OCTOPUS: a Scalable, Reliable and Cost-Efficient Solver Orchestrator for Optimization Services

Mattia Oriani, Marco Giorgini, Roberto D'Elia, Federico Naldini, Nicola Di Cicco, Fabio Lombardi
OPTIT, Italy
E-mail: {mattia.oriani, marco.giorgini, roberto.delia, federico.naldini, nicola.dicicco, fabio.lombardi}@optit.net

***Abstract*—The ever-increasing automation of complex decision-making processes through Operations Research (OR) raises the need for specialized management systems. To support multiple vertical industries, OR service providers must handle a diverse portfolio of solvers having widely varying resource, computational, and availability requirements. To address this fundamental challenge, we present OCTOPUS, a state-of-the-art cloud-native solver orchestrator. Leveraging Kubernetes, OCTOPUS features event-driven autoscaling to align resources with application demand, dedicated queue management to meet service-level agreements, and robust reliability mechanisms to prevent service interruptions. This demonstration showcases OCTOPUS's capabilities in a production-like environment, illustrating its full end-to-end workflow and commenting on its real-time performance.**

***Index Terms*—Operations Research, Optimization, Kubernetes, Microservices, Service Orchestration**

I. INTRODUCTION

Operations Research (OR) is one of the cornerstones of modern decision-making processes. OR optimization algorithms, ranging from general nonlinear solvers to specialized metaheuristics, are routinely being used to make critical business decisions in multiple industries, such as logistics [1], energy systems [2], telecommunications [3], and so on.

Cloud computing, with emphasis on microservices architectures, plays a crucial role in the modern OR stack. An OR service provider typically supplies APIs exposing multiple solvers for high-level optimization problems (e.g., Google's OR API [4]). In this context, microservices architectures allow us to abstract optimization algorithms and user-facing application components as loosely-coupled services, making the OR-based decision-making system suitable for cloud-native deployments.

However, the operational requirements of decision-making systems introduce non-trivial technical challenges in the design of a microservices architecture. For example, solvers for different business use-cases might have vastly different requirements in terms of, e.g., execution time (e.g., one minute vs. several hours) and CPU/RAM consumption, which requires fine-grained resource management. Moreover, the system should guarantee near-zero interruptions or downtime for critical API endpoints and active solvers, especially for long-running executions. Finally, as cloud infrastructure is billed by usage, the system should avoid resource over-provisioning while guaranteeing the above-mentioned requirements.

Despite the availability of cloud-based OR solutions [4], [5] and research on orchestrating predictive analytics [6], there is presently no off-the-shelf orchestration platform that can seamlessly handle heterogeneous OR solvers while supporting business-driven scalability and reliability requirements.

To address this challenge, we present OCTOPUS, a state-of-the-art OR services orchestrator. Specifically, OCTOPUS is a cloud-native application based on Kubernetes (K8s) for managing, executing and monitoring a portfolio of optimization algorithms, enabling full control and observability over the complete OR lifecycle. In this paper, we illustrate OCTOPUS's design and demonstrate its end-to-end workflow.

II. OCTOPUS ARCHITECTURE

OCTOPUS is an orchestrator for the execution of business optimization algorithms (henceforth, "solvers") within a K8s environment. It supports synchronous and asynchronous solver execution requests via REST APIs, provides priority-based queue management for each solver to ensure Service-Level Agreement (SLA) compliance, and manages real-time communications throughout the full execution lifecycle. In the following, we assume the reader is familiar with core K8s concepts (Pod, Node, ReplicaSet, Deployment, Job, etc.). We refer to the official K8s documentation for detailed explanations [7].

OCTOPUS implements a microservices architecture. On the infrastructure level, we containerize solvers as Docker images and release them as K8s Deployments, thus treating solvers as full-fledged services rather than one-off tasks (like K8s Jobs). Adding new solvers to OCTOPUS is straightforward: we use a dedicated library to wrap standardized I/O and messaging interfaces on top of arbitrary solvers, allowing OCTOPUS to register and expose them as services. Upon startup, each solver microservice establishes a dedicated message queue for processing execution requests. This allows the microservice in charge of the execution strategy to identify all active solver containers and assign a specific execution to be run on a designated solver container instance. Figure 1 overviews the main OCTOPUS components.

