服务化解耦建议

王博

内容

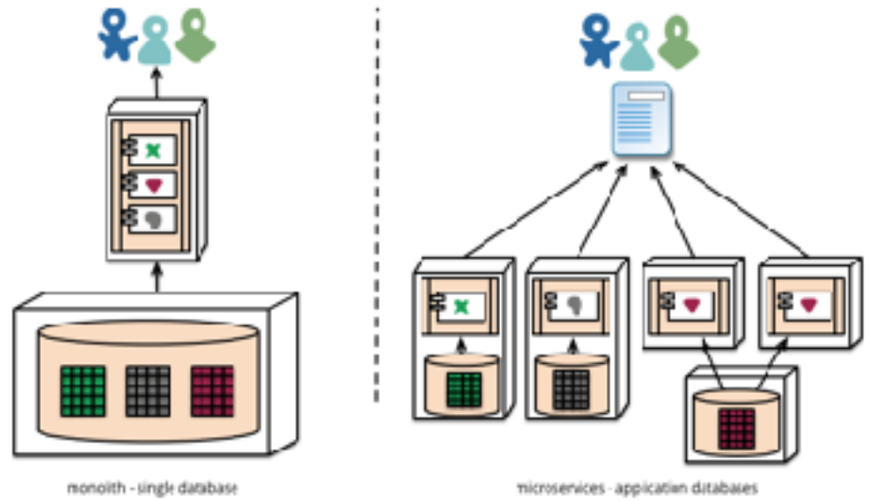
- 服务化解耦原则
- 服务化解耦建议
- 工程实践
- 架构问题

服务解耦原则

服务解耦的目标

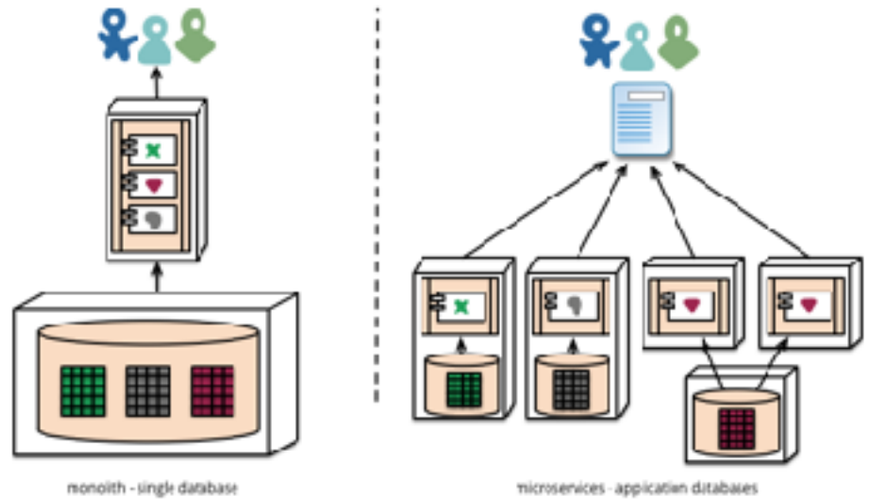
- 服务化带来软件组件生命周期的独立，针对以下目标：
  - 变化的频率不同：将频繁变动的和非频繁变动的解耦
  - 变化的原因不同：将由于不同原因变化的部分解耦
  - 性能的要求不同：将性能关键和非性能关键的部分解耦
  - 资源的占用不同：将资源占用粒度差异化的部分解耦

➤ 决策的依据：
- 对领域的深入了解和分析
- 需要数据的支撑

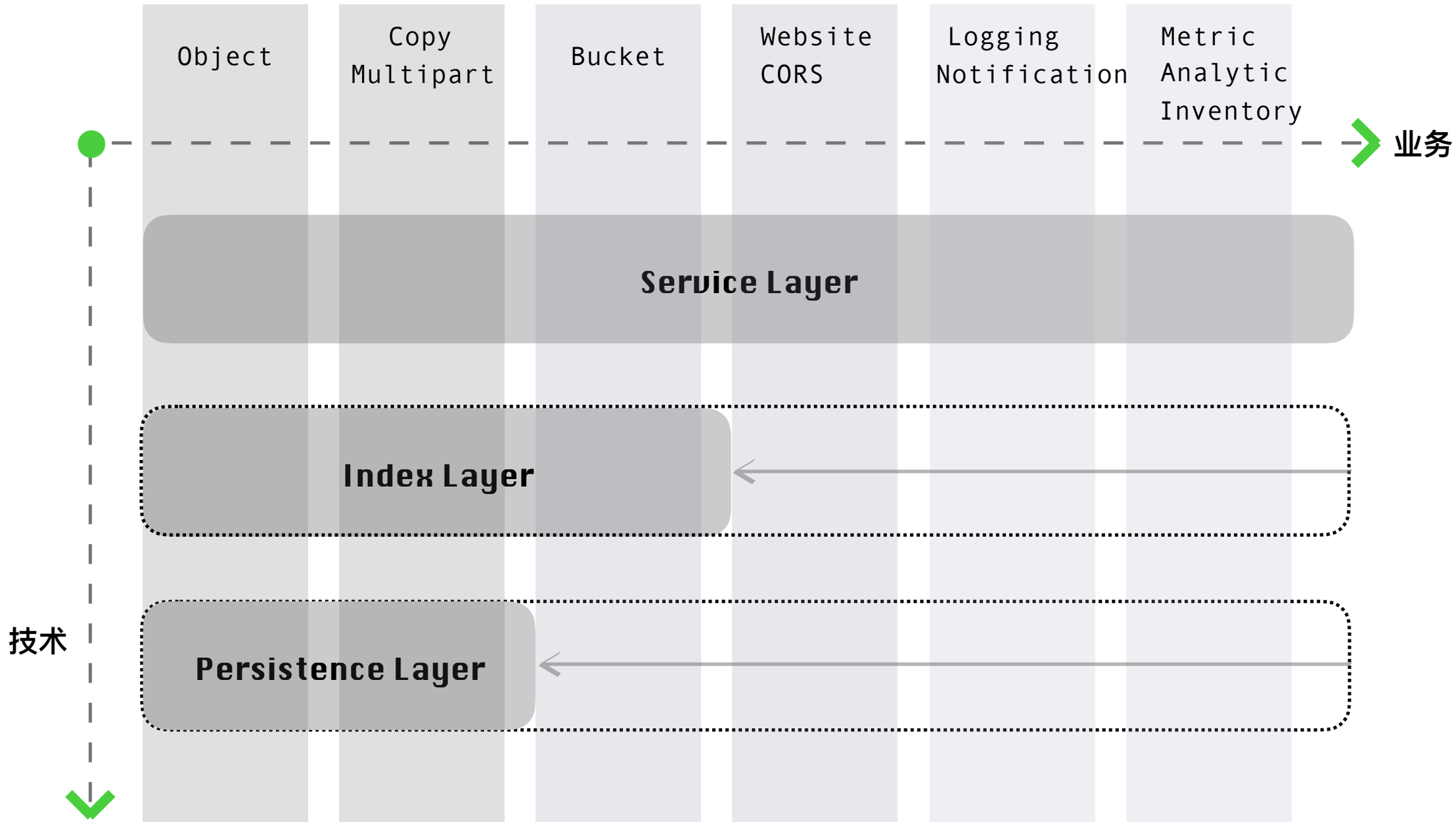


服务解耦的原则

- 关键是避免分布式带来的复杂性：
  - 需要分析数据一致性边界，沿着最终一致性的边界划分
  - 需要分析关键的性能路径，沿着不同性能路径的边界划分
  - 需要分析依赖关系，沿着稳定的接口和松耦合边界划分

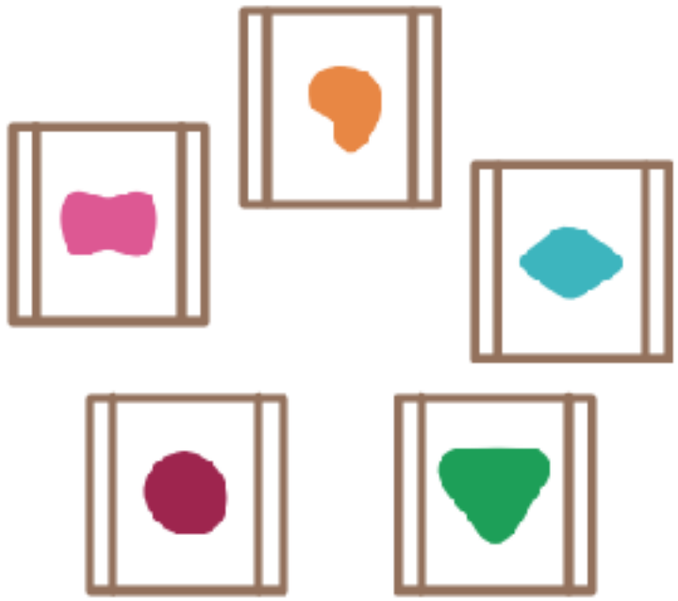


变化方向


Object
Copy
Multipart
Bucket
Website
CORS
Logging
Notification
Metric
Analytic
Inventory
业务
Service Layer
Index Layer
Persistence Layer
技术

