

Draft Version

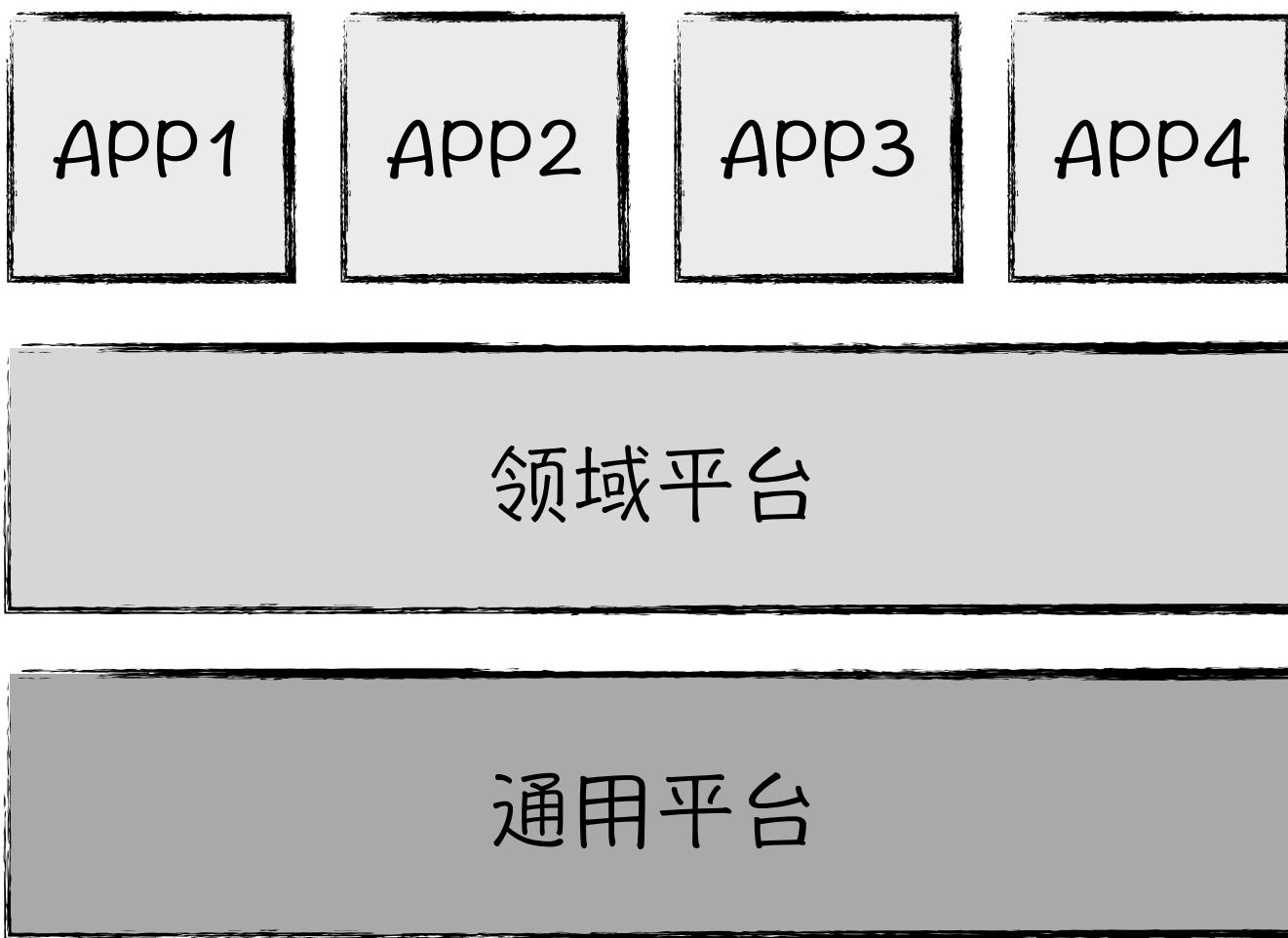
Some Thoughts on Platform Evolution

王博

CONTENT

- **Monolithic Platform**
- **Becoming Muddy**
- **Clean Architecture**
- **Component Platform**

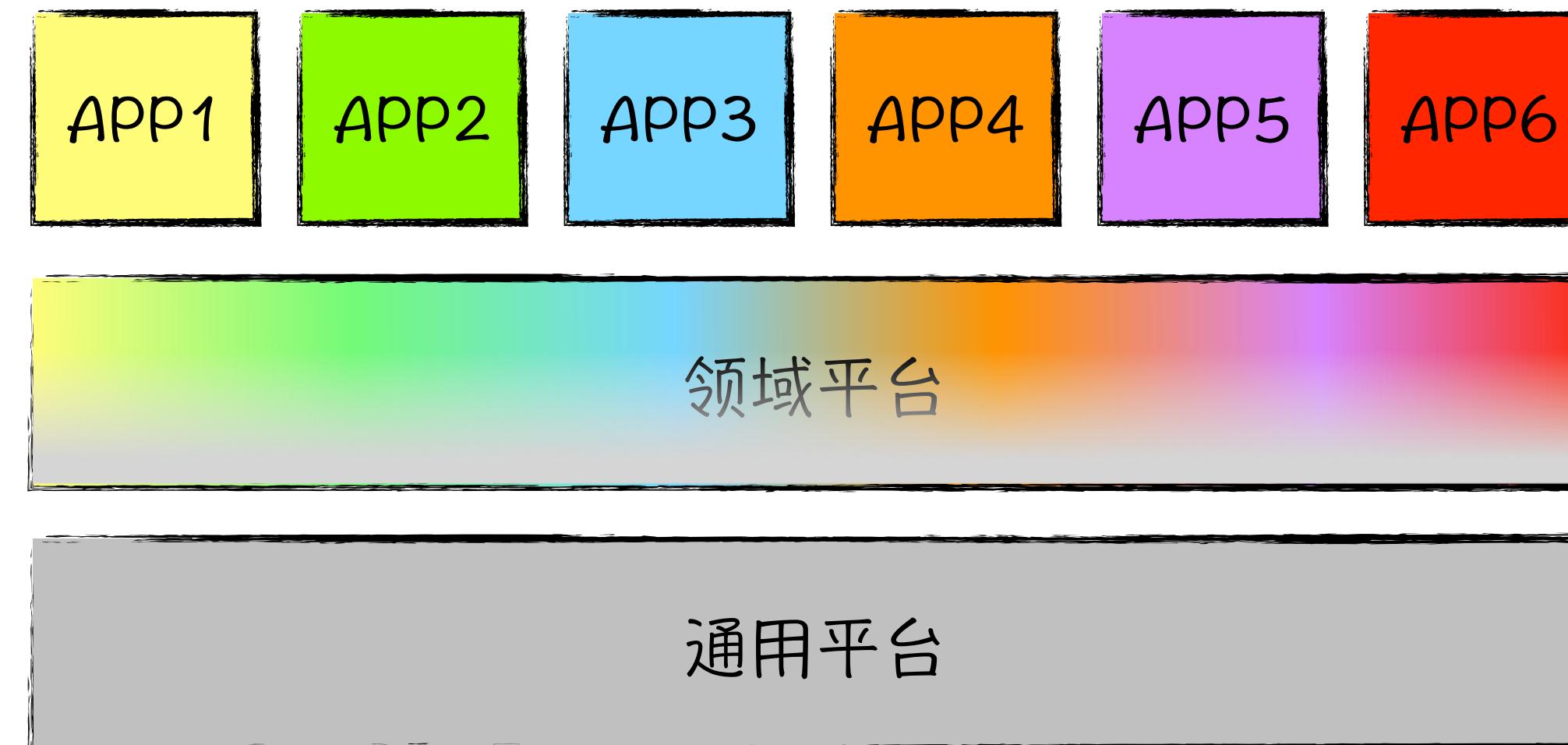
Monolithic Platform



单体平台化的**优点**:

