Splitting the monolith

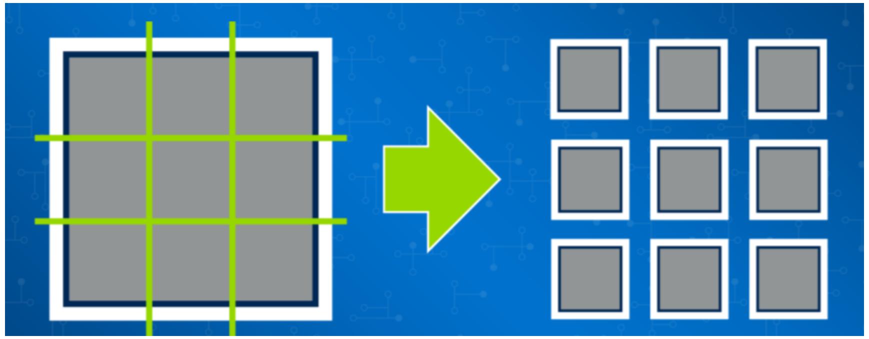
Antonio Brogi

Department of Computer Science
University of Pisa

The monolith grows over time

- new functionalities and lines of code constantly added
- ~~cohesiveness~~ all sorts of unrelated code kept together
- ~~loose coupling~~ small change can impact rest of monolith, need to redeploy the big beast

Split the monolith *when* it becomes a problem



Find the «seams»

line where two pieces of fabric are sewn together in a garment

portion of code that can be treated in isolation and worked on without impacting the rest of the codebase

Find the seams

Identify seams that can become **service boundaries**

- e.g., exploiting the notion of «bounded context»
  - some info does not need to communicated outside, some can be shared with other contexts
  - cell analogy: membranes define what is in and out and determine what can pass

-e.g., also exploiting namespace concepts of programming languages
  - Java’s packages

Example

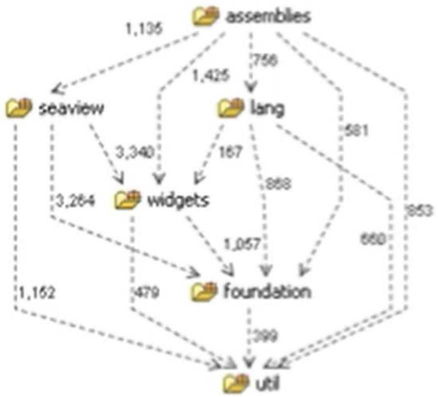
Online music retailer MusicCorp' monolith covers four contexts

- ***Catalog*** - Everything to do with metadata about the items offered for sale
- ***Finance*** - Reporting for accounts, payments, refunds, etc.
- ***Warehouse*** - Dispatching & returning of customer orders, managing inventory levels, etc.
- ***Recommendation*** - Patent-pending, revolutionary recommendation system

Organize code around the seams

Create **packages** representing these contexts,
and then move existing code into them

- Incremental refactoring and testing
- Code left over may identify new bounded context
- Use code to analyze dependencies between packages
  - Packages representing bounded contexts in organization should interact like real organization groups
    - e.g., *Warehouse* package should not depend on *Finance* package if no such dependency in real organization
  - Tools like Structure 101 graphically show inter-package dependencies



- Do the above incrementally, really

Where to start breaking?

Start from where we can get the most benefit

Typical drivers

- *Pace of Change*

  e.g., load of changes in inventory management coming, split out *Warehouse* seam first
- *Team Structure*

  e.g., delivery team split across two geographical regions
- *Security*

  e.g., all sensitive information handled by *Finance* code
- *Technology*

  e.g. team working on *Recommendation* system experimenting new algorithms in Clojure
- *Tangled dependencies*

  Pull out the seam that is least depended on, if possible

Databases

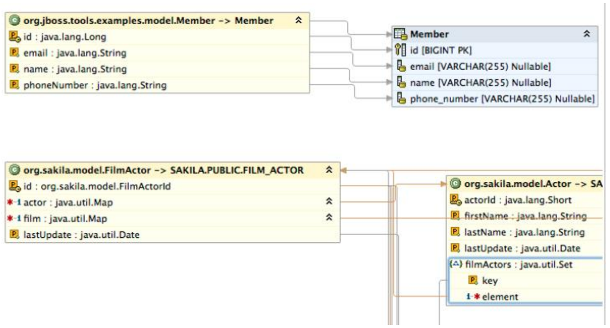
Databases often used to integrate multiple services

Need to find seams in our databases so that we can split them out cleanly

Databases

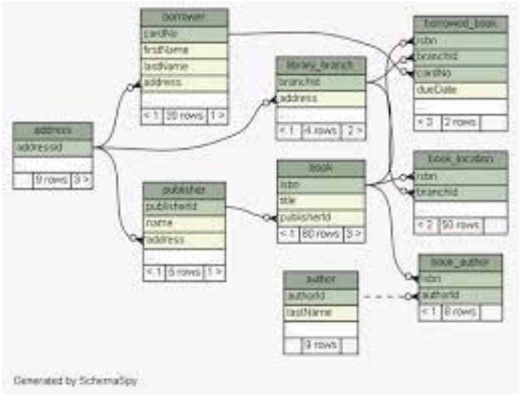
Understand which parts of code read to and write from the database