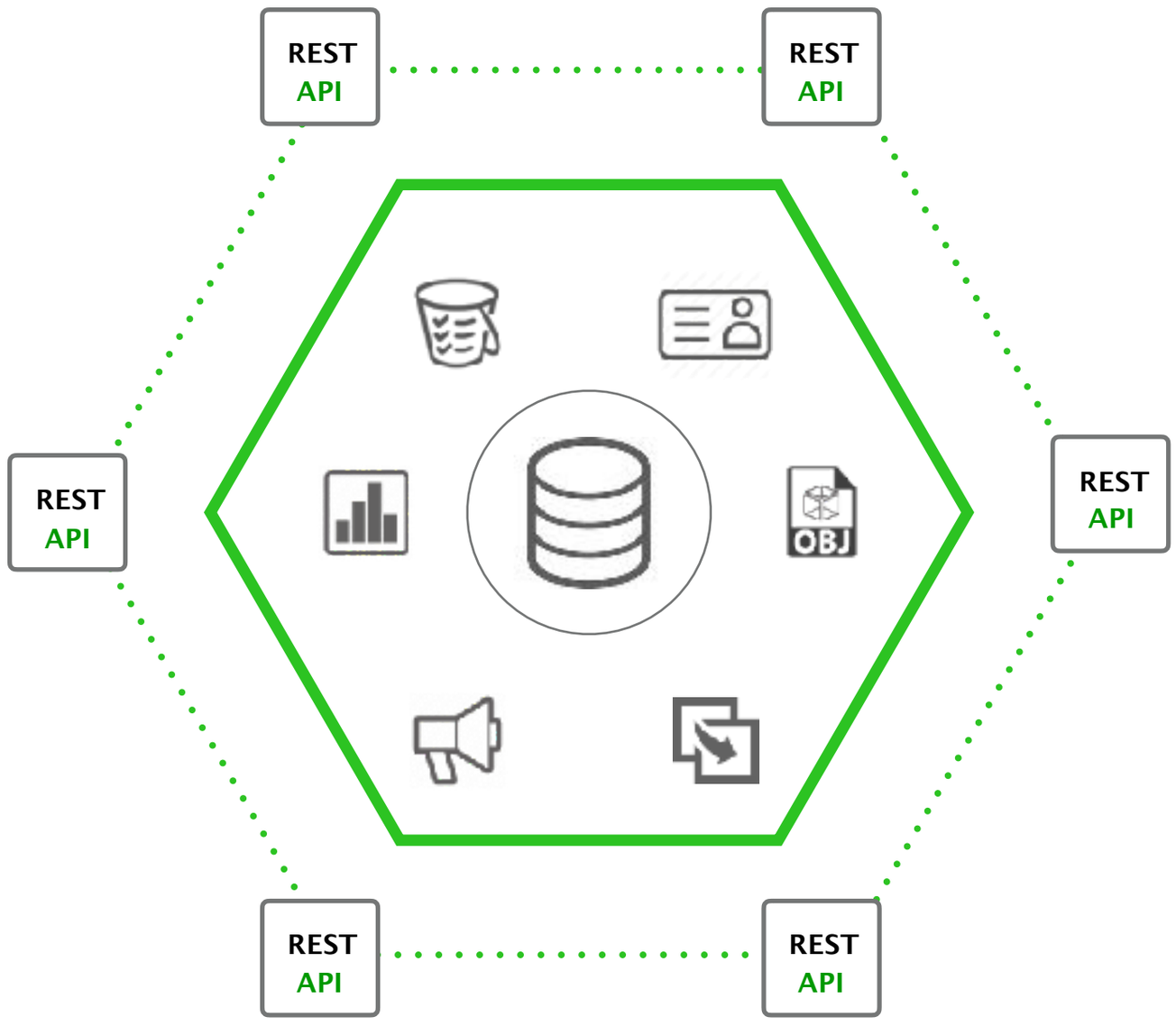
Microservices

common characteristics of this architectural style

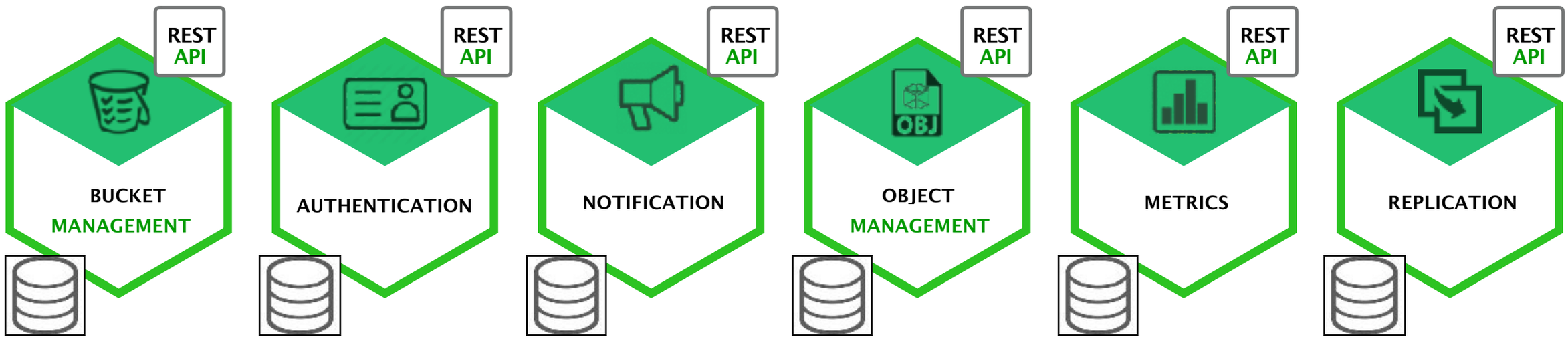


by James Lewis and Martin Fowler

現状



业务解耦



- 业务解耦的关键在于：
  - 数据和合理拆分
  - *API*的合理拆分

数据拆分

BUCKET MANAGEMENT

OBJECT MANAGEMENT

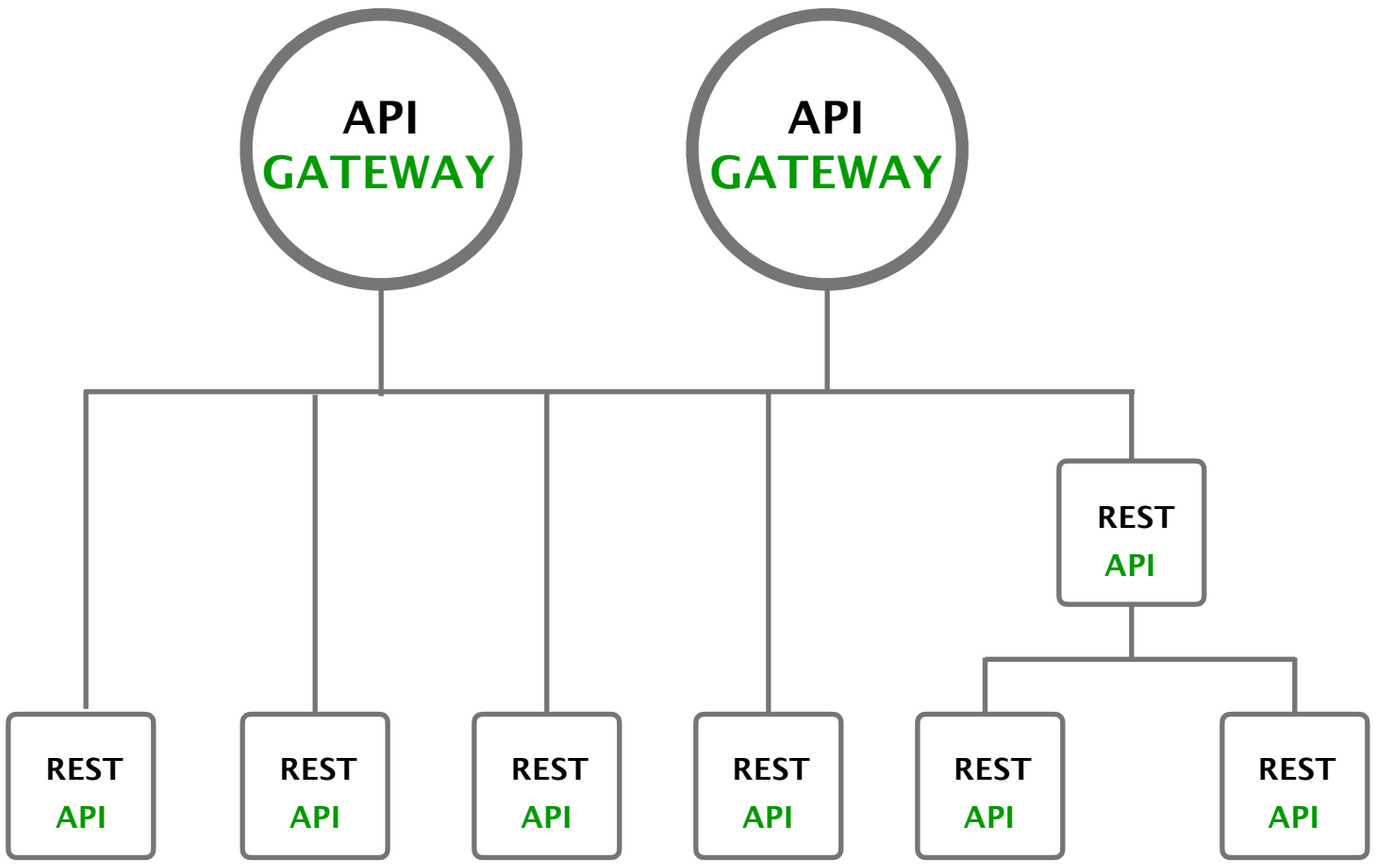
LIFECYCLE	CRR	AUTHENTICATION	WEBSITE	CORS	LOGGING	METRICS	ANALYTIC

Bucket : {name, versioning, lifecycle, CRR, policy, ACL, website, CORS, logging, metrics, analytic,…}

