Kubernetes 설계 개념

Source: 2018 Saad Ali, Google
Kubernetes Design Principles:
Understand the Why

개요

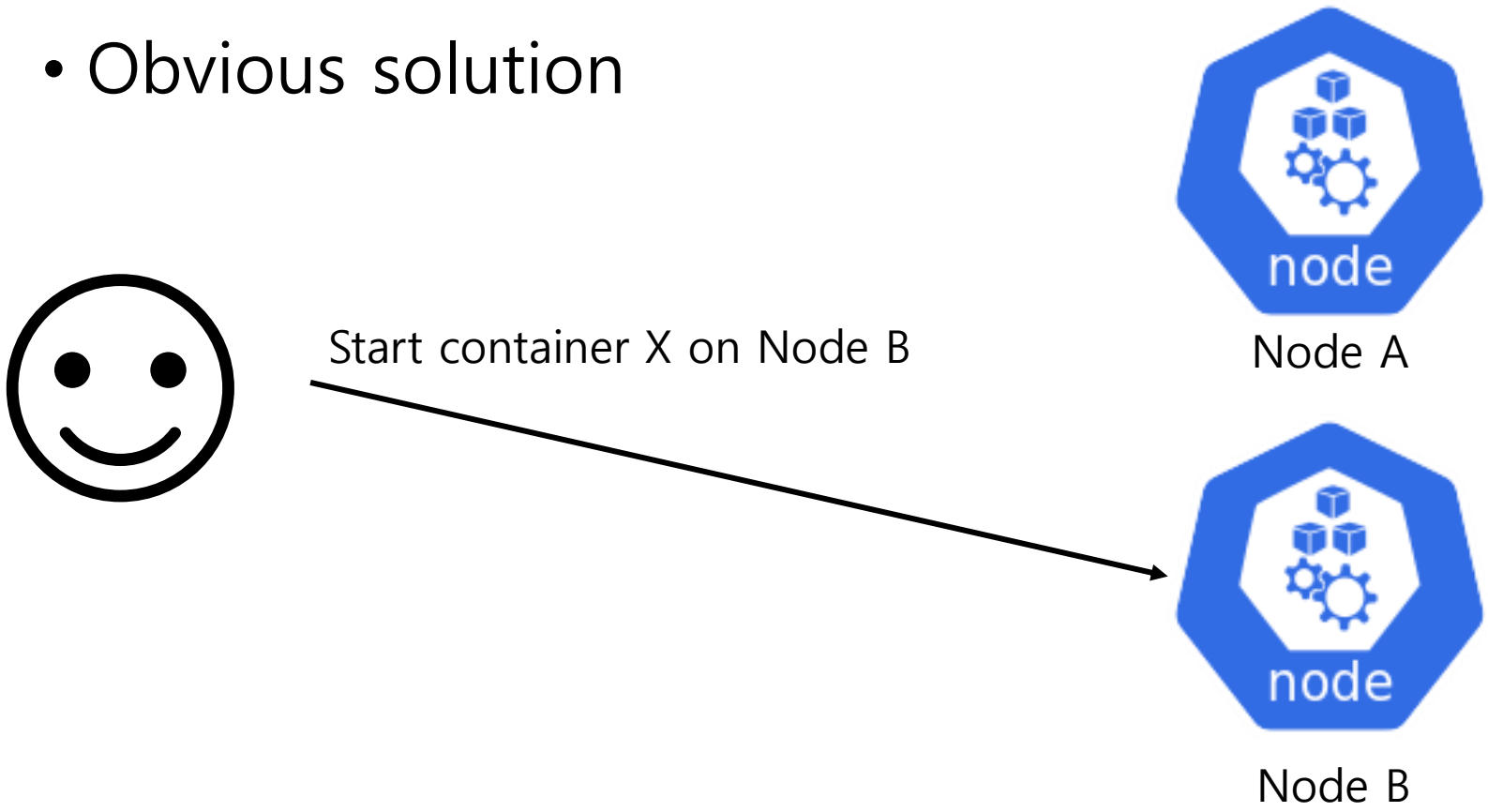
- Goal: A deeper understanding of Kubernetes
- Important tool for learning
  - Understand the problem
  - The **“why”** not just the “what”

Kubernetes

- Containerization was the key
  - Consistent, repeatable, reliable deployments on a **wide variety of systems**.
- Who will manage it?
  - You? Scripts? A system you write?
- **Kubernetes manages your cluster!**
  - **Deploys & monitors containerized workloads.**

How to deploy a workload?

- Obvious solution



How to deploy a workload?

- Obvious solution
  - Problems with this approach?

Start containers X on Node B

Node A

Node B

- What if:
  - Container crashes and dies?
  - What if node crashes and dies?
  - What if node B is momentary not healthy?

- User has to

- Monitor and store state of every container/node
- -"Catch up" any failed nodes that missed calls

- Complex, custom logic

Principle #1

Kubernetes APIs are declarative = 원하는 것을 서술하는 식

rather the imperative = 세부적인 명령 대신에

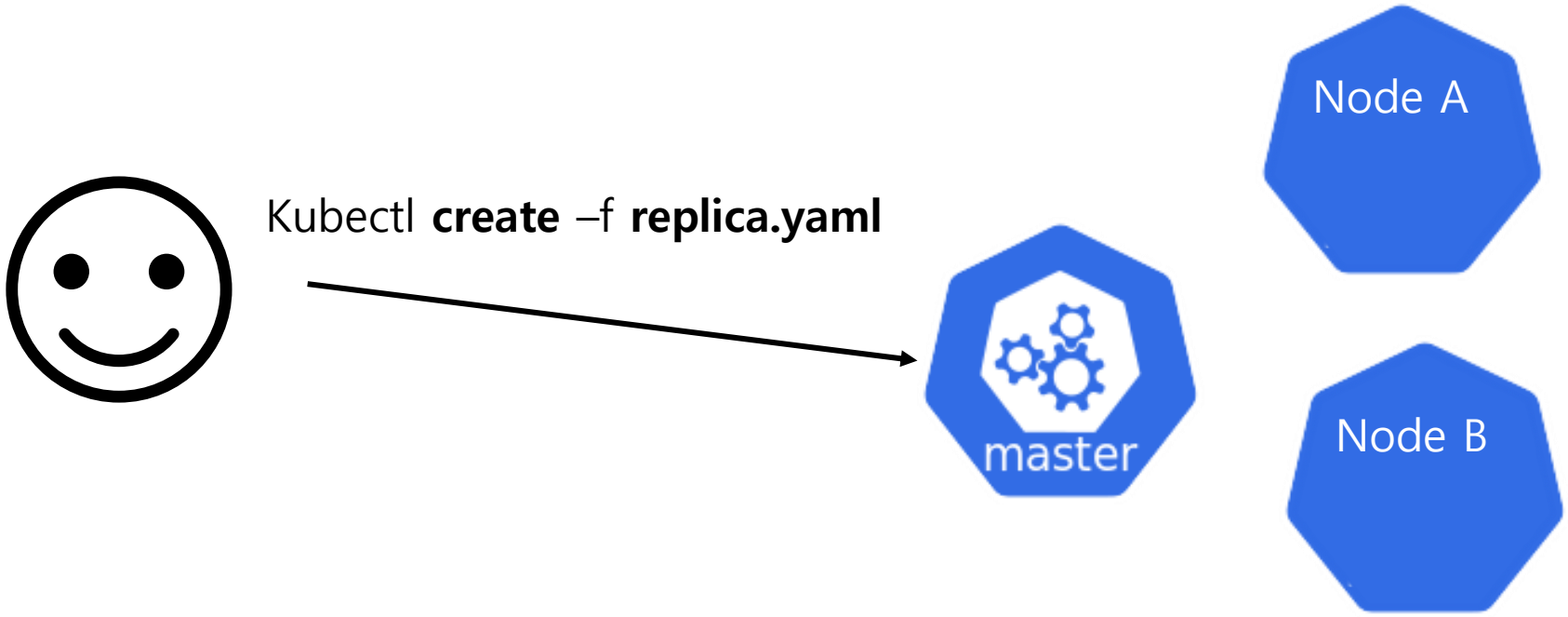
Declarative APIs

- Before:
  - **You:** provide **exact set of instructions** to drive to desired state
  - **System:** executes instructions
  - **You: monitor system**, and provide **further instructions** if it deviates.