A. Execution workflow

The process begins when OCTOPUS receives an execution request. After an information exchange that includes a specified solver and its inputs, the request is accepted. OCTOPUS then identifies the target solver for execution and proceeds to check for available container instances to serve the request. If no instances are available or active, the event-driven autoscaler

978-3-903176-75-1

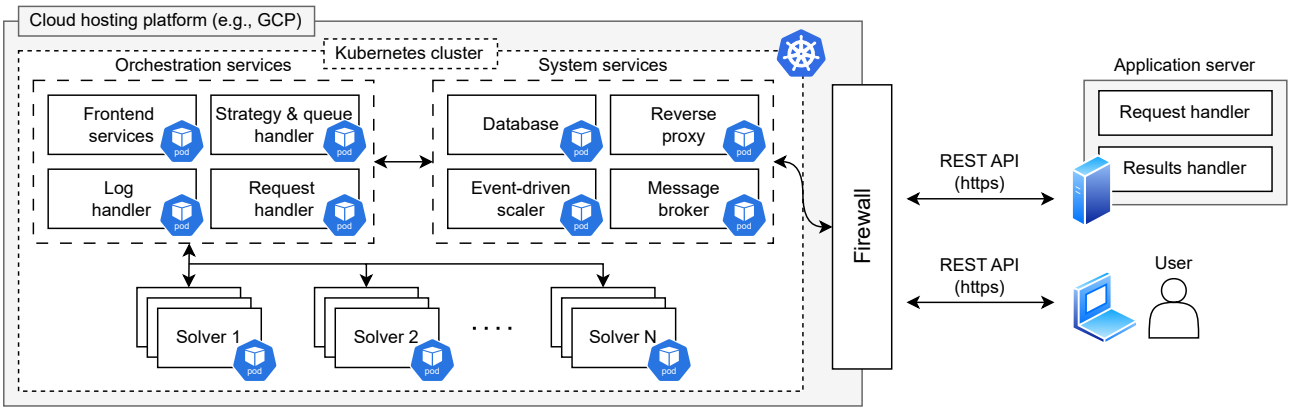


Fig. 1. Overview of OCTOPUS. End-users or managed applications request solver executions via REST APIs. A reverse proxy forwards requests to a message broker, which manages the different service queues. The event-driven scaler signals the number of solver Pods to be instantiated, according to the current request load. The orchestration services dispatch the solver execution, monitor the execution status, collect logs, and forward solver results.

(more details in Section II-B) detects the increased demand for a specific solver and signals K8s to scale up the number of Pod replicas for the corresponding Deployment. Then, a new solver instance is launched. Once OCTOPUS recognizes the availability of this new instance, it sends an asynchronous message prompting it to start the new request. The instance acknowledges its readiness to OCTOPUS, and following an internal data exchange to pass the necessary inputs, it confirms to OCTOPUS that the execution has started.

During execution, OCTOPUS receives real-time logs from the solver instances and displays them to the relevant users in the frontend interface. Each instance is capable of determining whether the execution has completed with or without errors, and sends a corresponding positive or negative status to OCTOPUS, along with any output files generated. In the event of an infrastructure-related failure where the entire container is shut down or restarted, OCTOPUS will detect the instance's termination through its existing monitoring mechanisms and recognize that the requested execution cannot be completed.

The workflow concludes by making the solver's results available to the requester, either through an application's user interface or by communicating them synchronously or asynchronously. The requester can then retrieve the solver's output files and supplementary data to complete the request.

We now describe in detail how OCTOPUS addresses our foundational requirements for a cloud-native decision-making system: scalability, cost-efficiency and reliability.

B. Scalability and cost-efficiency

K8s's Horizontal Pod Autoscaling (HPA) allows us to scale a Deployment's Pods according to resource usage (e.g., CPU, RAM). Though useful, it does not allow for fine-grained control of scaling events, which is necessary for SLA compliance and cost-efficiency. Consider a scenario with many concurrent long-running solver execution requests having low resource consumption. This would not trigger HPA, leading to all requests being assigned to the same replica, hampering not only performance (long waiting times due to queuing), but also reliability (a failure of that replica would cause all requests to fail). Similarly, consider a single short-lived execution request with high resource consumption: this would trigger HPA to instantiate additional replicas even though there is a single request in the system, leading to unnecessary costs.

