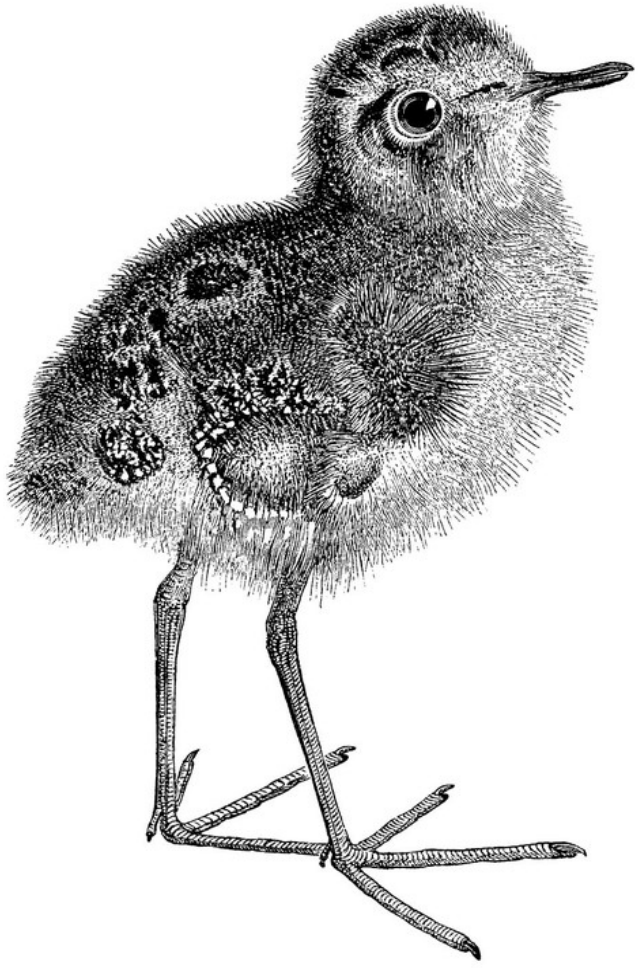
O'REILLY®

Designing Cloud Native Delivery Systems

Architectural Patterns for Highly-Distributed Software Delivery



Early Release

RAW & UNEDITED

Bryan Oliver &
Nic Cheneweth

Designing Cloud Native Delivery Systems

Architectural Patterns for Highly-Distributed Software Delivery

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Bryan Oliver and Nic Cheneweth

Designing Cloud Native Delivery Systems

by Bryan Oliver and Nic Cheneweth

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editors Megan Laddusaw

Development Editor: Angela Rufino

Production Editor: Clare Laylock

Interior Designer: David Futato

Illustrator: Kate Dullea

July 2026: First Edition

Revision History for the Early Release

- 2025-04-30: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098167110 for release details.

978-1-098-16705-9

[FILL IN]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Delivery Systems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form —the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at arufino@oreilly.com.

Introduction - The Perils of Delivery

In this book, we will be diving deep into Delivery Systems. But what do we mean by a "Delivery System"? In the physical world, they vary greatly. For example a Nuclear Power Plant requires an engineer in order to operate (and deliver energy to it's customers). But a toy water gun needs only a toddler to deliver water to it's unsuspecting victim. So what is it about the delivery of applications that requires we take a disciplined software engineering systems design approach, instead of simply configuring our deployments to pull the trigger on success and move on with our day? Lets start with some simple scenarios:

In todays api driven world, most organizations are in the business of shipping applications and apis. These apis serve traffic for customers (internal or external). At first, following agile principles,

most of them start out quite small. An api is designed, developed, packaged, and shipped to a runtime platform such as a VM or container orchestrator. It might also take traffic as a public endpoint, and write changes to a database. The api will also have some supporting services around it, such as a dns resolver, and a load balancer. And given the thousands of lessons learned about writing software in the last 20 years, we can safely assume the api is written to support multiple replicas deployed in parallel, so our load balancer can balance the requests and keep our api from being overrun.

When traffic increases on this application, our team was wise enough to deploy it with an autoscaler. The autoscaler will respond to the increase in traffic and resource usage, and create more replicas of the api to handle the new load. All is well!

Until the compute limit on the runtime environment gets hit. And if we don't have an autoscaling mechanism on our compute limits, we'll start to see scheduling failures in our api's autoscaler. And as replicas of our api begin to fail to deploy, the request count continues to pile up on the apis that are deployed. This leads to slower and slower response times, maybe even some replicas of our api crashes. If the runtime in use has a scheduler, the provisioned resources of the api that crashed suddenly become available, and another api that was trying to deploy grabs them on init after it had been failing to start for hours. Further hurting our already struggling api.

Our application team works quickly to increase the performance of the apis, because they can't wait for the infrastructure team to improve the compute bottlenecks. So they rollout the change to production quickly. Only, they realize there's an error in their hotfix. Now they have to manually roll it back and redeploy the old version of the application. To do this they have to revert the change, rebuild the application, and redeploy it to production, because the deployment of their application is tied to their static CI pipeline.