Object : {name, versionId, PLOG list, expire, CRR status, ACL,…}

- 数据的正交划分
- 关注一致性需求

API设计

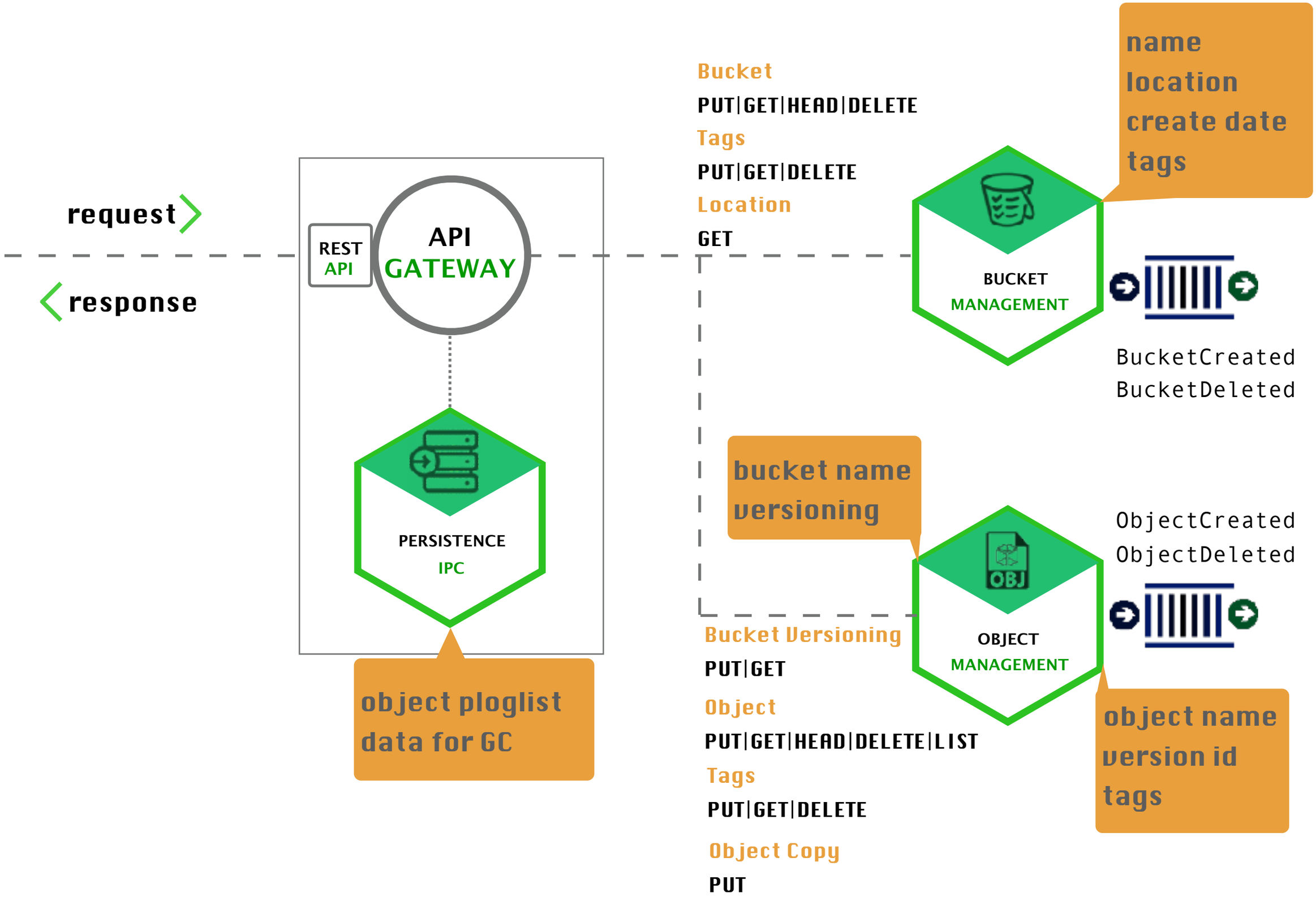


- *One size does not fit all*
- *Interface isolate principle*
- *Use facade pattern to convenient different users*
- *REST is not the only choice*
- *SYNC vs ASYNC*
- *P2P vs PUB/SUB*
- *Postel principle*
- *Idempotent design*
- *Semantic version*
- *…*

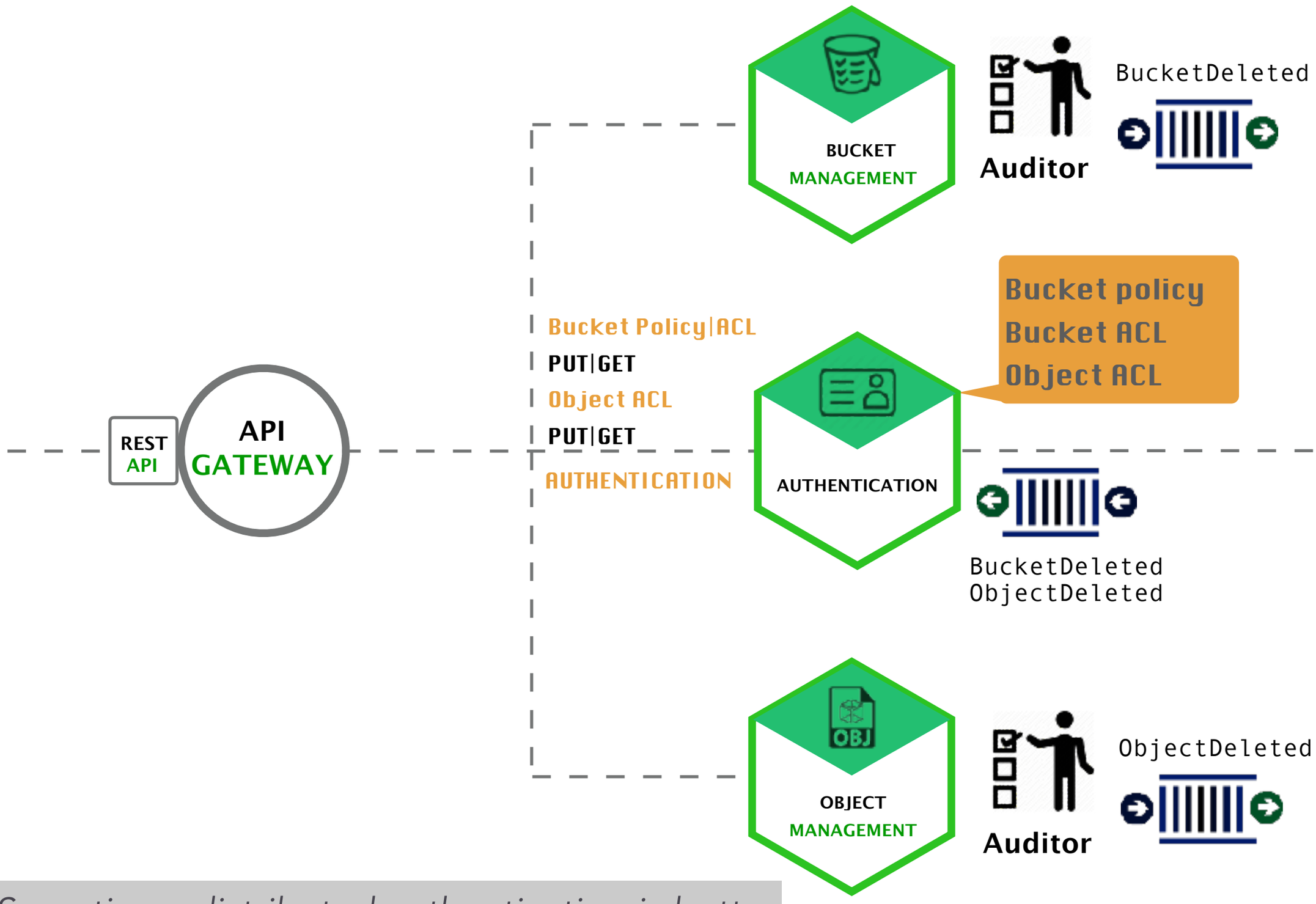
AWS S3 API

- Operations on Buckets
  - DELETE Bucket
  - DELETE Bucket analytics
  - DELETE Bucket cors
  - DELETE Bucket inventory
  - DELETE Bucket lifecycle
  - DELETE Bucket metrics
  - DELETE Bucket policy
  - DELETE Bucket replication
  - DELETE Bucket tagging
  - DELETE Bucket website
  - GET Bucket (List Objects) Version 2
  - GET Bucket accelerate
  - GET Bucket acl
  - GET Bucket analytics
  - GET Bucket cors
  - GET Bucket inventory
  - GET Bucket lifecycle
  - GET Bucket location
  - GET Bucket logging
  - GET Bucket metrics
  - GET Bucket notification
  - GET Bucket Object versions
  - GET Bucket policy
  - GET Bucket replication
  - GET Bucket requestPayment
  - GET Bucket tagging
  - GET Bucket versioning
  - GET Bucket website
- HEAD Bucket
- List Bucket Analytics Configurations
- List Bucket Inventory Configurations
- List Bucket Metrics Configurations
- List Multipart Uploads
- PUT Bucket
- PUT Bucket accelerate
- PUT Bucket acl
- PUT Bucket analytics
- PUT Bucket cors
- PUT Bucket inventory
- PUT Bucket lifecycle
- PUT Bucket logging
- PUT Bucket metrics
- PUT Bucket notification
- PUT Bucket policy
- PUT Bucket replication
- PUT Bucket requestPayment
- PUT Bucket tagging
- PUT Bucket versioning
- PUT Bucket website
- Operations on Objects
  - Delete Multiple Objects
  - DELETE Object
  - DELETE Object tagging
  - GET Object
  - GET Object ACL
  - GET Object tagging
  - GET Object torrent
  - HEAD Object
  - OPTIONS object
  - POST Object
  - POST Object restore
  - PUT Object
  - PUT Object - Copy
  - PUT Object acl
  - PUT Object tagging
  - Abort Multipart Upload
  - Complete Multipart Upload
  - Initiate Multipart Upload
  - List Parts
  - Upload Part
  - Upload Part - Copy

