

高效编排有状态负载—— TiDB 的云原生实践与思考

吴叶磊

Cloud Engineer @ PingCAP

自我介绍

吴叶磊。PingCAP Cloud 工程师，热爱云原生与开源技术，开发并维护 kubectl-debug, aliyun-exporter 等开源项目，同时也是专注于云原生技术的博客作者，现负责 TiDB Operator 研发。

mailto: wuyelei@pingcap.com

Blog: www.aleiwu.com

Github: [aylei](https://github.com/aylei)



为什么？

无状态应用 + Kubernetes 天生就能够在云环境中充分受益。
那么，我们能不能把这种成功复制到有状态应用上呢？

CRD 愈发成熟，Operator 遍地开花，Local PV 性能无忧。这个问题似乎有点简单啊🤔？

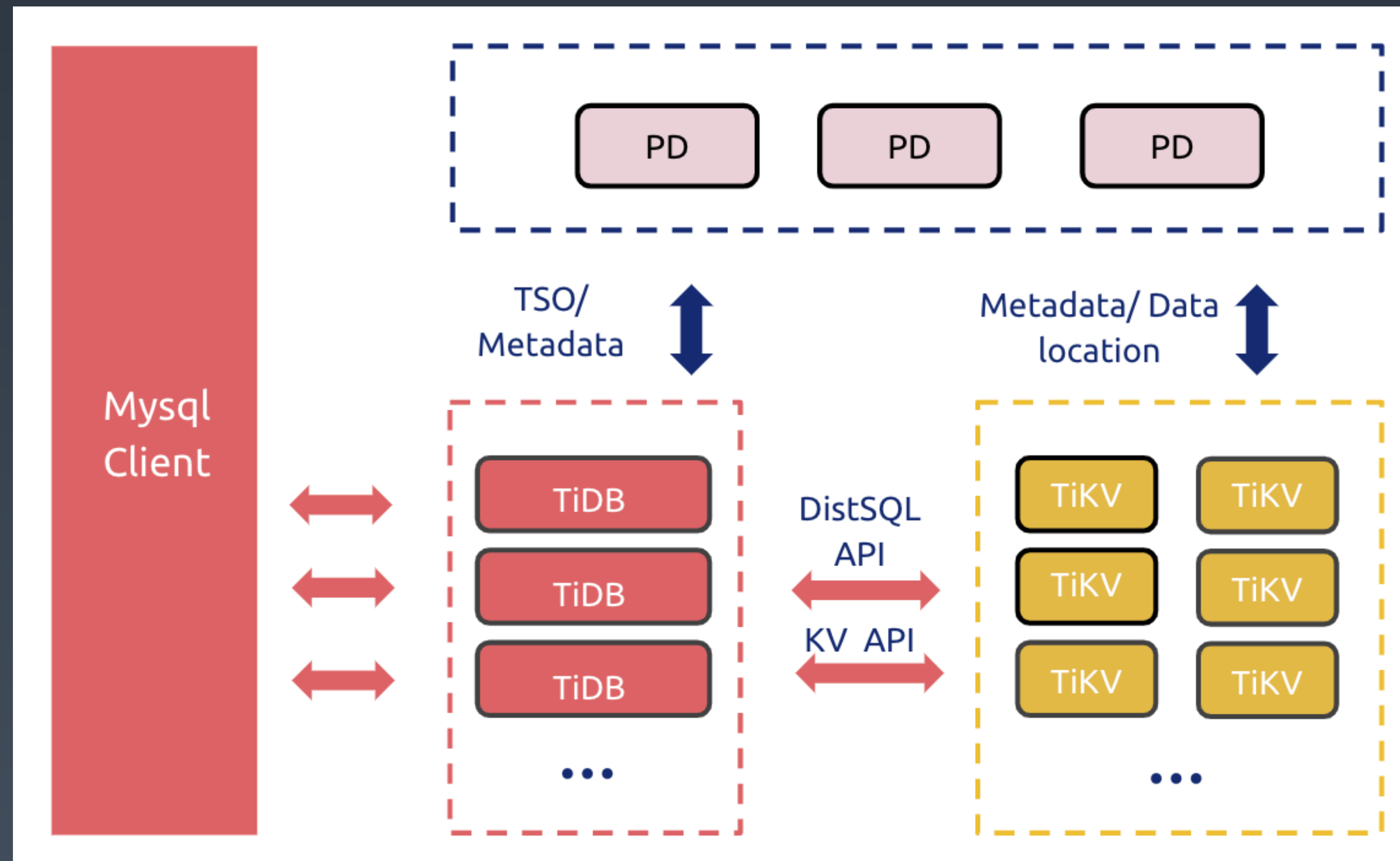
真的是这样吗？就让我们走近这场 talk 的主角 TiDB，看看它和云原生的“爱恨情仇”😂