- 统一运行时: 易于**共享**软硬件资源和业务资源;
- 统一开发过程: 易于代码**复用**, 版本管理简单;
- 依赖紧凑, 易于同步调用, **性能高**;

Becoming Muddy



当应用越来越多，且对平台的要求差异多样，单体平台必然会出现“**混浊化**”问题，继而影响到整体的响应力、灵活性、安全性和性能优化：

- 单体平台和应用的边界相对固化，当越来越多的应用加入后，应用变化不可避免的会侵入平台内部，必然导致平台代码重复率和散弹式修改问题变得严重，代码腐化快，测试和维护成本巨大；
- 应用感知变化快，平台感知慢。当平台支撑的应用越来越多，应用的隔离性、安全性、资源占用、发布周期等都会受单体平台集中牵制，整体响应力会越来越慢；
- 单体平台内部缺乏可裁剪设计，特性依赖与裁剪不易做到，针对特殊应用的端到端性能和资源优化变得困难；

Examples

阿里巴巴：业务平台化（三淘时期）

- 随着APP的增多，单体平台响应力不足；
- 单体平台难以满足多业务的灵活性要求：业务规则、隔离性、发布周期、差异化极致性能…；
- 难以快速支持业务创新速度；

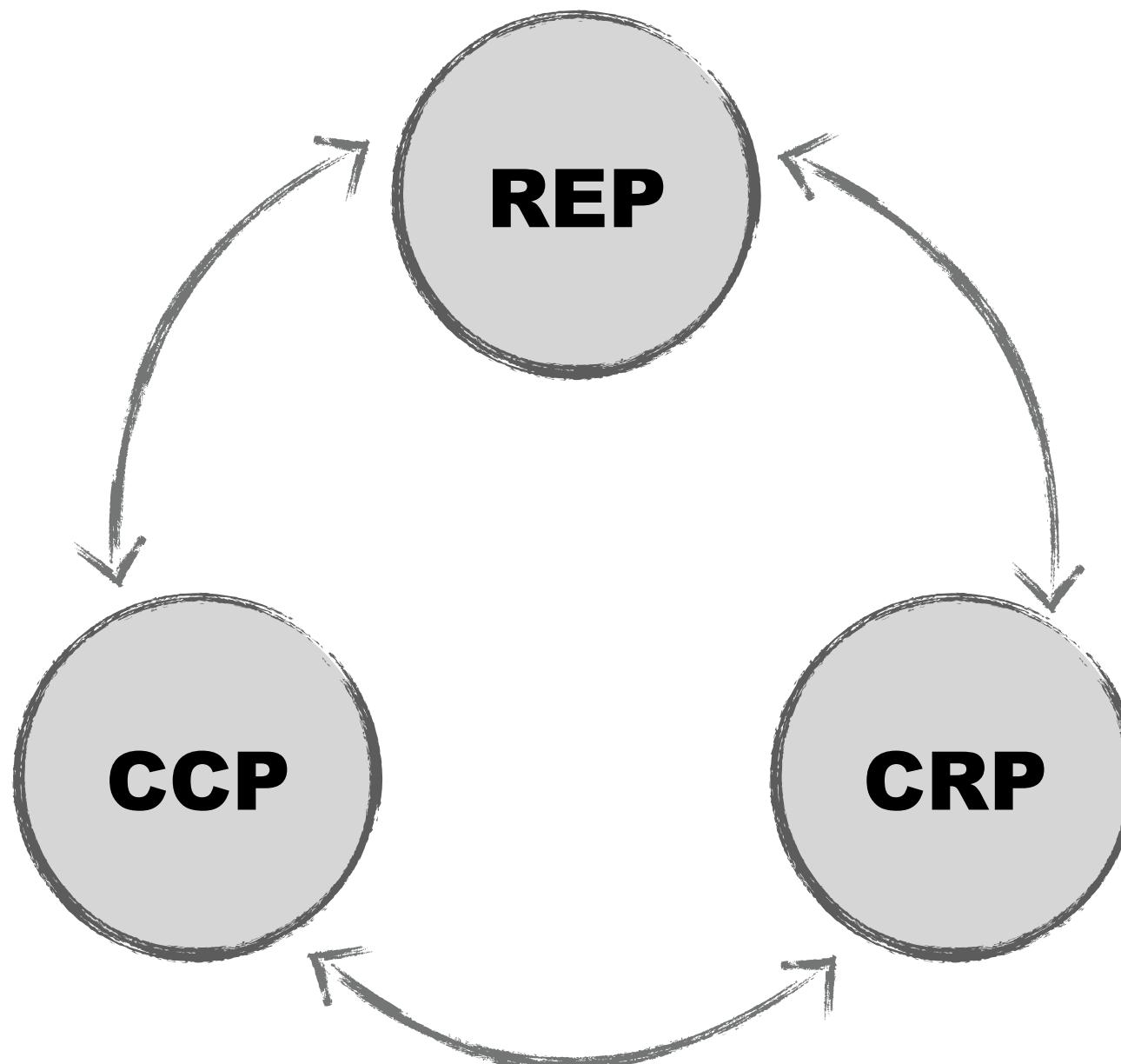
Java单体平台

- 资源占用越来越多：JDK 1.1, 10Mb -> JDK 8u77 达到227M；导致JAVA的应用场合越来越受限；
- 隔离性不足导致的安全性问题。可以通过Java平台的跨应用攻击；
- 代码质量随着历史兼容性和内部耦合混乱，下降严重；