- After:
  - **You:** define desired state
  - **System:** works to drive towards that state

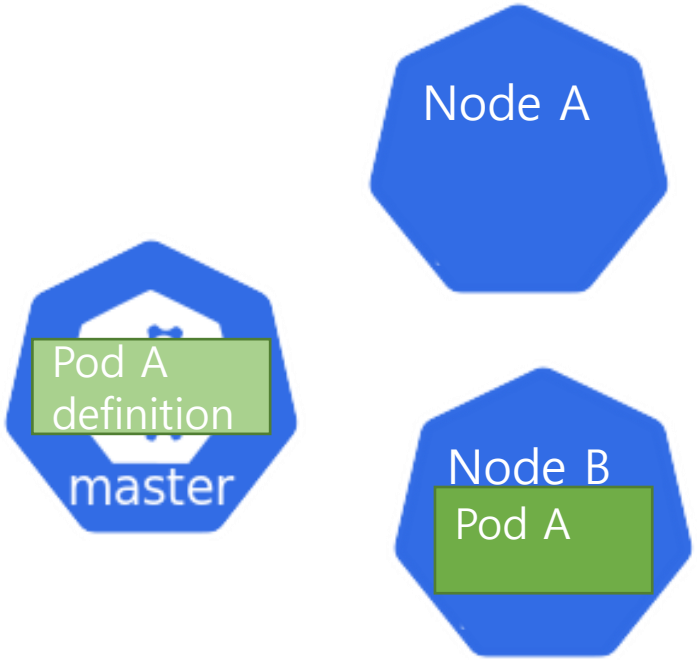
How to deploy a workload?

- The Kubernetes way!
  - **You:** create API object that is persisted on kube API server until deletion
  - **System:** all components work in parallel to drive to that state



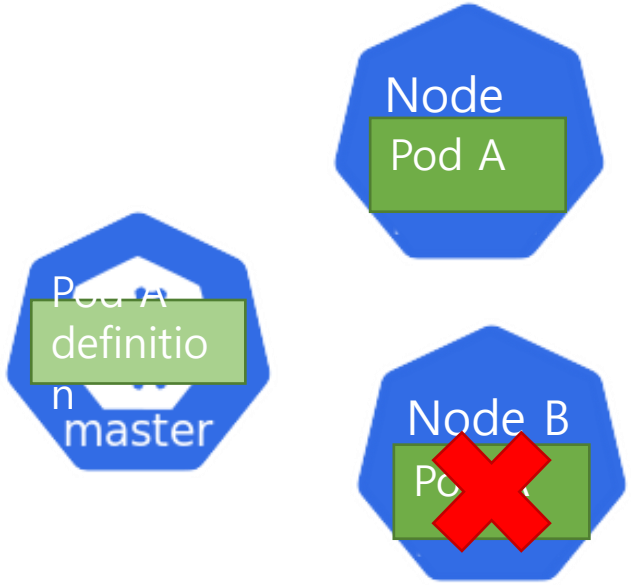
How to deploy a workload?

- The Kubernetes way!
  - **You:** create API object that is persisted on kube API server until deletion
  - **System:** all components work in parallel to drive to that state



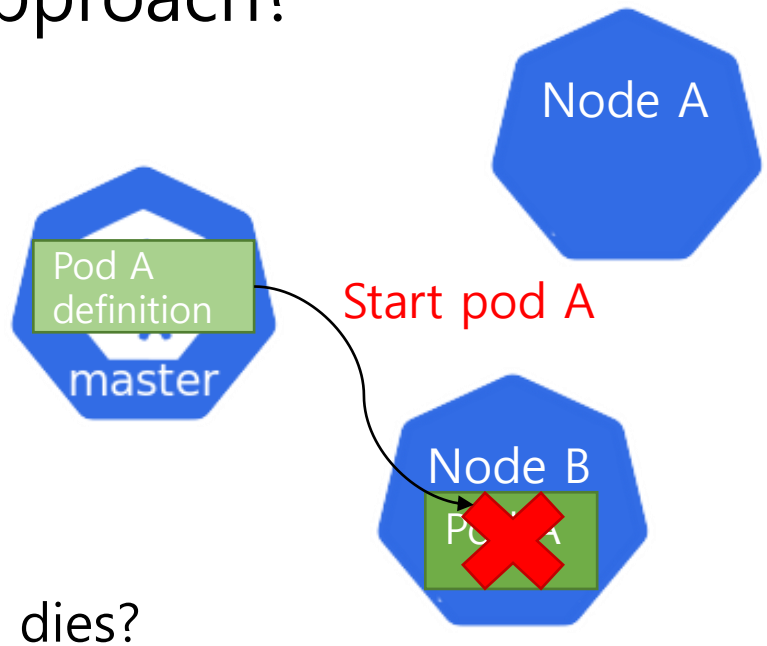
Why declarative over imperative?

- Automatic recovery!
- Example:
  - Step 1: Node failure
  - Step 2: System automatically moves pod to healthy node



How to deploy a workload?

- How does node figure out what to do?
- Problems with this approach?



- What if:
  - Container crashes and dies?
  - What if node crashes and dies?
  - What if node B is momentary not healthy?

- Master has to

- Monitor and store state of every component.
- "Catch up" any failed components that missed calls

Master becomes:

- Complex
- Fragile
- Difficult to extend

Principle #2

Principle #1 Kubernetes APIs are declarative rather the imperative

Principle #2 The Kubernetes control plane is transparent.
There are no hidden internal APIs.