目录

目标：以 TiDB 为例，探讨有状态应用如何落地云原生

- TiDB 简介
- TiDB Operator 的目标与挑战
- 控制器的复杂度权衡
- 正确使用 Local PV
- 总结

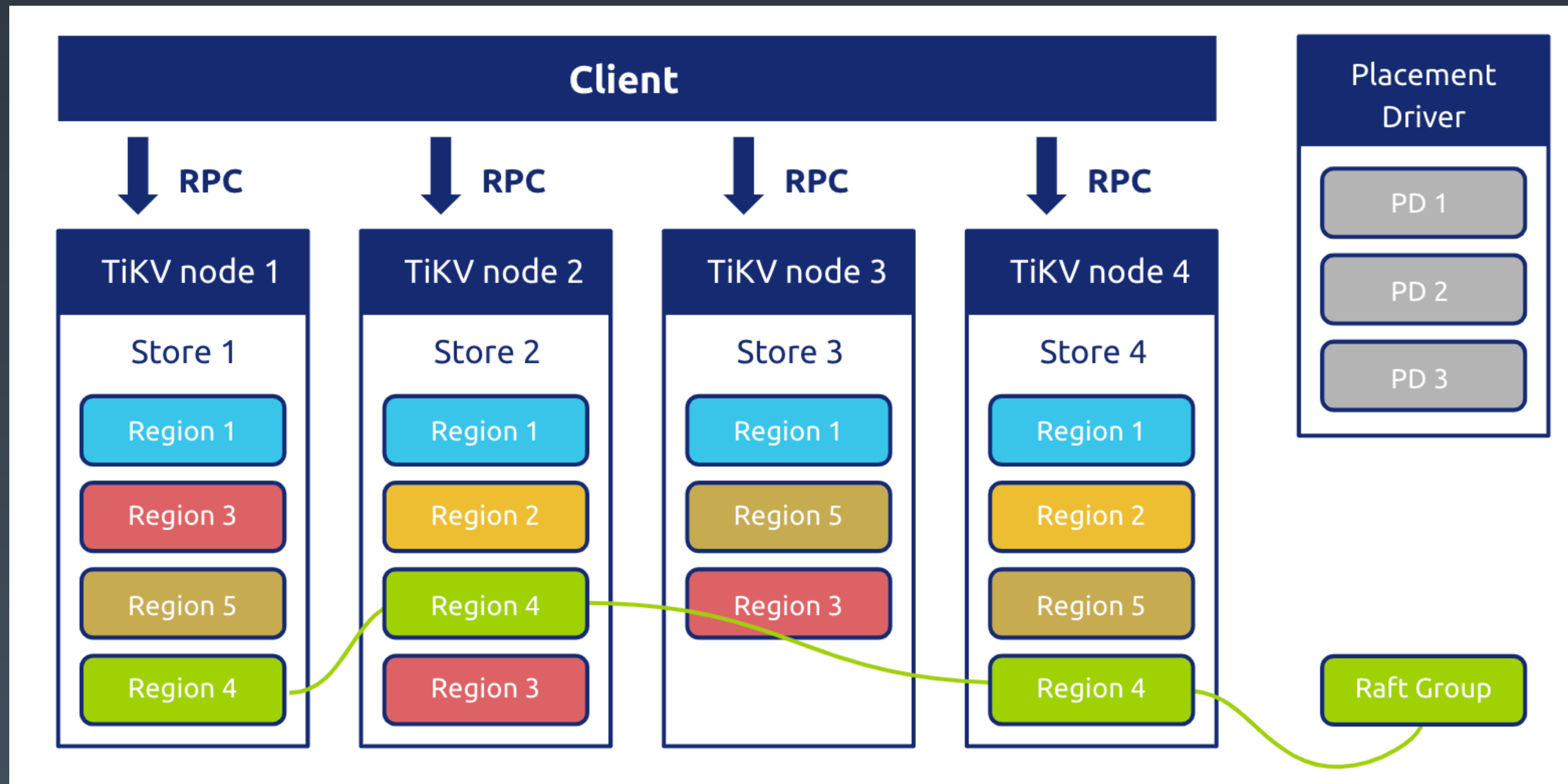
TiDB



TiDB, TiKV, PD 均单独组成集群水平伸缩。

- TiDB
 - 无状态
 - CPU 密集
- TiKV
 - CPU 密集
 - IO 密集
 - 有状态
 - 水平伸缩
- PD
 - 轻量
 - 有状态