Once they have fixed it, they find that they can't get their new fix deployed, it seems to be waiting in a scheduling queue. The increase in traffic has increased compute demand across the board. Other apis are also completeing for resources, and the infrastructure team is struggling to keep up with manually scaling the runtime to meet the increasing compute needs.

Definining Delivery Systems

Throughout this text, we will explore the common issues seen above and delve into how to solve them. But first, we need to have a clear understaning of what a Delivery System is.

The design of a delivery system comes down to 2 major areas, Deployment Patterns and Scheduling.

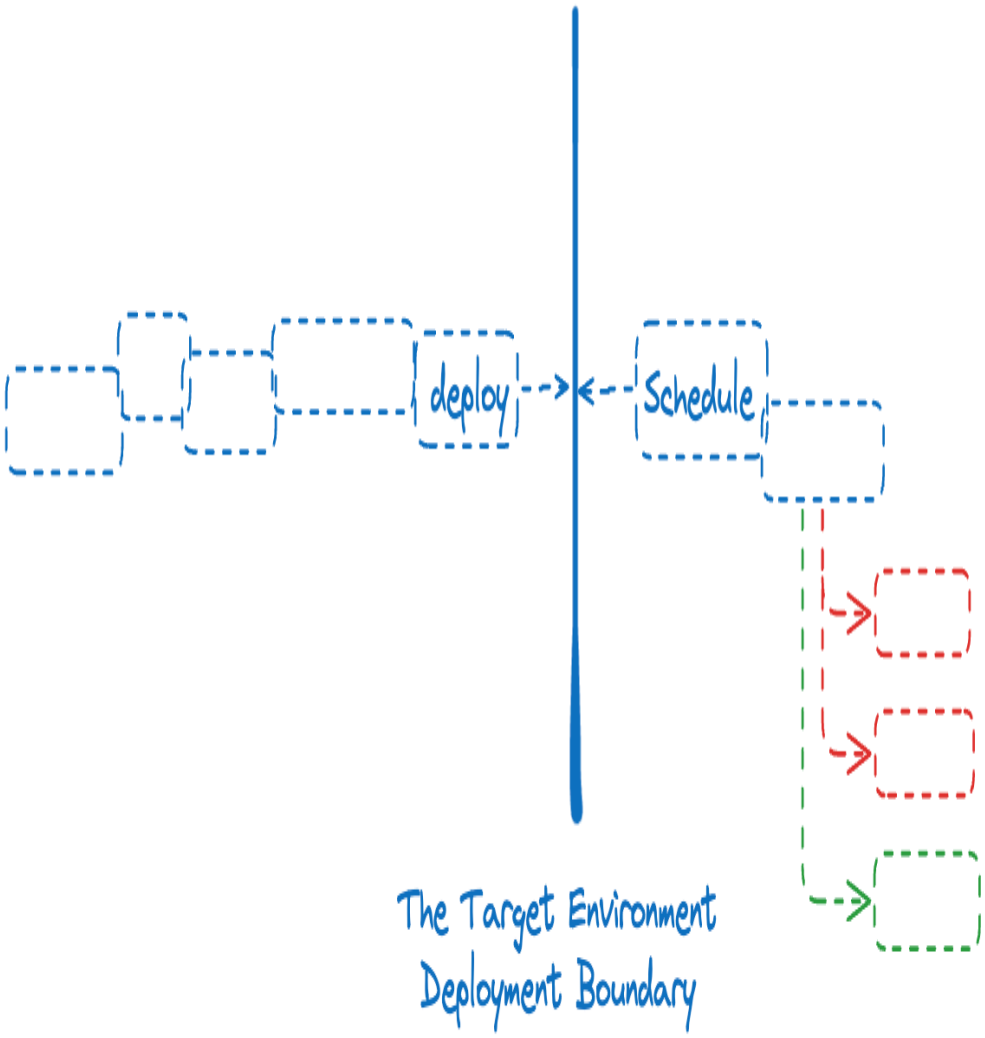


Figure 1-1. Delivery Systems are made of deployment patterns and scheduling algorithms

The business of defining how our software gets to the target runtime is handled by our deployment patterns. Once it gets there (successfully), the deployment must then be handled by an

algorithm that decides where the application will (physically) run. Sometimes this placement might be an abstraction (i.e. a scheduler that sits in front of a lower level scheduler), but almost always scheduling will be based on a number of factors (metadata, compute availability, location, task priority, proximity to other services, etc). This is all in the realm of scheduling.

When we properly combine the 2 areas, we can have a successful delivery system for our software. Failure to consider one side of the coin, will ultimately result in friction and problems down the line.

Why Delivery Systems

The term "Delivery System" is from our perspective, less commonly used. More commonly we hear terms like Continuous Integration, Continuous Delivery, Scheduling, Orchestration, CI/CD, and so on. And increasingly the title of devops engineer has become taboo, replaced with evolutions of that title like Site Reliability Engineer, Platform Engineer, or Cloud Engineer.