즉, API server 뒷단에서 다시 Master가 대신 명령하지 않는다

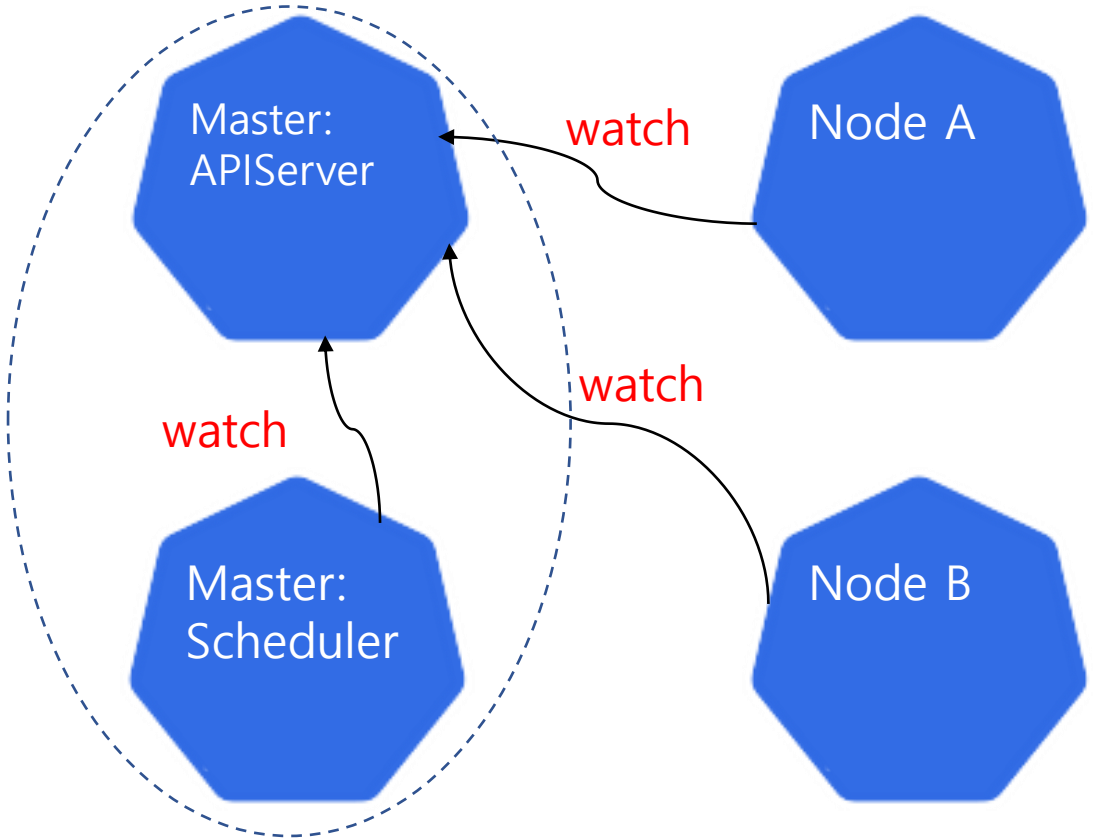
No hidden internal APIs

- Before:
  - **Master:** provide exact set of instructions to drive to node to desired state
  - **Node:** executes instructions
  - **Master:** monitor nodes, and provide further instructions if state deviates.

- After:
  - **Master:** define desired state of node
  - **Node:** works independently to drive itself towards that state

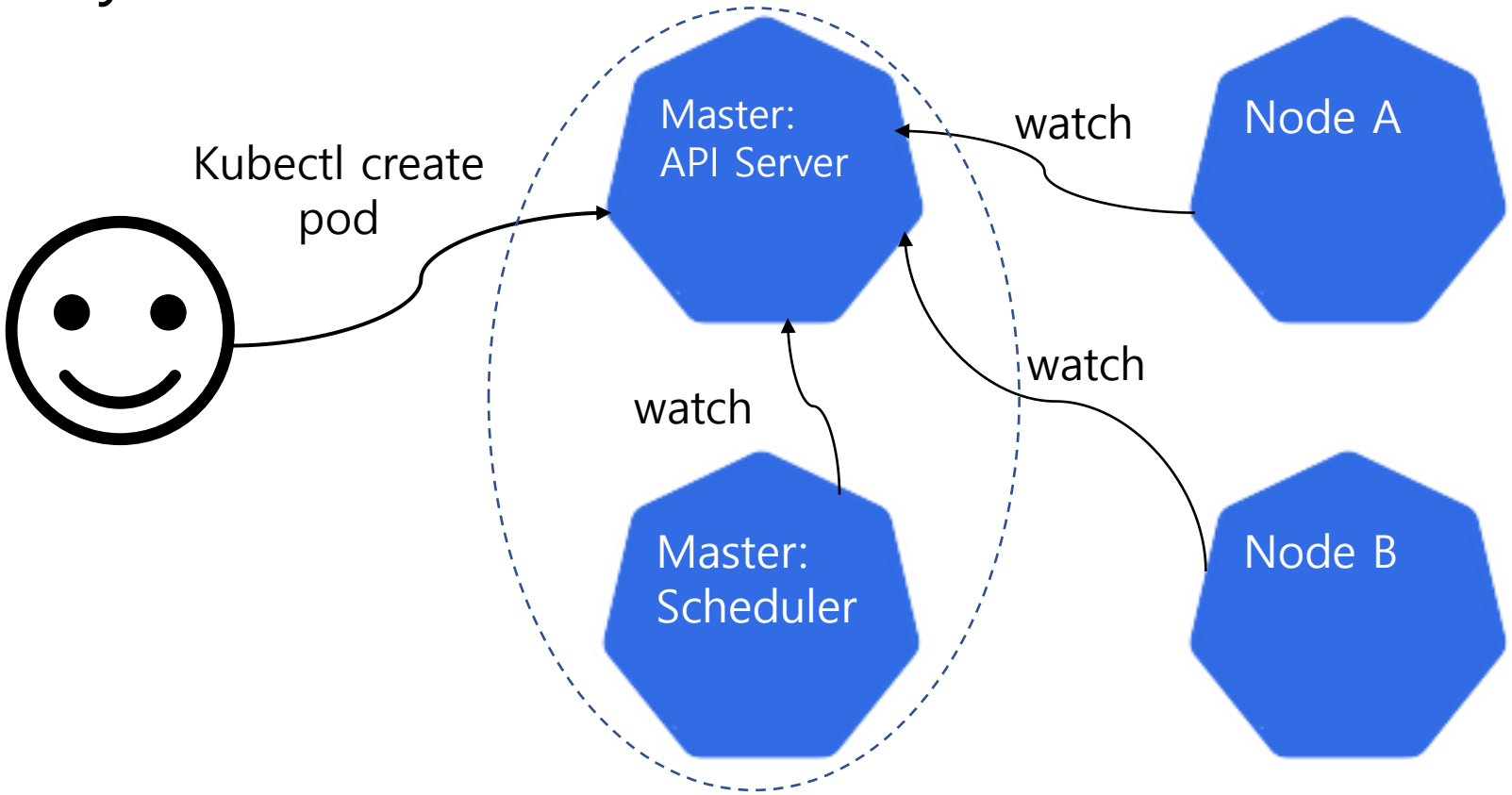
No hidden internal APIs

- **All components watch the Kubernetes API, and figure out what they need to do.**



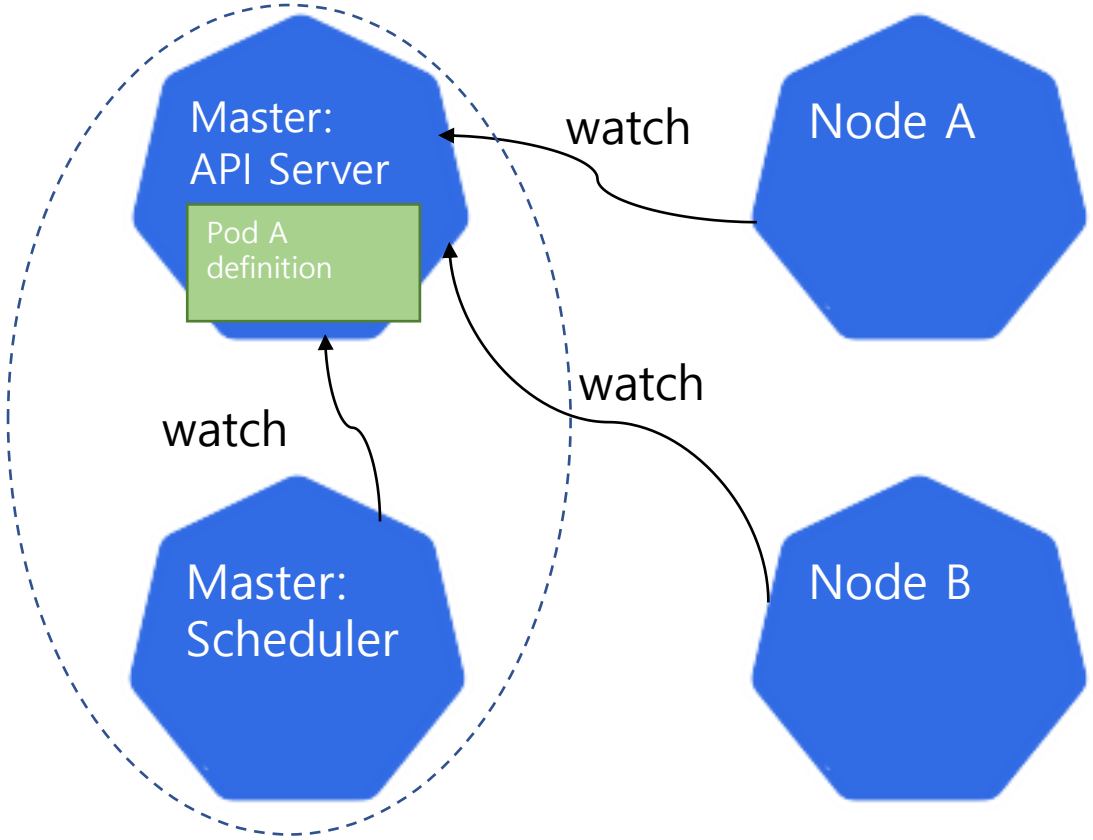
No hidden internal APIs

- All components watch the Kubernetes API, and figure out what they need to do.