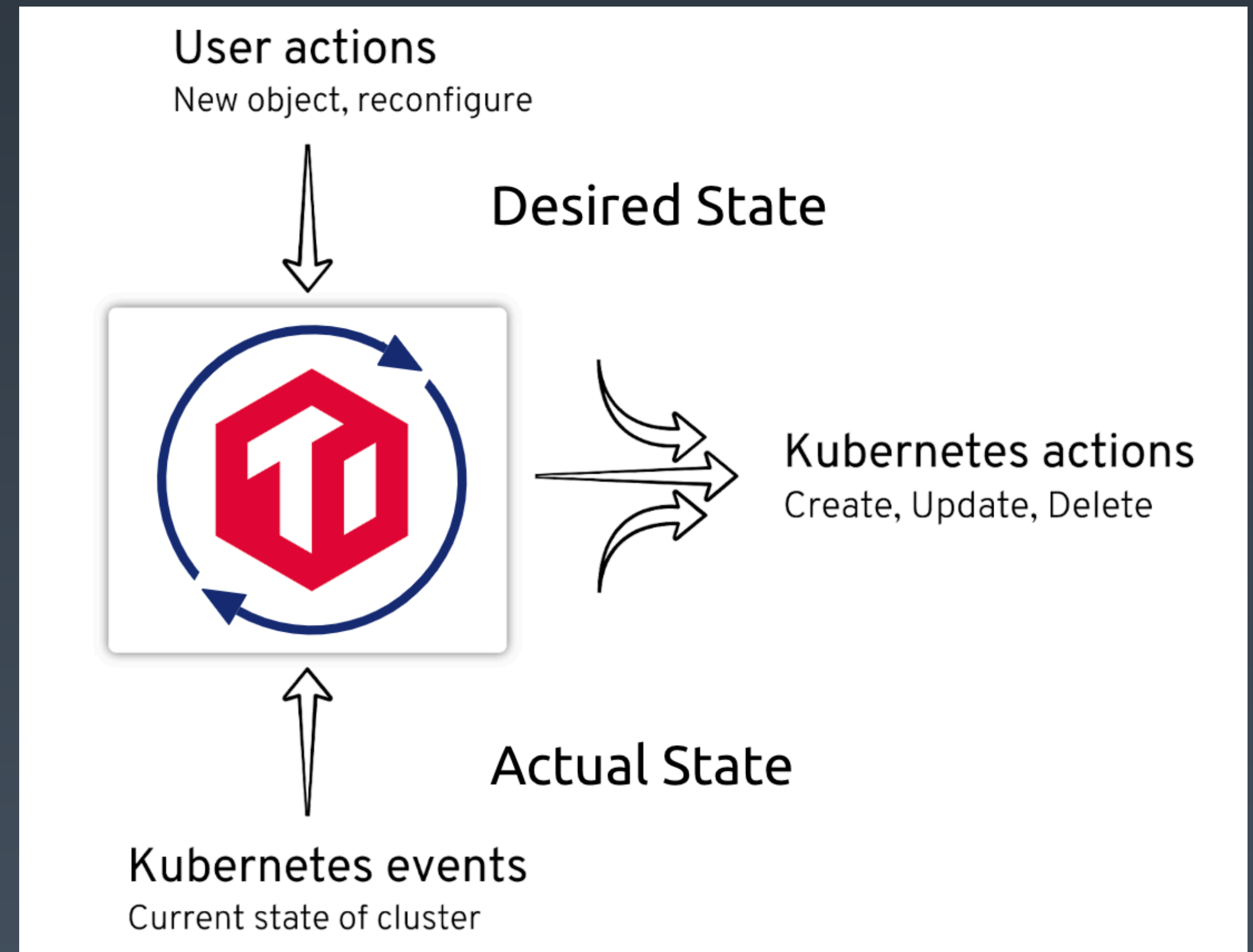
TiDB



Regions: 水平伸缩, 热点调度, 无需手工分片 🎉

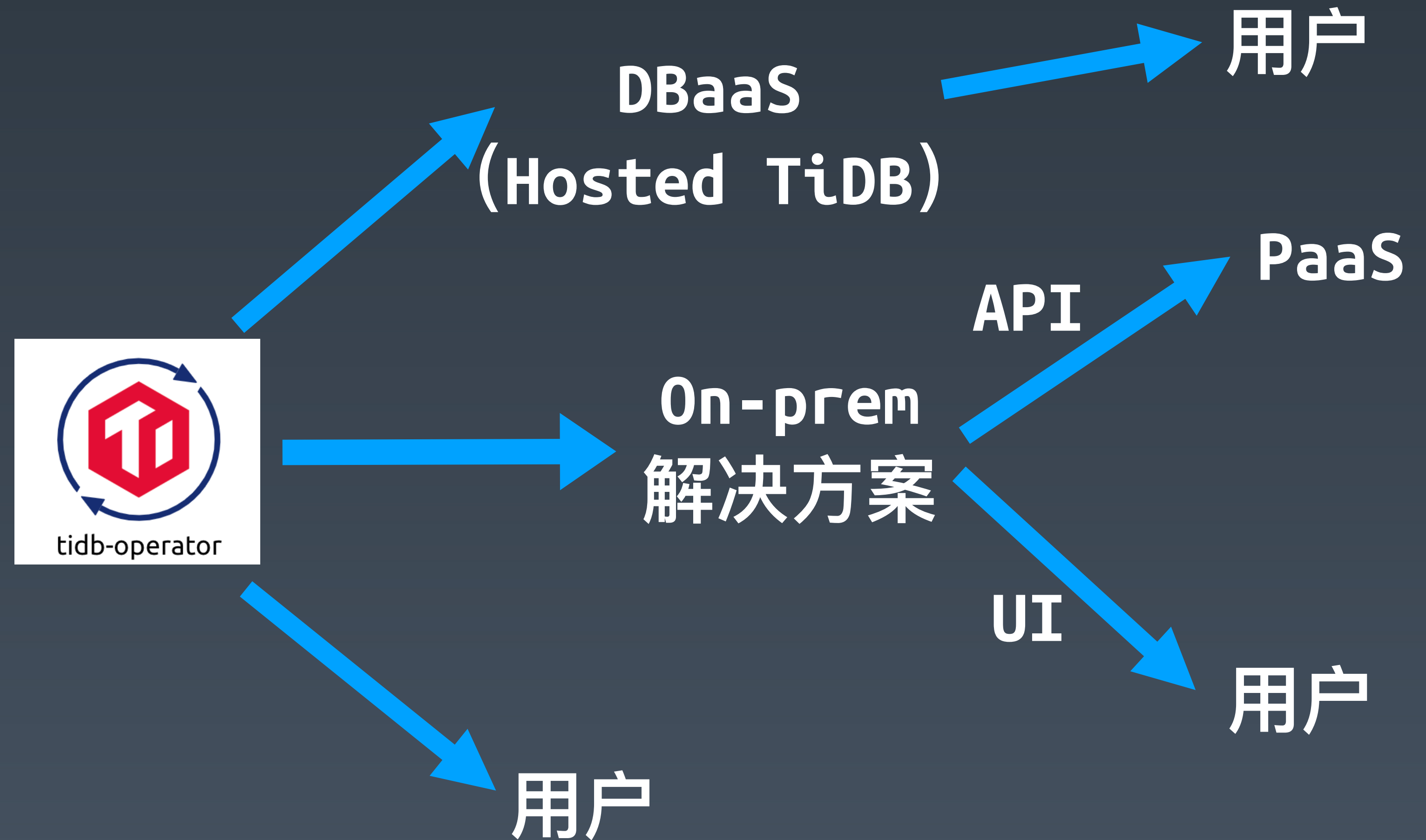
TiDB Operator

- Operator = CRD + Controller
- 控制器(Controller)不断对比 CRD 中记录的期望状态与实际状态, 并驱动实际状态向期望状态转移;
- CRD 解放用户: 声明式管理;
- Controller 解放运维: 将领域知识编写成代码进行自动化;



TiDB Operator - 我们想要什么?

- 一个内核，多种产品形态；
- 理论上，K8S + Operator + LocalPV 足以支撑我们的内核，然而现实并不像理论上那样一帆风顺。



挑战

- 场景严苛
 - “永不停机”是常见的诉求
 - 数据完整性是底线
- 编排分布式系统的复杂性
- 网络存储速度慢、单价高
- 本地存储易失：磁盘故障、节点故障
- 公有云、私有云、裸金属有不同的成本优化考量

控制器 — 剪不断，理还乱🧨

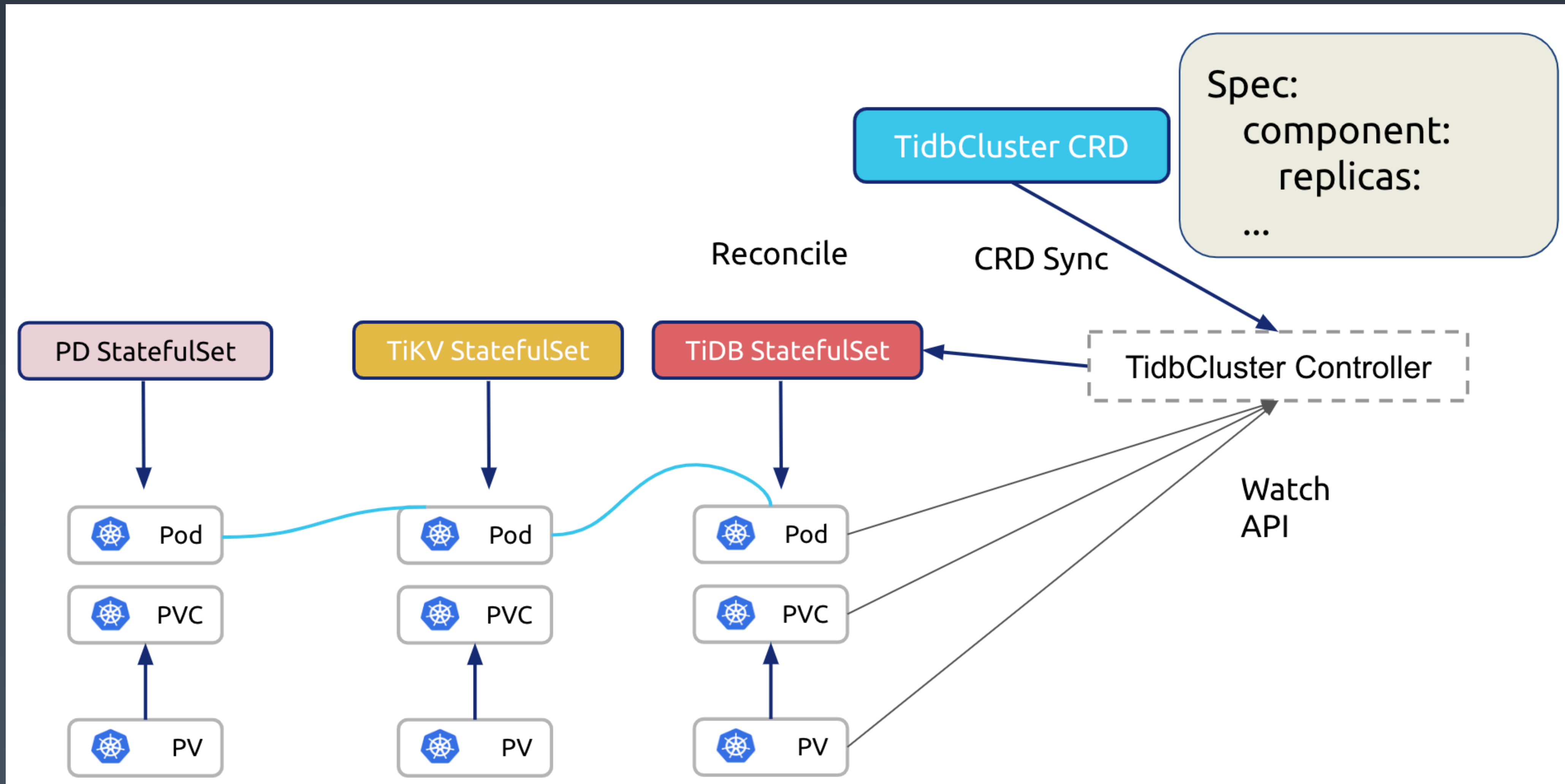
Pod vs StatefulSet

- Pod
 - 优点：灵活，可以自由控制所有编排细节：
 - 替换镜像原地升级
 - 按需决定升级顺序
 - 自由下线中间实例
 - 缺点：复杂，验证成本高，重新造一遍轮子。
- StatefulSet
 - 优点：能满足有状态应用的大部分通用编排需求；
 - 缺点：存在不必要的限制：
 - 无法指定 Pod 进行扩容
 - 滚动更新顺序固定
 - 滚动更新需要后驱 Pod 全部就绪