The complexity of application and cloud systems has exploded, and so to have the requirements we have in deploying to them. As developers we must now think about an insane number of factors and still deliver applications at tremendous (insurmountable?) speeds, with pressure set by expectations from leaders who want to make changes every day. And on top of all this, we must now think about how to deliver applications across many datacenters, zones, regions, and even continents. Soon we'll be sending continuously deploying our applications into outer space (*https://techport.nasa.gov/projects/11772*)!

As these complexities continue to grow exponentially, we felt it would be wise to document and explain the delivery systems we see in the wild today. This includes building a knowledge base about the deployment patterns and scheduling algorithms they are built on top of. This will by no means be a permanently "complete"

work on delivery systems, as all things this space will continue to evolve and improve over time.

Our hope is that readers gain a healthy understanding of delivery systems, thus enabling them to make better decisions when placed in difficult and demanding software focused organizations.

To quote Fundamentals of Software Architecture: "When studying architecture, readers must keep in mind that, like much art, it can only be understood in context. Many of the decisions architects made were based on realities of the environment they found themselves in."

This concept is essential to remember as we delve into the sea of delivery systems and schedulers. And no context is the same. We can only provide you with the tools and techniques to make better choices. While it is becoming rare for an engineering team to design a delivery system from scratch (although the authors must nod that this is how Argo and Spinnaker were born), each existing system has patterns and scheduling expectations that will affect the architecture of software. There are 100s of implementations of these systems, and more come out all the time.

The Importance of Evolution in Delivery Systems

Many a time, we'll see dedicated teams with multiple years of backlog when it comes to designing a better developer experience, and no end in sight. This author has seen it all, and many readers may be inclined to first think: the answer is abstraction! Just hide all of those pesky platform and pipeline details from the developers, they don't know what they are doing anyway. And a year later we have 100s of bugs filed by development teams that have no idea how to debug or triage their deployments, because all of those details have been abstracted away from them! When we abstract

away all of the details of delivery from development teams, we are signaling to them that they *don't have to care* about it anymore. This is no mistake either, software engineering principles call for abstraction as a means to provide the creators of apis and interfaces with a safe mechanism to ignore the implementation details of downstream dependencies.

The proper answer is evolution. We won't deep-dive on evolutionary architecture, there are other resources on that topic (including several O'Reilly books), but it's important to understand the essential concepts when it comes to designing a delivery system. As consumers of delivery platforms, we often see teams do a feature comparison. For example, we've seen a team build a feature matrix comparing Argo to Flux, or Jenkins to CircleCI.

And then immediately follow up with "you have an api right?" Any vendor worth their salt will nod their heads virgorously, of course we have apis! And we move along, happy that we can make changes to the technology we chose.

The irony of this type of decision making, is we often fail to consider simplicity and ability to change, when it comes to the decision. Those well-versed in software architecture principles are quick to think about how to design software that is easy to change. This is a core principle! But then we fail to apply the same type of thinking to our delivery systems. We tend to think of delivery systems as something we must configure with a heavy hand, so naturally customization is the first thought that comes to mind.

What if we applied a different set of principles to our delivery systems? Before we define them, lets ask ourselves some conrete (example) questions:

- Do we really need GitOps if there's only one production cluster?

- Is it true our developers aren't capable of using Helm, or have we simply not given them the trust and opportunity to do so? And is abstracting a templating language (ironically, usually with another form of templating) really going to help them, or hurt them?
- Are the teams actually asking for a deployment GUI? Or did we just assume that?

These are just some examples. The point is, teams responsible for delivery systems often make a *lot* of assumptions when it comes to their intended user base. This usually leads to designing a poorly optimized deployment process that doesn't actually meet the needs of their users, it just meets the desires of the team managing it.

For example, consider the first question. We'll see organizations adopt a "gitops" approach to delivery and deployment. This leads to the creation of many repositories, folders, branches, etc. that are all syncing to various downstream kubernetes environments. And the complexity that emerges from this decision is non-trivial. For some, it was the right choice. In highly distributed environments (i.e. 100s of clusters) it helps simplify the management of such a complicated topology. But for most, the old "you are not google" is sometimes forgotten. If the team has one, or 2 production environments, the complexity of introducing a GitOps process into the deployment workflow may not actually be worth it!

But audit! Compliance! Ok, sure. One could argue that by adopting something like GitOps, we get auditability of all of the changes across our environments.

Step back and consider that all of those environments were probably already created with infrastructure as code. And the changes to the applications being deployed on those environments have automated pipelines defined (in Git!), and the changes to those applications are tracked. A simple deployment pipeline's

history can be retained for however long you want. A simple pipeline run can also be rolled back.

The point is, we have to set aside "oh shiny" and actually think about our own environment. Just because Netflix had massive success with Spinnaker, doesn't mean it's the best microservices deployment system for your org. And a lot of the "benefits" or "features" in this cloud native space are already there in one way or another, if you look at the underlying pattern to understand what it's providing, and how.

