
Full Stack Development
Containers, Microservices and UI
9. Microservices

Recall In Real Life

- A hospital provides a health care service
- Made up of individual microservices
  - Laboratory
  - X-Ray
  - Pharmacy
  - Medical Staffing
- Each microservice:
  - Specializes in a specific domain activity
  - Only that microservice performs that domain activity (the pharmacy doesn't do x-rays for example)
  - Each microservice operates autonomously

Recall Interfaces In Real Life

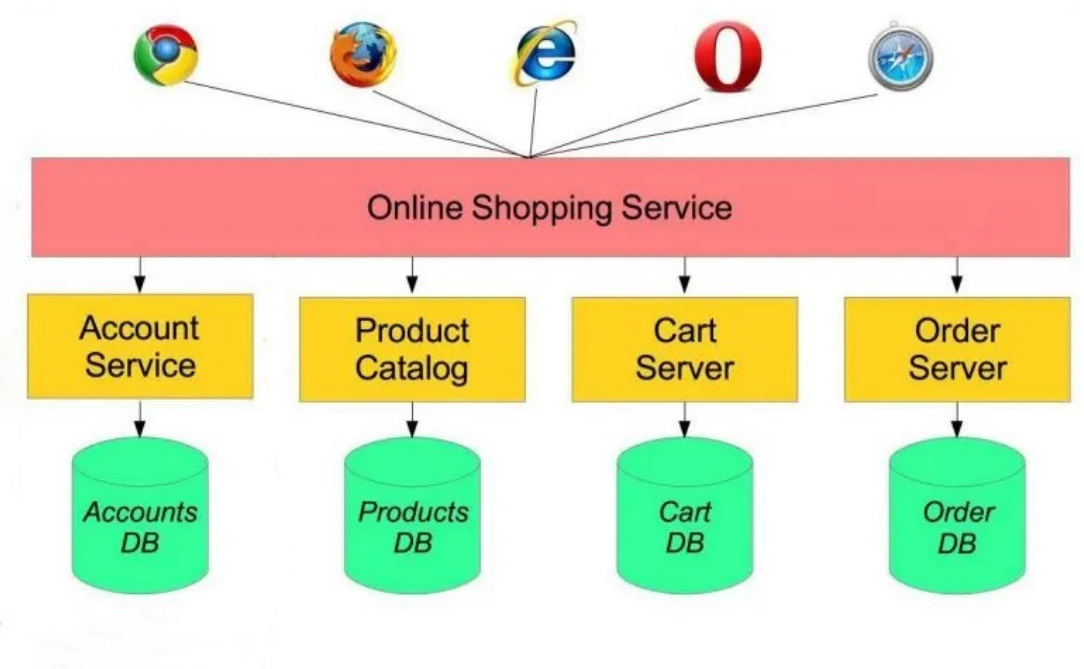
- Microservices request services from each other
  - They do not need to know the internal workings of the other microservice
- Requests are made through interfaces
  - Called APIs in software engineering
  - These are often paper based in the real world
  - Requisitions and official forms and paperwork used to make requests of a service
  - Response to a request is often a report of some kind
- Microservices make sense intuitively

COVID-19 VIRUS LABORATORY TEST REQUEST FORM[1]

Submitter information			
NAME OF SUBMITTING HOSPITAL, LABORATORY, or OTHER FACILITY*			
Physician			
Address			
Phone number			
Case definition:[2]	☐ Suspected case ☐ Probable case		
Patient info			
First name		Last name	
Patient ID number		Date of Birth	Age:
Address		Sex	☐ Male ☐ Female ☐ Unknown
Phone number			
Specimen information			
Type	☐ Nasopharyngeal and oropharyngeal swab ☐ Bronchoalveolar lavage ☐ Endotracheal aspirate ☐ Nasopharyngeal aspirate ☐ Nasal wash ☐ Sputum ☐ Lung tissue ☐ Serum ☐ Whole blood ☐ Urine ☐ Stool ☐ Other:		
All specimens collected should be regarded as potentially infectious and you must contact the reference laboratory before sending samples. All samples must be sent in accordance with category B transport requirements.			
Please tick the box if your clinical sample is post mortem ☐			
Date of collection		Time of collection	
Priority status			
Clinical details			
Date of symptom onset:			
Has the patient had a recent history of travelling to an affected area?	☐ Yes ☐ No	Country Return date	
Has the patient had contact with a confirmed case?		☐ Yes ☐ No ☐ Unknown ☐ Other exposure:	
Additional Comments			

