as a semester-long group project (typically 3 students) with individual components. While the group collaborates on the main miniK8s features and cloud-native applications (either microservices or FaaS), each student also undertakes an individual assignment (picked from the optional features). Students also adhere to engineering best practices, using version control (Git), pull requests, and are encouraged to set up CI/CD pipelines. The evaluation encompasses functionality demonstrations, code and design quality, documentation, and individual contributions, assessed via mid-term and final presentations.

We also carefully designed the the course content to support the project. The course at Shanghai Jiao Tong University spans 16 weeks, with classes held twice weekly for 90 minutes each session. To effectively support the project, the lectures introduce fundamental Kubernetes (K8s) concepts, including pods, scheduling, services, and declarative programming paradigms, alongside other cloud

computing techniques relevant to K8s, such as cloud storage, networking, and applications. Specifically, the lectures focus on the key ideas and architectures of these related techniques but omit intricate design details, while emphasizing their practical application in real cloud systems like K8s or miniK8s.

Meanwhile, while implementation choices remain flexible in this project, we enforce structured progress management. Specifically, each team must submit an opening report, an interim report, and a final report. The opening report requires teams to detail project plans, identify potential challenges with mitigation strategies, and define member roles and responsibilities. Teaching assistants (TAs) and faculty then review these plans to identify potential issues, ensuring smoother project execution. After 4–6 weeks, the interim report documents current progress, technical adjustments since the initial plan, and encountered bottlenecks. Finally, the final report serves as the primary evaluation mechanism for group scoring. This multi-stage reporting system enables effective faculty guidance, helping students successfully implement their project concepts.

Finally, we explicitly encourage students to incorporate real-world best practices like Git and CI/CD into their projects, with adherence impacting final scores. Students must grant TAs access to their Git repositories, enabling progress tracking through metrics like commit review quality and CI pipeline implementation. Furthermore, rather than assigning uniform scores, we differentiate individual grades based on documented contributions (stated in final reports) and Git commit history. This approach incentivizes active participation from all team members throughout the project lifecycle.

4 Experience

4.1 Evaluation

We have used miniK8s as the project for our semester-long cloud computing course at Shanghai Jiao Tong University over the past four years. Each semester comprises approximately 40–50 undergraduate students, typically in their final semester of their undergraduate period (sixth semester), where this course serves as a capstone experience in their undergraduate curriculum.

Our university performs annual post-course surveys to evaluate curricular effectiveness and identify improvement opportunities. During the first academic year of miniK8s’s design and implementation (2021–2022), we collected responses from 53 students. Subsequent years yield 45 responses (2022–2023) and 36 responses (2023–2024), with the 2024–2025 spring semester report currently pending analysis. This paper focuses on two metrics critical to miniK8s’s objectives: *teaching method effectiveness* and *student performance outcomes*. Both metrics use a 1-point evaluation scale (1=optimal results), with institutional assessment teams providing comparative benchmarking against all courses each academic year.

Additionally, we (the course faculty members) conduct a supplemental survey addressing miniK8s-specific research objectives. This instrument features Likert-scale items (1=strong disagreement, 5=strong agreement) assessing claims including: “miniK8s significantly improves my understanding of cloud-native system concepts” and “miniK8s effectively enhances my engineering competencies in real-world system analysis, large-scale development, and collaborative workflows”.

Next, we organize findings around research questions (RQs), supplementing them with case studies and pedagogical reflections from our four-year experience.