Defining Deployment Patterns

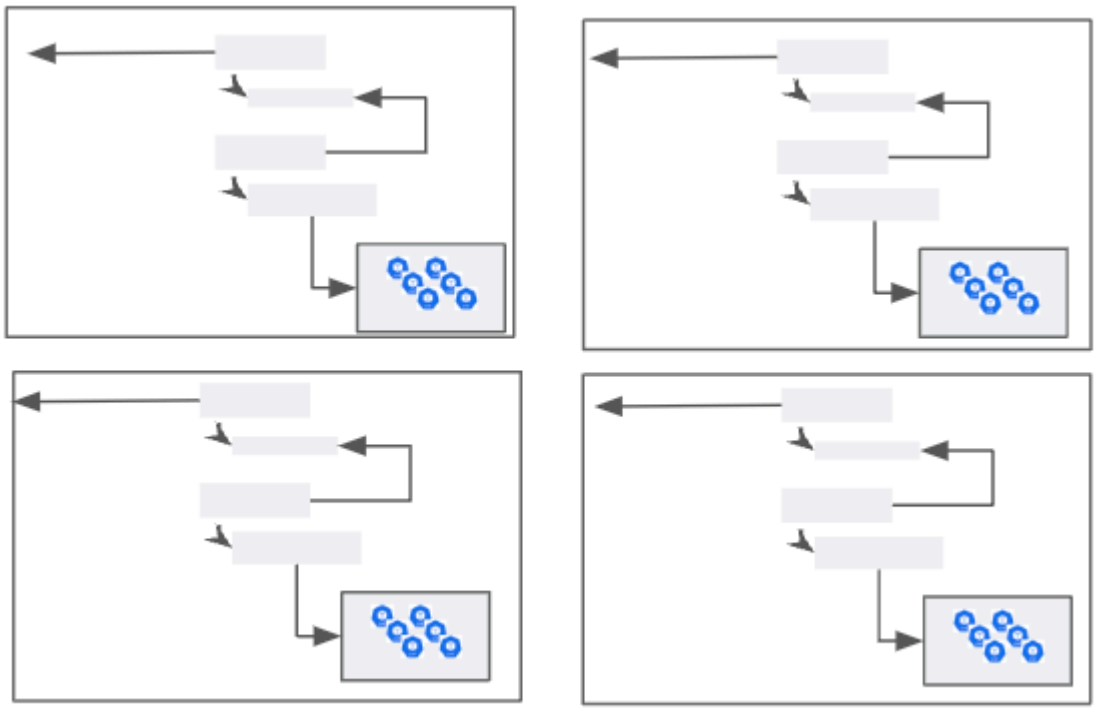


Figure 1-2. Patterns help us deliver software to target environments that make up varying distribution problems

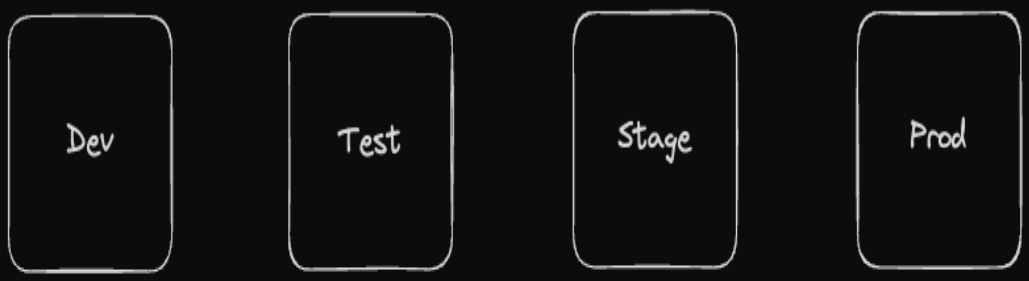
We've mentioned 2 components to delivery systems, these are deployment patterns and schedulers.

A Deployment Pattern is a combination of architectural choices and controllers that solve a unique application distribution problem.

Distribution problems can vary greatly. For example, we might need to deploy to 1500 IoT devices in a forest and within the same company we also need to get a container deployed to every major AWS region. Both of these problems present very different (and both challenging) puzzles to solve. In the forestry example we will need to consider a high degree of fault tolerance, localized failover mechanisms, and some form of autonomous decision making at the local level. In the AWS region example we'll need to build extremely robust load and scaling tests that run against our regional deployments.

By identifying the architectural patterns behind a delivery system we can break down the value and tradeoffs each provides and make better choices for our distribution problem.

