aws

将单体分解为微服务

AWS 规范性指导

AWS 规范性指导: 将单体分解为微服务

Table of Contents

将单体分解为微服务

Tabby Ward 和 Dmitry Gulin、Amazon Web Services (AWS)

2023 年 4 月（文档历史记录）

迁移到 Amazon Web Services (AWS) 云具有许多优势，包括技术和业务灵活性、新的收入机会和降低的成本。要充分利用这些优势，您应该通过将整体应用程序重构为微服务来不断实现组织软件的现代化。此过程包括三个主要步骤：

- 将单体分解为微服务：使用本指南提供的分解模式将整体应用程序分解为微服务。
- 集成微服务：使用 AWS 无服务器服务将新创建的微服务集成到微服务架构 中。
- 为微服务启用数据持久性：通过分散数据存储，促进微服务之间的多语言持久性。

现代化通常涉及两种类型的项目：

- Brownfield 项目涉及在现有或遗留系统的背景下开发和部署新的软件系统。
- Greenfield 项目涉及从头开始为全新的环境创建系统，而不涉及任何遗留代码。

对于棕地项目，应用程序现代化之旅的第一步是将投资组合中的整体分解为微服务。

大多数应用程序最初都是为特定业务用例设计的单体。如果单体架构不强制模块化设计，那么对于那些在既定领域知识的范围内没有明确定义职责的应用程序来说，单体结构仍然是有效选择。作为单个部署单元的整体结构的核心特征也可以帮助减少设计缺陷，例如紧密耦合或缺乏内部结构。

尽管对于某些用例来说，单体可能是一个有效的选择，但它通常不适合现代应用程序。单体架构的内部结构定义不明确会使代码难以维护，这使得新开发人员的学习过程曲折，并导致额外的支持成本。高耦合和低内聚会显著增加添加新功能所需的时间，并且您可能无法根据流量模式扩展单个组件。单体还需要多个团队为一个大型版本进行协调，这增加了协作和知识转移的负担。最后，您会发现，随着您的业务或用户群的增长，添加新功能或打造新的用户体验变得困难。

为了避免这种情况，您可以使用分解模式来分解单体应用程序，将它们转换为多个微服务，然后将其迁移到微服务架构。微服务架构将应用程序结构化为一系列松耦合服务。微服务旨在通过支持持续交付和持续部署（CI/CD）流程来加速软件开发。

在开始分解过程之前，您应评估要分解哪些单体。确保包含存在可靠性或性能问题的单体，或者在紧密耦合的架构中包含多个组件。我们还建议您充分了解单体架构的业务用例、其技术及其与其他应用程序的相互依赖关系。

本指南适用于应用程序所有者、企业主、架构师、技术主管和项目经理。它讨论了以下六种用于分解单体的云原生模式，并描述了每种模式的优势和劣势：

- 按业务能力分解
- 按子域分解
- 按交易分解
- 按团队提供的服务模式
- strangler fig 模式
- 按抽象模式分支

该指南是内容系列的一部分，该系列涵盖了推荐的应用程序现代化方法 AWS。该系列还包括：

- 实现云端应用程序现代化的策略 AWS
- 分阶段实现云端应用程序现代化的方法 AWS
- 评估 AWS 云端应用程序的现代化准备情况
- 使用 AWS 无服务器服务集成微服务

目标业务成果

在将整体分解为微服务后，您应该期待以下结果：

- 将您的单体应用程序高效过渡到微服务架构。
- 快速调整飘忽不定的业务需求，而不中断核心活动，例如高可扩展性、改进的弹性、持续交付和故障隔离。
- 加快创新，因为每项微服务都可以单独测试和部署。