RQ1: To what extent does miniK8s enhance cloud-native systems comprehension? Figure 3-a presents the average Likert-scale ratings for miniK8s’s effectiveness in conveying cloud-native concepts. Students consistently report strong agreement, with Pod Scheduling and Controller mechanisms receiving the highest rating (4.53/5, including 6.67% students for 3, 33.33% for 4, and 60% for 5), followed by Declarative Methods (4.47, including 6.67% students for 3, 40% for 4, and 53.33% for 5), and Service/Ingress Networks (4.33, including 6.67% students for 1, 6.67% students for 3, 26.67% for 4, and 60% for 5). By contrast, security-related ConfigMap (3.6, only 33.3% for 5) and Storage (3.9, only 40% for 5) yield significantly lower ratings, correlating with their status as non-required features in miniK8s’s lab activities (Table 1), where only a small group of students opt to implement them. To gather qualitative feedback, we include the open-ended survey question: “Describe a case where miniK8s most helped you understand a conceptual topic.” Students’ responses align with the Likert-scale ratings, e.g., “with miniK8s I understand pod, replicaset, and hpa.”

We extend our analysis to miniK8s’s alignment with Washington Accord engineering competencies [7] in Figure 3-b. Students strongly affirm that miniK8s: (1) enhances understanding of real-world complex systems (4.4/5), (2) develops large-scale system implementation capabilities (4.27), and (3) improves team collaboration skills (4.07). These results demonstrate systemic educational benefits of miniK8s across both technical and professional practice dimensions.

RQ2: How can the teaching methods benefit student performance? The university’s post-course survey includes two miniK8s-relevant metrics: *teaching method effectiveness* (student perceptions of pedagogical quality) and *student performance outcomes* (aggregate measures of engagement, classroom dynamics, and perceived difficulty). Figure 3-c and Figure 3-d present comparative distributions across 307 courses during miniK8s’s inaugural implementation (2021–2022). Our course achieves exceptional scores within the [0.9, 1.0] excellence range (the best): 0.95 (teaching methods) and 0.93 (performance outcomes). This trend maintains consistency through subsequent iterations, with 2023–2024 results reaching 0.98 and 0.96 respectively.

While miniK8s introduces significant implementation challenges, with many students identifying it as their most challenging undergraduate project, the sustained superiority in the survey demonstrates strong student recognition of miniK8s’s pedagogical value.

RQ3: How do student groups perform in the miniK8s project? (A Case Study) Across four years of practice, student groups consistently demonstrate remarkable technical ambition in miniK8s, frequently exceeding faculty expectations for undergraduate system projects. We analyze two representative cases (30% of groups achieve similar sophistication).

The first group designs and implements Kubernetes, which carefully extends our recommended architecture with serverless computing supports [17, 26, 35], as shown in Figure 4. Kubernetes implements all Kubernetes core components: an API Server handles RESTful [31] resource orchestration while etcd maintains cluster state persistence. The scheduler monitors Pod/Node lifecycle events,

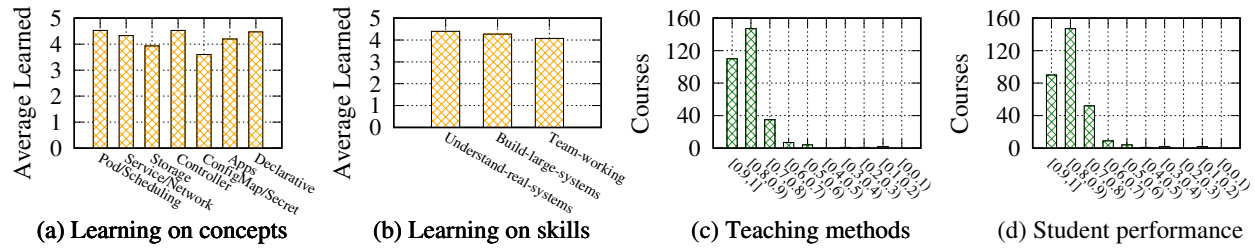


Figure 3: A quantitative study of how miniK8s improves the teaching. (a) and (b) present the Likert-scale ratings. (c) and (d) present the distribution across 307 courses in the university, and miniK8s achieves the score in the best range.