If this is all you are doing



Why do you need this

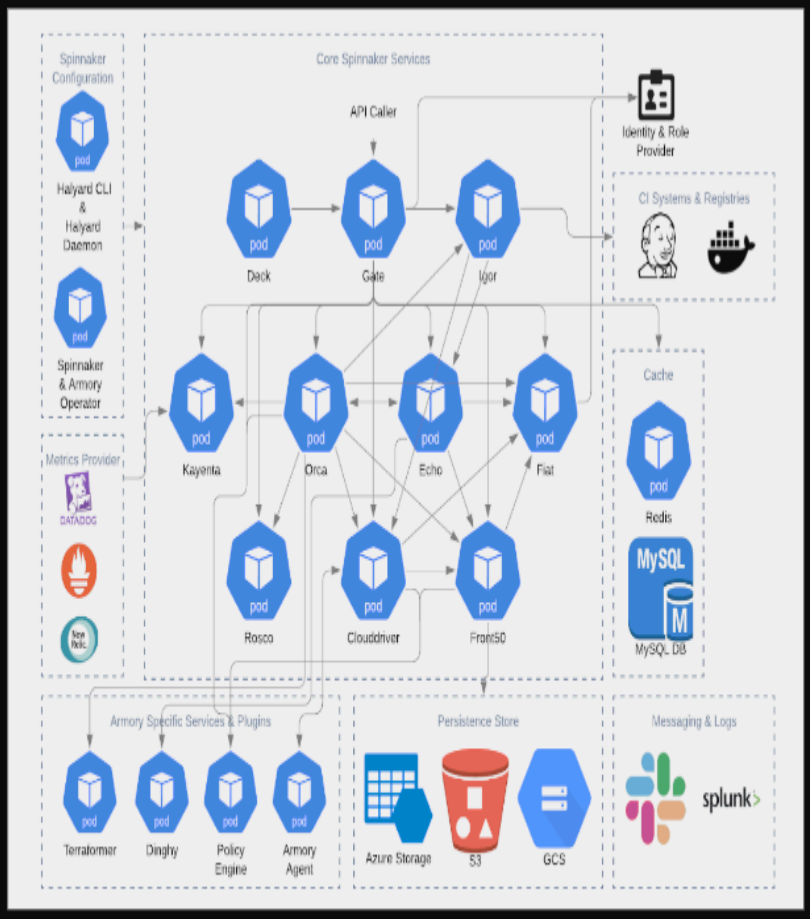


Figure 1-3. Deployment platform complexity

At Thoughtworks, we will often defer the decision to adopt a deployment platform until we absolutely *have* to. Meaning, many delivery challenges are solved with the resident CI system in place, and introducing a dedicated (seperate) deployment system just complicates matters. For many traditional enterprises that are just deploying APIs and Frontends, this is a perfectly reasonable (and manageable) solution. It's low cost, low complexity, and very easy to change. This works when the enterprise only has a small handful of target environments to deploy applications to. As this number grows (i.e. companies that are global and have a cloud presence in many regions) the ability to change and manage simple pipelines becomes harder.

By adopting technologies that provide simple interfaces to the environments we are targeting, it's easy to pickup a rollout technology down the road. For example, investing in helm allows us to deploy to kubernetes environments with any (major) CI system. However, nearly all notable cloud native deployment systems *also* support helm charts. So we can very comfortably defer the decision of using a dedicated deployment system, because we know we can continue to use the templates and charts when the day comes that such a system is actually needed.

This is the nature of evolutionary thinking when it comes to delivery systems. The reader should approach this topic with a sense of skepticism. A delivery system, per se, is not just the likes of Spinnaker, Argo, Flux, etc. Any major CI system *can also* meet the needs of a delivery system. And almost certainly in most contexts, the decision to use a dedicated deployment system can be deferred, because the ability to deploy is already inherent in any CI and CD tool.

If you take away one lesson from this chapter, it's:

Delivery systems are not exclusive to deployment platforms. Delivery systems are any tool that can meet the requirement of deploying your software to a target environment.