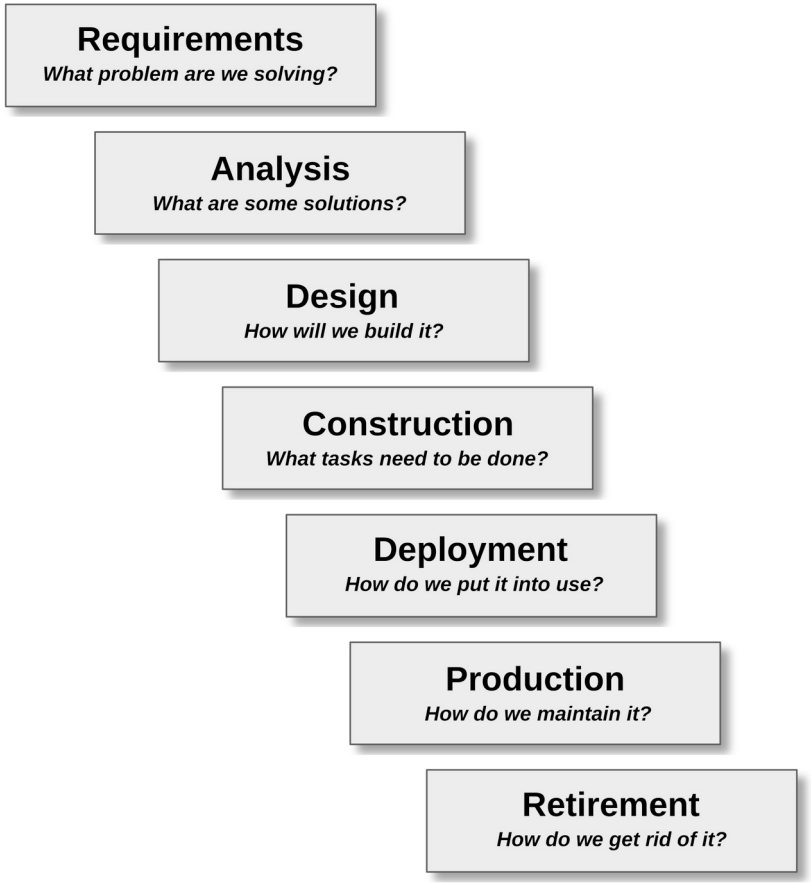
Recall Implementing the Solution

- Software architectural design choice
  - Enterprise applications are deployed as a set of microservices
  - Each microservice represents a clearly defined area of domain functionality
  - Each service manages requests from other services through an API
- Microservices are deployed as individual applications
  - Microservices are isolated from each other
  - They only communicate through APIs
  - They require an infrastructure to deploy them and ensure that they are all running and available
  - As an application executes, it makes requests of various microservices