分解单体的模式

决定分解应用程序组合中的一个单体之后，您应该选择适当的分解模式并将其引入您的组织。

> **Note**
>
> 您可以使用多种模式来分解一个单体。例如，您可以按业务能力分解单体，然后使用子域模式对其进行更多分解。

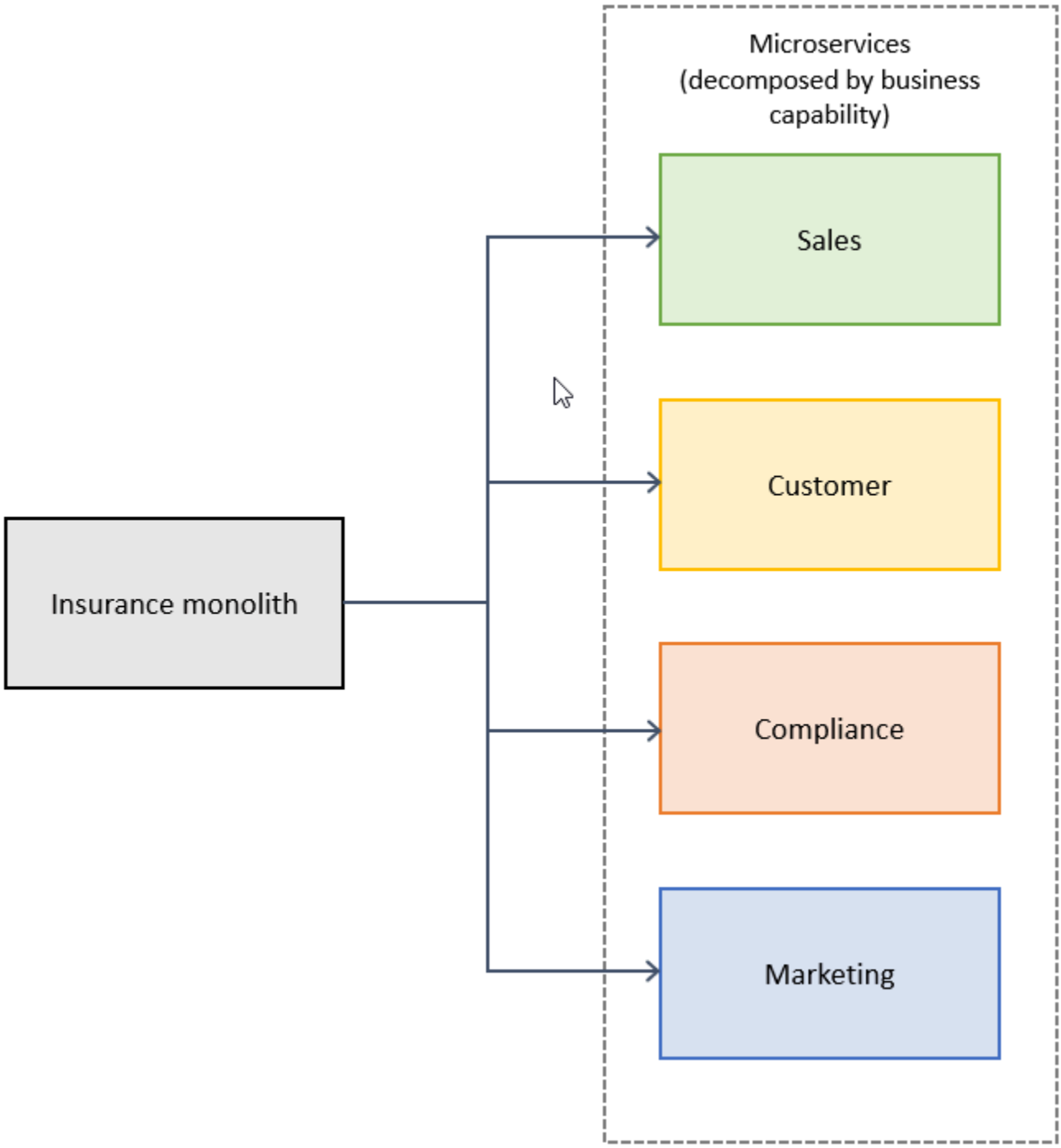
主题

按业务能力分解

您可以利用组织的业务流程或能力来分解一个单体。业务能力是指企业如何创造价值（例如，销售、客户服务或营销）。通常，一个组织具有多种业务能力，这些能力因部门或行业而异。如果您的团队对组织的业务部门有足够的洞察力，并且每个业务部门都有域界专家 (SME)，则使用此模式。下表说明了使用此模式的优势和劣势。

优点	缺点
• 如果业务能力相对稳定，则生成稳定的微服务架构。 • 开发团队是跨职能的，围绕提供业务价值而不是技术功能进行组织。 • 服务是松散耦合的。	• 应用程序设计与商业模式紧密结合。 • 需要对整体业务有深入的了解，因为可能很难确定业务能力和服务。