- e.g., use tools like Hybernate


org.jboss.tools.examples.model.Member -> Member
id : java.lang.Long
email : java.lang.String
name : java.lang.String
phoneNumber : java.lang.String
Member
id [BIGINT PK]
email [VARCHAR(255) Nullable]
name [VARCHAR(255) Nullable]
phone_number [VARCHAR(255) Nullable]
org.sakila.model.FilmActor -> SAKILA.PUBLIC.FILM_ACTOR
id : org.sakila.model.FilmActorId
actor : java.util.Map
film : java.util.Map
lastUpdate : java.util.Date
org.sakila.model.Actor -> SA
actorId : java.lang.Short
firstName : java.lang.String
lastName : java.lang.String
lastUpdate : java.util.Date
filmActors : java.util.Set
key
element

Detect database-level constraints

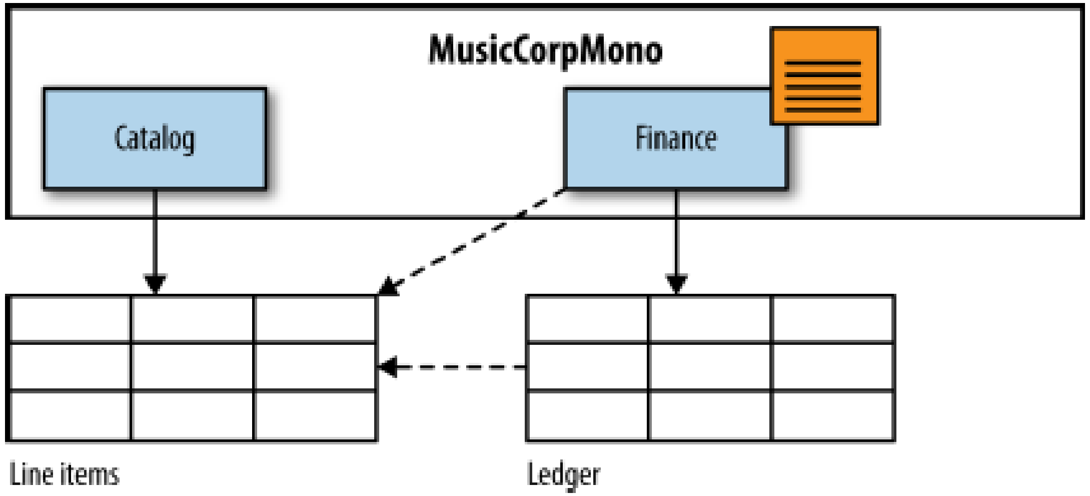
- e.g., foreign-key relationship across tables used by different parts of code
- e.g., use tools like SchemaSpy



Some tables may be used from different bounded contexts, too

Let’s see some examples of database refactoring …

Databases example: **Breaking foreign key relationships**

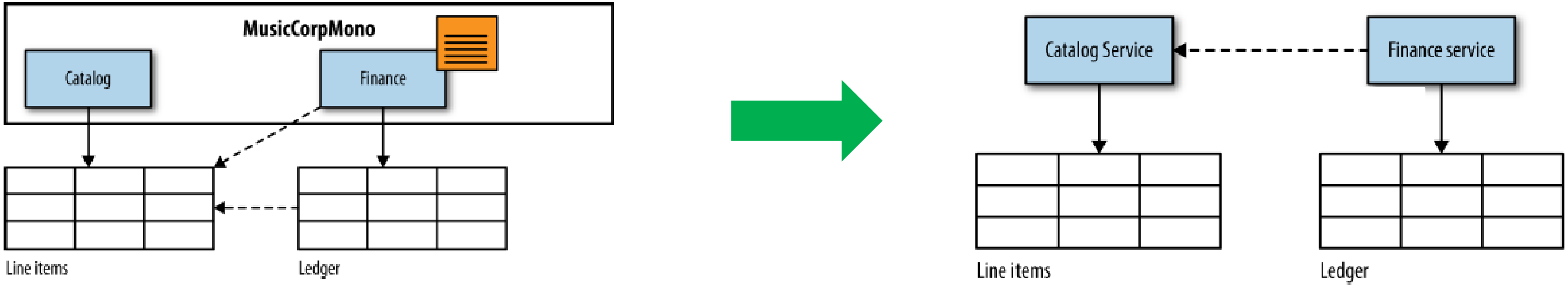


Finance code uses *ledger table* to track financial info

Catalog code uses *line item table* to store info on an album

To generate monthly reports like
“We sold 400 copies of Lady Gaga’s Joanne and made 1,300 USD”
Finance code accesses *line item table*, and a foreign-key relationship from *ledge table* to *line item table* exists