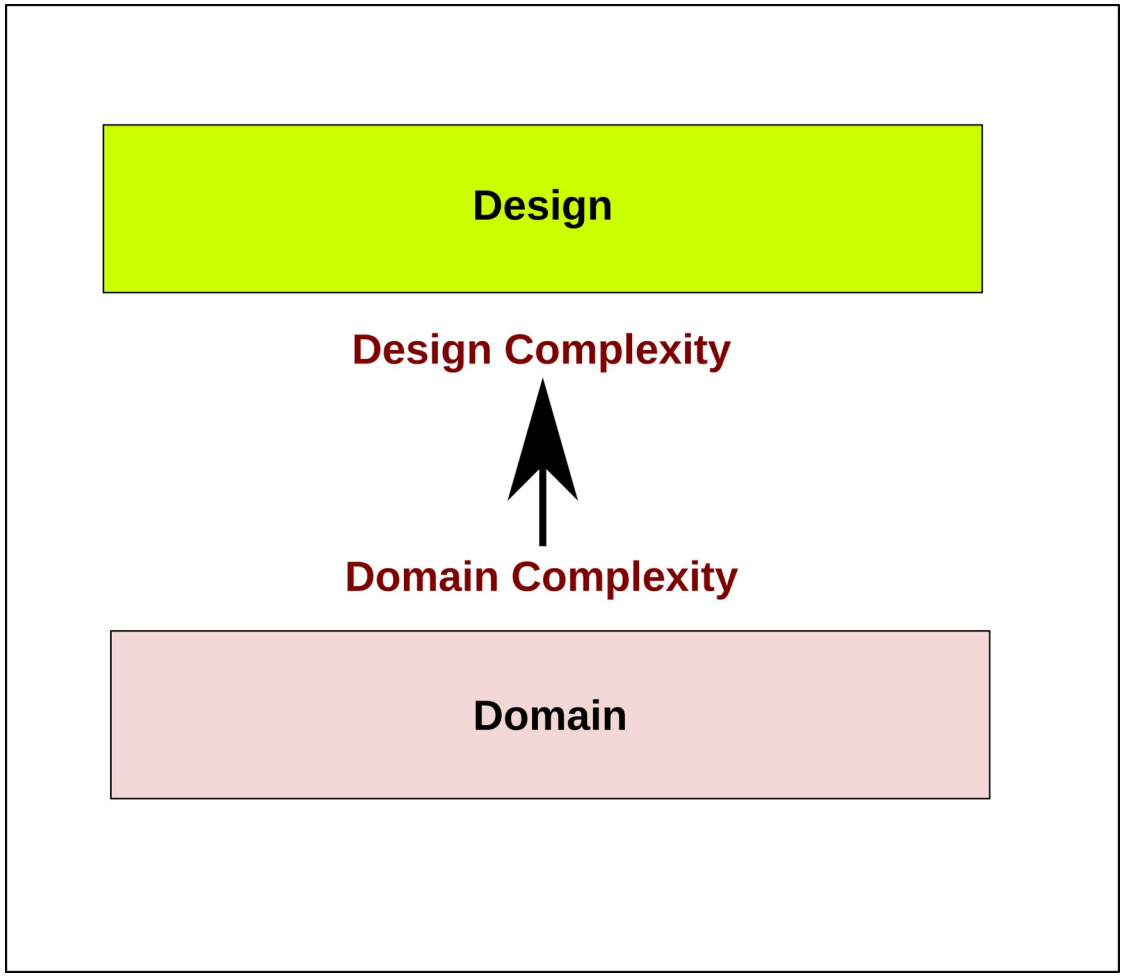


Requirements

- Understand what sort of functionality the clients of a microservice will need
- identify the domain objects, like transactions, will be passed through interfaces
- Identify the real world constraints on the domain objects
- The different contexts of the domain
- To do this, we employ a domain model
  - An abstraction of the domain that we can base a design on



The Complexity Problem



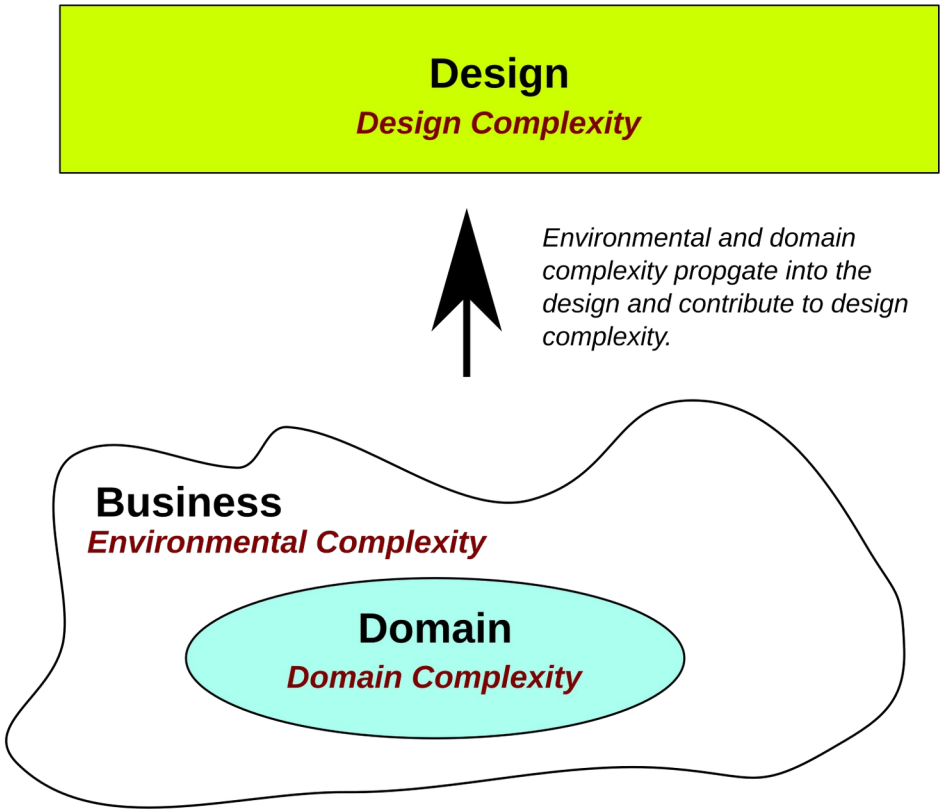
- There are two kinds of complexity we have always been concerned with
  - Domain complexity - trying to automate a domain that is inherently complex
- Design complexity
  - Designs that are not elegant (simple and effective)
  - We can eliminate a lot of design complexity with good design and engineering practices
- We cannot make domain complexity go away, we can only manage it

> *Controlling complexity is the essence of computer programming.* *Brian Kernigan*

Design Complexity

- Design complexity is a result of design choices
- Often due to over-engineering or mis-engineering solutions
- Also due to bad architectural and technology choices
- Managing design complexity will be touched on throughout the course
- Design complexity is dealt with extensively in software development methodologies and practices
  - Design Patterns for example

Types of Complexity



- Not properly managed, environmental and domain complexity directly propagate into design complexity