在下图中，保险单体根据业务能力分解为四个微服务。



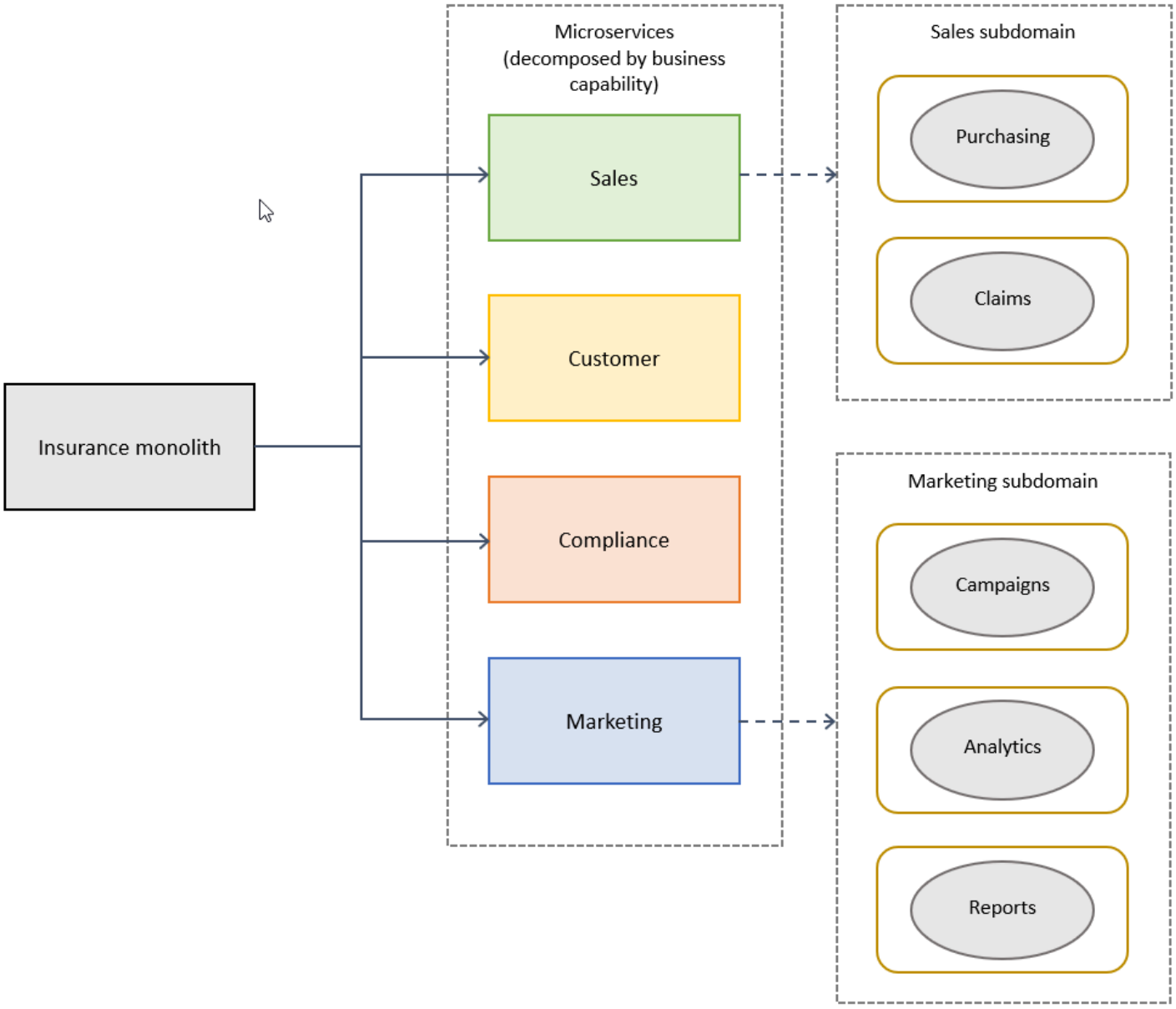
按子域分解

这种模式使用域驱动设计 (DDD)子域来分解单体。这种方法将组织的域模型分解为单独的子域，这些子域被标记为核心（业务的关键差异化因素）、支持（可能与业务相关但与差异化因素无关）或通用（不是特定于业务）。这种模式适用于现有的单体系统，这些系统在子域相关模块之间具有明确定

义的边界。这意味着您可以通过将现有模块重新打包为微服务来分解整体结构，而无需大量重写现有代码。每个子域都有一个模型，该模型的范围称为有界限上下文。微服务是围绕这种界限上下文开发的。下表说明了使用此模式的优势和劣势。

优点	缺点
• 松耦合架构提供了可扩展性、弹性、可维护性、可延展性、位置透明性、协议独立性和时间独立性。 • 系统变得更具可扩展性和可预测性。	• 可能会创建太多的微服务，这使得服务发现和集成变得困难。 • 业务子域很难识别，因为它们需要对整体业务有深入的了解。