Pod vs StatefulSet

- 在 TiDB Operator 中，我们两种方案都尝试过：
 - 直接操作 Pod 需要维护相当多的代码。更重要的是，控制器逻辑复杂、控制循环变多后，代码的正确性变得相当难验证。
 - 操作 StatefulSet 极大降低了复杂度，从而直接提升了控制器的可靠性。
 - 最终，我们选择用灵活性来交换可靠性。

Pod vs StatefulSet



运维操作

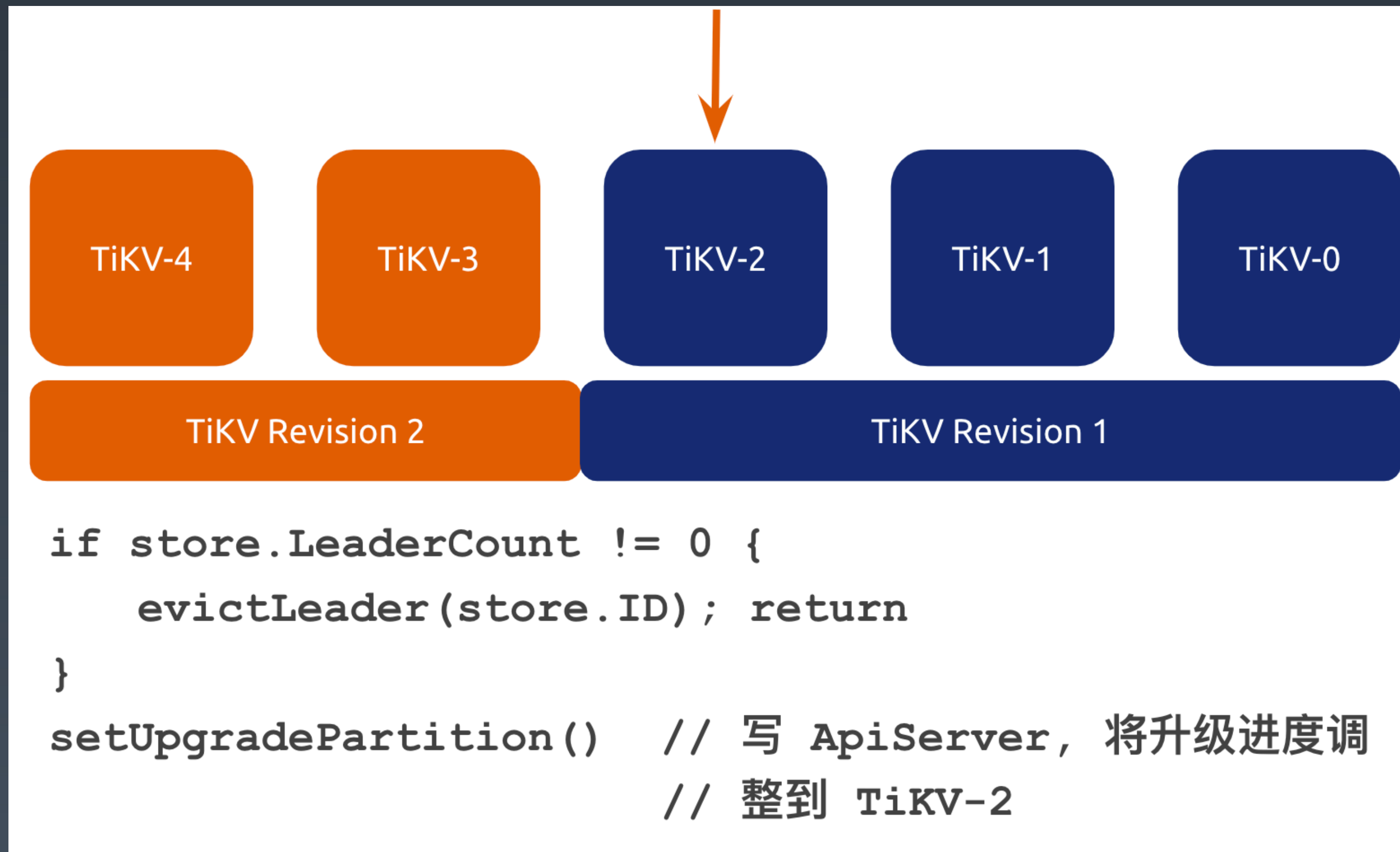
- 滚动升级
 - PD: transfer leader
 - TiKV: evict region leaders
 - TiDB: evict DDL owner
- 扩缩容
 - 扩容 TiKV: 执行下线流程, 将所有 region 搬迁到其它节点
- 故障转移
 - 新增实例补齐副本数
 - 旧实例恢复后自动释放新实例
- 本质上, 我们是要在控制 Pod 时同时协调应用层的状态, 那对于类似的场景, 该怎么做自动化呢?

方案一：Container Hook

```
spec:
  containrs:
  - name: my-awesome-container
    lifecycle:
      preStop:
        exec:
          command: ["/bin/sh", "-c", "/pre-stop.sh"]
```