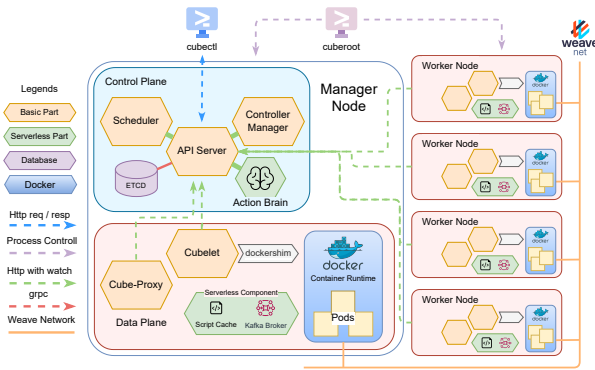


Figure 4: A case of students' submissions: Kubernetes. The architecture figure is drawn by the students (with permissions granted).

complemented by a controller managing ReplicaSet deployments. The key difference compared with other submissions is the *Action Brain* subsystem, a runtime engine enabling FaaS workflows. Specifically, Kubernetes treats a single function as an *Action* and the Pod executing an action as the *Actor*. To support the function chain, which is a workflow requiring different functions to cooperate, Kubernetes will utilize *Action Brain* to define the calling relationship among functions and utilize Kafka [28] (an open-sourced message queue) to connect different Actors. As a result, Kubernetes can support complicated serverless applications, and support dynamic scaling and easy-to-use programming for serverless apps.

The second group demonstrates comprehensive implementation of minik8s core components through Rust-based system programming, a notable choice given Golang and C++'s stronger community support during 2021-2022. This technical decision also surprises faculty members, as Rust's Kubernetes ecosystem tools remain under active development at the time. Their solution implements Kubernetes' kube-proxy functionality using rust-iptables for dynamic management while enforcing code quality through automated rustfmt formatting. The students further explore Rust's cloud-native potential by extending their minik8s with GPU service orchestration and serverless function support, demonstrating the language's versatility in heterogeneous computing environments. Their development process establishes rigorous engineering

practices through Gitflow workflows and Conventional Commit standards via an integrated toolchain (JetBrains Space, Gitee, Cargo Workspaces). The team's collaboration proves exceptionally effective, with 35 of 37 code-reviewed pull requests achieving peer approval (≥ 1 reviewer signoffs).

4.2 Limitations

The miniK8s project still exhibits limitations requiring continuous improvement. First, assessment proves challenging in such an open-ended project framework, as different teams typically adopt divergent approaches and architectures for implementation. To ensure effective evaluation, our course allocates 4 TAs each semester to guide students and assess final projects. A second challenge involves debugging complexities: students frequently report networking issues, such as connectivity failures between their three dedicated virtual machines. Resolving these network problems often demands substantial effort in code debugging, sometimes requiring teaching assistant intervention. These challenges highlight the need for purpose-built infrastructure tools [32] to better support project implementation.

5 Conclusion

miniK8s is an educational project designed to provide undergraduates with a practical understanding of cloud infrastructures. Its pedagogical innovation lies in emphasizing the integration of existing open-source components, teaching how complex systems are composed, and fostering critical integration skills. By implementing and managing familiar k8s abstractions through YAML and a custom CLI, students both learn the core concepts inside cloud systems and gain tangible, industry-relevant experience. We believe miniK8s offers a compelling alternative to traditional systems projects, demystifying complex technologies and cultivating a valuable practical skill set.

Acknowledgments

We thank the anonymous SIGCSE TS reviewers for their constructive feedback, and all the TAs (Qingyuan Liu, Ziming Zhao, Zhenyu Yang, Jingkai He, Zhuoran Wei, Rongchuan Liu, Xiaoqian Ma, Yuhan Li, Bojun Ren, Haocheng Wang, etc.) for improving miniK8s. This work was supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China, and Fundamental Research Funds for the Central Universities. Corresponding authors: Haibo Chen and Binyu Zang.