Databases example: **Breaking foreign key relationships**

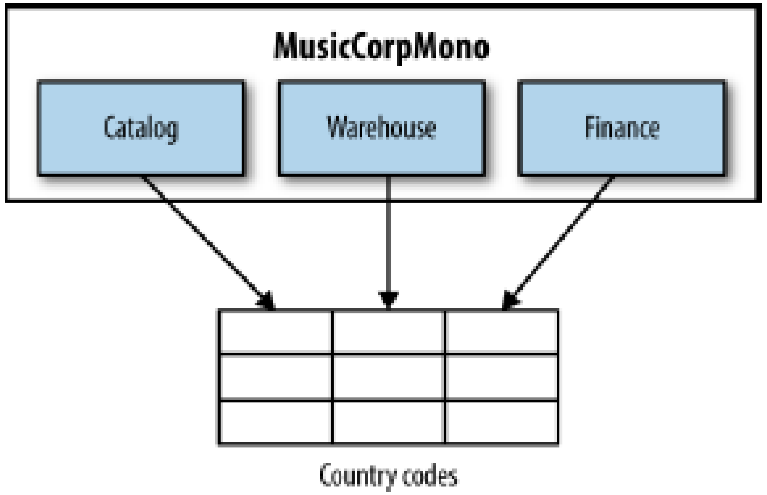


Solution: Expose data via API in the *Catalog* package and make two database calls to generate the report

- overhead introduced (yes)
- foreign key relationship lost
  - constraint that will have to be managed in the services rather at the database level
  - may* need to implement consistency check across services, or to trigger actions to clean up related data

*choice often not based on technology, e.g., how to handle an order that refers to old -now invalid- catalog ID

Databases example: **Shared *static* data**



Solutions

(1) Duplicate table for each package (and then service)

- potential consistency challenge

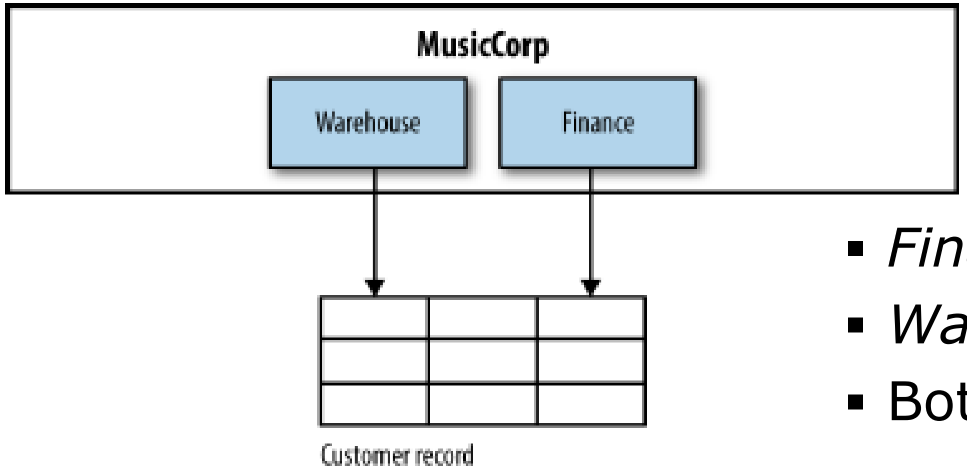
(2) Treat this shared static data as code (e.g., configuration file)

- consistency issues remain (though perhaps easier to change config files than db tables)

(3) Push static data into a separate service

- overkill?

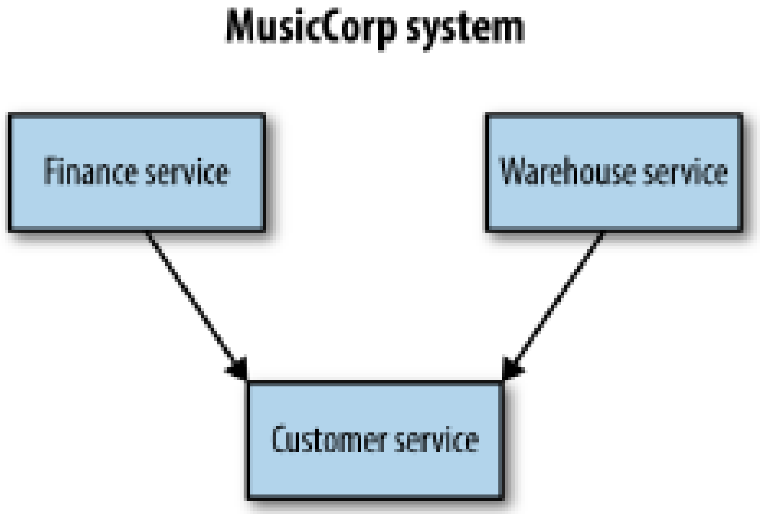
Databases example: **Shared *mutable* data**