- 问题：
 - 缺乏全局信息。比如，脚本中无法区分当前是缩容操作还是滚动更新；
 - 存在 Grace period，我们更希望下线失败时等待运维介入；

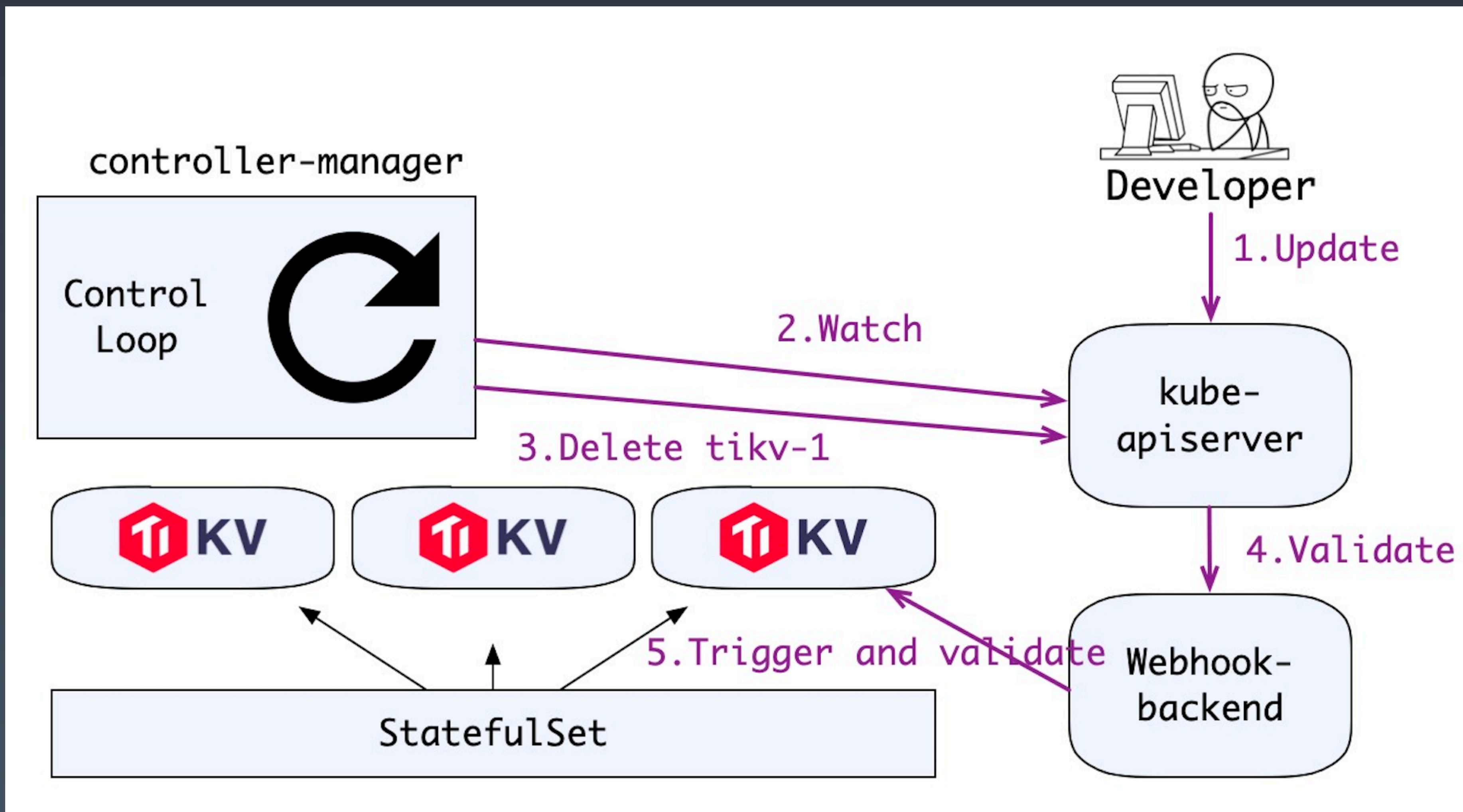
方案二：Controller



问题：

- 复杂，边界情况多。
e.g.
 - 扩容和滚动升级同时进行
- 使用了 StatefulSet 的 Partition 控制优雅升级，迫使我们还需要另外 partition 原有的金丝雀发布功能

方案三： Admission Webhook



方案三： Admission Webhook

- 控制器只协调 Pod 层面状态；
- 通过 Validating Webhook 在控制器的 API 调用中寻找切点；
- 只有应用层状态协调完毕后，才允许操作；
- 问题
 - 新增组件会增加 Operator 本身的运维负担和错误模式；
- 优点
 - 解耦：更容易编写和测试

Local PV — 想说爱你不容易 😊💧

各种存储类型

- 本地临时(ephemeral)存储
 - 生命周期: Pod
 - e.g. emptyDir, secret
- 远程持久存储
 - 生命周期: 与集群无关
 - e.g. rbd, cloud persistent disk
- 本地持久存储
 - 生命周期: 磁盘或节点
 - e.g. local, hostpath

各种存储类型

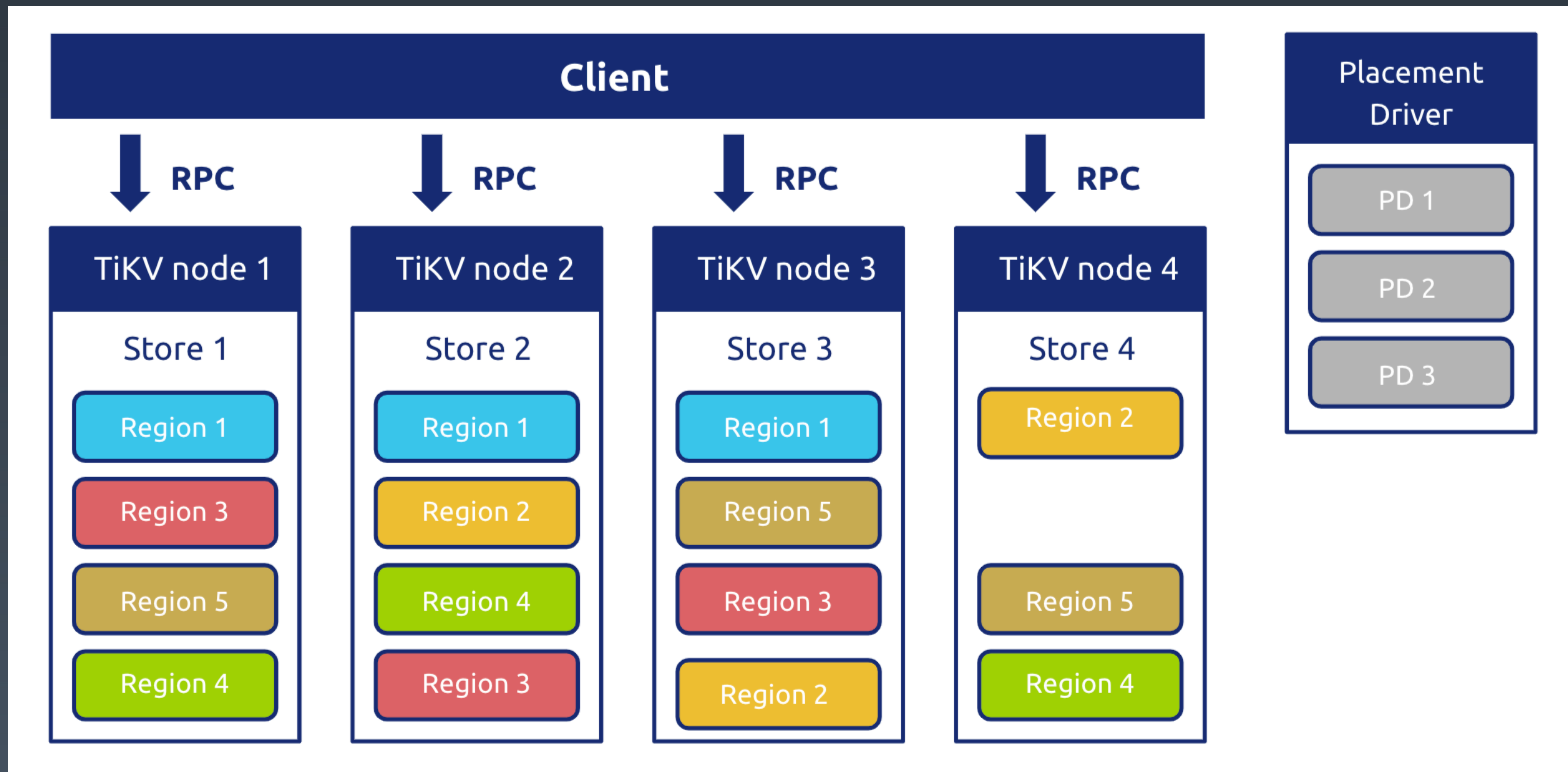
- 远程存储的问题：
 - （通常）性能较差，难以满足高 IO 需求的有状态应用
 - （通常）单位成本较高，大部分远程存储本身会做三副本
- hostpath 的问题：
 - 生命周期不受 Kubernetes 管理
 - 难以管理，需要手动限制 Pod 总是调度到同一个节点上
- Local PersistentVolume：
 - 数据易失（相对于远程存储，没有三副本保障）
 - 节点故障会影响数据访问
 - 相比于远程存储，难以垂直扩展容量