阿里巴巴：业务中台化

Java 9版本平台模块化

Clean Architecture



CCP：共同闭包原则：

将为相同目的而修改的代码放在一起，为易维护性而组合；

CRP：共同复用原则：

将共同复用的代码放一起，将为不同目的而复用的代码进行分离；为避免不必要的发布而切分；

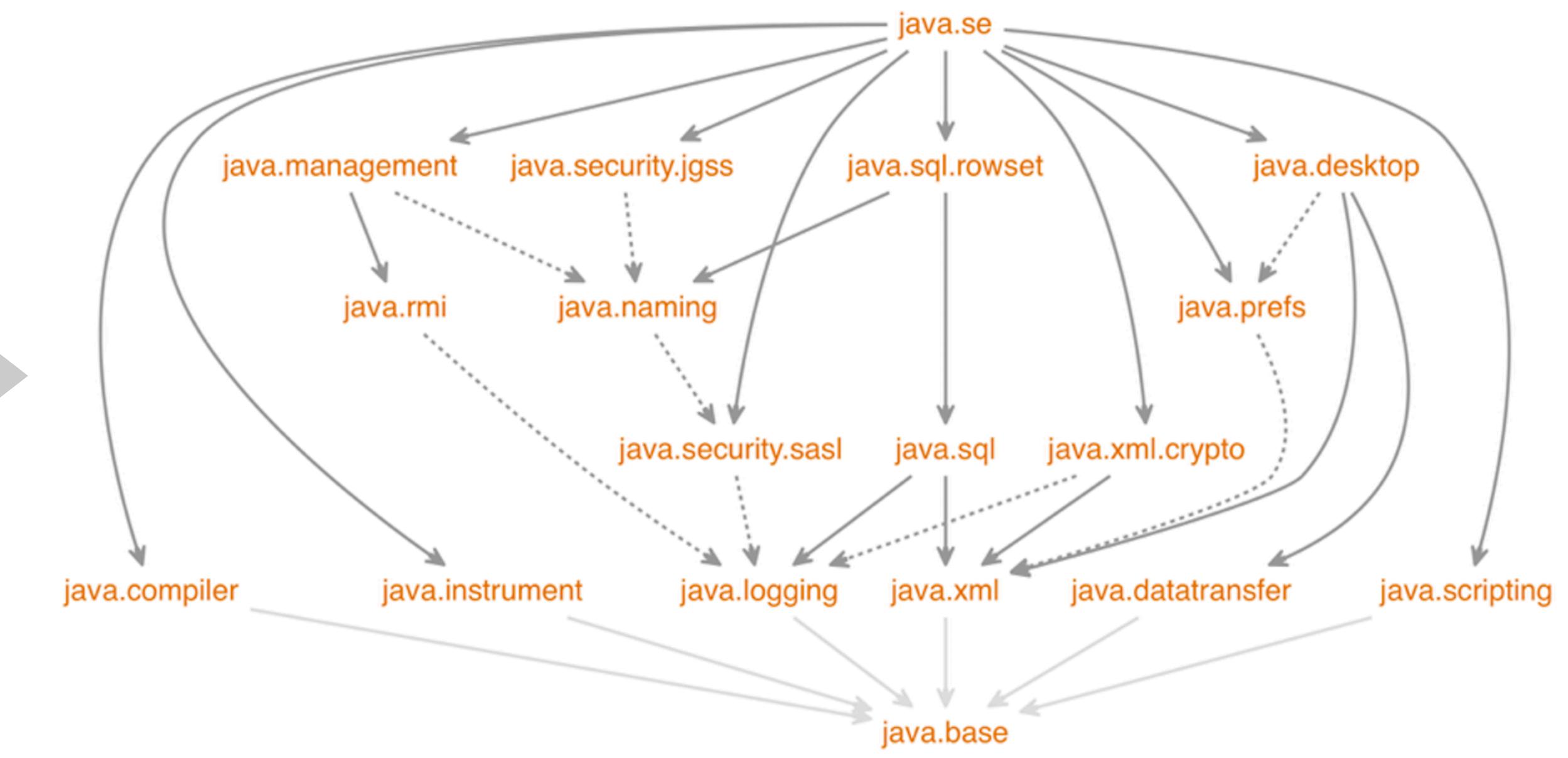
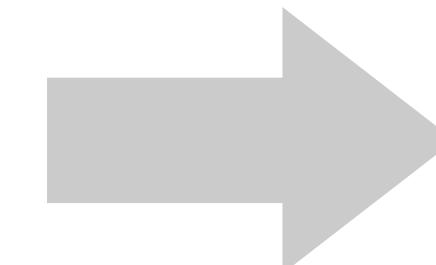
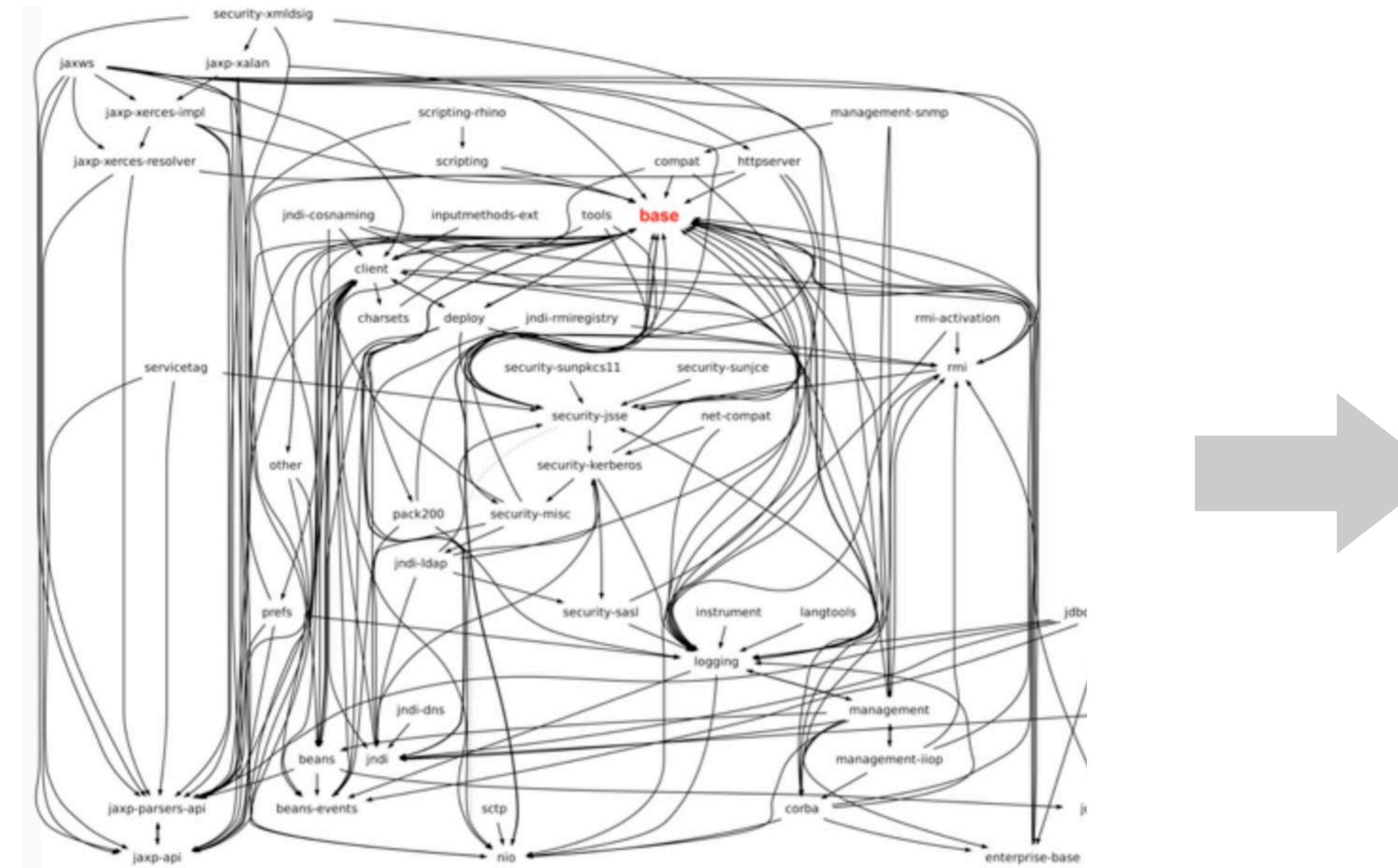
REP：复用/发布等价原则：