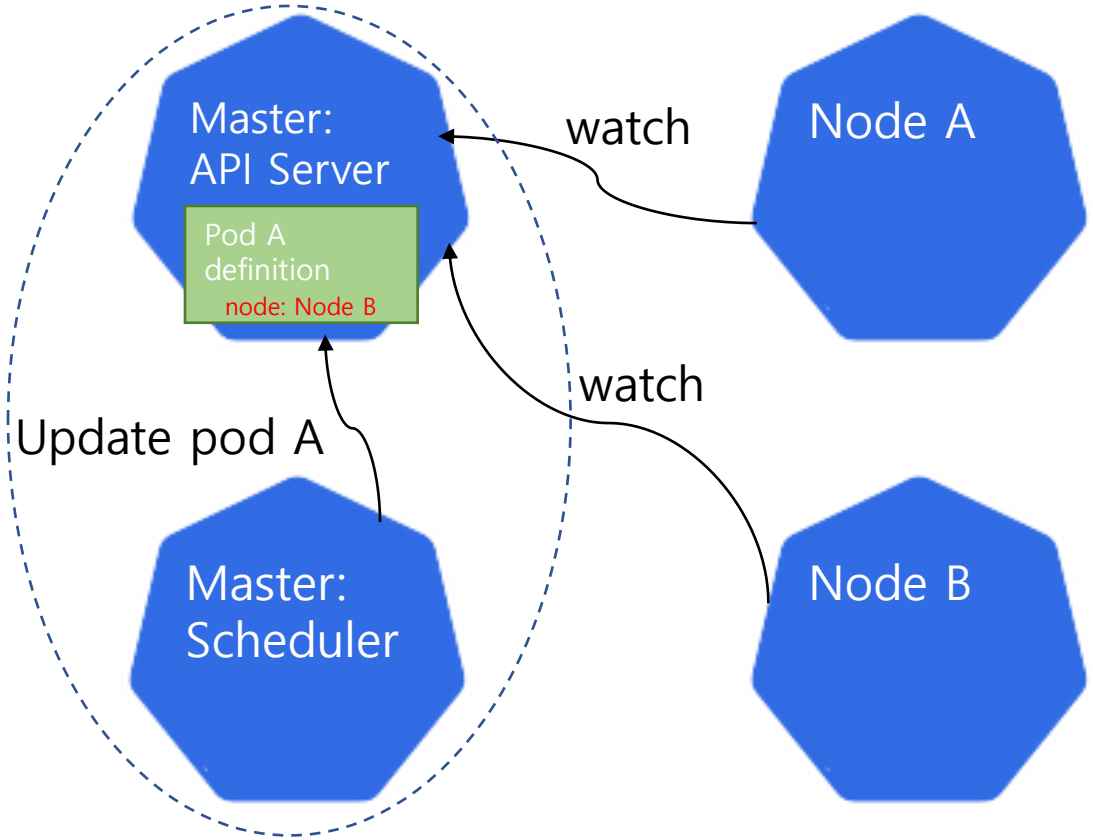
No hidden internal APIs

- All components watch the Kubernetes API, and figure out what they need to do.



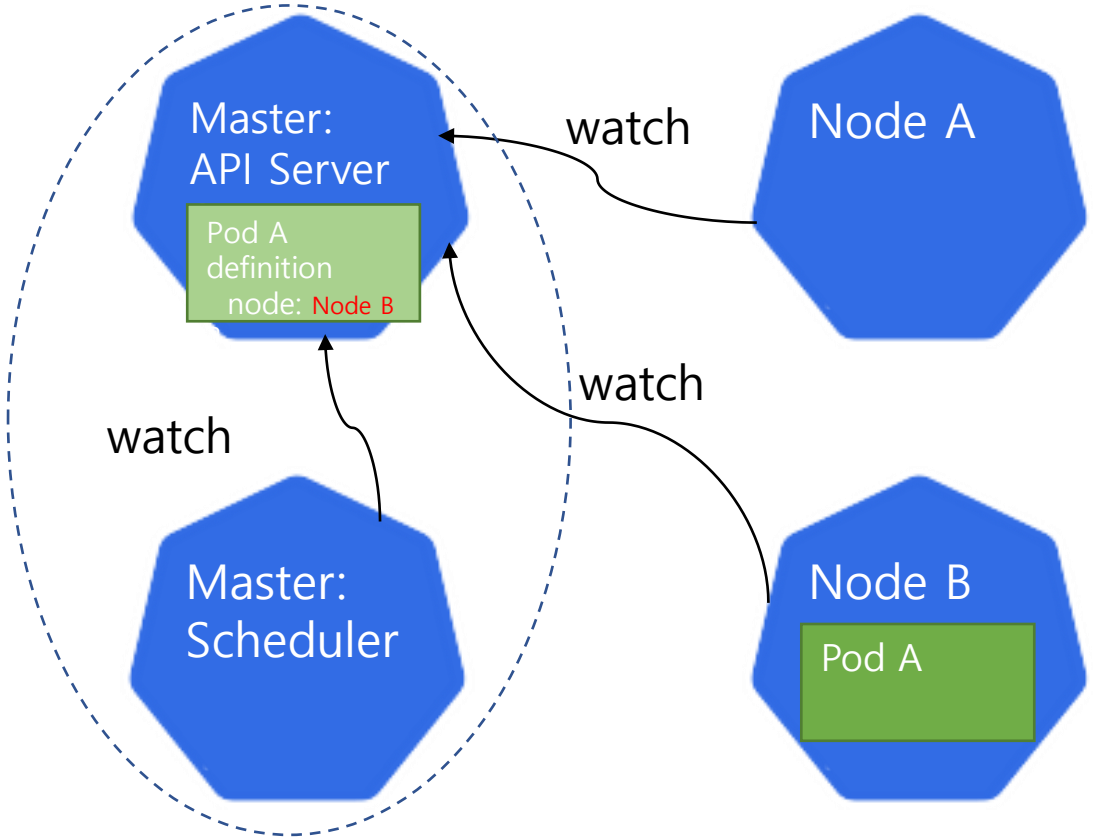
No hidden internal APIs

- All components watch the Kubernetes API, and figure out what they need to do.



No hidden internal APIs

- All components watch the Kubernetes API, and figure out what they need to do.



Why No hidden internal APIs?

- Declarative API provides the same benefits to internal components
- Resulting in a Simpler, more robust system that can easily recover from failure of components.
  - No single point of failure.
  - Simple master components.

- Also **makes Kubernetes composable and extensible.**
  - Default component not working for you?
    - Turn it off and replace it with your own.
  - Additional functionality not yet available?
    - Write your own and to add it.

Kube API Data

- Kubernetes API has lots of data that is interesting to workloads
  - Secrets – Sensitive info stored in KubeAPI
    - e.g. passwords, certificates, etc.
  - ConfigMap – Configuration info stored in KubeAPI
    - e.g. application startup parameters, etc.
  - DownwardAPI – Pod information in KubeAPI
    - e.g. name/namespace/uid of my current pod.