BASE SKETCH

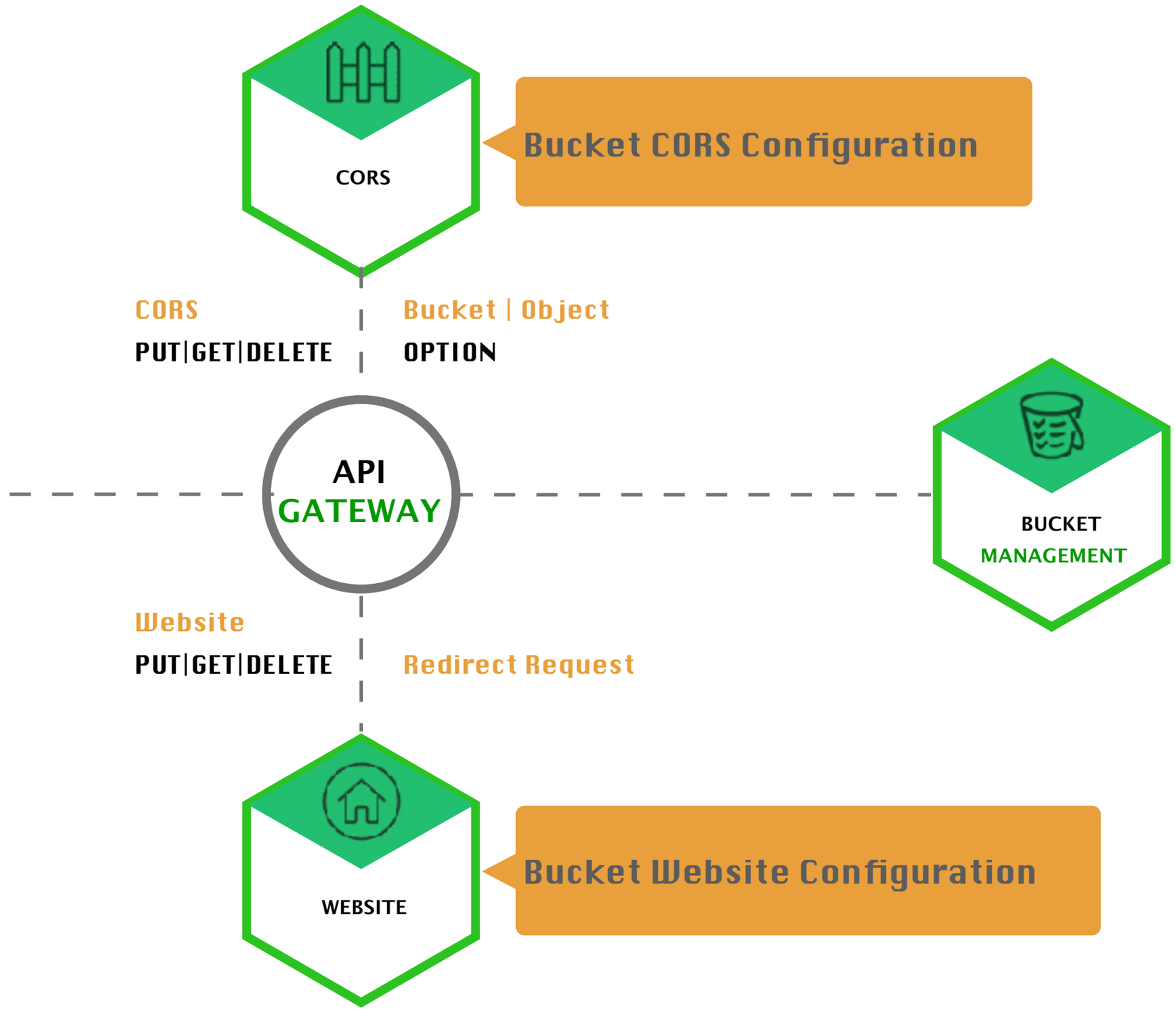

request
response
REST API
API GATEWAY
PERSISTENCE
IPC
object ploglist
data for GC
Bucket
PUT|GET|HEAD|DELETE
Tags
PUT|GET|DELETE
Location
GET
name
location
create date
tags
BUCKET
MANAGEMENT
BucketCreated
BucketDeleted
bucket name
versioning
OBJ
OBJECT
MANAGEMENT
ObjectCreated
ObjectDeleted
Bucket Versioning
PUT|GET
Object
PUT|GET|HEAD|DELETE|LIST
Tags
PUT|GET|DELETE
Object Copy
PUT
object name
version id
tags