组件的最小复用粒度等同于其发布的最小粒度；为平衡发布和复用边界而切分或组合；

一般情况下，软件项目的重心会随着时间进行移动；开始会为了易维护性而将代码都放在一起，随着软件面临变化的增多，以及对隔离性和灵活性的要求，需要逐步对代码进行分离，最终向可独立复用性和可独立发布性的平衡点靠近。这过程中伴随着不断对软件的细化设计以及拆分与重组。

通过细化设计与拆分，让软件的内部边界保持适应性和灵活性，不同部分可以根据不同的复用粒度进行组合或者覆盖，提高了整体的灵活性和响应力，避免了软件腐化，同时最大化兼顾了软件的资产复用。

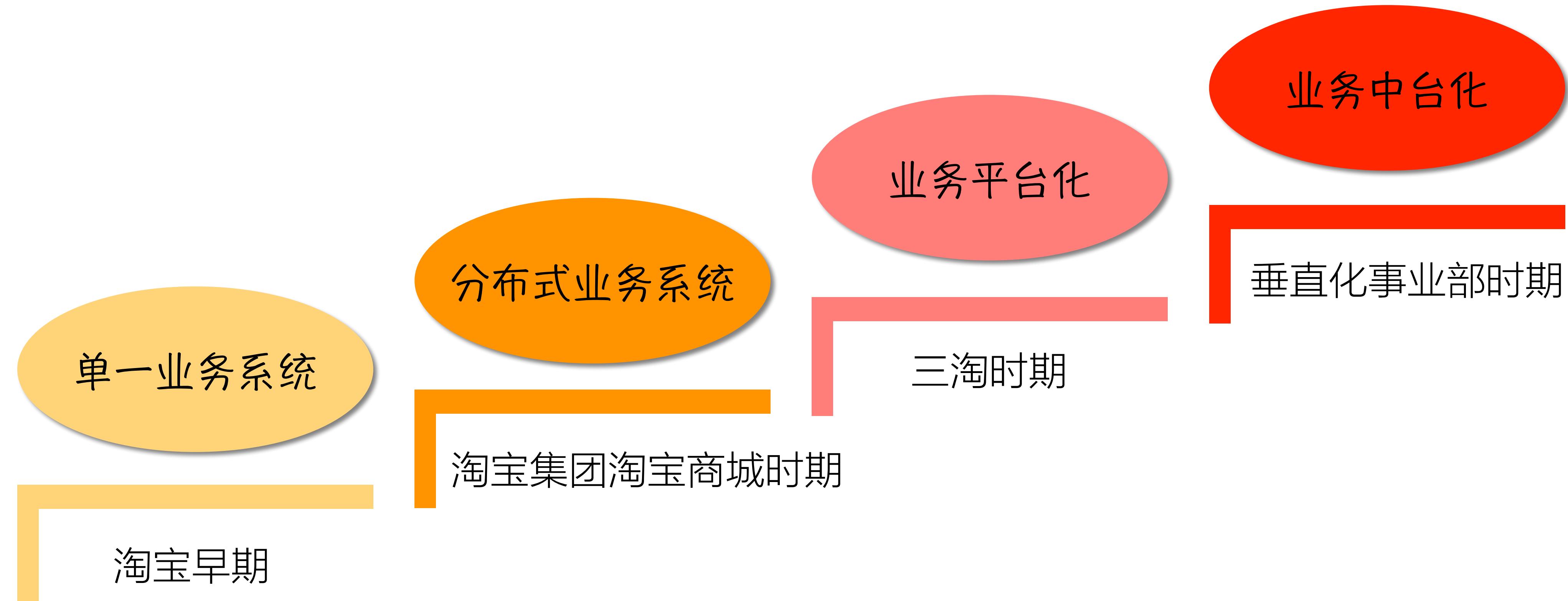
Example : Java Platform Module System



Java平台的模块化过程，从Java 7（2011年）开始到Java9（2017年）完成。

主要工作：1) 语言支持模块化特性（显示化的接口发布与依赖声明）；2) JDK平台的模块化设计拆分与重构；解决了Java单体平台资源占用大，难以裁剪、隔离性差（安全）等问题，使得Java可以适应更多场合（例如IOT等）；