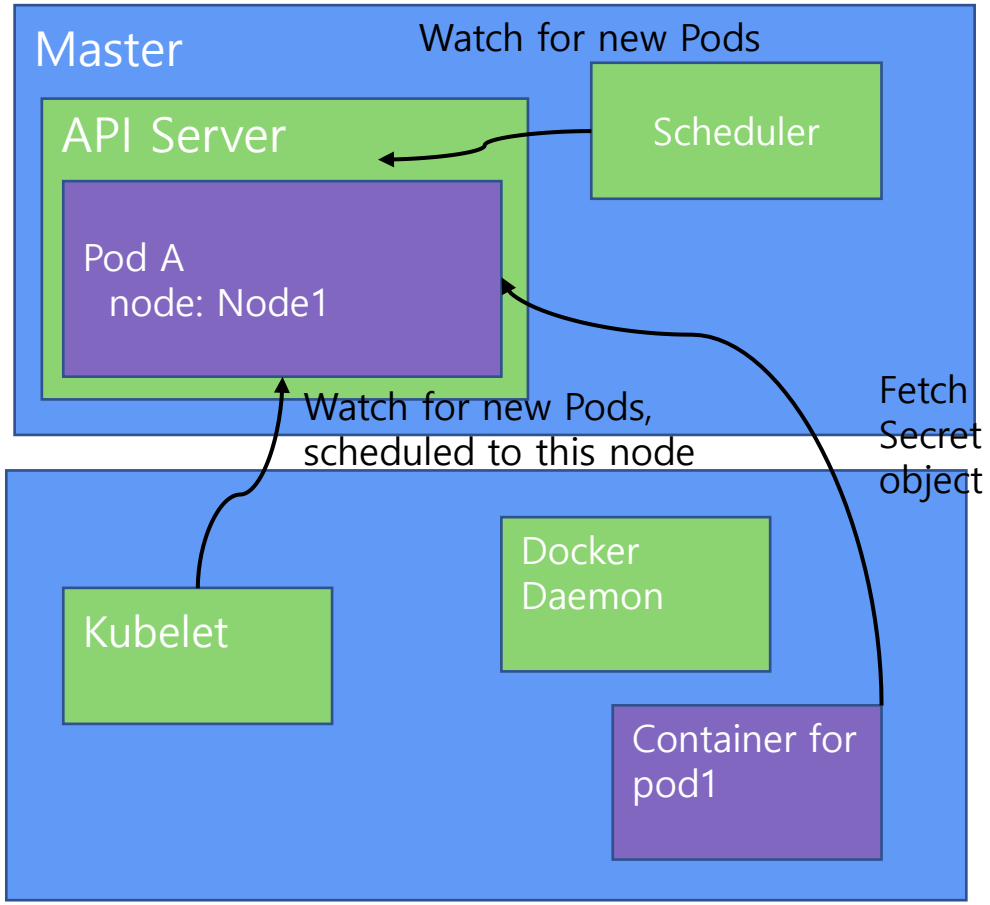
Fetching Kube API data

- How does application fetch secrets, config map, etc. information?
- **Principle:** No hidden internal APIs.

- **Obvious solution:** Modify app to read directly from API Server.

  -> Not good for legacy applications

  So, principle #3



Principle #3

Meet the user where they are.
(i.e. Supporting Legacy Applications)

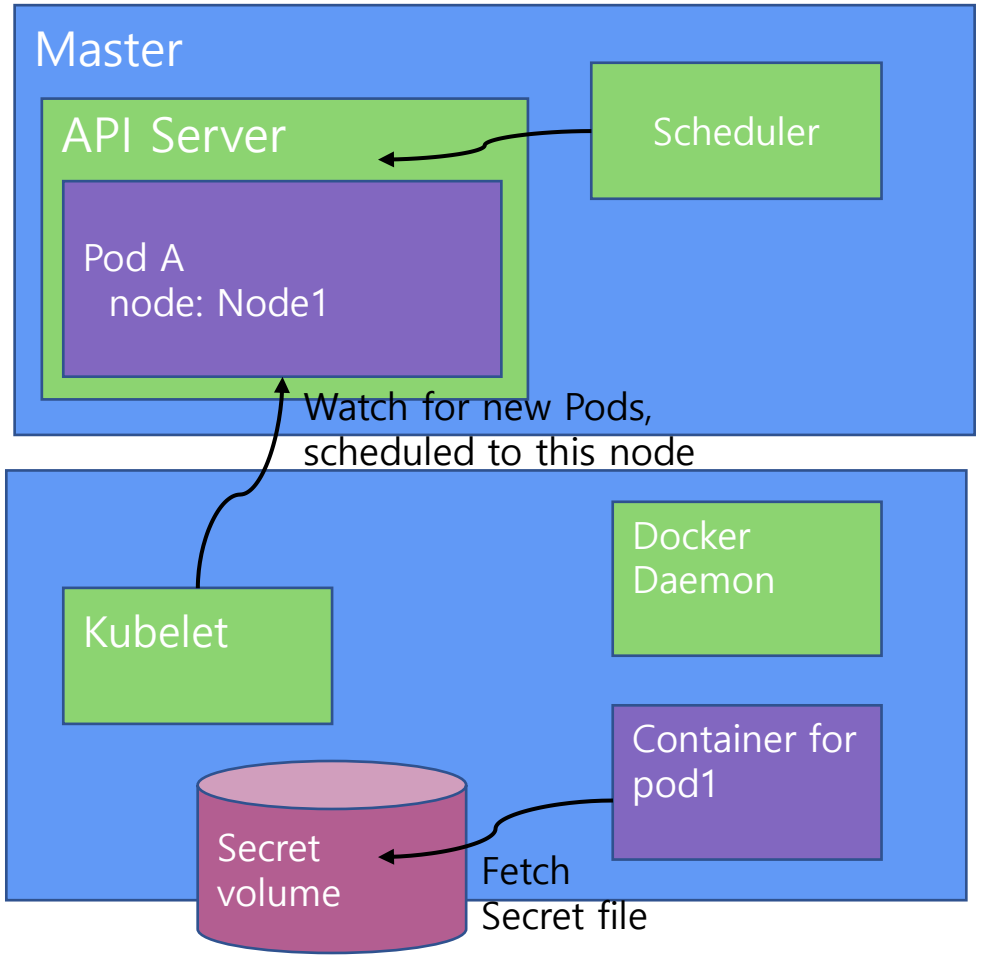
기존 응용들을 수정없이 돌릴 수 있게 하자

Why meet the user where they are?

- Minimize hurdles for deploying workloads on Kubernetes.
- Increases adoption.