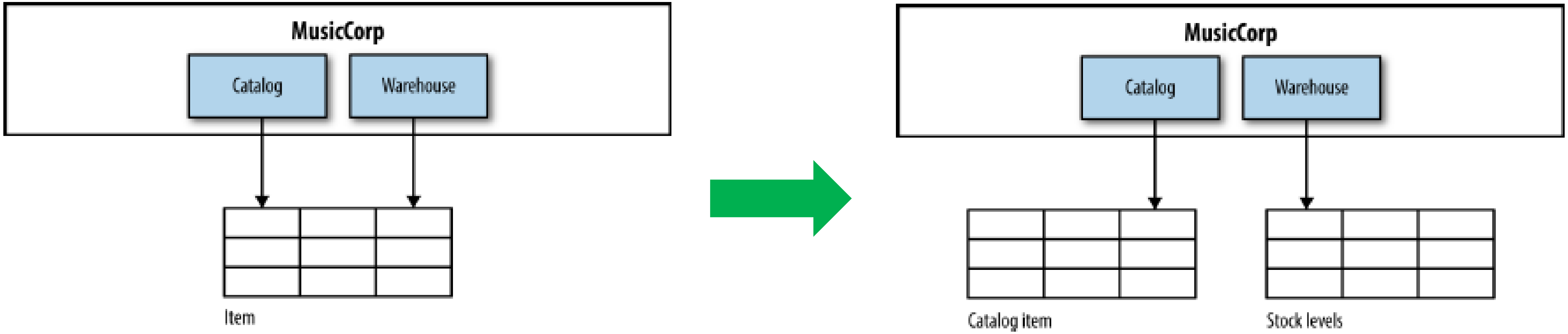
- *Finance* code tracks customers’ payments and refunds
- *Warehouse* code updates orders dispatched and received
- Both *Finance* and *Warehouse* access *customer record table*

Domain concept – *Customer* - implicitly modeled in the database.

Solution: Create new *Customer* package (service, later), and make *Finance* and *Warehouse* invoking its API



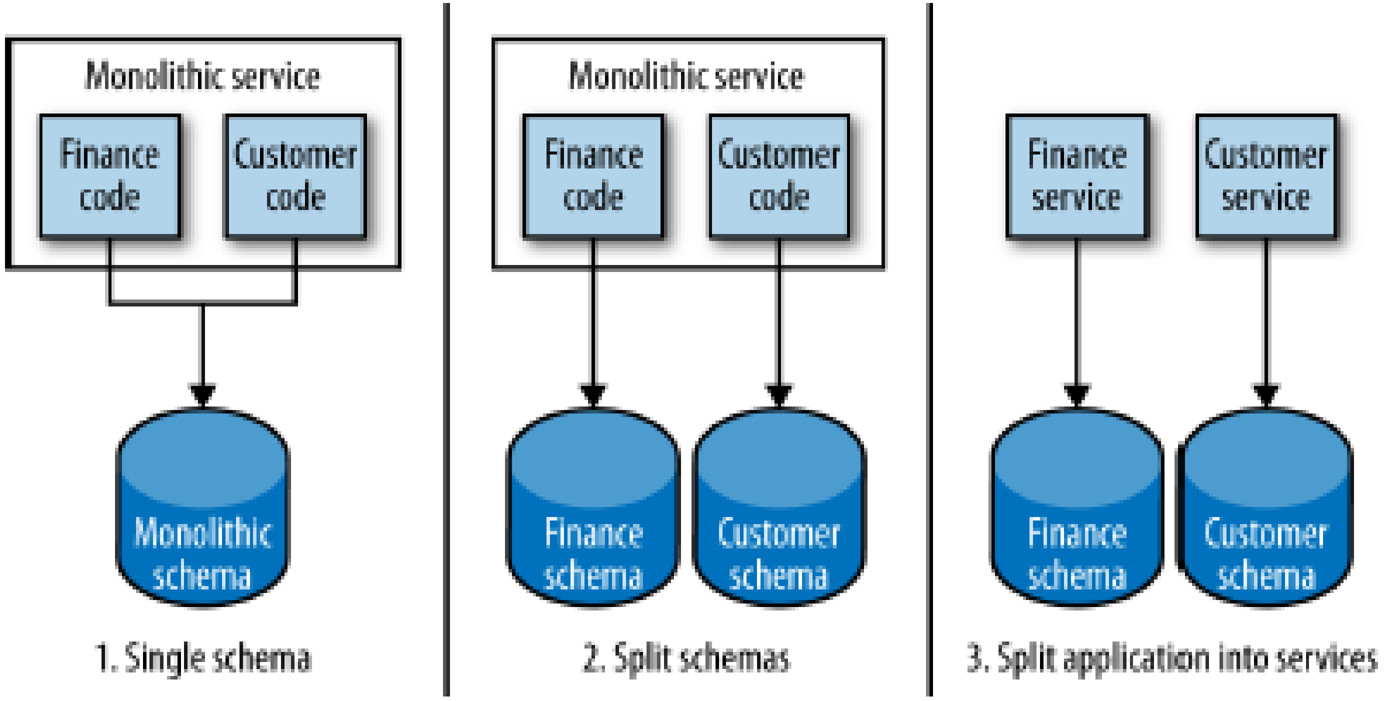
Databases example: **Shared tables**



- *Catalog* needs to store name and price of record we sell
- *Warehouse* needs to keep electronic record of inventory
- Both info kept in same *item table*

Solution: Store the two concepts separately, split table in two

Tip: First split schemas, then split services



1. Single schema 2. Split schemas 3. Split application into services

Splitting schemas (potentially) increases number of database calls, may break transactional integrity