Domains of Delivery

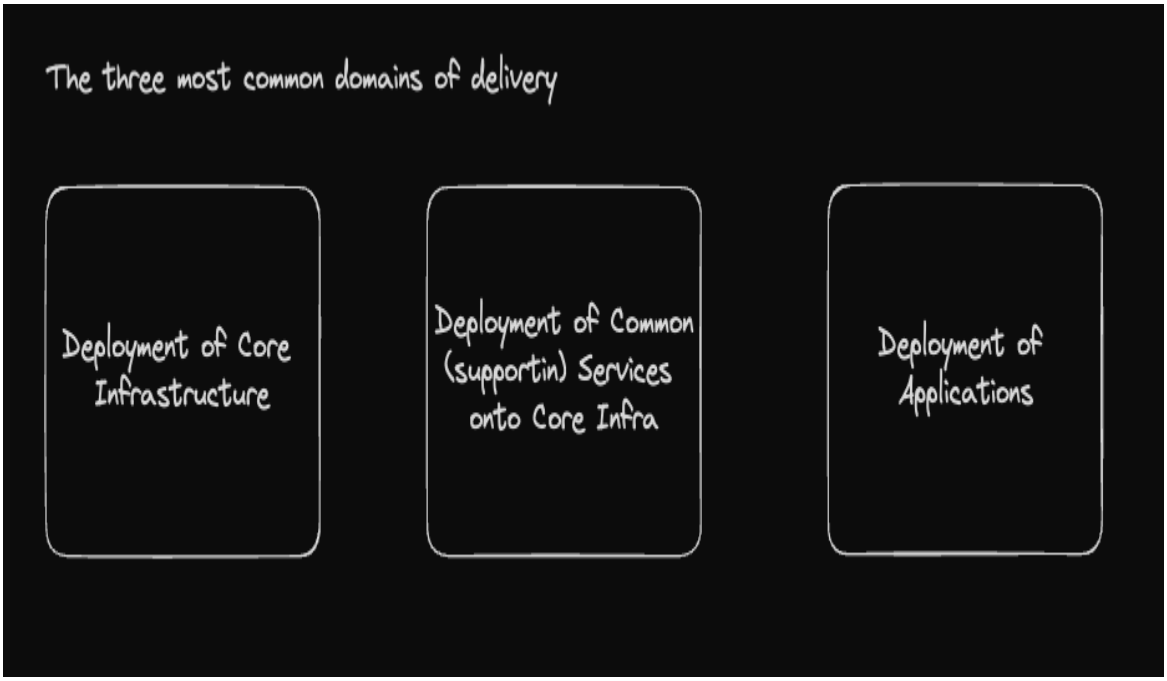


Figure 1-4. Domains of delivery

One of the mistakes we'll often see made is the assumption that the selection of a delivery system is a unilateral solution to all areas of "delivery" across an organization.

This is a mistake.

Each area of delivery has unique requirements and problems. Even within the *same* area of delivery we will see very different delivery problems in front of us. This goes back to earlier why we make massive "feature" checklists to certify that the delivery checks every little box (can it run terraform, and helm, and apis, and etc.).

If we apply *domain thinking* to delivery, we can boil the space down into 3 key domains: Infra, Supporting Services (often coupled with

the infra), and Applications. Now, within each of these domains is a myriad of problemsets. For example the delivery patterns of a Machine Learning application will be almost alien to that of a RESTful api. But for the sake of simplicity we'll consider those as unique subdomains underneath applications.

> Domain thinking is a concept that has stood within the software industry for decades. Pioneered and documented by the likes of Eric Evans, Martin Fowler, and many others, Domain Driven Design continues to be applied to all areas of technology and software, simplifying conversations and driving a common language when it comes to defining boundaries, models, and systems. To read more, checkout *https://martinfowler.com/bliki/DomainDrivenDesign.html* or the Domain Driven Design (Eric Evans) book.

Consider for example the following:

- Our infrastructure is deployed with Terraform
- Our supporting services are deployed with Flux and Bash scripts
- Our applications are deployed with Helm Charts

It's easy to take this example and map the 3 examples to our 3 domains in Figure 3. The execution of Terraform falls under our Deployment of Core Infrastructure domain. The execution of bash scripts falls under our Deployment of Common Services domain. And the execution of Helm Charts falls under our Deployment of Applications domain.

Each domain, has quite unique requirements. For example, terraform is both declarative and stateful. It requires the execution and approval of a plan, and it syncs its state with a statefile(s) as infrastructure is created/updated/deleted. Deployments lock

changes to infrastructure until complete (out of neceisity, we wouldn't want 2 runs to modify the same server!).

Now consider the requirements of our common services. We are executing Flux and Bash scripts here. The scripts are necessary to bootstrap flux onto our Kubernetes environments. Once installed, Flux will reconcile the deployment of our common services onto the cluster. Ther requirements of this type of deployment are *extremely* different from the Terraform one! We don't have state in this scenario, and the definitions of our deployments are not declarative. Flux continuously reconciles the resources onto our environment, whereas Terraform will run drift detection only when we are executing a Plan (not continuously). The removal of an application from our cluster (say manually) will be remediated almost immediately by the controllers that Flux deploys, always watching that our desired servies are present.

Lastly, our Application domain. This requires the execution of Helm Charts onto our downstream environments. While there are some parallels to the supporting services domain, the difference between Application Deployments and Supporting Services are in how they are managed and rolled out. For example, our Applications take HTTP traffic from real users. So it's wise to run canary tests when deploying new versions of our application. Many of our supporting servies, however, do not have the ability to run Canary tests. So in that domain we rely on things like infrastructure testing, user acceptance testing (the user being developers), api incompatibility scanning, etc.

From just these 3 oversimplified examples, we begin to show the complexity delivery systems present, even to the simplest of organizations. It's easy to buy a delivery tool and say it must be used across the organization. But time and experience have proven it's much wiser to select the right tool and pattern for the job (domain!).

Defining Schedulers

It is often that case that we will see long discussions about the pairings of various CI and CD systems. But oft overlooked is the scheduling and orchestration algorithms used on the target environments. Whether it's Kubernetes or Serverless, there are scheduling algorithms working behind the scenes when your delivery system provides a deployment to a target environment. And if the target environment doesn't have the available resources for your deployment, then who cares what delivery system you are using! It's just going to sit there anyway. Schedulers are an extremely critical component to a complete delivery system. Without them, our deployments are just empty promises.