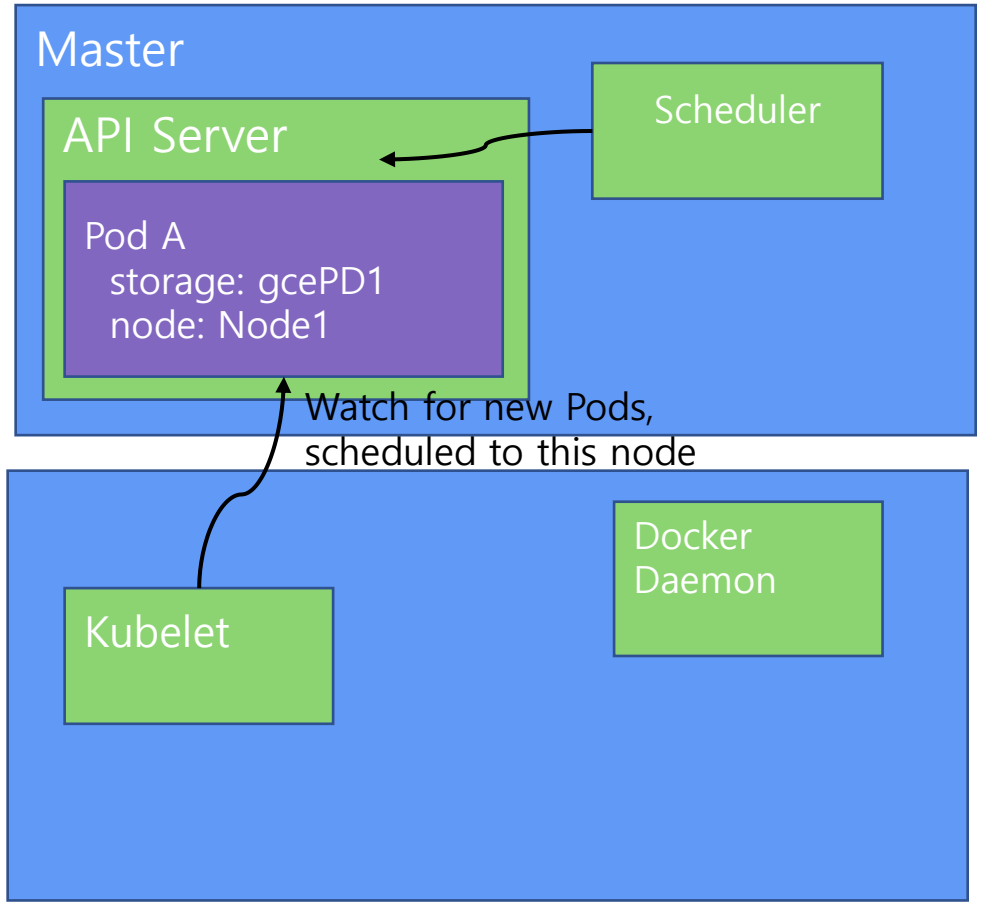
Meet the user where they are.

- Before:
  - App must be modified to be Kubernetes aware.
- After:
  - If app can load config or secret data from file or environment variables it doesn't need to be modified!
- But, not good solution, because PoD's could be deleted after application's mission completion
  - So,…. Remote storage for stable data storage



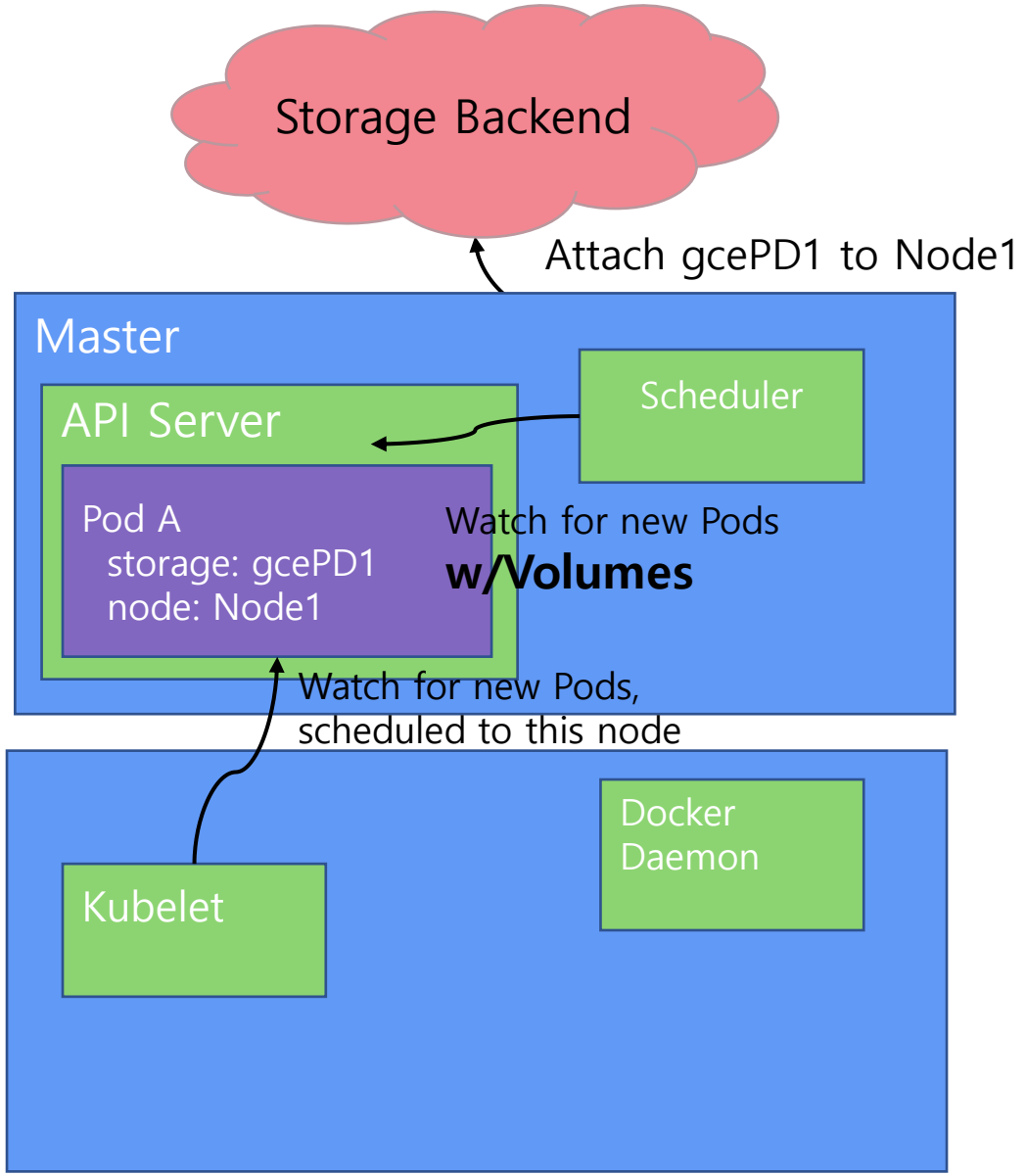
Remote Storage

- Could directly reference a remote volume (GCE PD, AWS EBS, NFS, etc.) in pod definition.
- Kubernetes will automatically make it available to workload