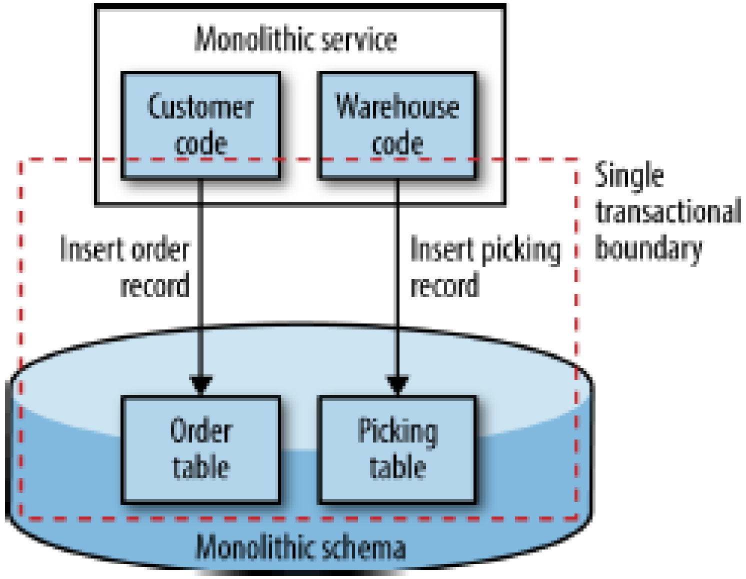
Splitting schemas but keeping application code together makes reverting changes easier

Transactions

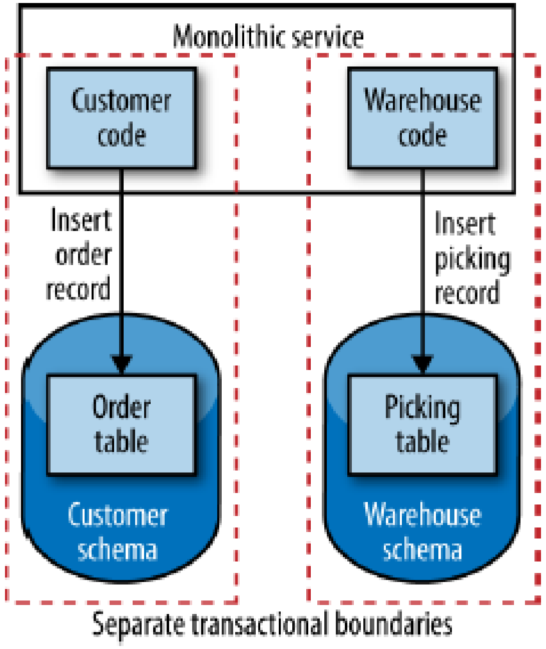
All-or-nothing transactions are useful, multiple tables updated «at once», always stay in consistent states

Example- When a new order:

- *order table* updated
- *picking table* updated for *Warehouse* team



Single transaction in a monolithic schema



Transactional safety lost with two separate schemas

What if *insert order record* succeeds but *insert picking record* fails?

Transactions: Solutions (1)

Abort the entire operation, with compensating transaction(s)

- e.g., if *insert order record* succeeds but *insert picking record* fails, then
  - unwind the committed transaction in the *order table*
  - report failure via UI

Questions

- Where the logic to handle compensating transactions?
  - *Customer* service, *Warehouse* service, elsewhere?
- What if compensating transaction fails? (retry)
- What if we need to enforce consistency of 5 (rather than 2) ops?

Transactions: Solutions (2)

Employ *distributed transactions*

- *Transaction manager* orchestrates the various transactions being done
- *Two-phase commit*
  - Voting phase: each participant tells manager whether its local transaction can go ahead
  - If transaction manager gets a *yes* vote from all participants, it tells them all to go ahead and perform their commits – otherwise it sends a rollback to all parties

Vulnerabilities

- If transaction manager goes down, pending transactions never complete
  - locks on resources can lead to contention and inhibit scaling
- If one participant fails to respond during voting, everything blocks
- What if a commit fails after voting?
  - hp: participants make commit work eventually