下图显示了保险单体在按业务能力分解后如何将其分解为子域。



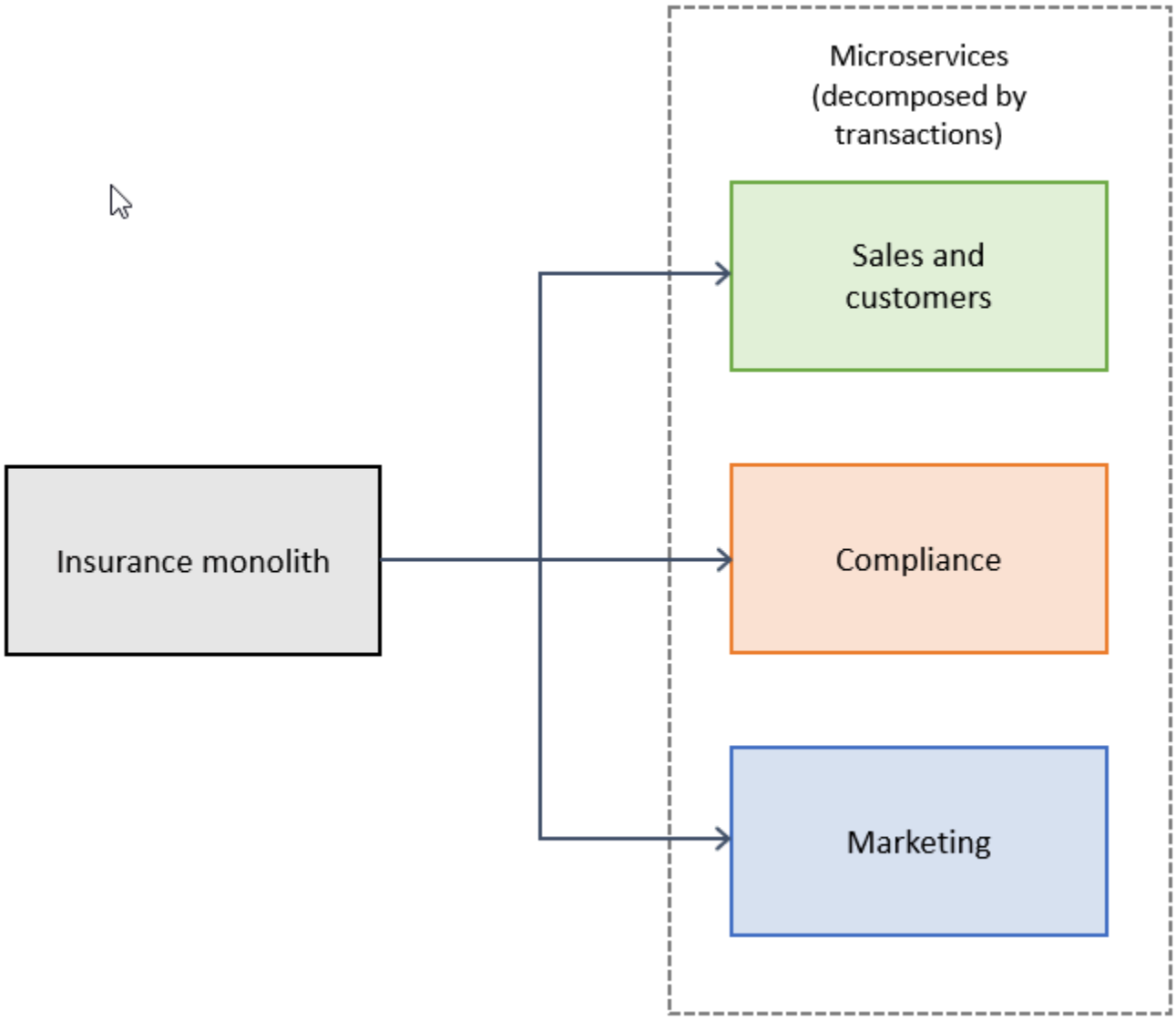
该插图显示，销售和营销服务被分解为较小的微服务。采购和索赔模型是销售的重要业务差异化因素，它们分为两个独立的微服务。通过使用活动、分析和报告等辅助业务功能来分解营销。

按交易分解

在分布式系统中，应用程序通常必须调用多个微服务才能完成一项业务事务。为避免延迟问题或两阶段提交问题，您可以根据事务对微服务进行分组。如果您认为响应时间很重要，并且不同的模块在打包后不会形成整体结构，则这种模式是合适的。下表说明了使用此模式的优势和劣势。

优点	缺点
• 响应时间更短。 • 您无需担心数据一致性。 • 提高可用性。	• 可以将多个模块打包在一起，这可以形成一个单体。 • 多个功能是在单个微服务中实现的，而不是在单独的微服务中实现的，成本和复杂性因此而增加。 • Transaction-oriented 如果业务域的数量和它们之间的依赖关系很高，则微服务可以增长。 • 对于同一个业务领域，可能会同时部署不一致的版本。