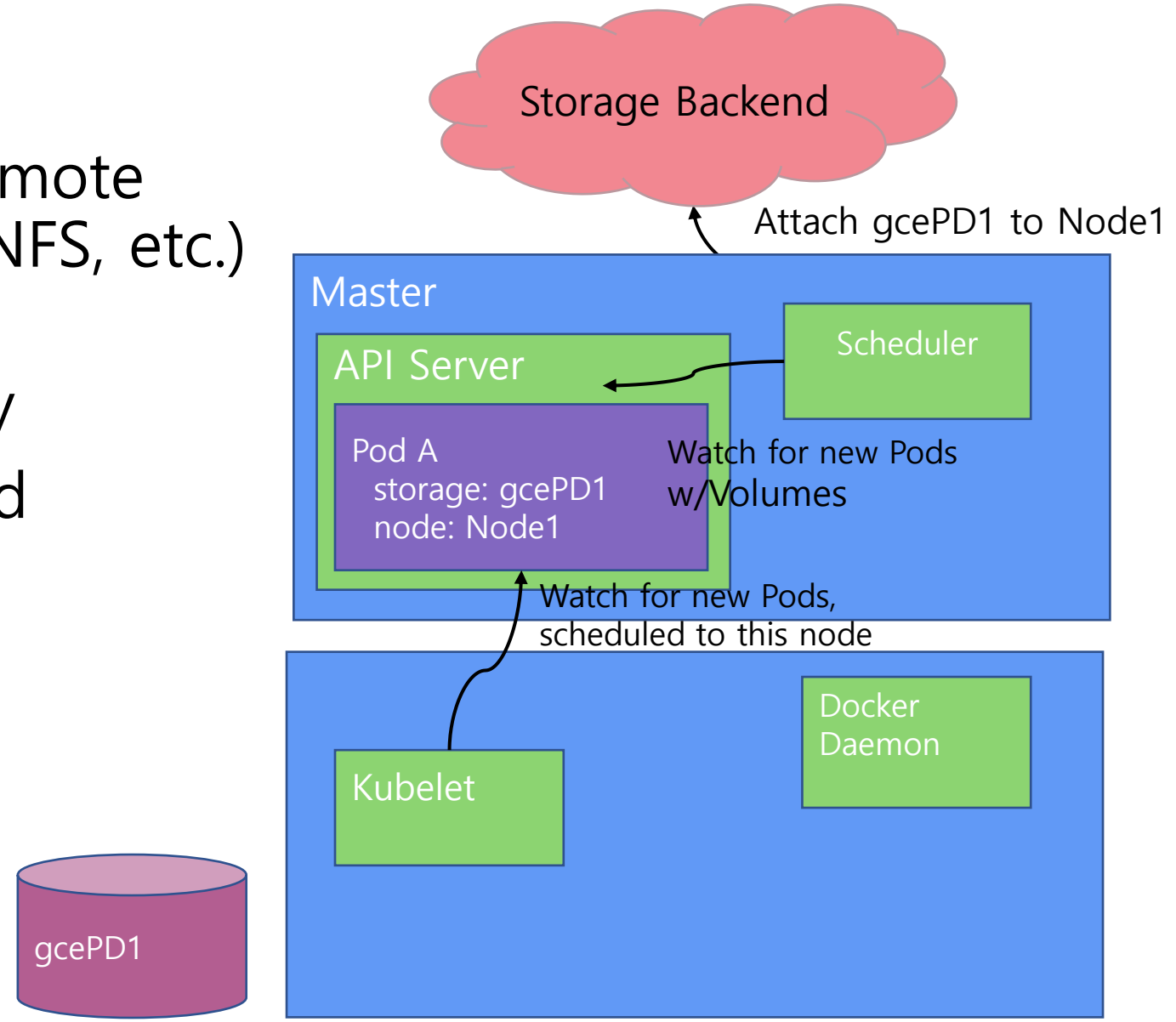
Remote Storage

- Could directly reference a remote volume (GCE PD, AWS EBS, NFS, etc.) in pod definition.
- Kubernetes will automatically make it available to workload



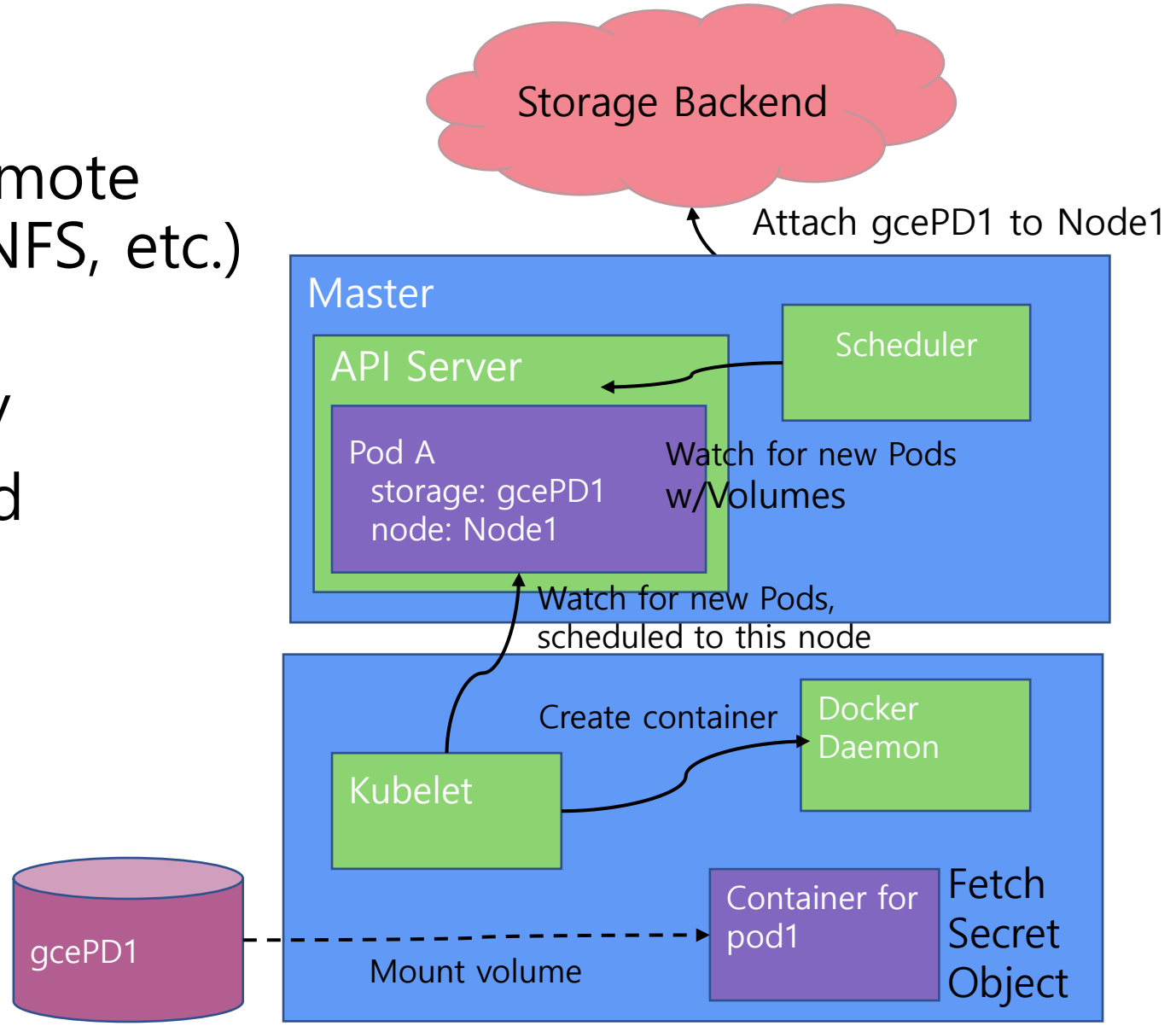
Remote Storage

- Could directly reference a remote volume (GCE PD, AWS EBS, NFS, etc.) in pod definition.
- Kubernetes will automatically make it available to workload