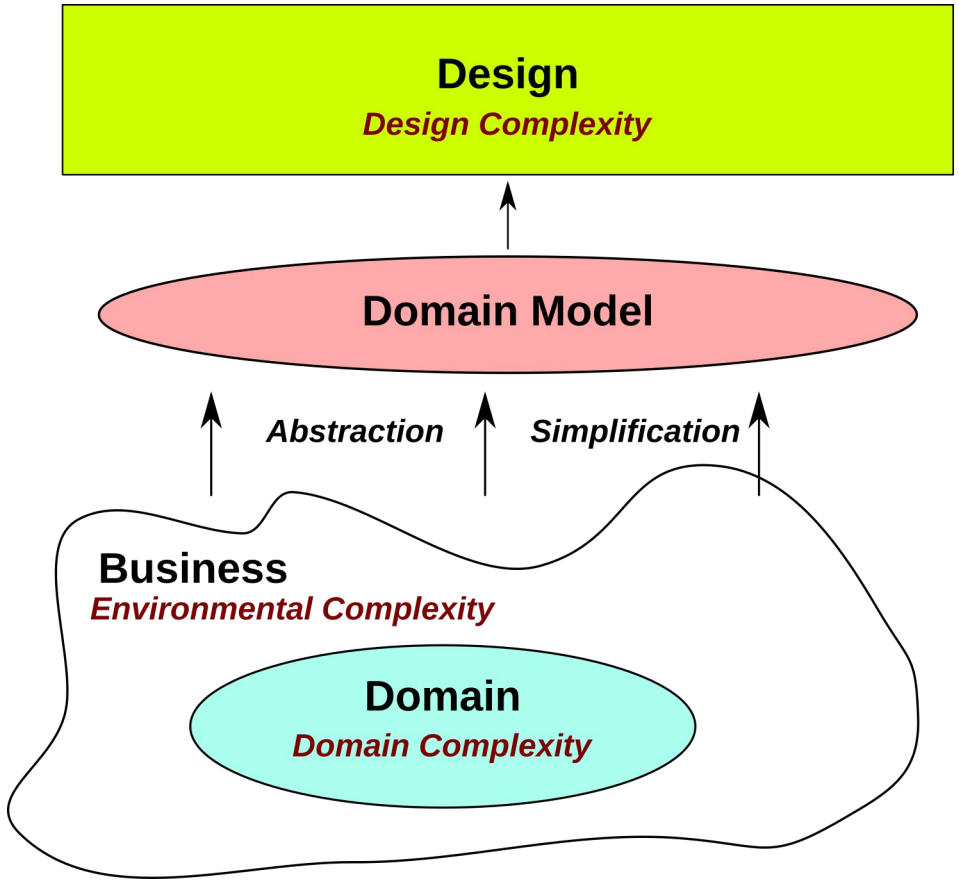
The most important single aspect of software development is to be clear about what you are trying to build.
Edsger Dijkstra

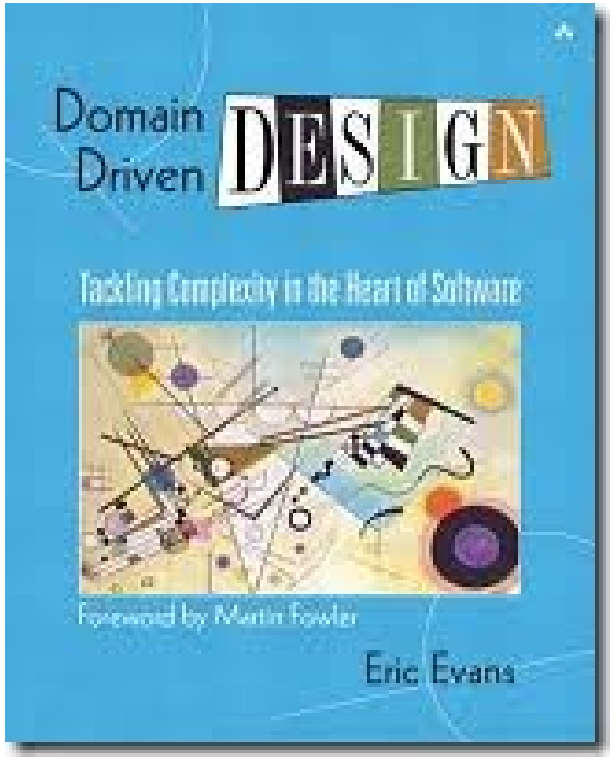
First, solve the problem. Then, write the code
Donald Knuth