在下图中，保险单体根据交易分解为多个微服务。



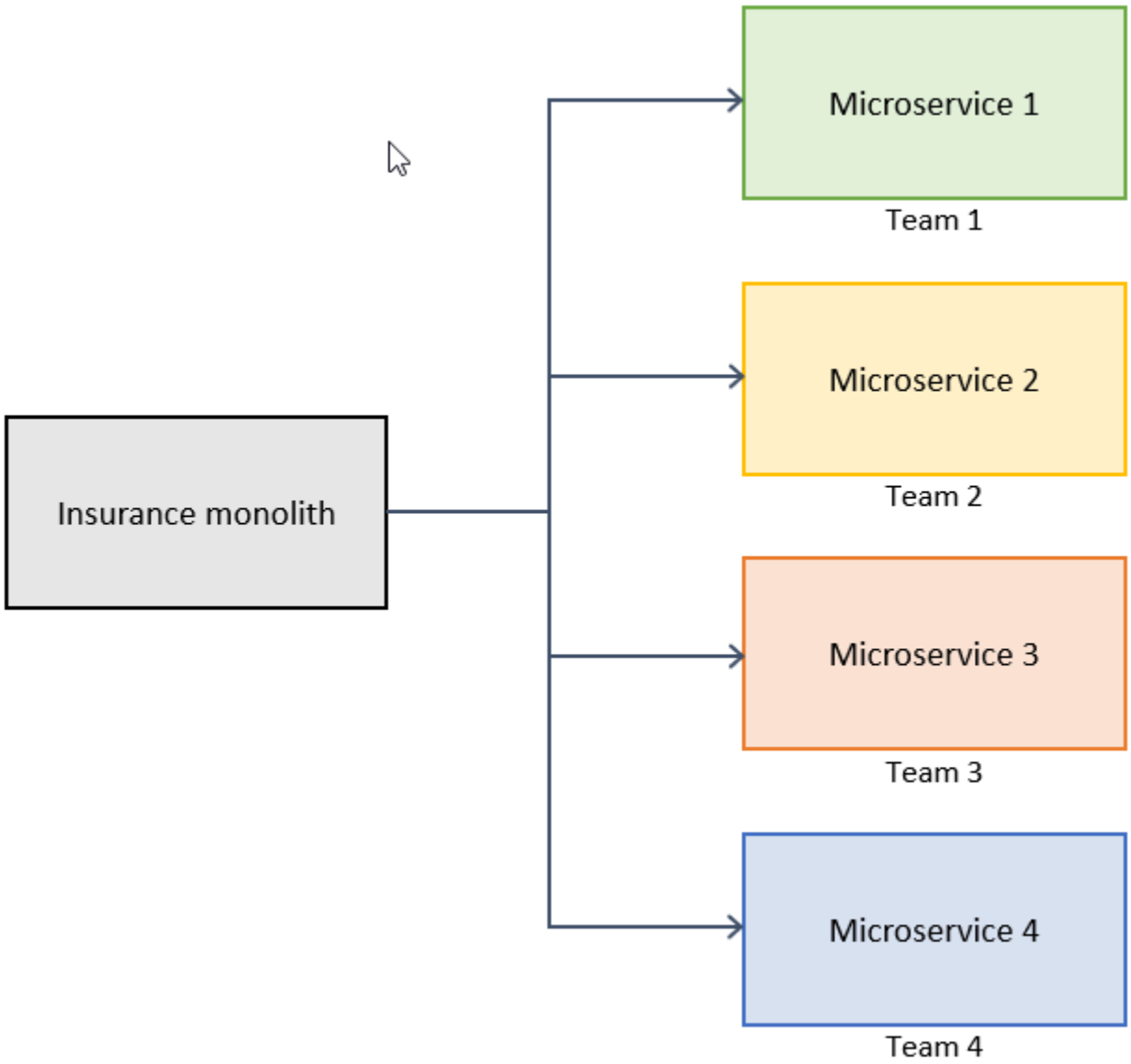
在保险系统中，索赔申请通常在提交后标记给客户。这意味着，如果没有客户微服务，理赔服务就不可能存在。销售和客户打包在一个微服务包中，业务交易需要与两者进行协调。

按团队提供的服务模式

每个团队的服务模式不是按业务能力或服务分解单体，而是将其分解为由各个团队管理的微服务。每个团队都对一项业务能力负责，并拥有该能力的代码库。该团队独立开发、测试、部署或扩展其服务，主要与其他团队互动以协商 API。我们建议您将每项微服务分配给一个团队。但是，如果团队足够大，则多个子团队可以在同一个团队结构中拥有单独的微服务。下表说明了使用此模式的优势和劣势。

优点	缺点
• 团队独立行动，很少协调。 • 代码库和微服务并非由多个团队共享。 • 团队可以快速创新和迭代产品功能。 • 不同的团队可以使用不同的技术、框架或编程语言。注意：这些应该隐藏在定义明确且稳定的公共 API 后面。	• 根据最终用户的功能或业务能力调整团队可能很困难。 • 要交付更大、更协调的应用程序增量，还需要付出额外的努力，尤其是在团队之间存在循环依赖关系的情况下。

下图显示了如何将单体拆分为由各个团队管理、维护和交付的微服务。



Strangler fig 模式

本指南中到目前为止讨论的设计模式适用于在全新环境中开发的项目的分解应用程序。那么涉及大型单体式应用程序、在遗留系统中开发的项目呢？ 将以前的设计模式应用于它们会很困难，因为在积极使用它们的同时将它们分解成较小的部分是一项艰巨的任务。