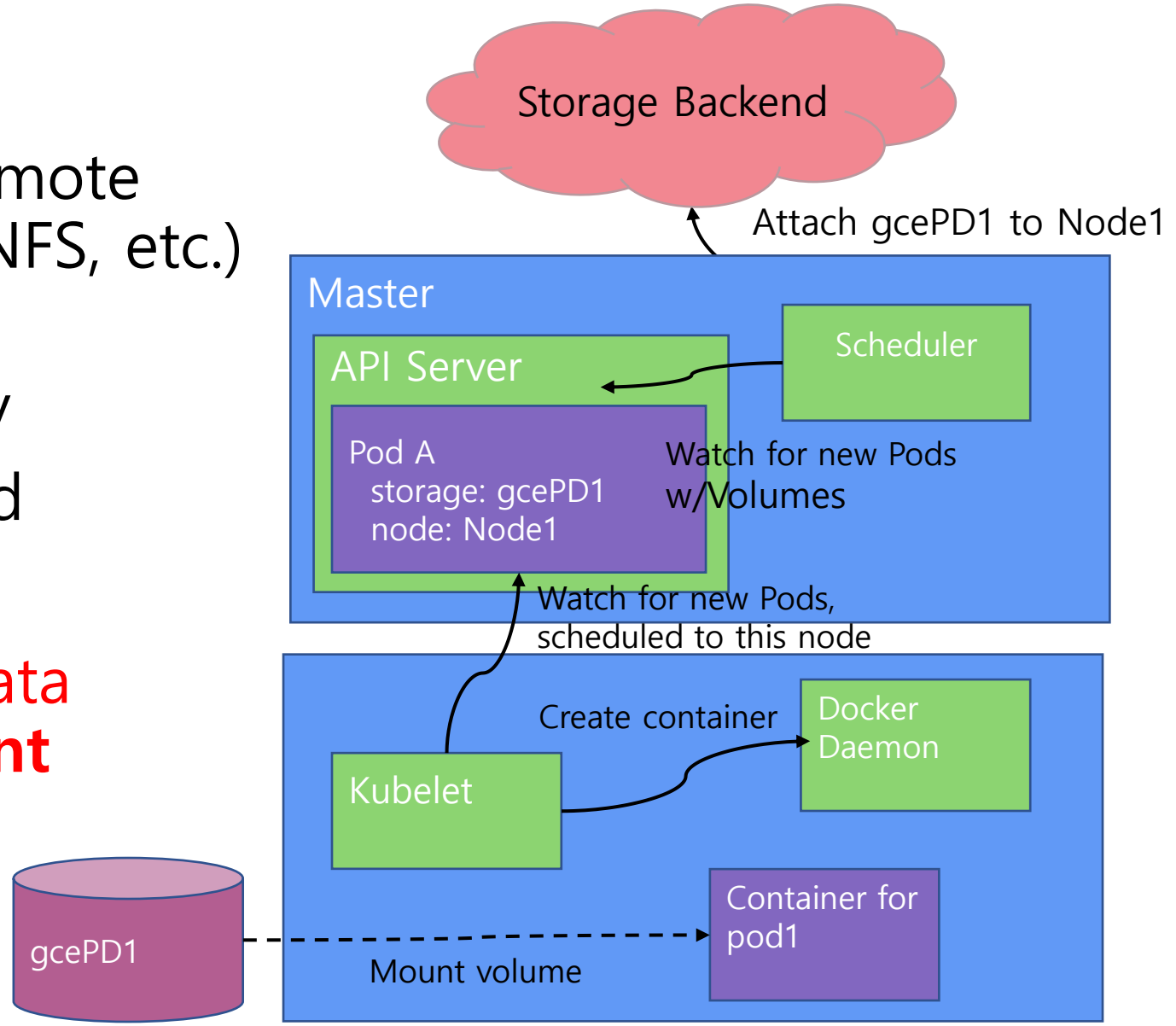
Remote Storage

- Could directly reference a remote volume (GCE PD, AWS EBS, NFS, etc.) in pod definition.
- Kubernetes will automatically make it available to workload



Remote Storage

- Could directly reference a remote volume (GCE PD, AWS EBS, NFS, etc.) in pod definition.
- Kubernetes will automatically make it available to workload
- Even If PoD's deleted ..the data stored in Volume is **persistent**

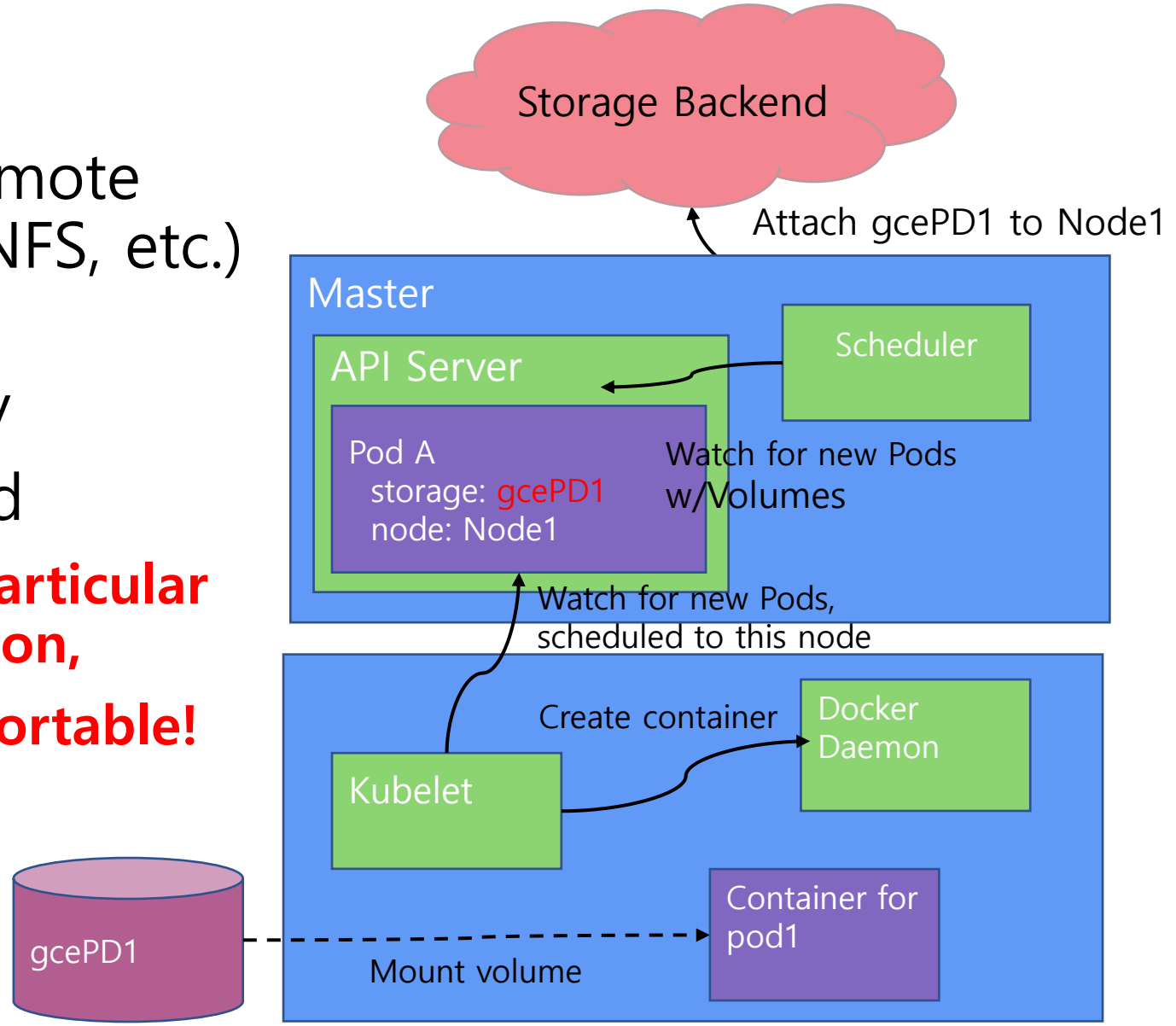


Remote Storage

- Could directly reference a remote volume (GCE PD, AWS EBS, NFS, etc.) in pod definition.
- Kubernetes will automatically make it available to workload

But…If you directly reference a particular type of disk inline of PoD definition,

the PoD definition is no longer portable!



Principle #4

Workload portability

PoD의 이식성을 강화(스토리지 측면에서도)

PVC/PV

- PersistentVolume and PersistentVolumeClaim Abstraction
- **Decouple** storage **implementation from** storage **consumption**

Why Workload Portability?

- Decouple distributed system application development form cluster implementation
- Make Kubernetes a true abstraction layer, like an OS.

Kubernetes Principles introduced

- 1. Kube API declarative over imperative.
- 2. No hidden internal APIs
- 3. Meet the user where they are: Remote storage
- 4. Workload portability: PV/PVC