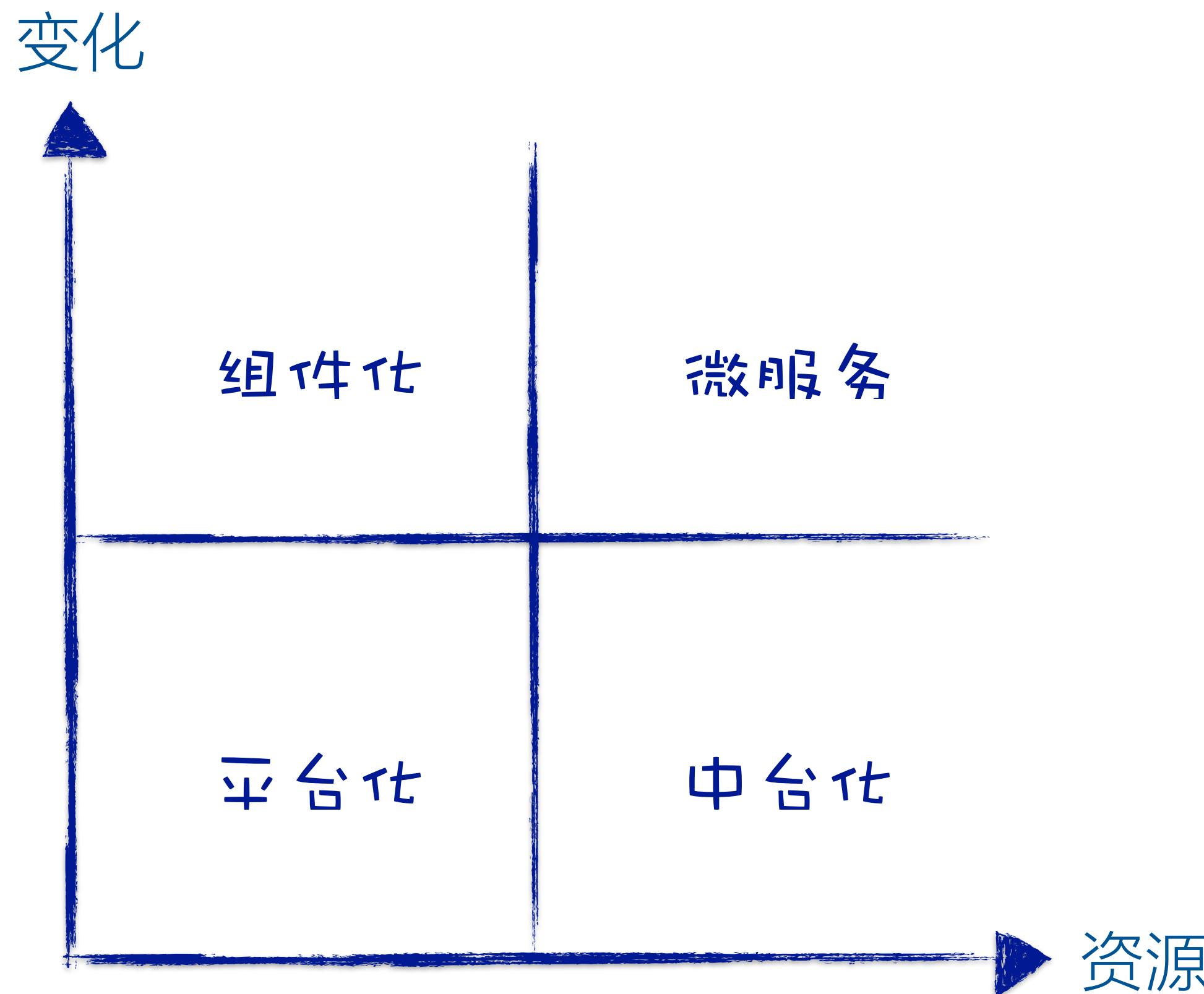
Examples : 阿里巴巴中台化演进



阿里巴巴中台化战略，借助云和服务化技术，将平台拆分成不同类型的可独立运维的中台服务。

在满足复用性的前提下，同时保证不同业务的隔离性与灵活性要求。

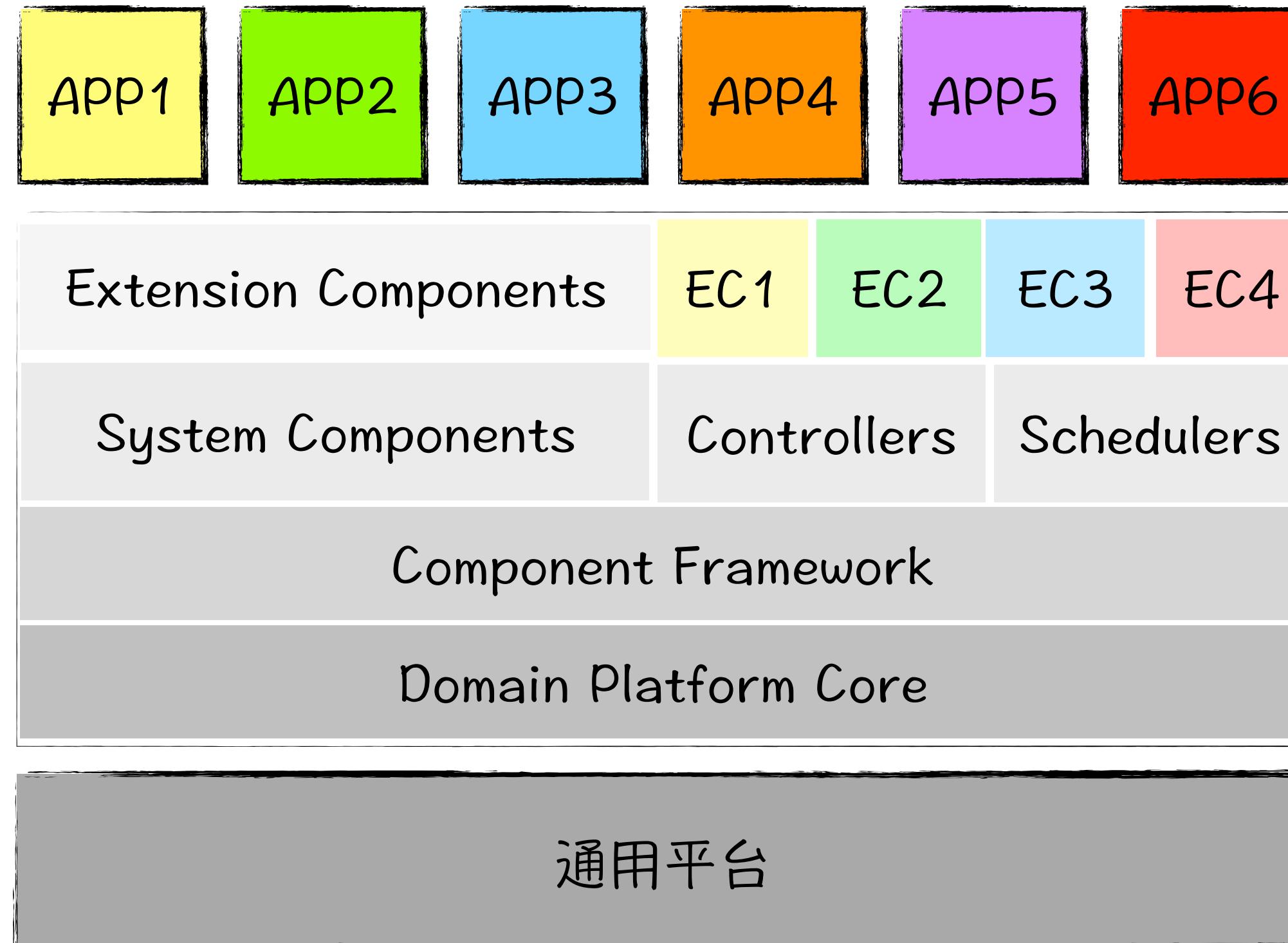
Constraints



中台化以云和服务化技术为基础，保留了平台化能力复用的同时，降低了黑盒复用的成本（不关心被依赖方的部署、发布、升级、弹缩等运维活动），同时提升了业务的隔离性与灵活性；