Strangler fig 模式是一种流行的设计模式，由 Martin Fowler 推出，他的灵感来自于某种类型的无花果，它在树的上部分支上自己播种。现有的树最初成为新无花果的支撑结构。然后无花果将其根送到地面，逐渐包裹原树，只留下新的，自我支撑的无花果。

这种模式通常用于通过将特定功能替换为新服务来逐步将单体式应用程序转换为微服务。目标是让传统版本和新的现代化版本共存。新系统最初由现有系统支持并覆盖现有系统。这种支持使新系统有时间进行扩展，并有可能完全取代旧系统。

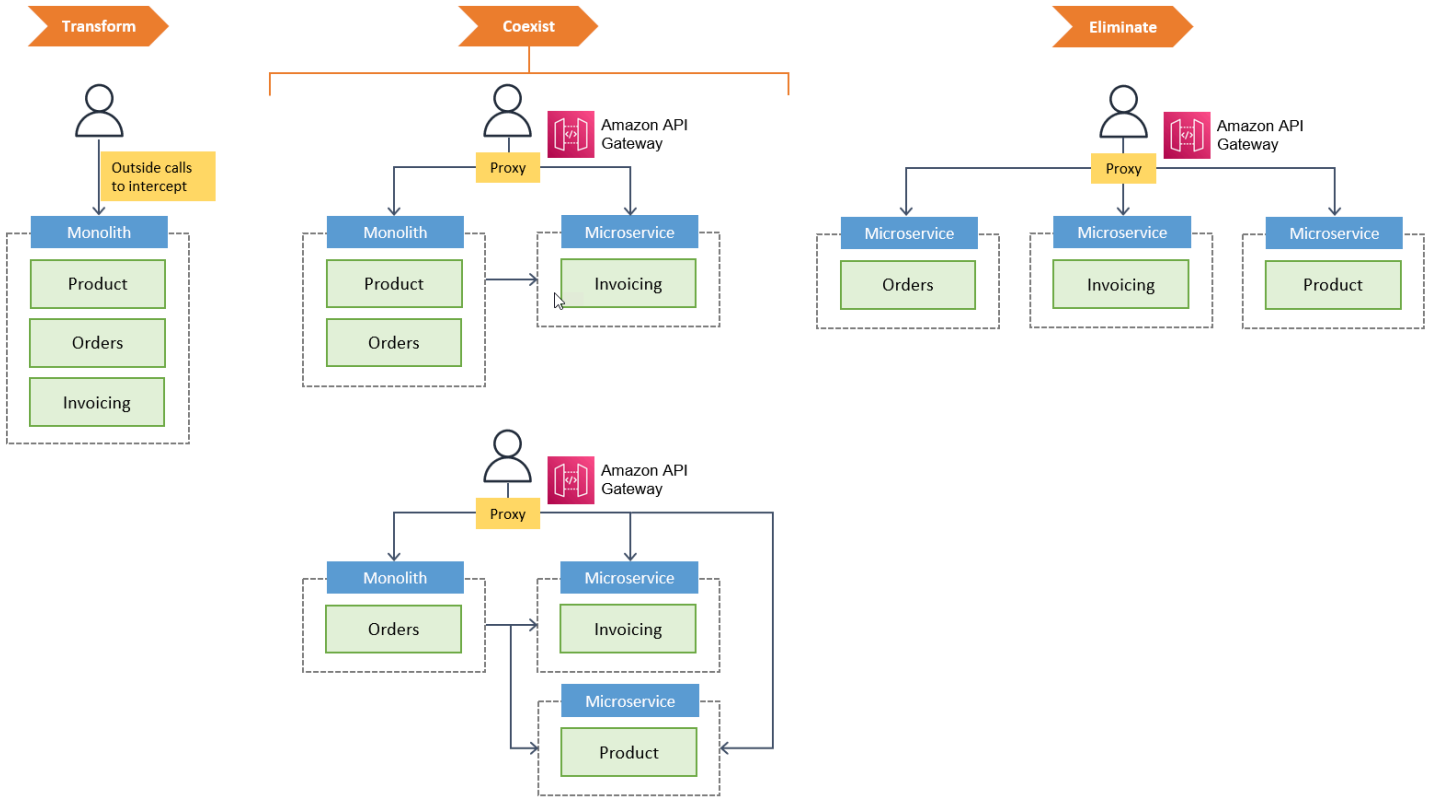
通过实现 strangler fig 模式从单体应用程序过渡到微服务的过程包括三个步骤：转换、共存和消除：

- 转换：通过移植或与旧版应用程序并行重写来识别和创建现代化组件。
- 共存：保留单体式应用程序以进行回滚。通过在单体外围加入 HTTP 代理（例如 Amazon API Gateway）来拦截外部系统调用，并将流量重定向到现代化版本。这可以帮助您逐步实现功能。
- 消除：当流量从传统单体重定向到现代化服务时，旧功能将在单体结构中停用。