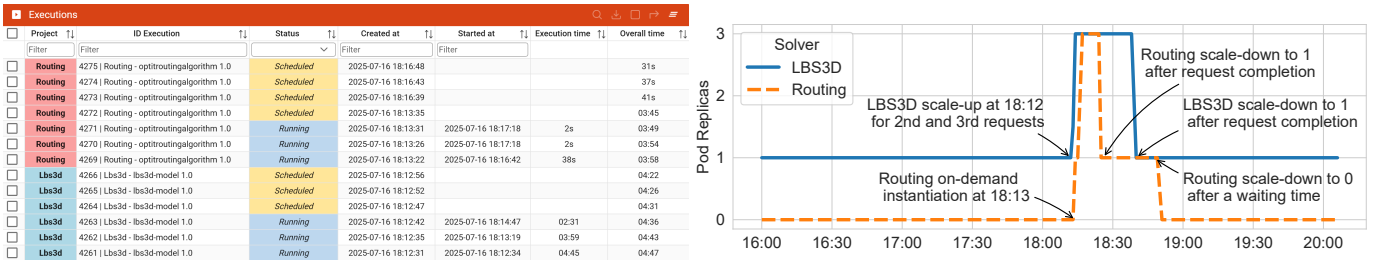
Event-driven autoscaling. To overcome these limitations, we leverage KEDA [8] for implementing autoscaling based on application-level events. Specifically, to guarantee parallelism and resource isolation, we dimension the number of solver Pods based on the number of execution requests accepted in the system, in accordance with solver-level SLAs. In this context, we trigger KEDA by continuously monitoring a database view returning the number of replicas required by a specific solver. The database is polled every few seconds, to ensure that scale-up requests are processed promptly. Conversely, we introduce a larger waiting time before scale-down, ensuring that resource deprovisioning does not introduce disruption and avoiding repeated scaling events in the presence of fluctuating traffic. This implementation, which matches instantiated resources with application-level demand, strikes a good trade-off between SLA compliance and cost-efficiency.

C. Reliability

We focus on two aspects of reliability particularly relevant for decision-making systems: *a)* API microservices must always be reachable and respond to user requests, and *b)* solver microservices must not be migrated or terminated during execution. These requirements cannot be guaranteed by the default K8s configuration. Indeed, during a scale-down event, K8s can move Pods between Nodes via pod eviction and migration. As this process moves only the containers but not their state, it will disrupt ongoing solver executions and application-level communications. As such, to satisfy our reliability requirements, we implement the following measures.

Pod disruption budget. We explicitly enforce that critical system and orchestration Pods must have at least one active replica at all times. The absence of these replicas, even if for a few milliseconds (e.g., due to migrations or restarts) can disrupt communications and lead to service interruption.

K8s annotations. We leverage the safe-to-evict K8s annotation to ensure that Pods with running solvers cannot be evicted from their Node during a scale-down event. However, this



(a) OCTOPUS GUI. Executions are sorted by request creation time.

(b) Pod allocation over time, with events from Fig. 2a.

Fig. 2. Using OCTOPUS for launching multiple solvers in parallel and monitoring the K8s cluster's resources.

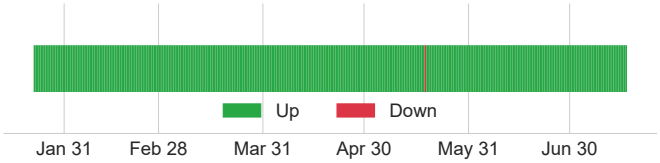


Fig. 3. Uptime (99.9%) of OCTOPUS's orchestration services as of July 2025.

only controls the Node autoscaler, but not the Pod autoscaler within an individual Node (i.e., KEDA). We therefore need the following additional reliability mechanism.

PreStop hooks. We implement a set of actions that each solver Pod must execute before stopping: *1)* notify OCTOPUS that the Pod is about to be terminated, such that no further optimization requests are forwarded to that Pod; *2)* if the solver is currently executing, we wait until it terminates. To ensure termination, we set a worst-case timeout (profiled for each solver) after which we stop the solver Pod irrespective of its state.