但是在资源和性能约束强的系统中，某些特性（独立运维等）需要为性能和资源让路，因此最终软件**组件化**（结合了模块化和服务化的平衡策略），将会是可取之道。

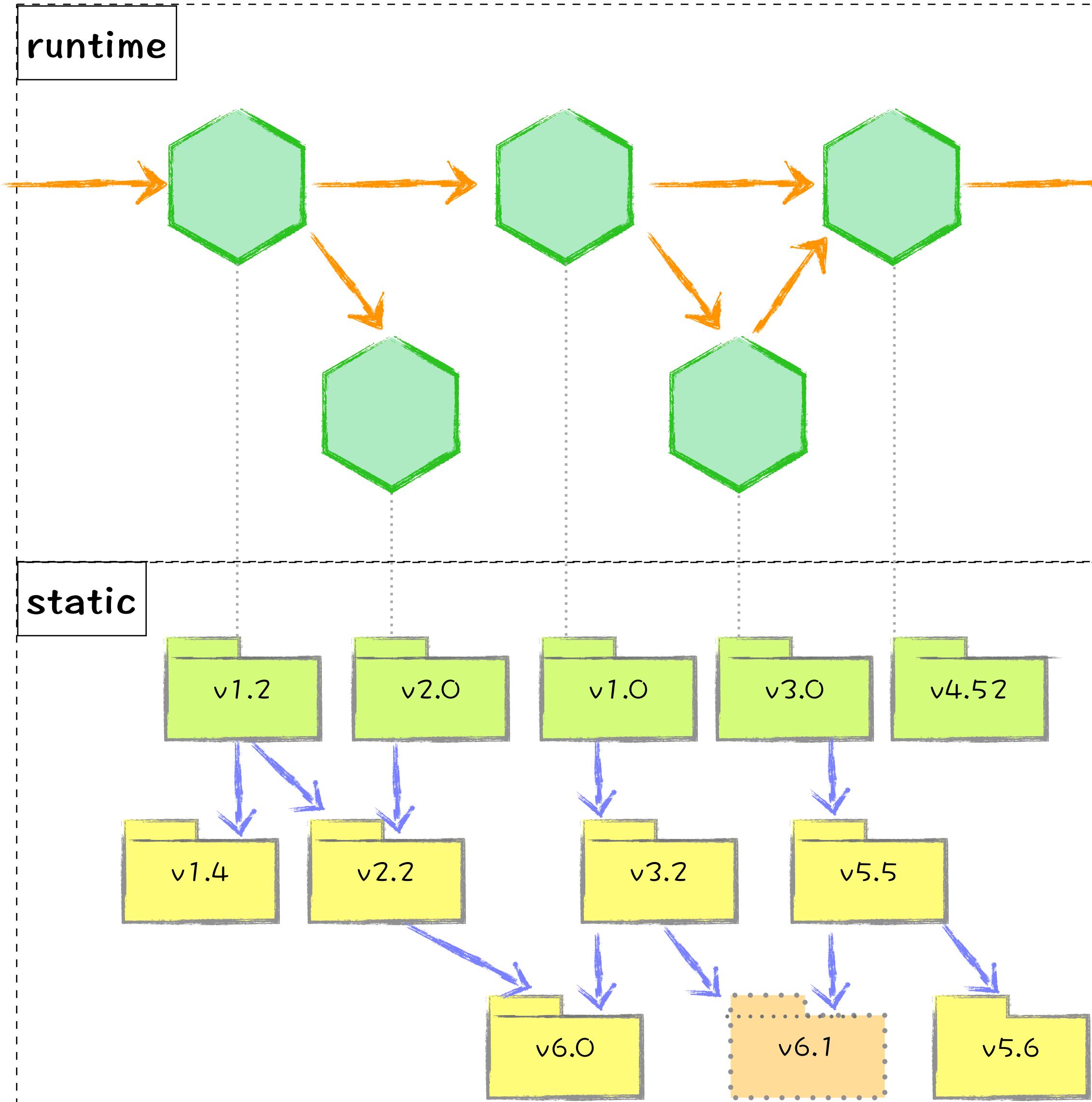
Component Platform



核心特点:

- 弹性边界，增强APP的自主**响应力**，平台趋于智能调度和控制。APP可扩展和定制平台资源、配置和算法等差异部分，平台负责在整体约束下进行加载、解释和调度，灵活应对资源的共享和分离问题；
- 平台具备独立的局部交付和组合能力，增强了整系统的**灵活性**；
- 平台内**隔离性**变强，提升了安全性，同时降低了APP之间的互相影响；
- 最大化兼顾了单体平台的**复用性和性能**；
- 系统组成部分变多，配置变灵活，需要借助一定的治理框架和开发工具进行**复杂度控制**；

Technology Challenges



组件（插件与服务）治理与运行时框架：

- 通信（注册与发现）； 2) 生命周期管理； 3) 资源管理与隔离；
- 部署编排与依赖自动安装（包括冲突解决）；
- 组件替换时机与业务中断级别：1) 重启替换； 2) 运行期有损； 3) 运行期无损（灰度发布）；
- 轻量级沙盒；
- 安全与鉴权机制；
- 动态依赖追踪与管理；