AUTHENTICATION

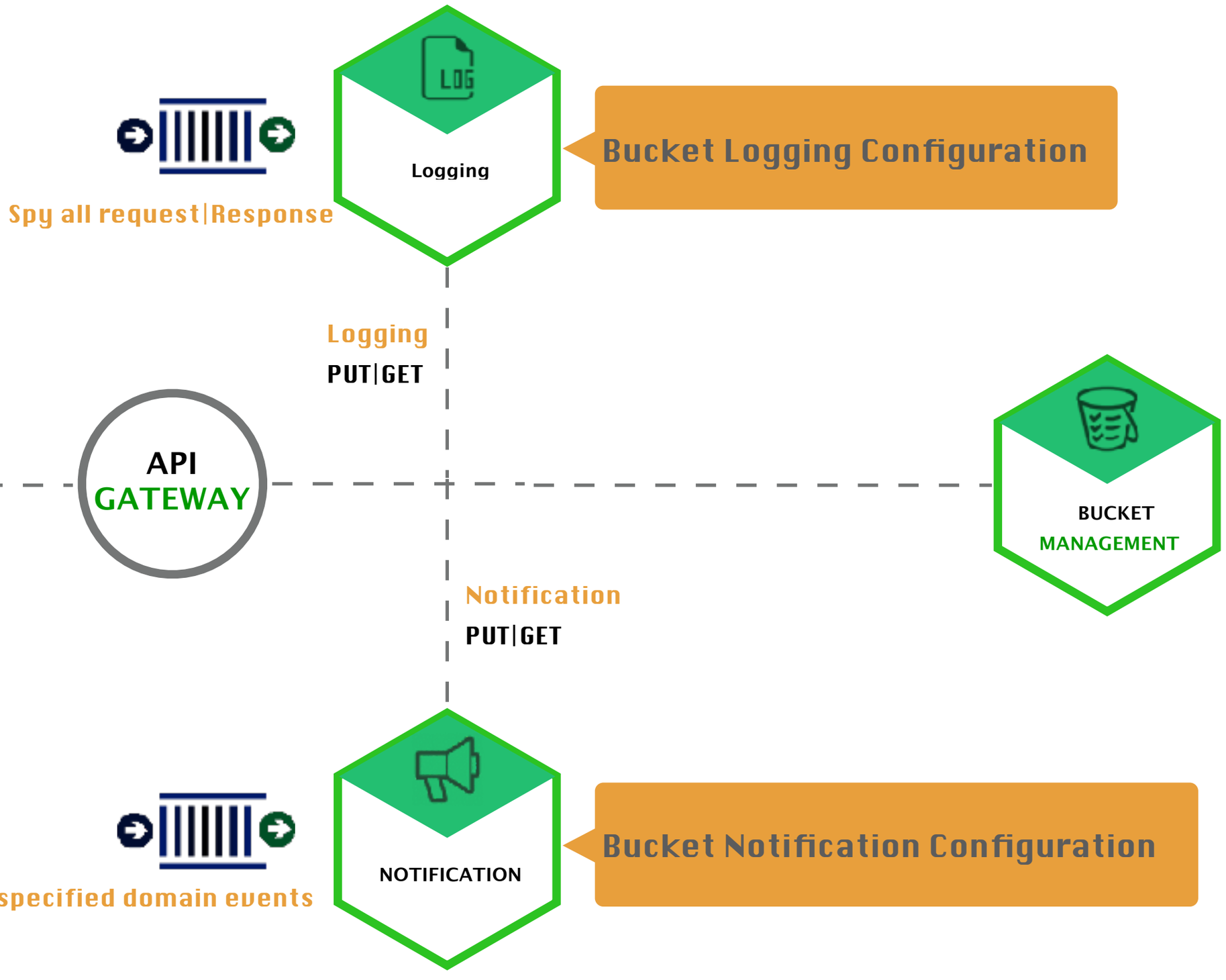


➤ *Sometimes distributed authentication is better*

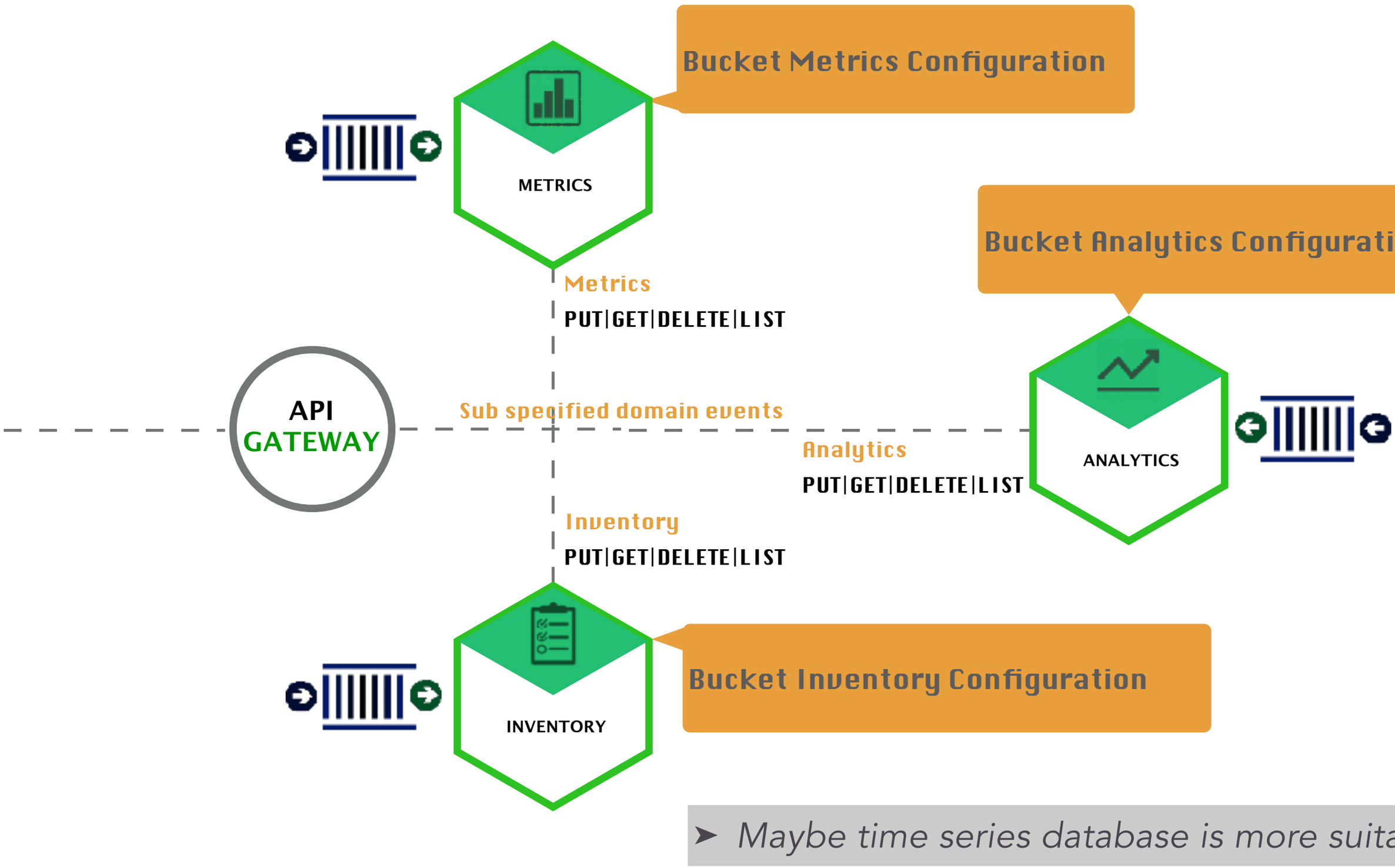
WEBSITE AND CORS



LOGGING AND NOTIFICATION

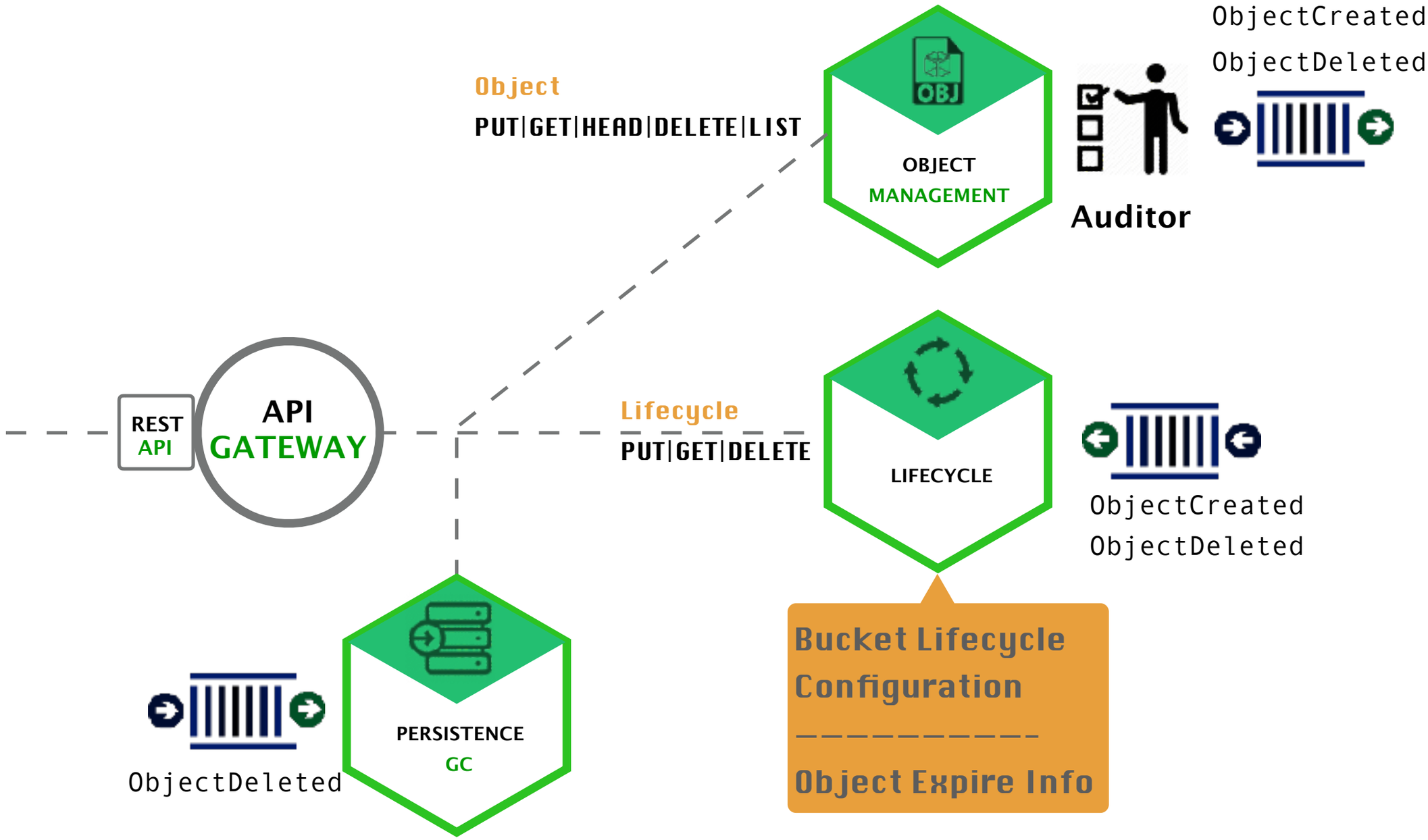


STATISTICS AND ANALYTICS



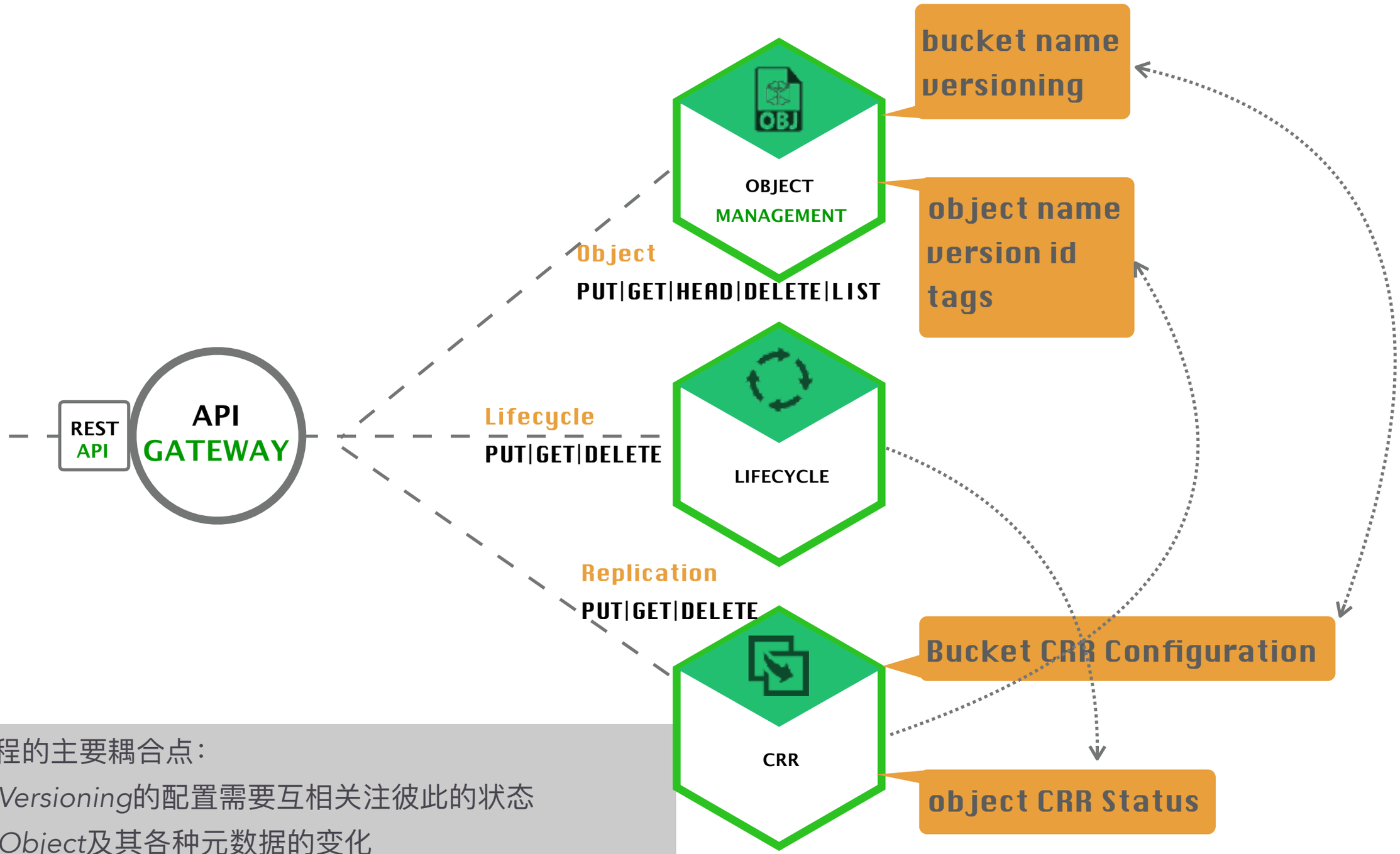
➤ *Maybe time series database is more suitable*