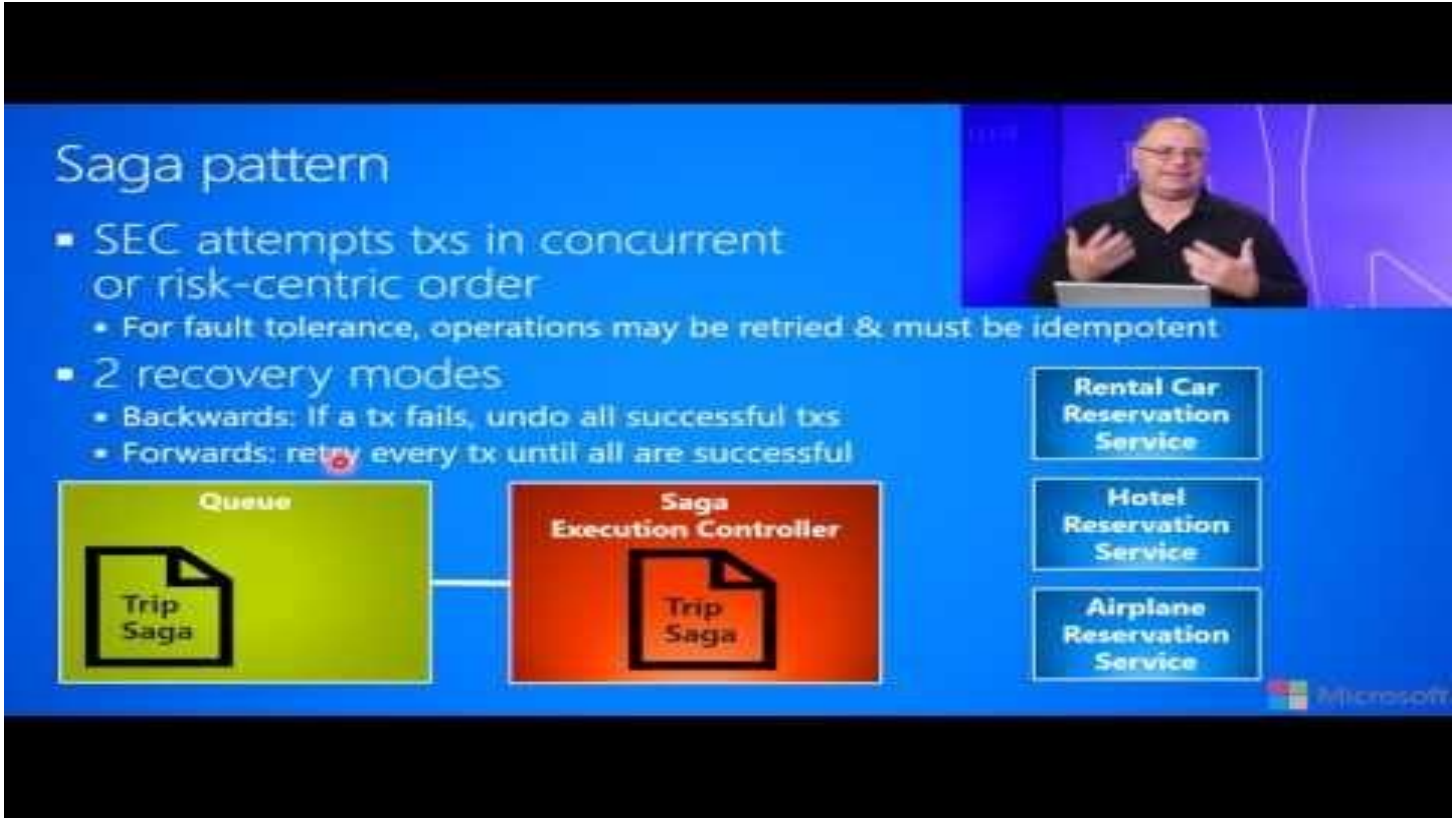
Transactions: Solutions (3)

TRY AGAIN
LATER

eventual consistency

Implementing eventual consistency

The saga pattern



Saga pattern

- SEC attempts txs in concurrent or risk-centric order
  - For fault tolerance, operations may be retried & must be idempotent
- 2 recovery modes
  - Backwards: If a tx fails, undo all successful txs
  - Forwards: retry every tx until all are successful

Queue

Trip Saga

Saga Execution Controller

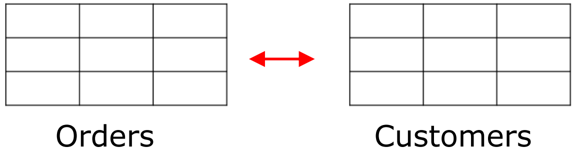
Trip Saga

Rental Car Reservation Service

Hotel Reservation Service

Airplane Reservation Service

Eventual consistency



Implementing eventual consistency

(a) Client determines when data is consistent

(b) Use fault-tolerant message queues & Saga pattern

- Saga Execution Controller attempts transactions in parallel or according to «risk»
  - retries (idempotent) operations
- Recovery models
  - Backward model: if a transaction fails undo all succesful transactions
  - Forward model: retry every transaction till all succeed

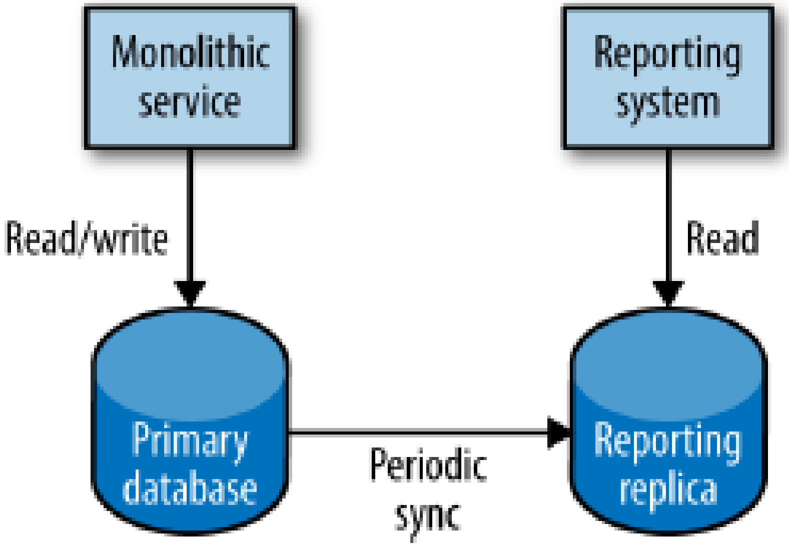
Rental Car Reservation Service
Hotel Reservation Service
Airplane Reservation Service