下表说明了使用 strangler fig 模式的优势和劣势。

优点	缺点
• 允许从一项服务顺利迁移到一项或多项替代服务。 • 在重构到更新版本时保持旧服务正常运行。 • 提供在重构旧服务的同时添加新服务和功能的能力。 • 该模式可用于 API 的版本控制。 • 该模式可用于未升级或不会升级的解决方案的传统交互。	• 不适用于复杂度低、体积小的小型系统。 • 不能在无法拦截和路由对后端系统请求的系统中使用。 • 如果设计不当，代理层或外观层可能会成为单点故障或性能瓶颈。 • 需要为每个重构的服务制定回滚计划，以便在出现问题时快速安全地恢复到以前的工作方式。

下图显示了如何通过将 strangler fig 模式应用于应用程序架构来将单体拆分为微服务。两个系统并行运行，但是您将开始将功能移到单体代码库之外，并使用新功能对其进行增强。这些新功能使您有机会以最适合自己需求的方式架构微服务。在全部被微服务取代之前，您将继续从单体上剥离这些功能。此时，您可以取消单体应用程序。这里要注意的关键点是，单体和微服务将在一段时间内共同存在。



按抽象模式分支

当您能在单体周边拦截调用时，Strangler fig 模式效果很好。但是，如果您想对存在于遗留应用程序堆栈中更深层次且具有上游依赖关系的组件进行现代化改造，我们建议按抽象模式分支。这种模式使您能够对现有代码库进行更改，以允许现代化版本与旧版本安全共存，而不会造成中断。

要成功使用抽象分支模式，请按照以下过程操作：

1. 识别具有上游依赖关系的单体组件。
2. 创建一个抽象层，用于表示待现代化的代码与其客户端之间的交互。
3. 抽象到位后，将现有客户端更改为使用新的抽象。
4. 在单体结构之外使用经过重新设计的功能，创建新的抽象实现。
5. 准备就绪后，将抽象切换到新的实现。
6. 当新的实现为用户提供了所有必需功能并且不再使用单体架构时，请清理旧的实现。

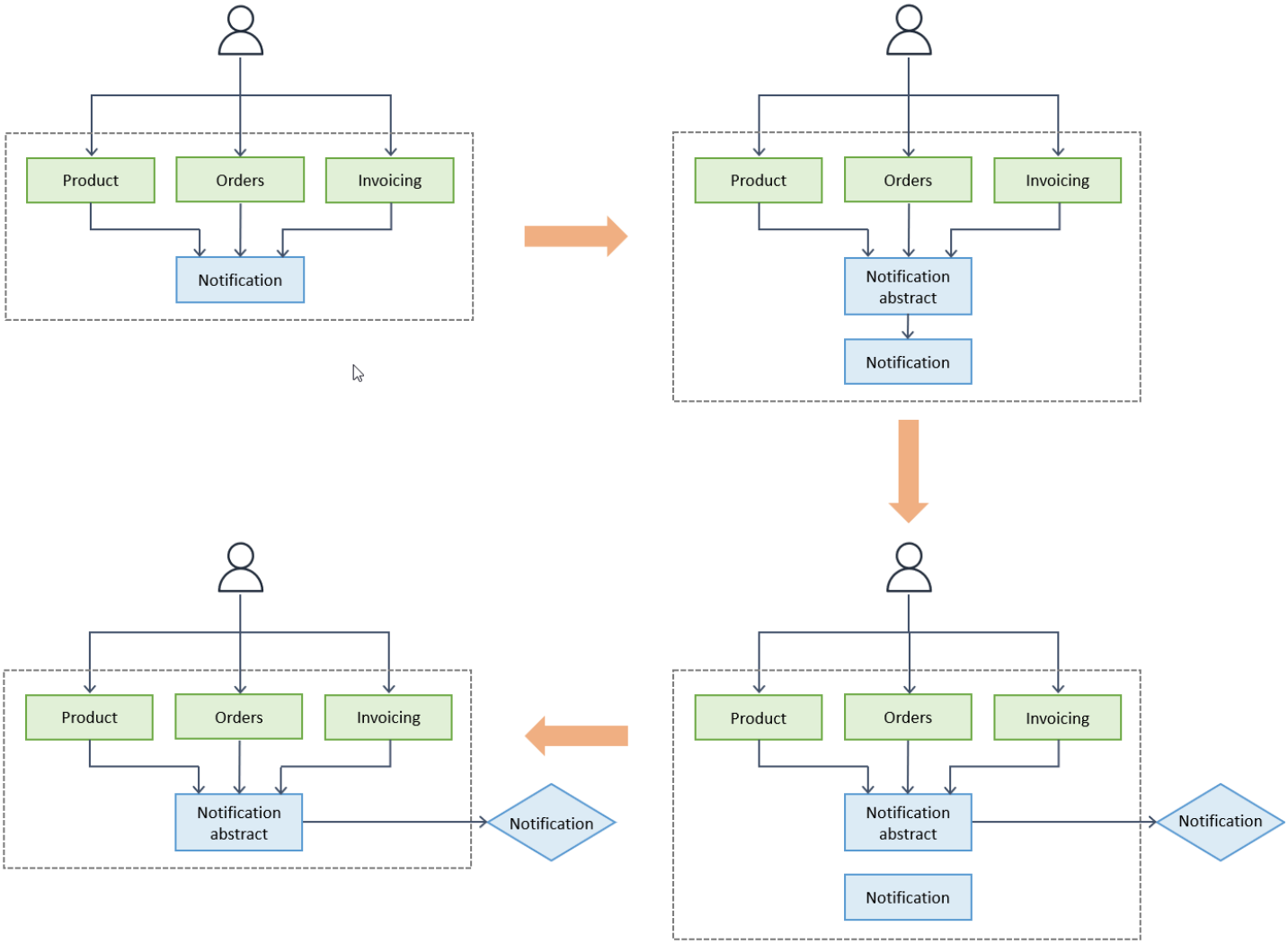
抽象分支模式经常与功能切换混淆，后者还允许您逐步更改系统。不同之处在于，功能切换旨在允许开发新功能，并在系统运行时保持这些功能对用户不可见。因此，在部署时或运行时系统使用功能切换来