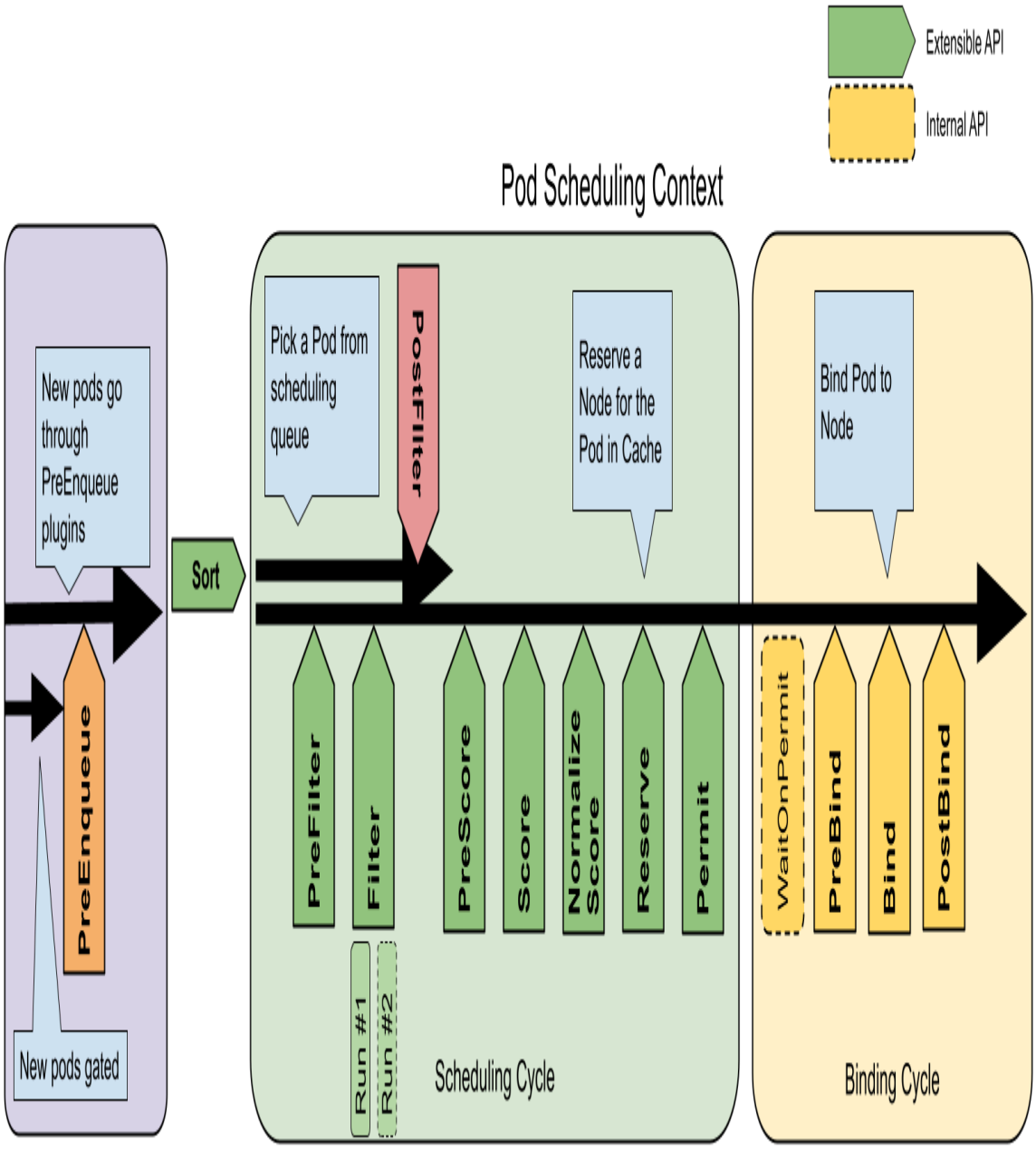


Figure 1-5. The kubernetes scheduler cycle

As an example, In Figure 4, we show the various interfaces the Kubernetes scheduling framework exposes. In it, we are permitted to deploy Plugins to each of these interfaces, allowing us to manipulate and change the outcome of pod scheduling on a given cluster. This is an extremely *powerful* capability, and an important one, because no two workloads are the same. While many teams

may get along just fine with default scheduling algorithms and settings, knowing that they can be changed is incredibly useful knowledge. As teams grow and scale into larger distributed platforms, it becomes more and more necessary to understand the fundamentals of scheduling.

In designing a delivery system, it's important for us to understand the downstream scheduling requirements. Failure to do so can result in waiting for deployments, critical time we don't have. We'll take a look at common scheduler patterns for general computing, web/http, data intensive, and AI/ML workloads.