III. DEMONSTRATION

The demonstration showcases OCTOPUS in a production-like scenario, illustrating the workflow described in Section II. For the demo, we deploy OCTOPUS on Google Cloud Platform (GCP). We consider two productionized solvers for real-world use-cases from logistics: LBS3D and Routing, which solve variants of the 3D Bin Packing Problem [9] and the Vehicle Routing Problem [1], respectively. LBS3D must always have one Pod active, while Routing is instantiated on-demand, illustrating OCTOPUS's capability of handling diversified service-level requirements. For both LBS3D and Routing, we assume an SLA mandating up to three parallel solver executions.

The attendees will interact with OCTOPUS using a commodity laptop, either through a GUI application or by directly querying the REST APIs. The GUI allows a user to request one or multiple solver executions and monitor their status (e.g., scheduled, running) in real time. We also provide access to the GCP admin dashboard, allowing attendees to monitor in detail the internal status of OCTOPUS's K8s cluster. For example, attendees can observe in real time resource provisioning and deprovisioning in specific Deployments in response to one or multiple solver requests. Finally, we provide interactive solution visualizers for both LBS3D and Routing, allowing attendees to observe OCTOPUS's workflow end-to-end.

Fig. 2a illustrates the GUI that attendees will use to launch solver executions. In the screenshot, we observe six competing execution requests for LBS3D and Routing. Fig. 2b shows

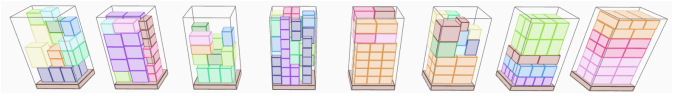


Fig. 4. Interactive visualizer for LBS3D solutions.

the number of active Pods, extracted from GCP's dashboard, for LBS3D and Routing during the requests' lifetime. As Routing is an on-demand service, requests are queued and a scale-up event is triggered. Conversely, LBS3D must always be available: indeed, the earliest request is served in a few seconds, while a scale-up event is triggered for handling the others. For both solvers, in Fig. 2b we observe around 18:15 two rapid scale-up events which, as described in Section II-B, promptly match the instantiated resources with application-level demand, up to the maximum three parallel executions.

As requests are processed, OCTOPUS frees resources according to the waiting time logic described in Section II-B. For Routing, we can observe the waiting time elapsed before completely deprovisioning the service, which would allow us to serve intermittent demand without reprovisioning from scratch. Conversely, LBS3D Pods are scaled down but never deactivated, since by SLA we always have one active Pod.

Fig. 3 shows the uptime of OCTOPUS's orchestration services extracted from GCP's dashboard. Thanks to the mechanisms described in Section II-C, we suffered only one downtime event (a GCP incident on 19/05) in the past six months.

Finally, Fig. 4 shows the interactive visualizer of LBS3D. Attendees can inspect a 3D rendering of the item loading configurations in the bins, and search for specific items.

REFERENCES

[1] P. Toth and D. Vigo, *Vehicle Routing: Problems, Methods, and Applications*. Society for Industrial and Applied Mathematics, 2014.
[2] C. Bordin, A. Gordini, and D. Vigo, "An optimization approach for district heating strategic network design," *European Journal of Operational Research*, vol. 252, no. 1, pp. 296–307, 2016.
[3] Y.-F. Liu *et al.*, "A survey of recent advances in optimization methods for wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 42, no. 11, 2024.
[4] "Google OR API," https://developers.google.com/optimization/service.
[5] "Gurobi cloud," https://www.gurobi.com/solutions/gurobi-instant-cloud/.
[6] V. Reddy Chintapalli *et al.*, "Orchestrating edge- and cloud-based predictive analytics services," in *2020 European Conference on Networks and Communications (EuCNC)*, 2020, pp. 214–218.
[7] "Kubernetes documentation," https://kubernetes.io/docs/home/.
[8] "KEDA: Kubernetes event-driven autoscaler," https://keda.sh/.
[9] T. G. Crainic, G. Perboli, and R. Tadei, "Extreme point-based heuristics for three-dimensional bin packing," *Informs Journal on computing*, vol. 20, no. 3, pp. 368–384, 2008.