选择特定功能或一组功能在应用程序中是否可见。抽象分支是一种开发技术，可以与功能切换结合使用，以便在新旧抽象之间切换。

下表说明了按抽象模式使用分支的优势和劣势。

优点	缺点
• 允许在出现任何问题时进行可逆的增量更改（向后兼容）。 • 当您无法在单体边缘拦截对单体的调用时，可以提取单体深处的功能。 • 允许多个抽象共存于软件系统中。 • 通过使用中间验证步骤调用新旧功能，提供了一种实现回退机制的简便方法。 • 支持持续交付，因为在整个重组阶段，您的代码始终处于工作状态。	• 如果涉及数据一致性，则不适合。 • 需要对现有系统进行更改。 • 可能会增加开发过程的开销，尤其是在代码库结构不佳的情况下。（在许多情况下，上行空间值得付出额外的努力，而重组规模越大，考虑按抽象模式使用分支就越重要。）

下图显示了保险单体中通知组件的分支抽象模式。它首先为通知功能创建摘要或接口。以较小的增量更改现有客户端，以使用新的抽象。这可能需要在代码库中搜索与通知组件相关的 API 的调用。您可以将通知功能的新实现作为单体架构之外的微服务进行创建，并将其托管在现代化的架构中。在您的单体中，您新创建的抽象接口充当代理并调用新的实现。您可以将通知功能移植到新的实现中，在经过全面测试并准备就绪之前，新实现将保持非活动状态。当新的实现准备就绪时，您可以切换抽象来使用它。您需要使用一种可以轻松切换的切换机制（例如功能切换），这样在遇到任何问题时可以轻松切换回旧功能。当新的实现开始向用户提供所有通知功能并且不再使用单体时，您可以清理较旧的实现并删除可能已实现的任何切换功能标志


Product
Orders
Invoicing
Notification
Product
Orders
Invoicing
Notification abstract
Notification
Product
Orders
Invoicing
Notification abstract
Notification
Product
Orders
Invoicing
Notification abstract
Notification
Notification

分解单体常见问题解答

本节提供了有关分解单体常见问题的答案。

您能用多种模式来分解一个单体吗？

是的，您可以使用多种模式来分解一个单体。最常见的方法是使用按业务能力分解模式分解单体，然后使用按子域分解模式对其进行更多分解。

将巨石分解为微服务会如何影响流程？ DevOps

由于在对应用程序进行更改后不必重新部署所有内容，因此必须对添加到部署过程中的新建微服务拥有支持和所有权。这可能会使 DevOps 过程变得更加复杂。

资源

相关指南

- 实现云端应用程序现代化的策略 AWS
- 分阶段实现云端应用程序现代化的方法 AWS
- 评估 AWS 云端应用程序的现代化准备情况
- 使用 AWS 无服务器服务集成微服务

其他 资源

- 使用 Cop AWS ilot、Amazon ECS、Docker 和，将单体应用程序分解为微服务 AWS Fargate

文档历史记录

下表介绍了本指南的一些重要更改。如果您希望收到有关未来更新的通知，可以订阅 RSS 源。

变更	说明	日期
添加了新模式	添加了有关 strangler fig 和抽象分支模式的信息。	2023 年 4 月 6 日
初次发布	—	2021 年 1 月 11 日

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。