The Reporting Database

Reporting typically needs to group together data from across multiple parts of organization in order to generate useful output, e.g.,

- combine data from general ledger with descriptions sold items from catalog
- combine purchase histories and customer profiles to get behavior of customers

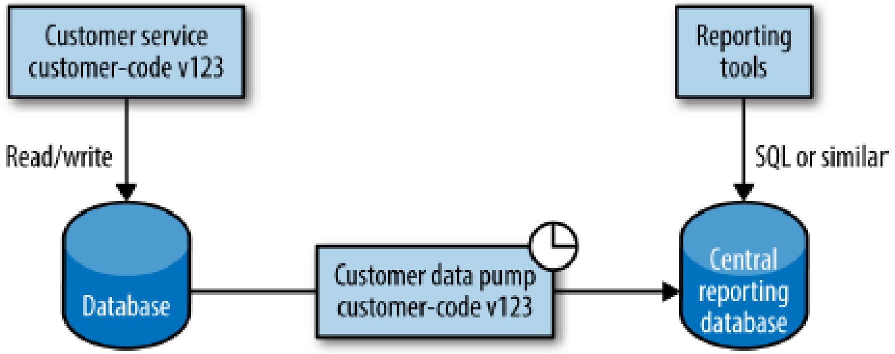
In monoliths, reports typically run on read replica of primary database



Reporting on multiple systems

Pull data from source systems via API calls

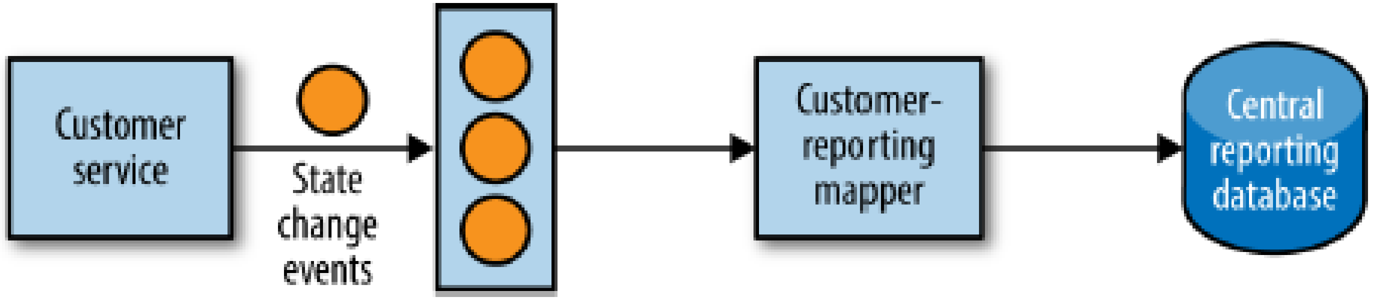
- does not scale well with large volumes of data
- exposed APIs may not be designed for reporting (e.g., no way to retrieve all customers)
  - inefficient, may generate load for target service



Data Pumps

- data pushed to the reporting system
- data pump maps service db to reporting schema
- coupling worth to pay to make reporting easier
  - data pump should be built & managed by the team managing the service
  - data pump version-controlled together with service
- can be piggybacked by backup operations (Netflix “backup data pumps”)