LIFECYCLE



➤ *Composite design: service level reuse*

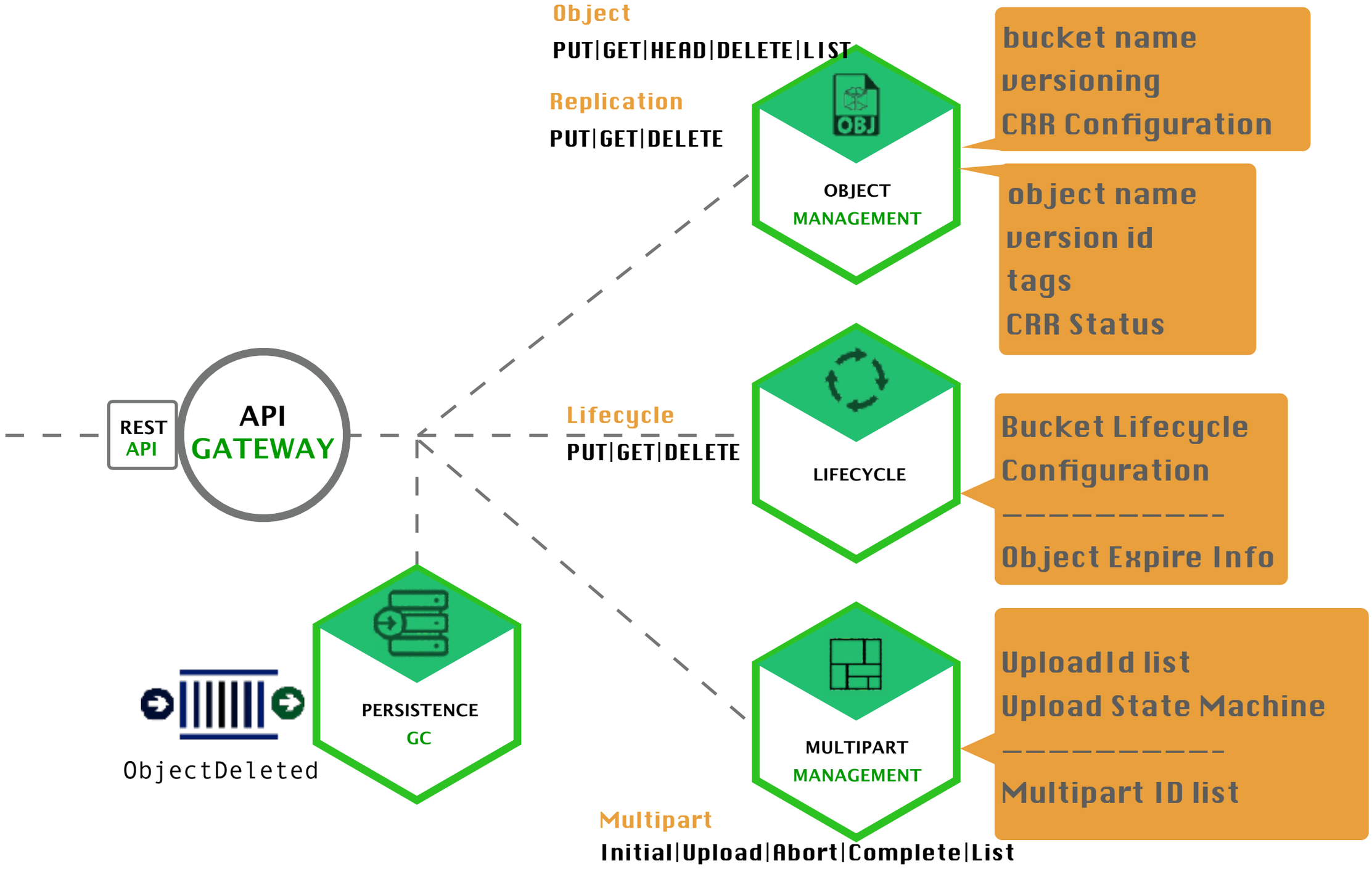
CRR ?



➤ *CRR*和主流程的主要耦合点：

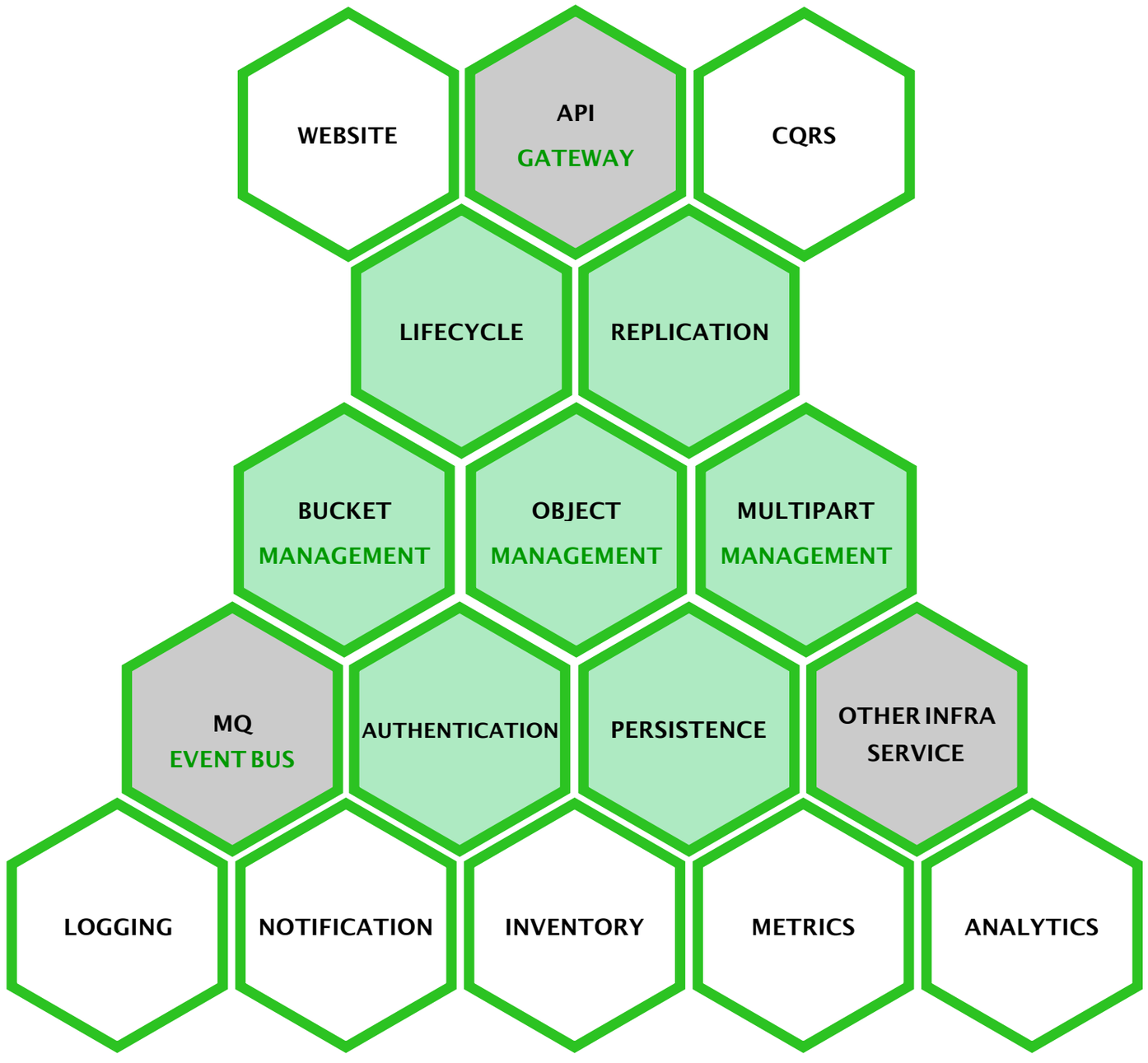
- *CRR*的配置和*Versioning*的配置需要互相关注彼此的状态
- *CRR*需要关注*Object*及其各种元数据的变化
- 需要关注*Object*的删除原因
- *Object*的删除需要关注该对象的当前*CRR*状态
- *CRR*和*Object*之间面对并发上有许多一致性约束需要更仔细的考虑