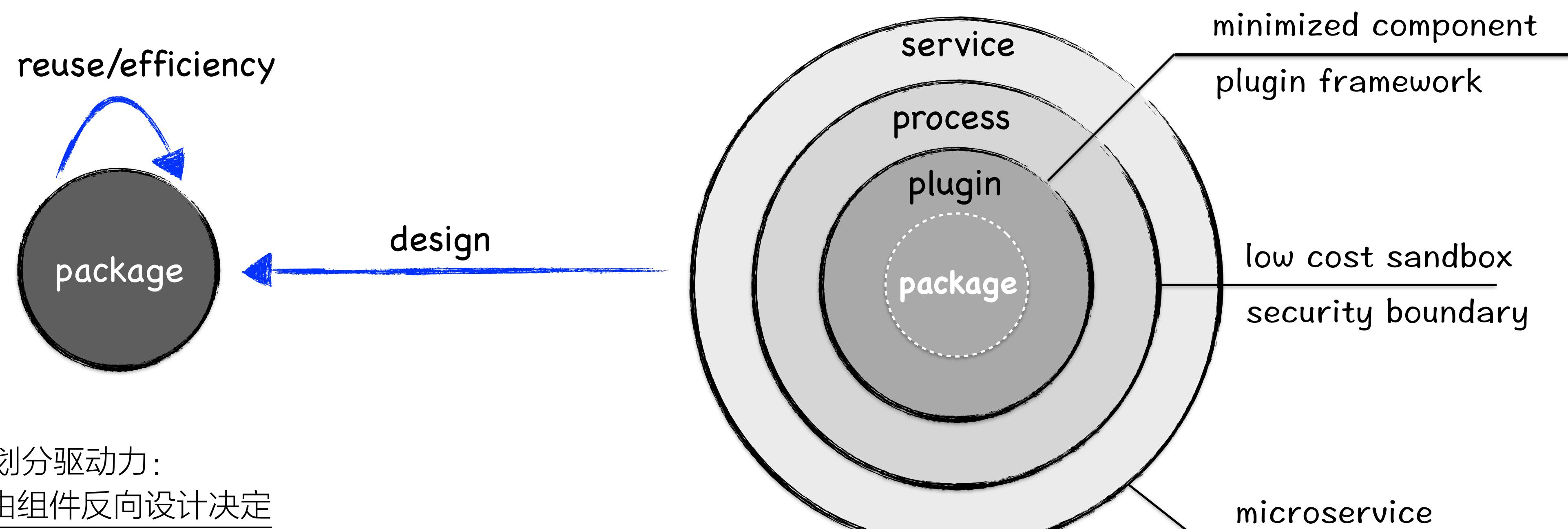
模型驱动：

- 从动态到静态全局的可视化依赖管理
- 模型驱动设计，指导组件和包的划分
- 模型辅助性能指标的量化分解与验证

开发态依赖管理机制：

- 组件与软件包关系管理追踪；
- 软件包版本管理（可定制的内部交付与外部交付策略）；
- 自动化的依赖管理、集成与追踪，以及冲突解决；
- 以包交付为中心的团队级独立流水线和devops能力；
- 以包为中心内建的个人标准化高效开发环境和基础设施；