References

- [1] 2025. 6.828: Learning by doing. <https://pdos.csail.mit.edu/6.828/2014/overview.html>. Referenced Jun 2025.
- [2] 2025. Cloud Native Computing Foundation. <https://www.cncf.io/>. Referenced Jun 2025.
- [3] 2025. CS 5450 - Spring 2024 Networked and Distributed Systems - Assignments. <https://cs5450.github.io/assignments/>. Referenced Jun 2025.
- [4] 2025. K3s - Lightweight Kubernetes. <https://k3s.io/>. Referenced Jun 2025.
- [5] 2025. KubeEdge. <https://github.com/kubeedge/kubeedge>. Referenced Oct 2025.
- [6] 2025. Kubernetes: Production-Grade Container Orchestration. <https://kubernetes.io>.
- [7] 2025. Washington Accord. <https://www.internationalengineeringalliance.org/accords/washington>. Referenced May 2025.
- [8] S Tanenbaum Andrew and Bos Herbert. 2015. *Modern operating systems*. Pearson Education.
- [9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. 2009. *Above the clouds: A berkeley view of cloud computing*. Technical Report. Technical Report UCB/EECS-2009-28, EECS Department, University of California.
- [10] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. *Research advances in cloud computing* (2017), 1–20.
- [11] Ken Birman. 2022. CS5412 Cloud Computing. <https://www.cs.cornell.edu/courses/cs5412/2022fa/>.
- [12] Charles B Border. 2013. Cloud computing in the curriculum: fundamental and enabling technologies. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 147–152.
- [13] Eric A. Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) (SoCC '15). Association for Computing Machinery, New York, NY, USA, 167. doi:10.1145/2806777.2809955
- [14] Jeremy Cloud. 2013. Decomposing Twitter: Adventures in ServiceOriented Architecture. <https://www.infoq.com/presentations/twitter-soa/>.
- [15] Adrian Cockcroft. 2016. Evolution of Microservices. <https://www.slideshare.net/slideshow/evolution-of-microservices-craft-conference/61466608>.
- [16] containerd Authors. 2025. containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability. <https://containerd.io/>.
- [17] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefer. 2021. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*. 64–78.
- [18] etcd Authors. 2025. etcd - A distributed, reliable key-value store for the most critical data of a distributed system. <https://etcd.io/>.
- [19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.
- [20] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.
- [21] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).
- [22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS] <https://arxiv.org/abs/1902.03383>
- [23] Dac-Nhuong Le, Souvik Pal, and Prasant Kumar Pattnaik. 2022. OpenFaaS. *Cloud computing solutions: architecture, data storage, implementation and security* (2022), 287–303.
- [24] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 122–1225.
- [25] Zilu Liang and Mario Alberto Chapa-Martell. 2018. A Top-Down Approach to Teaching Web Development in the Cloud. In *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, 32–39.
- [26] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM international conference on distributed and event-based systems*. 73–84.
- [27] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux j* 239, 2 (2014), 2.
- [28] Neha Narkhede, Gwen Shapira, and Todd Palino. 2017. *Kafka: the definitive guide: real-time data and stream processing at scale*. " O'Reilly Media, Inc."
- [29] Emma Nieh, Zijian Zhang, and Jason Nieh. 2025. ezfs: A pedagogical linux file system. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*. 861–867.
- [30] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.
- [31] Leonard Richardson and Sam Ruby. 2008. *RESTful web services*. " O'Reilly Media, Inc."
- [32] Aditya Saligrama, Cody Ho, Benjamin Tripp, Michael Abbott, and Christos Kozyrakis. 2025. Teaching Cloud Infrastructure and Scalable Application Deployment in an Undergraduate Computer Science Program. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*. 1015–1021.
- [33] Asma Shakil and Paul Denny. 2024. Enhancing Student Engagement in Large-Scale Capstone Courses: An Experience Report. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. 722–728.
- [34] James Turnbull. 2018. *Monitoring with Prometheus*. Turnbull Press.
- [35] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 30–44.