Environmental Complexity

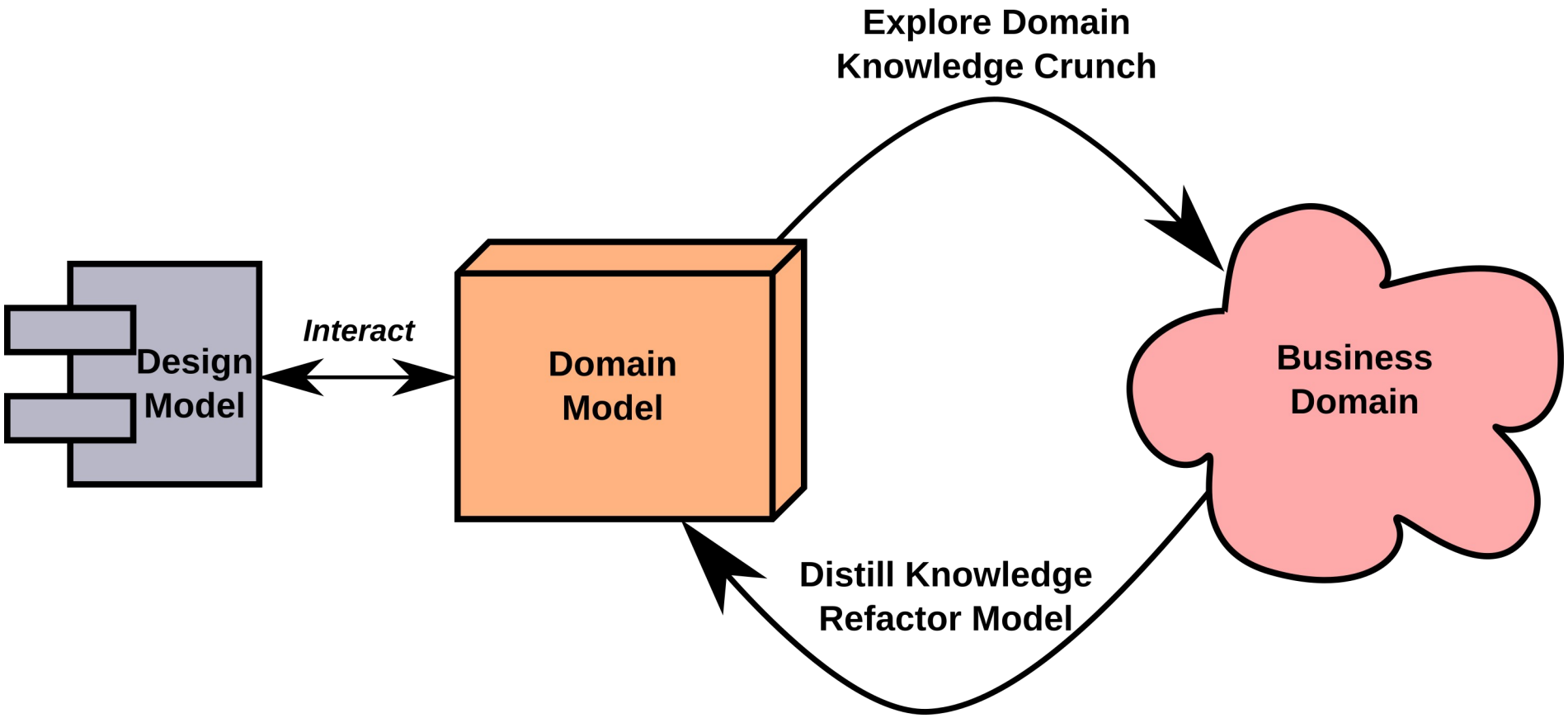
- The environment can be complex because:
  - No one understands the business
  - Conflicting or contradictory business processes and rules
  - Lots of administrative overhead (bureaucracy)
  - The organization is badly managed
  - What people do is not the same as what the process says they do
  - There is more than one domain all mixed in together
- And other factors...
- There exist a number of practices and processes in business analysis and software development to manage the environmental complexity - these are outside the scope of this course

The Domain Model





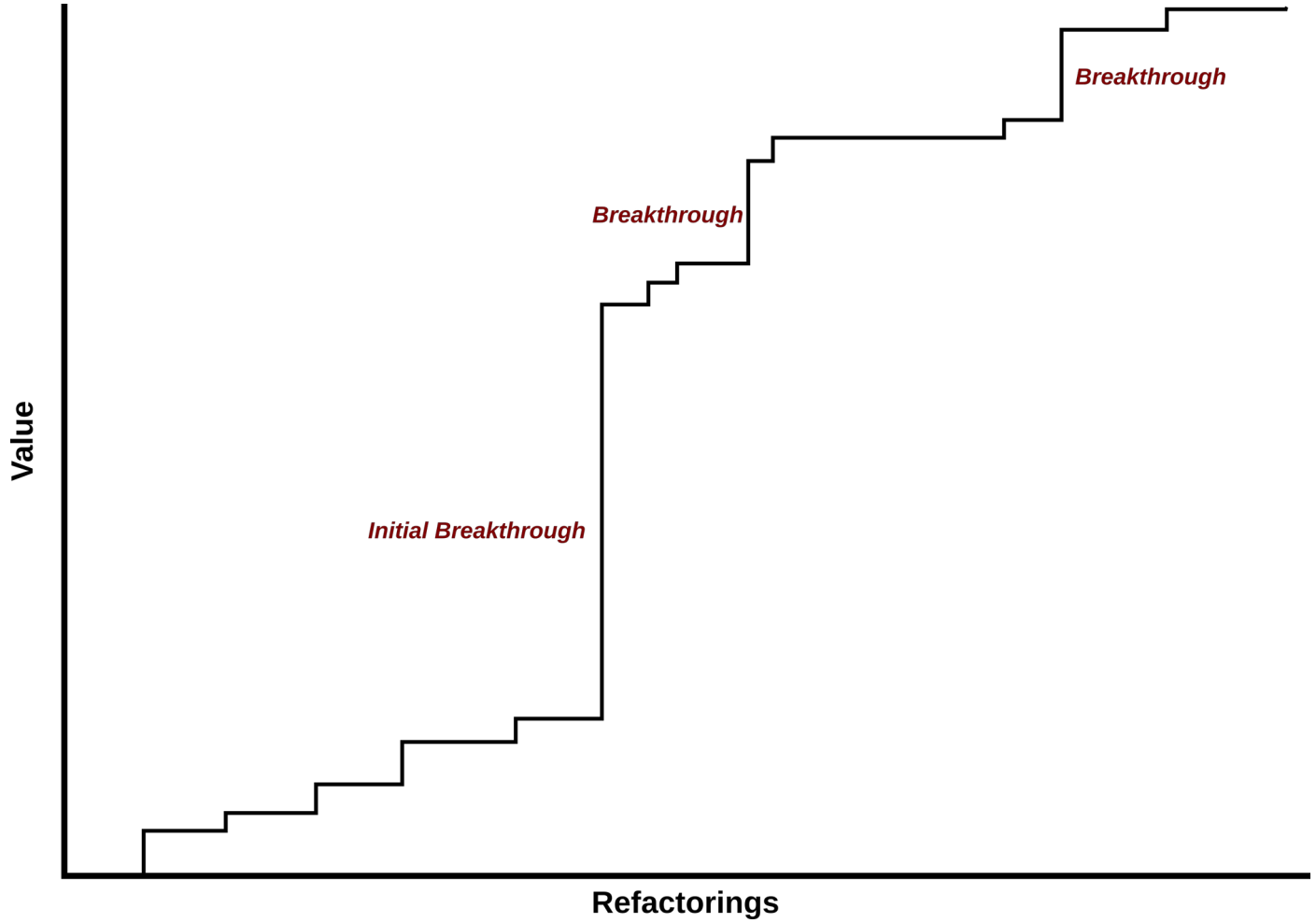
The Knowledge Crunching Iterative Process