使用 Local PV

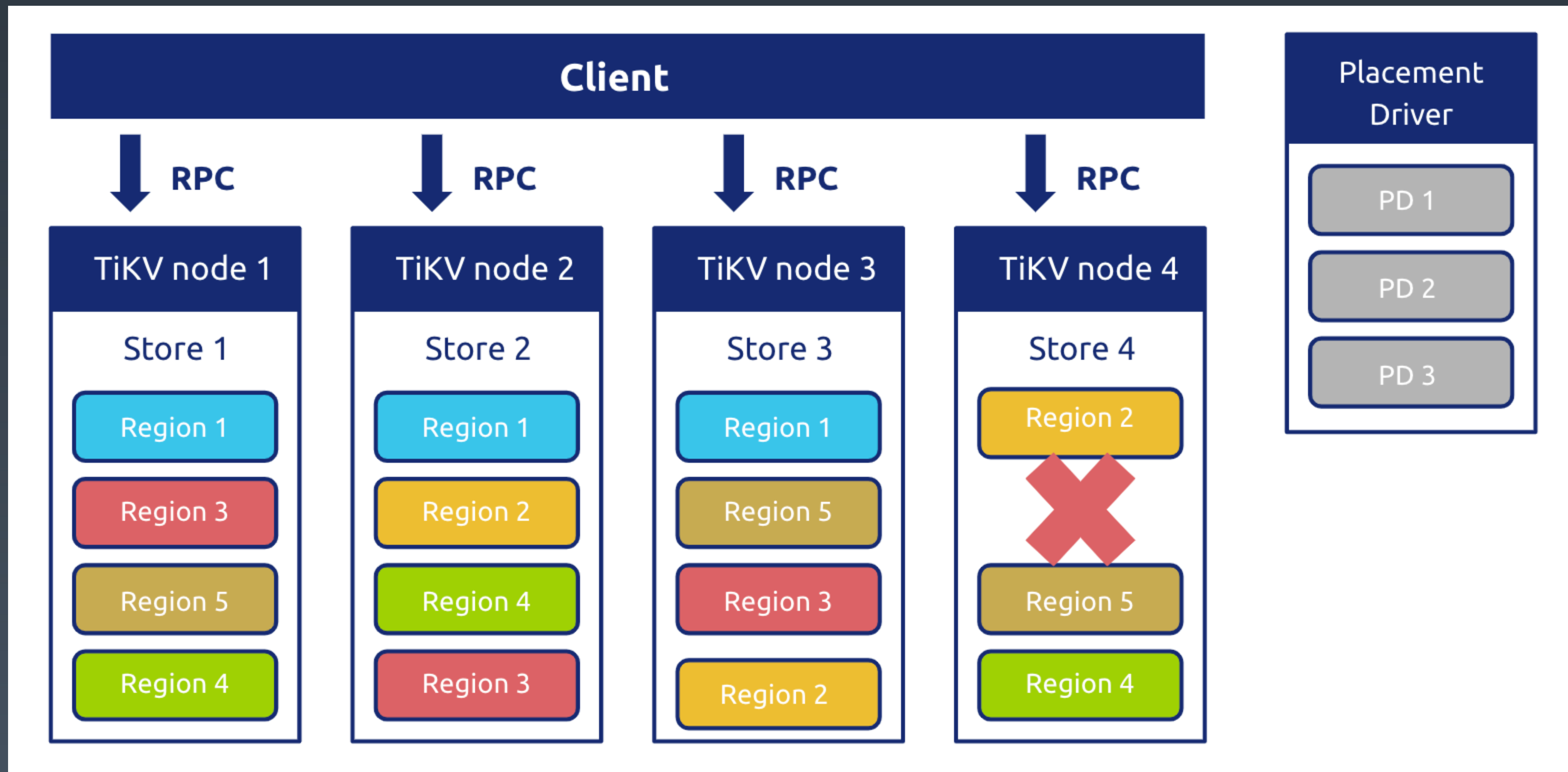
还好，TiDB 设计时充分考虑了这些问题。

- 必须在应用层实现数据冗余
 - TiKV 默认每个 Region 三副本
 - 当副本缺失时，TiKV 能自动补齐副本数
- 节点故障影响数据访问
 - Raft Leader Election
- 难以垂直扩展
 - TiKV 支持水平扩展