Reporting on multiple systems



Event Data Pumps

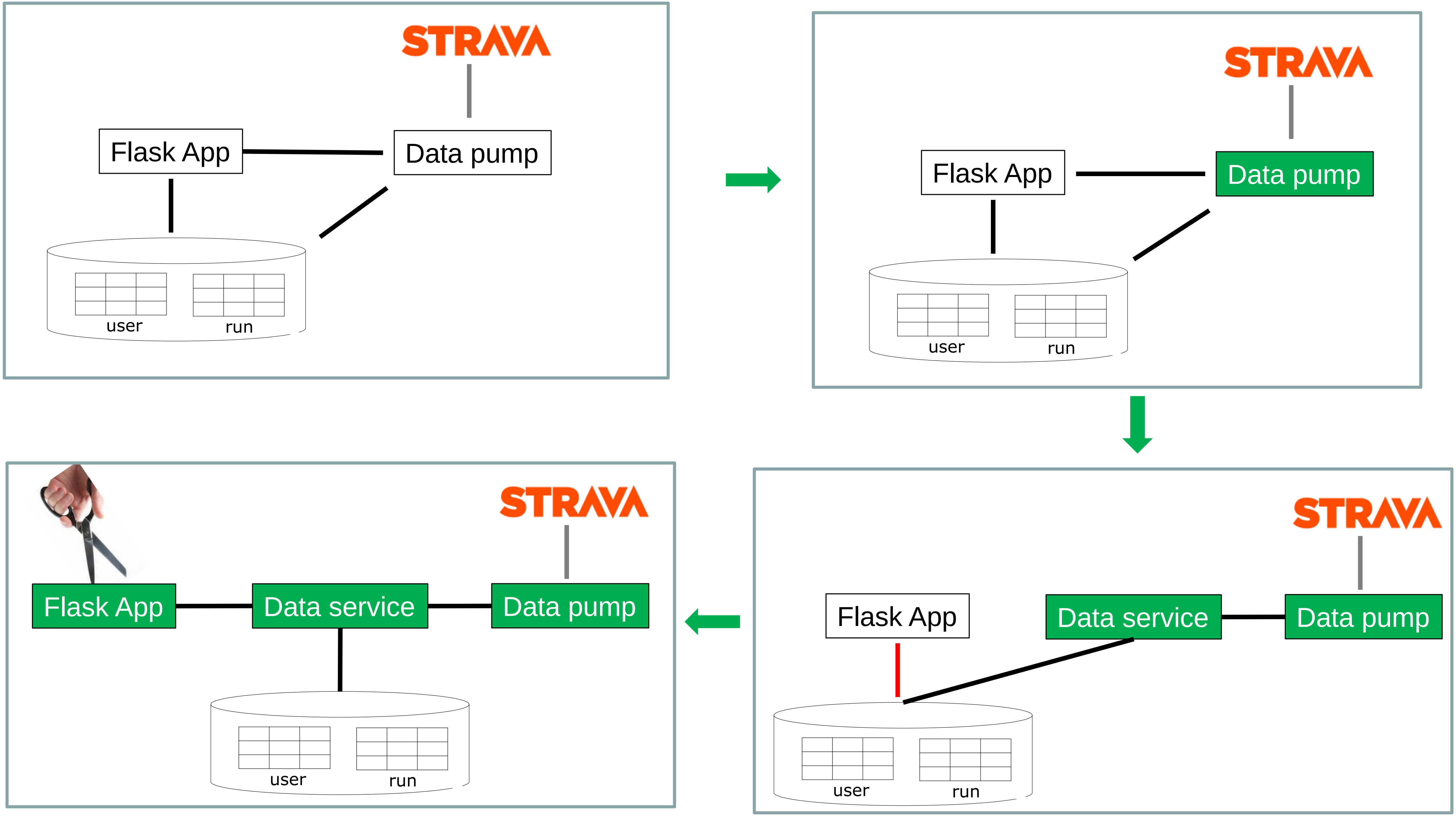
- microservices can emit events based on the state change of entities that they manage
  - e.g., Customer service may emit event when a customer is created/updated/deleted
- event subscriber pumps data into reporting database
- no coupling with db of source microservice, just binding to events emitted by service (meant to be exposed)
- temporal nature of events makes it easier to flow data faster to reporting system (than periodically scheduling a data pump)
- event data pump needs not to be managed by team managing the service
- drawback: all required information must be broadcasted as events, may not scale well for large volumes of data

Concluding remarks

Planning small, incremental changes makes it easier to understand impact of each change and how to mitigate costs of (unavoidable) mistakes

Growing a service till it needs to be split is completely OK

But we should understand *when* it needs to be split – and split it – before the split becomes too expensive


STRAVA
Flask App
Data pump
user
run
STRAVA
Flask App
Data pump
user
run
STRAVA
Flask App
Data service
Data pump
user
run
STRAVA
Flask App
Data service
Data pump
user
run

OpenAPI Initiative

OpenAPI Initiative (Linux Foundation Collaborative Project) aims at creating a standardized, vendor neutral description format of REST APIs

- Open API 2.0 f.k.a. Swagger (current version 3.0.2)
- Simple (JSON-based) description language to specify HTTP API endpoints, how they are used, and the structure of data that comes in and out

OpenAPI members

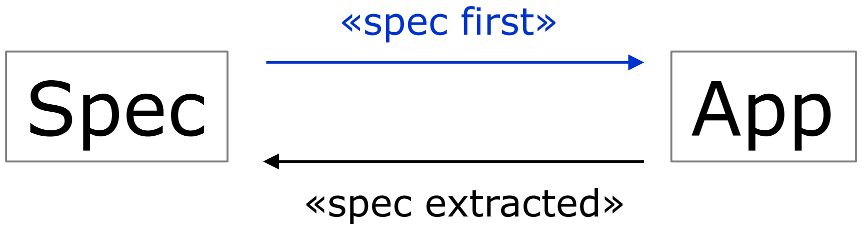
Industry adoption (2016)

/* Simple example: One endpoint /api/users_id supporting GET to retrieve list of user Ids */

```
swagger: "2.0"
info:
  title: BipBip Data Service
  description: returns info about BipBip data
  license:
    name: APLv2
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  version: 0.1.0
  basePath: /api
  paths:
    /user_ids:
      get:
        operationId: getUserIds
        description: Returns a list of ids
        produces:
        - application/json
        responses:
          '200’:
            description: List of Ids
            schema:
            type: array
            items:
              type: integer
```

OpenAPI 2.0

E.g., Connexion framework for Flask
automagically handles HTTP requests based
on OpenAPI 2.0 Specification of your API


«spec first»
Spec
App
«spec extracted»