Component Based Design



包划分驱动力：

- 由组件反向设计决定
- 由可复用的粒度决定
- 由包规模或组织结构决定

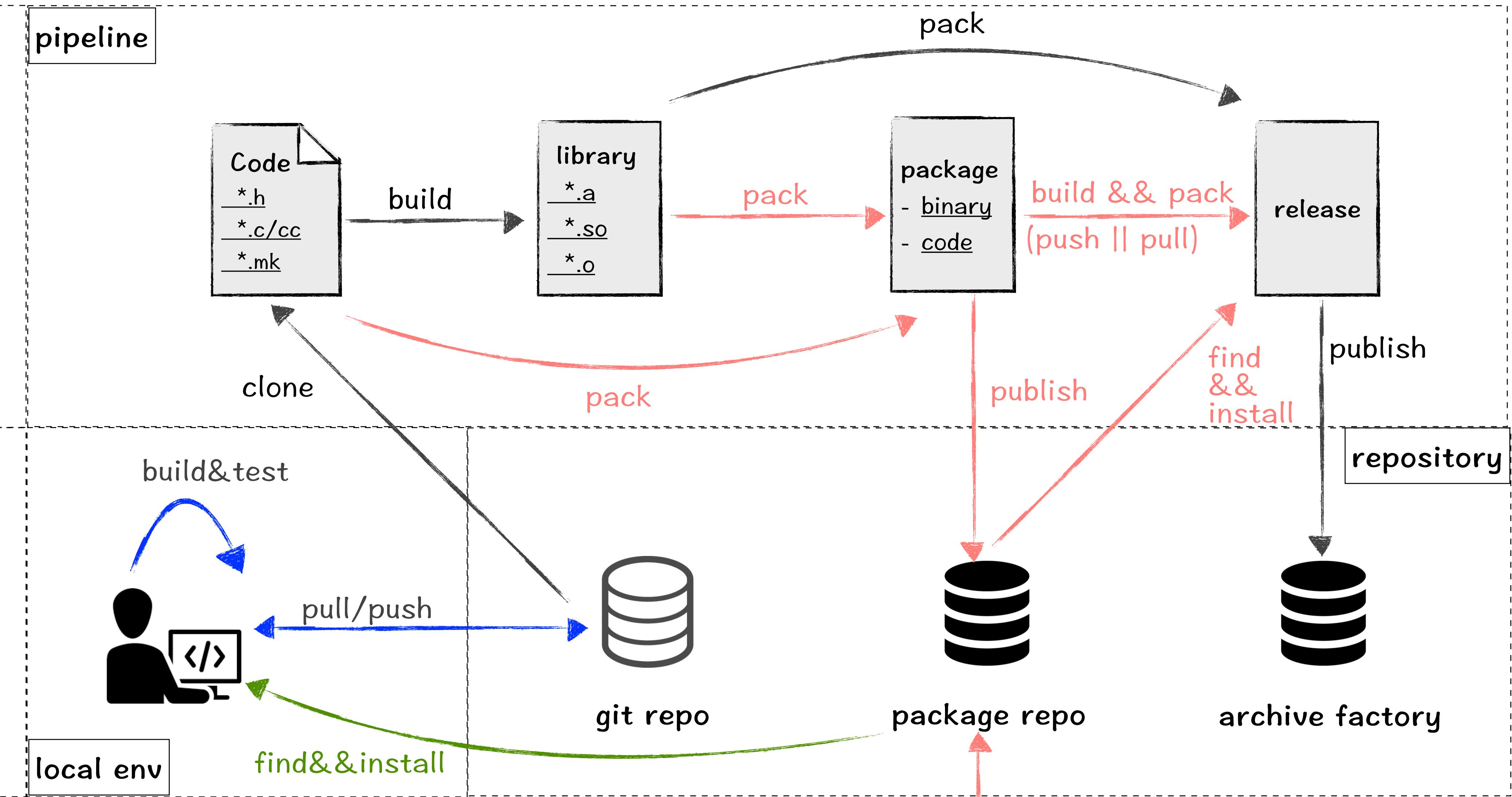
组件划分驱动力：

- 最小的独立交付和替换的边界决定；
- 不同安全级别和边界决定；
- 受系统的软硬件部署视图、资源以及性能的约束；

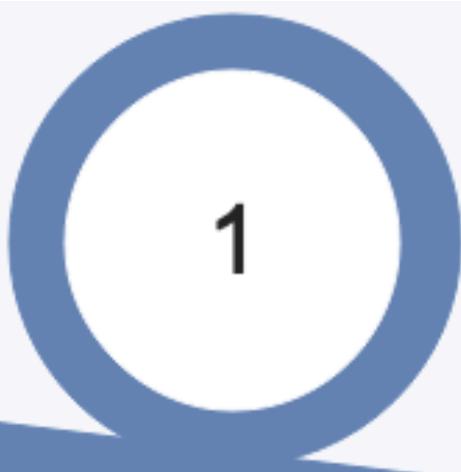
Component Based Development

以包交付为目标，使能了团队级独立流水线和团队devops能力。自动化的依赖管理、集成和追溯，整体高效运维；

参考现代化语言工具链，以包为中心内建个人现代化的标准开发环境。统一独立开发、构建、功能和契约测试、性能和安全测试等开发最佳实践和高效基础设施。

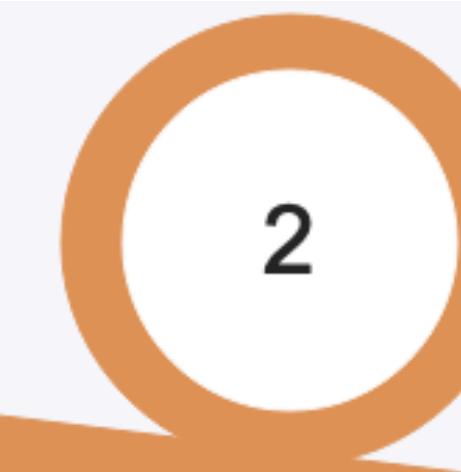


以包和包仓库为标准，统一了第三方依赖的获取和追踪，并为开放性和社区化做好准备



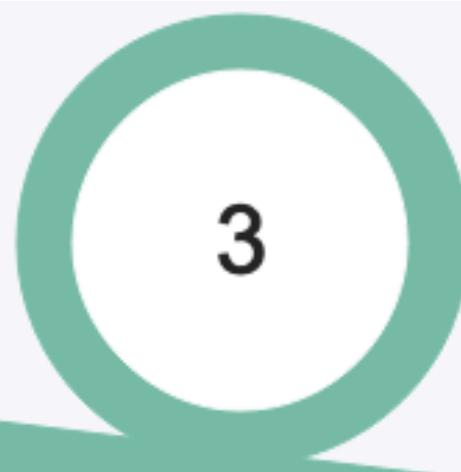
独立应用

- 应用独立开发，隔离性强，交付独立；
- 相似功能软件重复投资，复用性差，成本大；
- 不能共享运行时资源，共同部署时资源消耗大、浪费大；



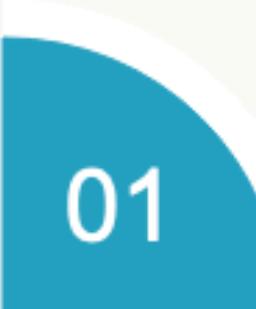
单体平台

- 相同功能统一开发，复用性强，避免重复投资；共享运行时可以最大化复用系统资源，统一部署时开销小；
- 在支持应用数不多，且变化不剧烈的时候，是一种平衡了投入和产出的选择；
- 当支持的应用变多，变化剧烈时，带来的问题是互相干扰，安全性和单独应用的响应力和灵活性下降。平台代码质量下降快，极致的端到端性能优化难以实施；



组件化平台

- 增强APP的自主响应力，平台趋于智能调度和控制。APP可扩展和定制平台资源、配置和算法等差异部分，平台负责在整体约束下进行加载、解释和调度，灵活应对资源的共享和分离问题；
- 平台具备独立的局部交付和组合能力，增强了整系统的灵活性；平台内隔离性变强，提升了安全性，同时降低了APP之间的互相影响；
- 最大化兼顾了单体平台的复用性和性能
- 系统组成部分变多，配置变灵活，需要借助一定的治理框架和开发工具进行复杂度控制；



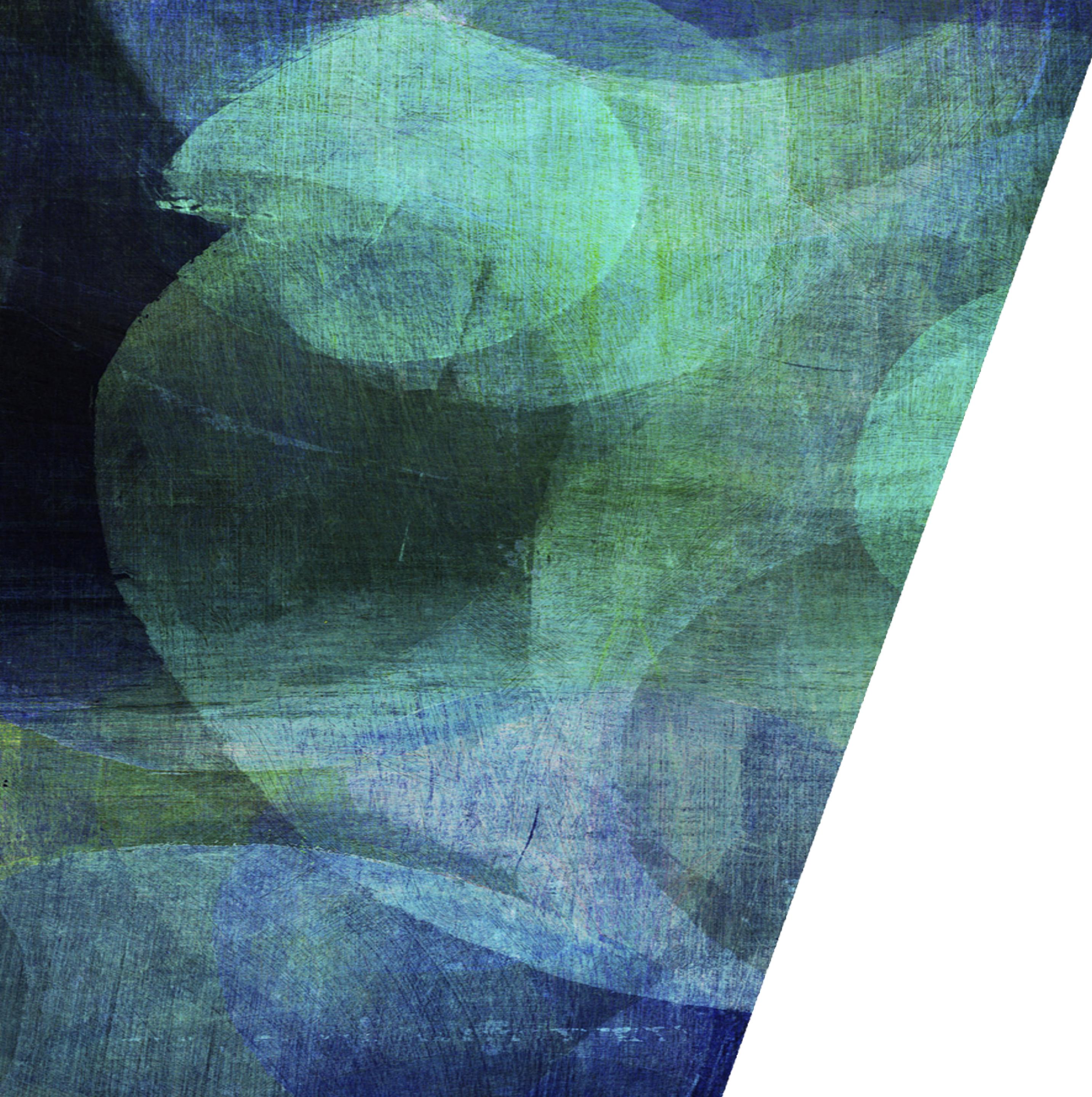
论证应用变多，差异大，对平台和系统带来的问题的真实性和紧迫性；



论证新的技术出现（硬件、框架、工具：如服务化、包管理、轻量级沙盒等）能够解决新平台从运行时到开发态的问题挑战；



论证业界有先行者的成功案例；



THANKS