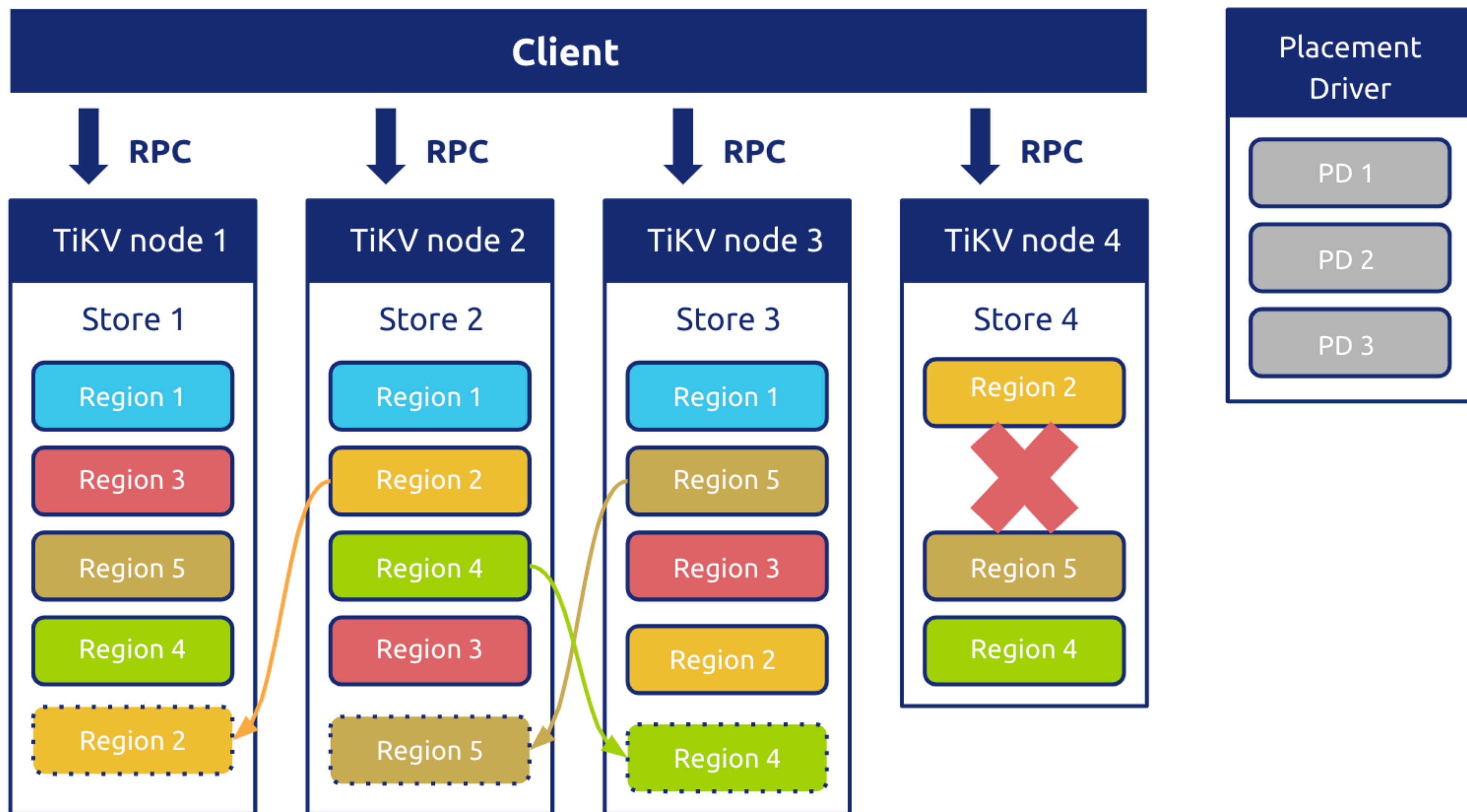
基石：TiKV 数据调度



基石：TiKV 数据调度



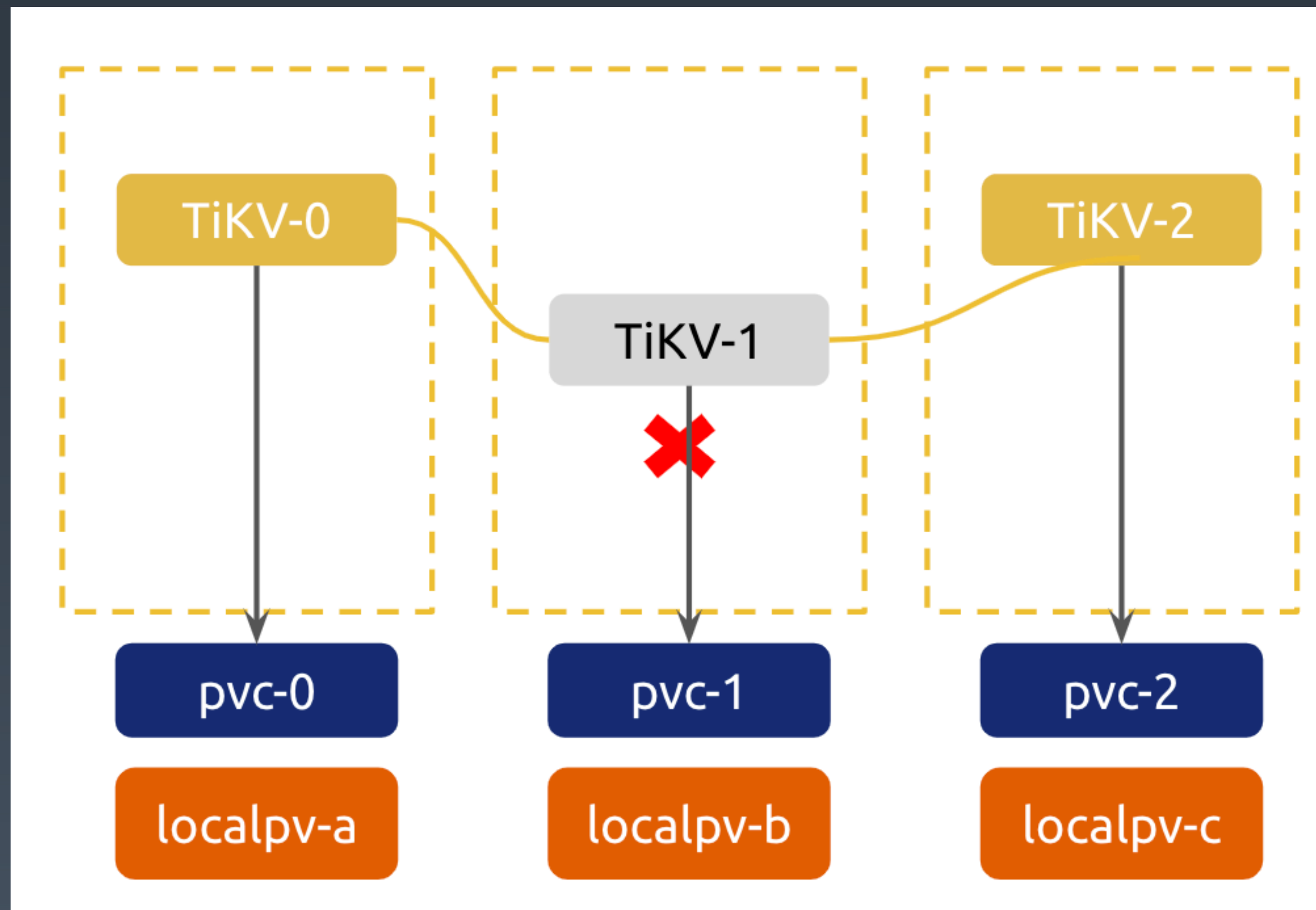
基石：TiKV 数据调度



基石：TiKV 数据调度

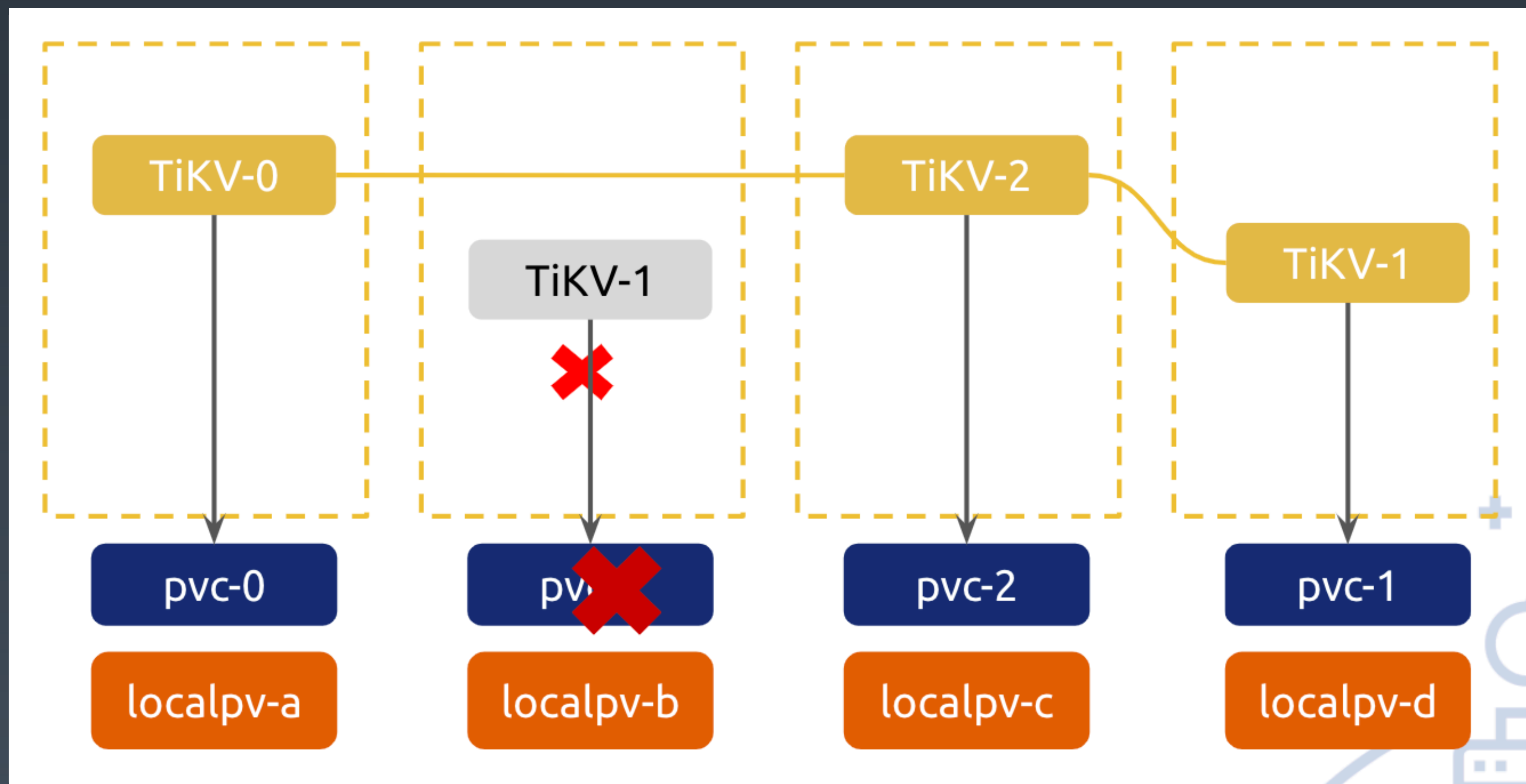
- 存储层的多副本和数据自动调度是 TiDB 能够使用 Local PV 的前提，也是水平伸缩的关键。
- 当然，在发生节点或磁盘故障时，由于旧 Pod 无法正常运行，我们需要控制器帮助我们进行恢复，确保有足够的健康 Pod 数来承提供足够的存储空间与 IO 能力，即 failover。