CRITICAL PATH



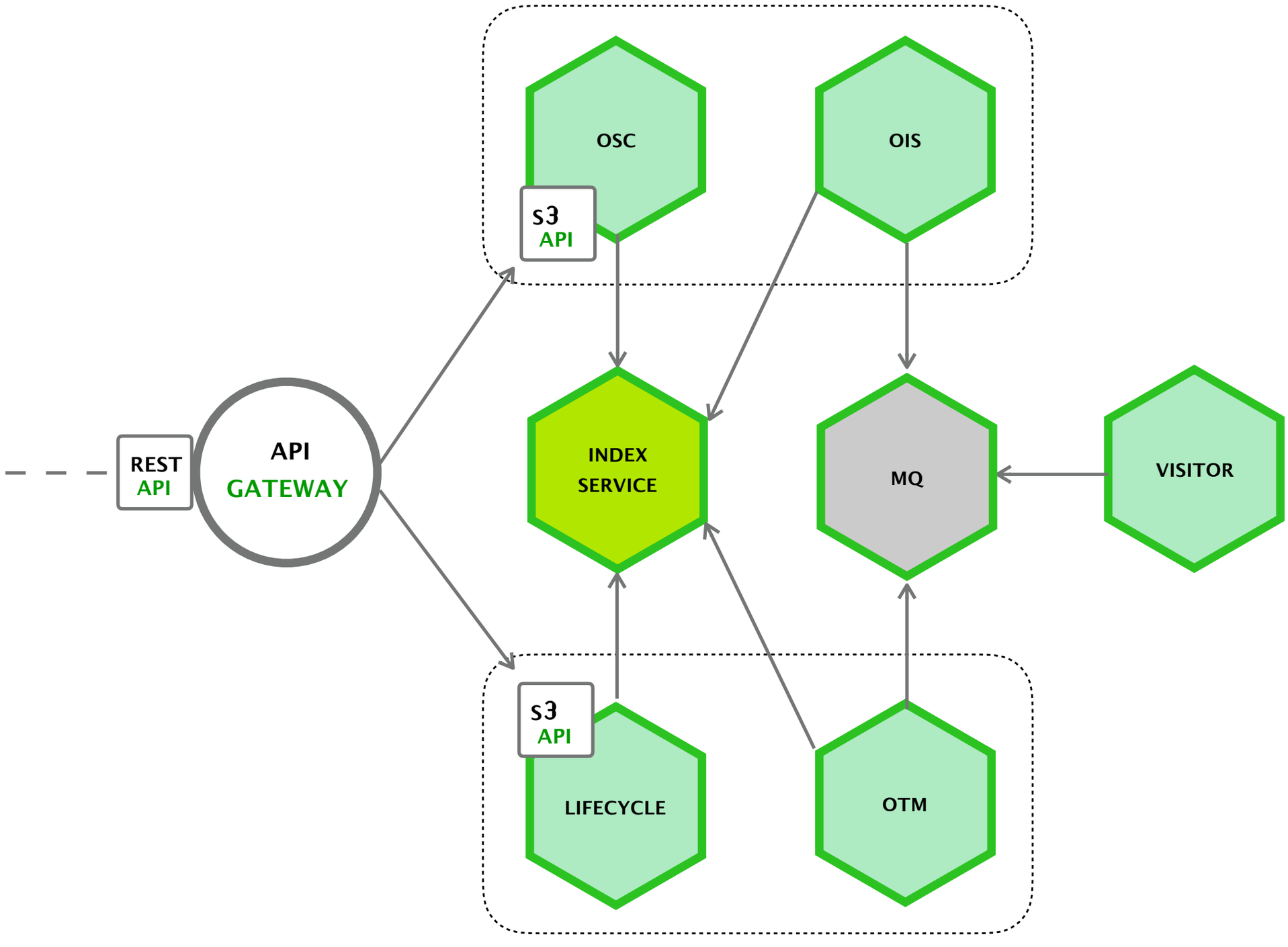
➤ 事实上，数据面的每个*feature*都需要更谨慎细致的分析

演进式拆分

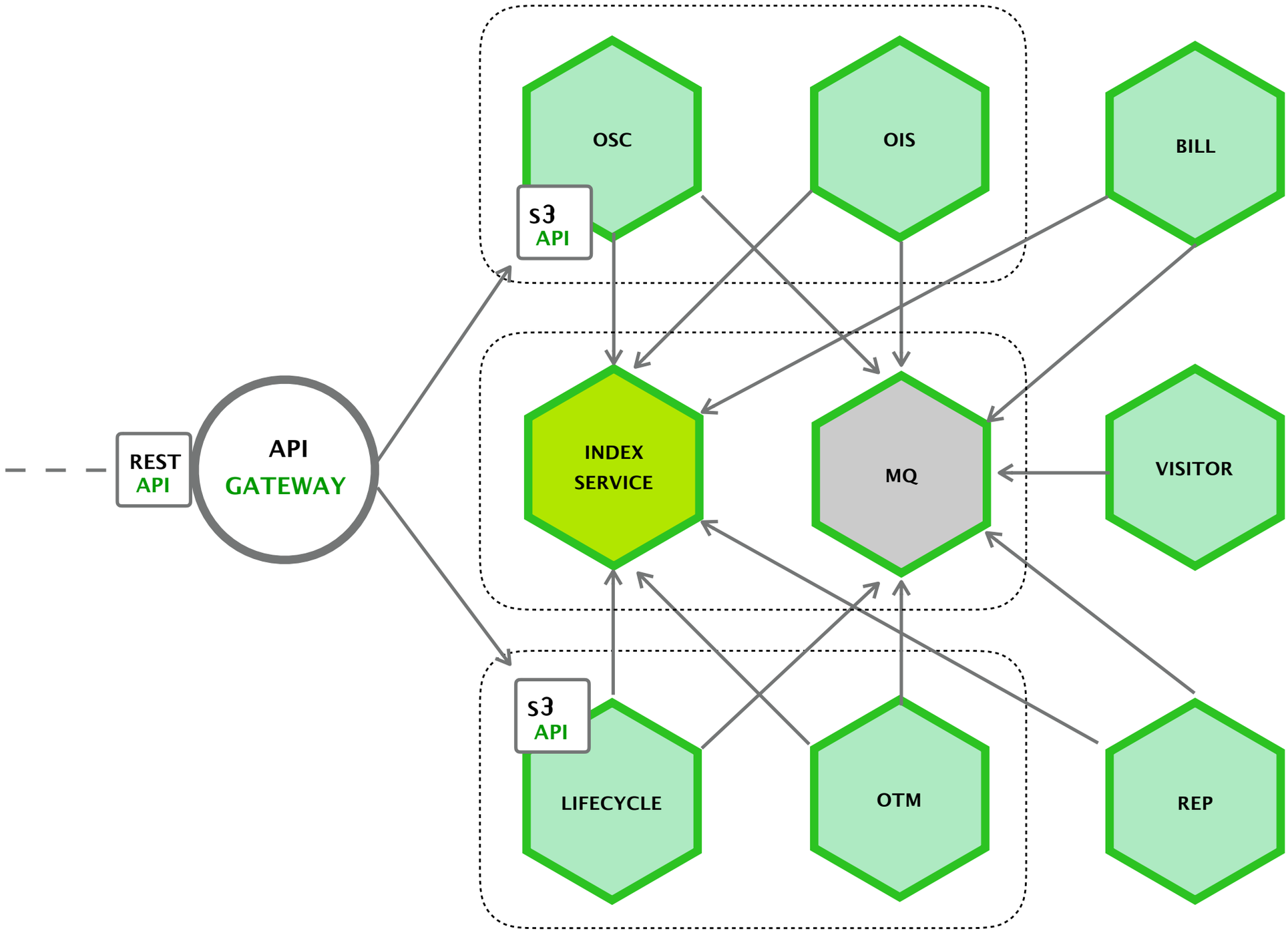


- 从非关键路径的特性开始，以较容易服务开始拆分，逐渐积累经验
- 性能关键的和强一致性的不要拆分
- 耦合较大，不确定的暂时不要拆分
- 优先依赖服务稳定的接口
- 通过演进式拆分逐步完善基础设施
- 新写的功能逐一分析