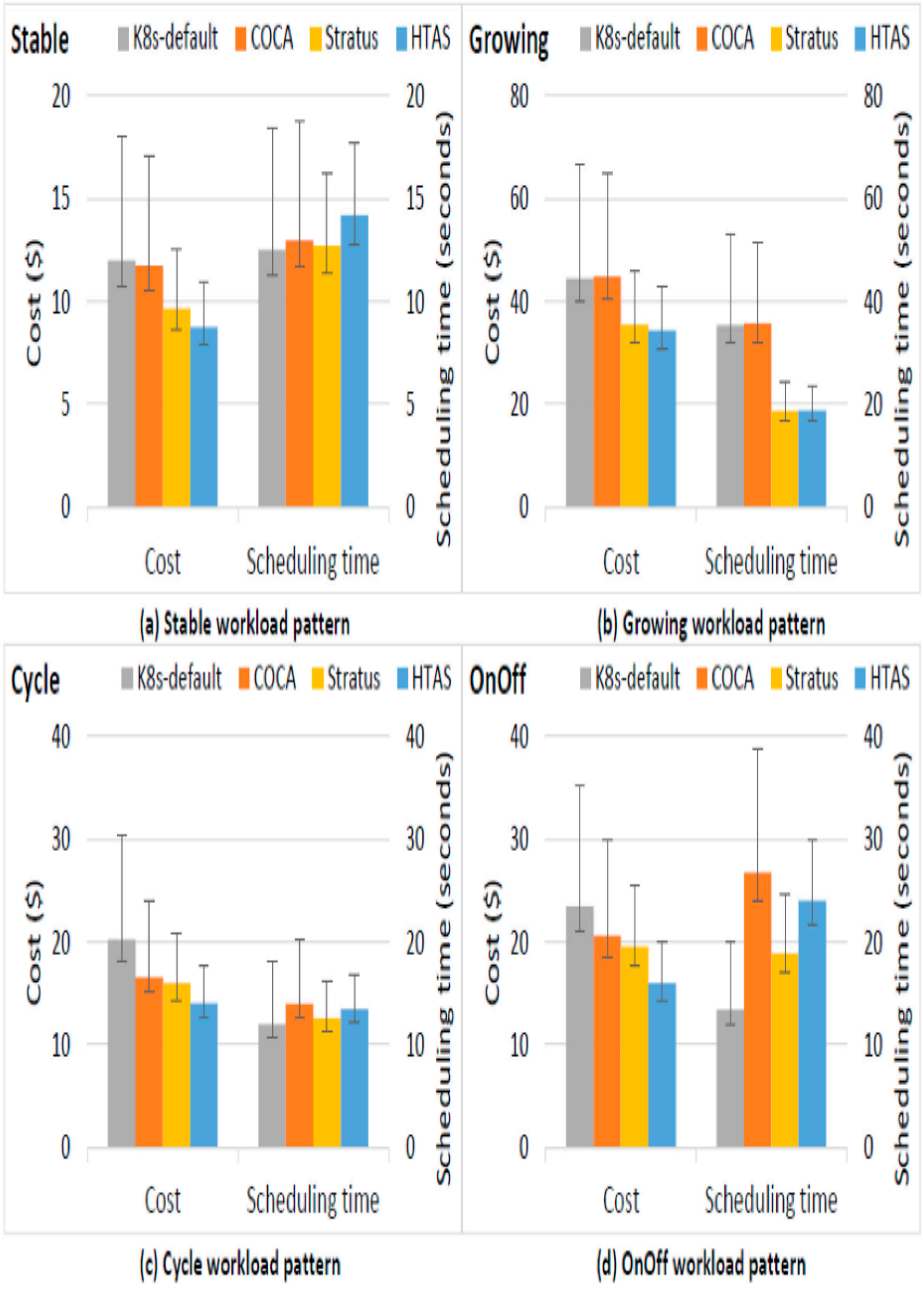


Fig. 3. Total cost and average task scheduling time.

Figure 1-6. Comparing scheduler algorithms is an interesting area of research

It's common to use the default when it comes to complex topics like scheduling. This is simply because designing and implementing a custom scheduler is *very hard*. So we'll take a look at the different schedulers available in the wild, both from a research perspective as well as active open source projects. In Figure 5 we see several scheduling algorithms compared across use-cases, and see that the default kubernetes scheduler actually doesn't do so well against several alternatives.

While deep-diving into scheduling is out of scope for this chapter, we'll later dive into why this is the case (and why it's not a bad thing), and how we can leverage this knowledge to our advantage when designing delivery systems.

Summary

In this book, we'll discuss and document the presently available Delivery Systems, represented through powerful combinations of Deployment Patterns and Schedulers. We'll discuss the Deployment Patterns in Part 1, the Schedulers in Part 2, and the combinations of each in Part 3 (defining the complete Delivery System approaches).

About the Authors

Bryan Oliver is an experienced engineer and leader reprising roles such as cloud engineer and engineering executive at publicly traded companies. He has spent his career developing mobile and back-end systems, building autonomous teams, and modernizing legacy companies to adopt modern techniques and cloud strategies. He is now focused on the delivery and cloud native space at Thoughtworks EMPC where he is a principal architect for delivery infrastructure.

Bryan is also a member of Kubernetes Sig Network, where he contributes to the network-policy-api subgroup. And as an experienced cloud native engineer he's given numerous cloud native focused talks at conferences including CdCon, GitopsCon, GitOpsDays, Codefreeze, DevOpsDays, and Refactr.tech.

Nic Cheneweth is a Principal Consultant and Technology Strategist at ThoughtWorks and is the founding infrastructure contributor to ThoughWorks Digital Platform Strategy. His undergraduate studies are in computer science, and he holds an MBA, doctorate, and post-doctorate degrees. With 30 years of executive leadership, consulting, and engineering experience in roles ranging from the courtroom to the boardroom, as a former CEO, VP, Chief Counsel, Director, or entrepreneur in startup, private, and publicly traded companies, Nic brings a unique perspective to technology strategy and implementation.