Breakthrough Iterations

- The process of understanding the domain is not linear.
- After number of refactorings a breakthrough insight often occurs.
- Results in a new more elegant model
  - "Versatility and explanatory power suddenly increase even as complexity evaporates - this sort of breakthrough is not a technique, it is an event" Eric Evans

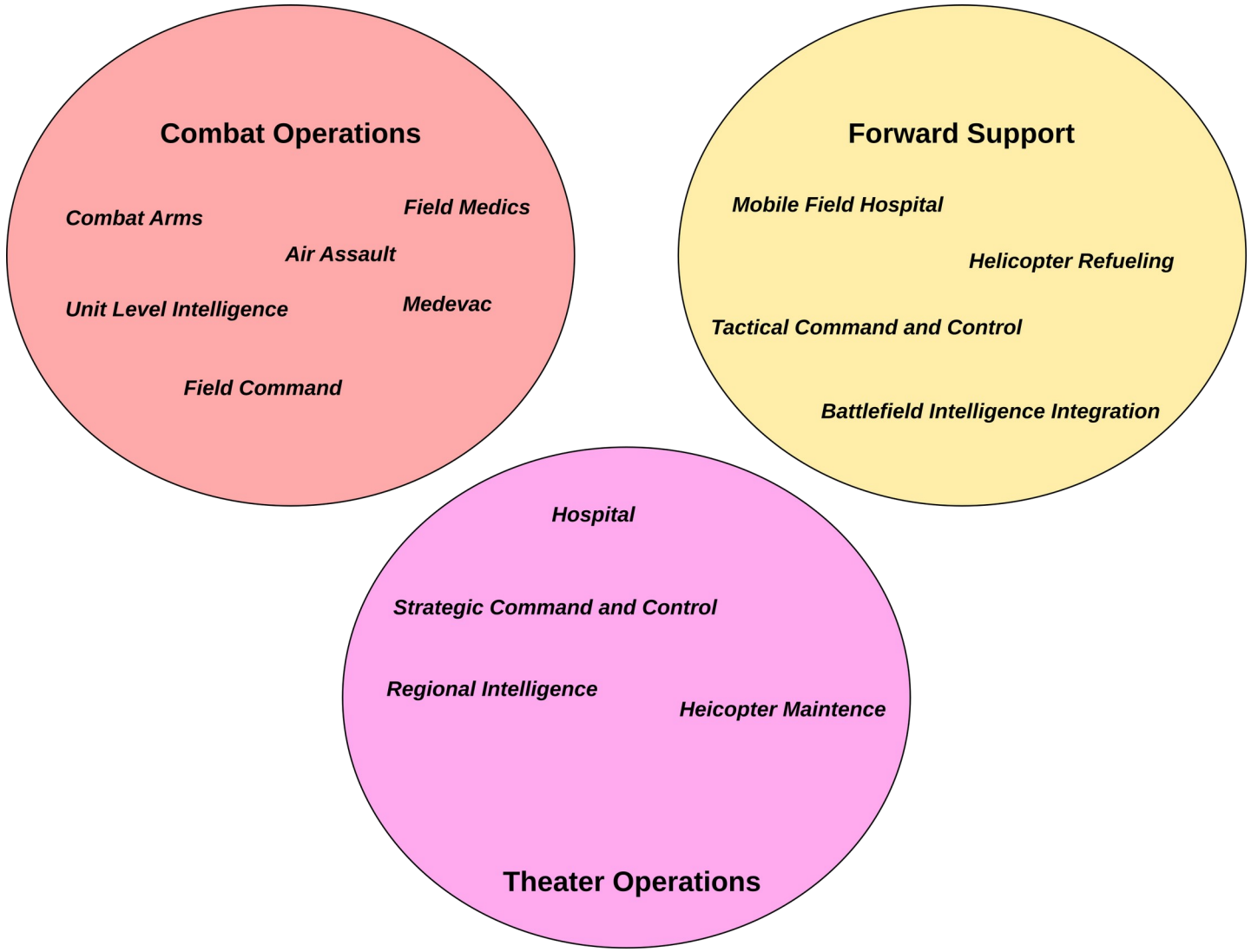
Breakthrough Iterations



Sub-Domains

- Sub-domains are the existing clusters, divisions or groupings that we see in the domain
  - They can have vague or fuzzy boundaries, which are often the cause of "turf wars"
  - Sub-domains exist in the business, we do not define them, we explore the business to discover them
  - We do not want to base the structure of our model on the structure of the sub-domains
  - Sub-domains may exist because of factors other than the business models and processes
    - Geographical co-location
    - Administrative regions and structures
    - Cross-disciplinary teams organized around a specific task or project
    - Legal or regulatory reasons
    - Historical mergers or acquisitions
  - These can be considered environmental complexity factors that we want to abstract out of our domain model