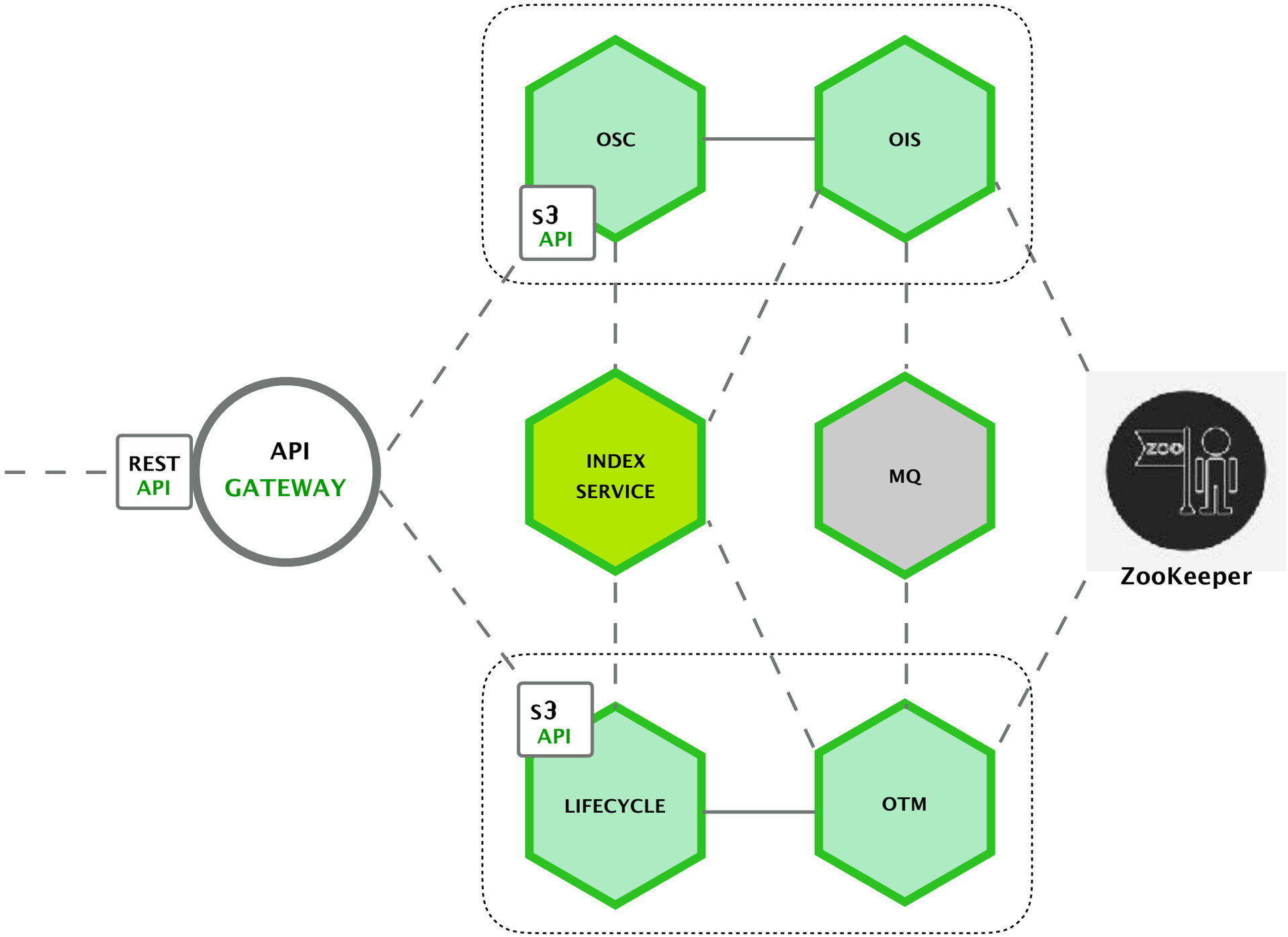
温冷迁移


OSC
S3
API
OIS
REST
API
API
GATEWAY
INDEX
SERVICE
MQ
VISITOR
S3
API
LIFECYCLE
OTM

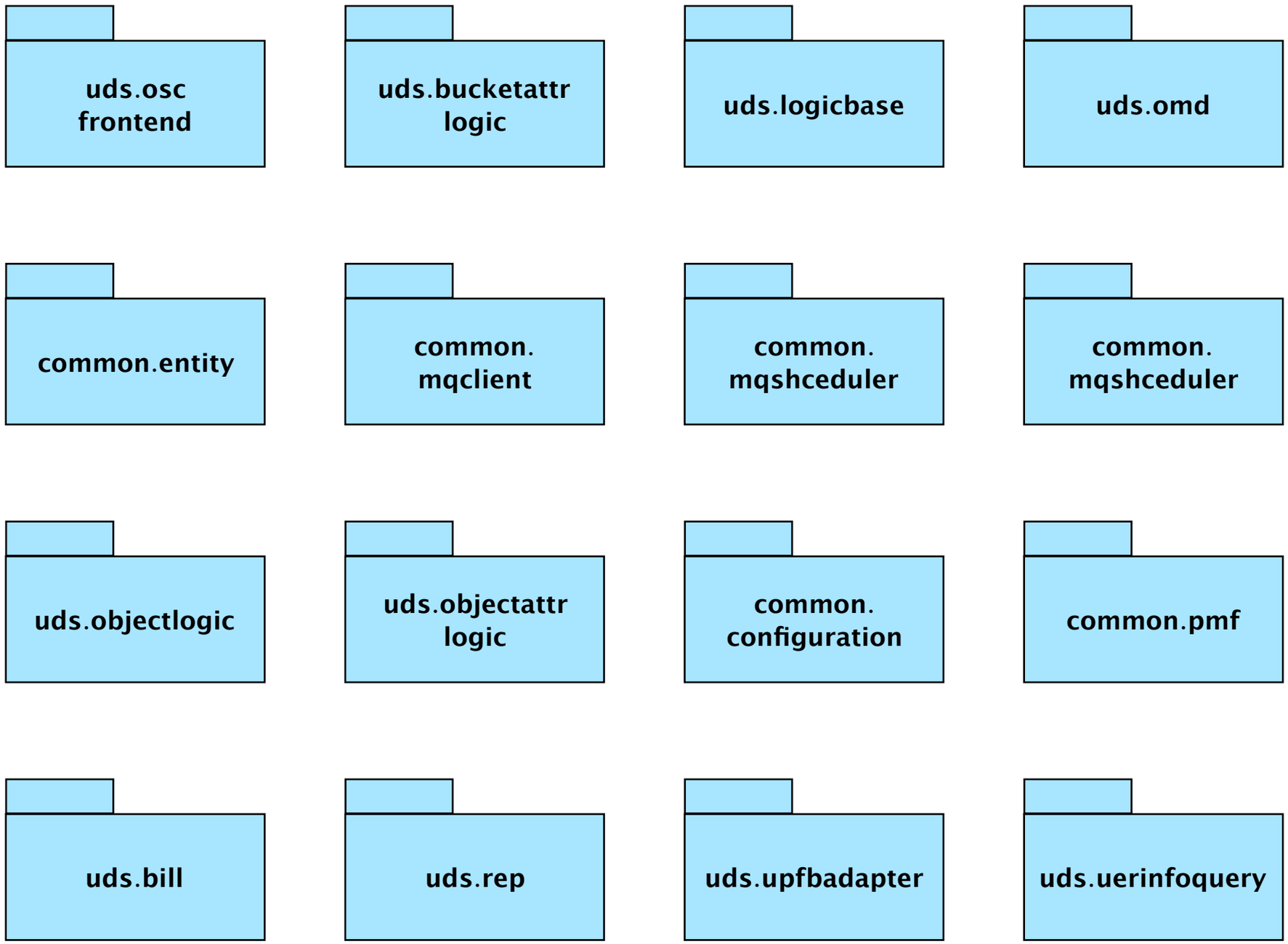
温冷迁移


OSC
S3
API
OIS
BILL
REST
API
API
GATEWAY
INDEX
SERVICE
MQ
VISITOR
S3
API
LIFECYCLE
OTM
REP

温冷迁移

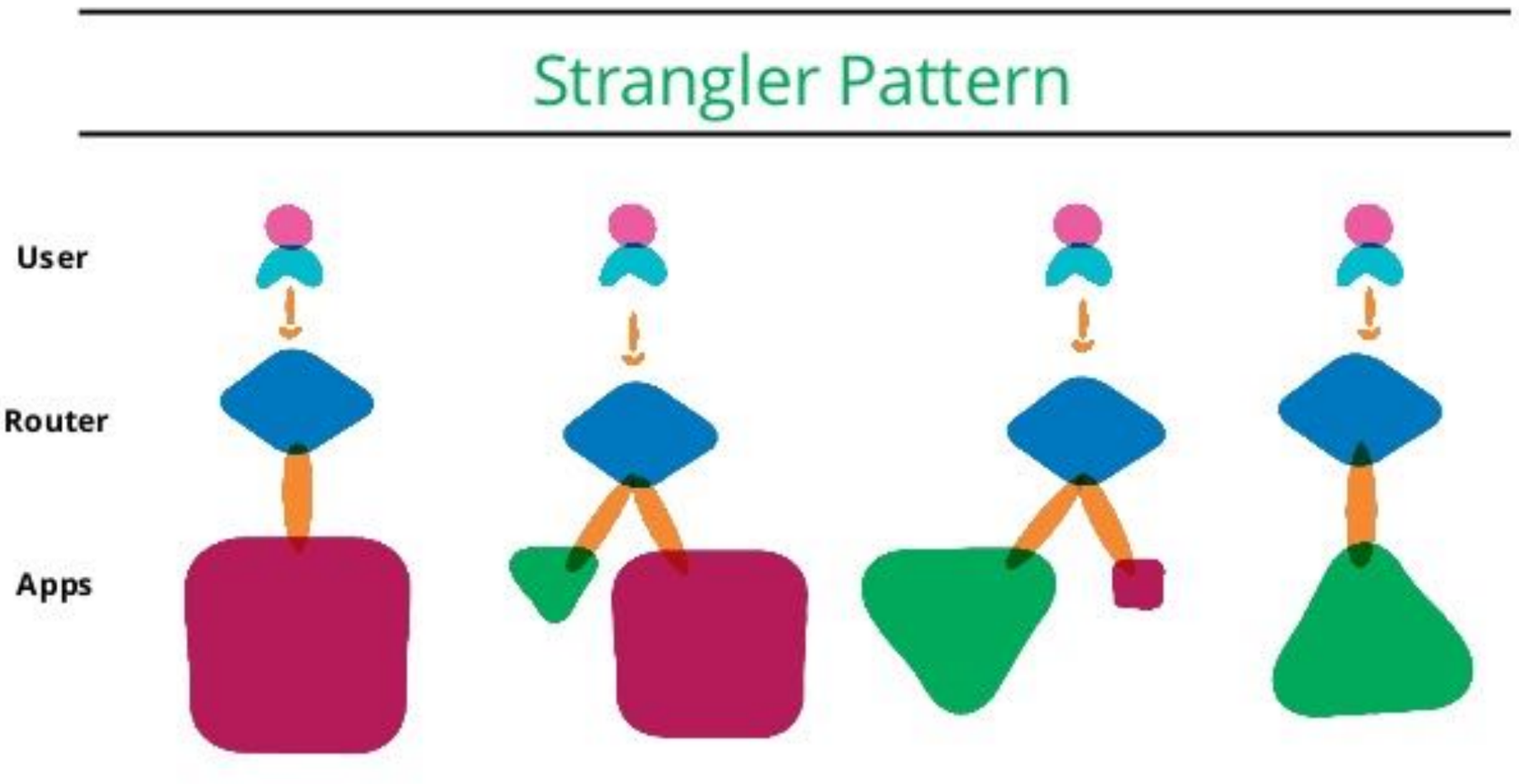

OSC
OIS
S3
API
REST
API
API
GATEWAY
INDEX
SERVICE
MQ
ZooKeeper
S3
API
LIFECYCLE
OTM

温冷迁移


uds.osc
frontend
uds.bucketattr
logic
uds.logicbase
uds.omd
common.entity
common.
mqclient
common.
mqshceduler
common.
mqshceduler
uds.objectlogic
uds.objectattr
logic
common.
configuration
common.pmf
uds.bill
uds.rep
uds.upfbadapter
uds.uerinfoquery

工程实践

演进模式



迭代交付能力

- 演进式设计
- 迭代规划，user story拆分，验收测试用例设计
- 迭代交付，基于迭代的计划、跟踪、验收和回顾
- 流水线要求

软件技能提高

- 正交设计培训
- OO设计编码能力培训，用好Java
- 重构培训：基于遗留系统的开发和优化能力
- 自动化测试技巧：高效的单元测试培训

架构问题

架构问题

- 前台进程和后台任务需要分离，不能因为后台的流量影响前台
- 进程弹缩粒度的绑定与固化（无法自匹配资源）
- 滥用的线程池
- 系统的缺乏易测性设计
- Index layer完成了 DB，MQ，LOCK，DISCOVERY等所有功能
- MongoDB 暴露了过多细节给Service Layer
- 粗暴的进程退出方式，有可能损坏系统资源

Questions?

ThoughtWorks

THANKS

e.wangbo@gmail.com