Failover



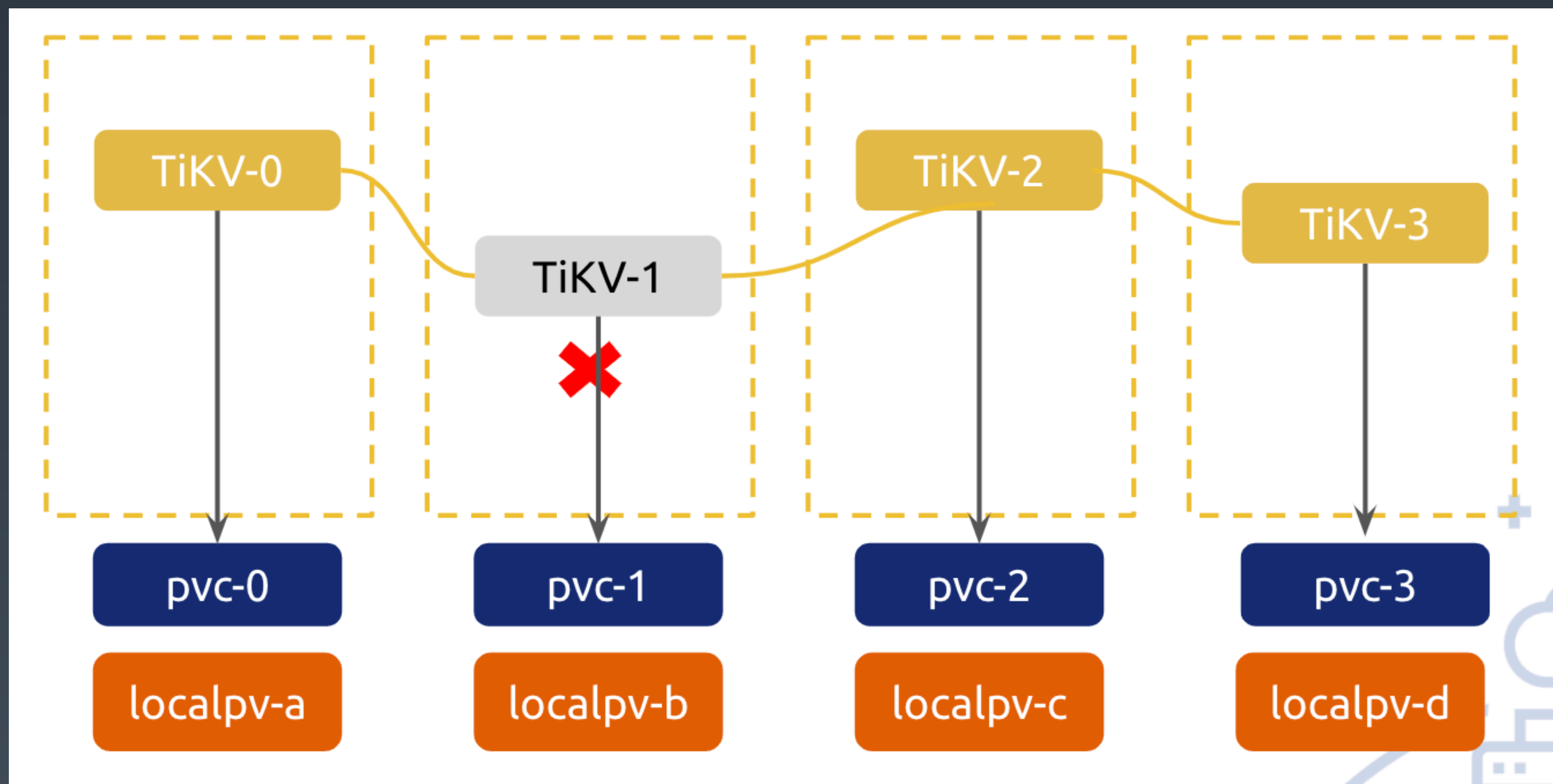
由于 TiKV 绑定了本地节点上的 PV，因此当节点恢复前，TiKV-1 将一直处于 Terminating 或 Pending 状态

能够确定节点状态



当运维或控制器确认 Node 已经宕机，删除 Node 对象时（如公有云上的 NodeController 会通过查询公有云 API 来删除已经释放的 Node），我们的控制器可以直接删除 pvc-1，解绑 PV。

无法确定节点状态



除节点物理故障外，也要考虑发生网络分区，TiKV-1 进程实际仍在运行的可能。节点状态未知的原因有很多，很难在原地进行恢复。此时控制器新增 Pod 进行补齐。

总结

有状态应用的云原生之路

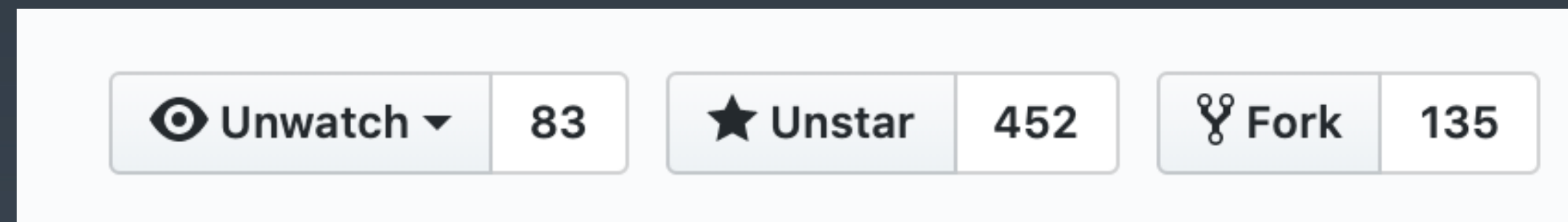
一些经验：

- Operator 本身的复杂度不可忽视；
- Local PV 能满足高 IO 性能需求，代价则是编排上的复杂度；
- 应用本身必须迈向云原生（**meets kubernetes part way**）；

当然，与拥抱云原生带来的红利相比，这些都是值得的！ 😊

眼见为实

tidb-operator 完全开源：




<https://github.com/pingcap/tidb-operator>

期待你的 issue 和 PR! 😎

联系我

邮箱：
wuyelei@pingcap.com



吴叶磊 

Hangzhou, Zhejiang



Scan the QR code to add me on WeChat

THANKS! | QCon th