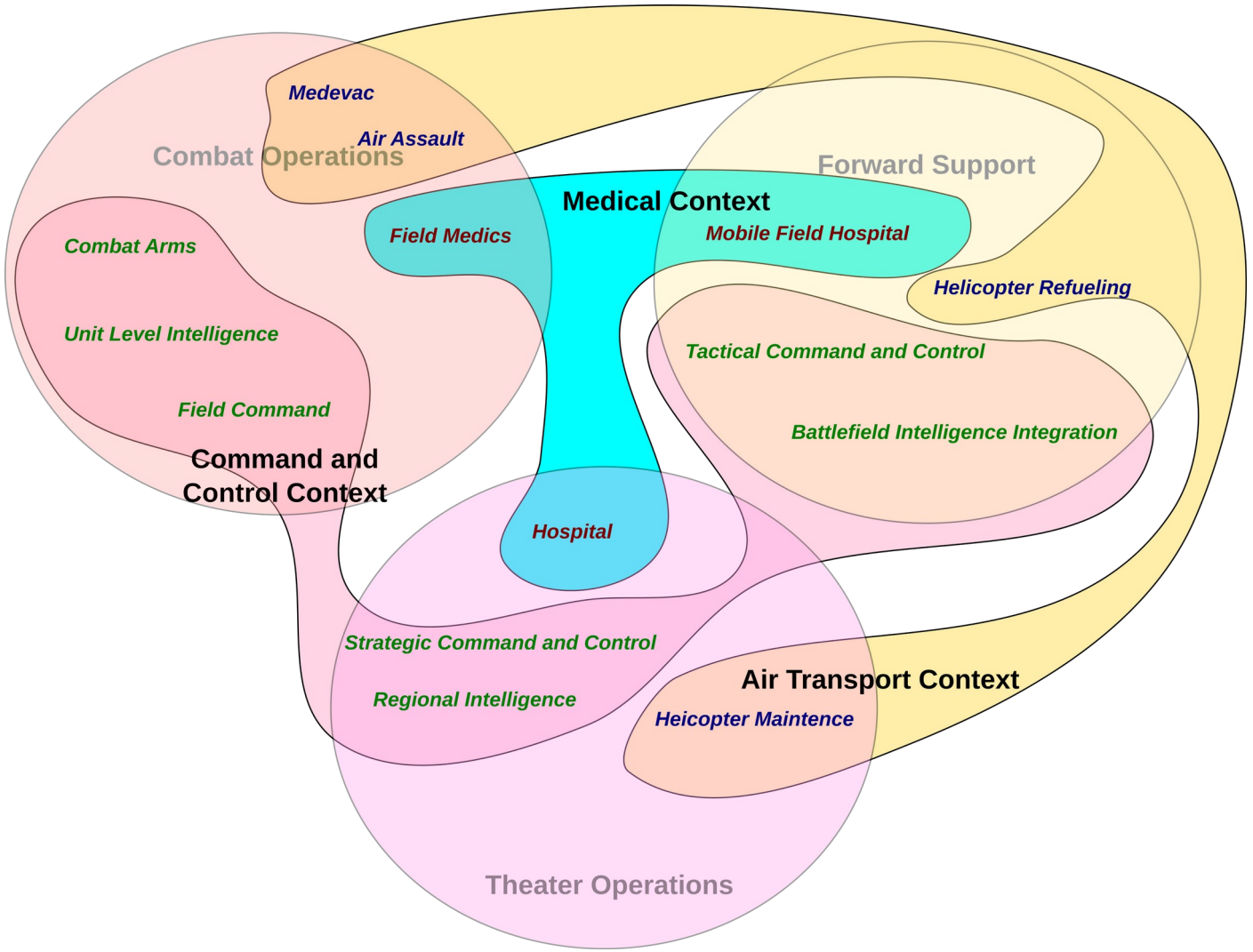
Military Sub-Domains Example



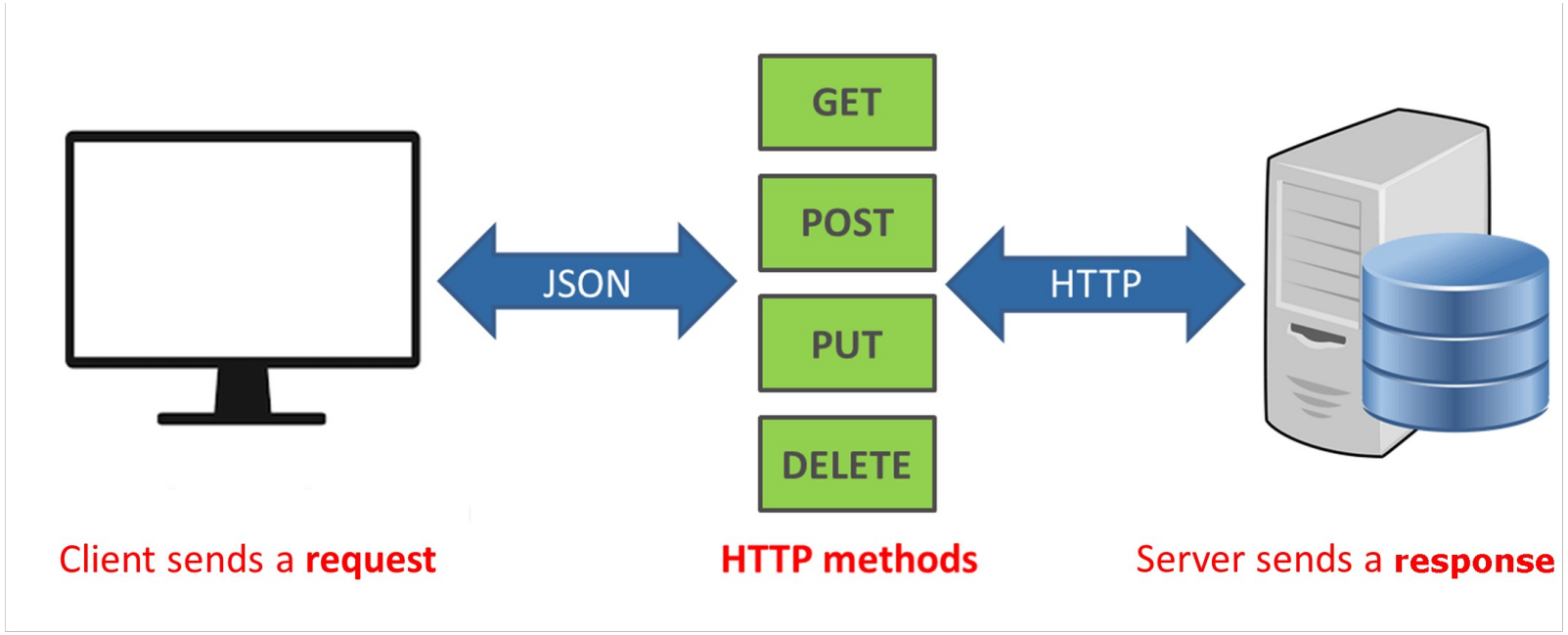
Sub-Domains

- A bounded context delimits the applicability of a particular model
  - A model is cohesive and unified within a bounded context
  - Different models apply in different contexts
  - We define contexts in our domain models to delimit where the important sub-models need to be developed
  - A context may appear in several sub-domains in the business
- A single sub-domain in the business may work across multiple contexts
- We do not model across context boundaries

Bounded Contexts



Restful Web Service



Restful Web Service Design

- Client-Server
  - There should be a separation between the server that offers a service, and the client that consumes it
- Stateless
  - Each request from a client must contain all the information required by the server to carry out the request
  - The server cannot store information provided by the client in one request and use it in another request
- Cacheable
  - The server must indicate to the client if requests can be cached or not

Restful Web Service Design

```
GET http://api.coolcars.io/cars/

GET http://api.coolcars.io/cars/{id}

DELETE http://api.coolcars.io/cars/{id}

POST http://api.coolcars.io/cars/
 {
  "make":"chevrolet",
  "model":"Silverado 3500",
  "year": 2004,
  "vin":"1GCJK33104F173427"
  }

Response: {”data”:{“id”: “8b7138db-0c7c-4e2e-
8494-bd5daf1788e0”}
```

Bad Restful Web Service Design

- The REST endpoints are verbs (imperative)
  - Creates a link between application logic and the web service API
  - Add or changing app logic requires changing the web service
  - For example, adding a “lease” capability to the business
- All business process logic, state information and interaction with the client should be done at the application layer
- Remember the hospital, patient interactions with X-ray, etc. are accessed by the doctor
- The “application” is “hospital visit”
  - The rest objects are the documents sent from department to department.

Bad Restful Web Service Design

```
GET http://api.coolcars.io/cars/

GET http://api.coolcars.io/cars/sell/{id}

GET http://api.coolcars.io/cars/buy/{id}

POST http://api.coolcars.io/cars/trade
{
  "to ": {
    "make": "chevrolet",
    "model": "Silverado 3500",
    "year": 2004,
    "vin": "1GCJK33104F173427"
  },
  "from": {
    "make": "honda",
    "model": "civic",
    "year": 1990,
    "vin": "2HGED6349LH506746"
  }
}
```

Web Service vs Microservice

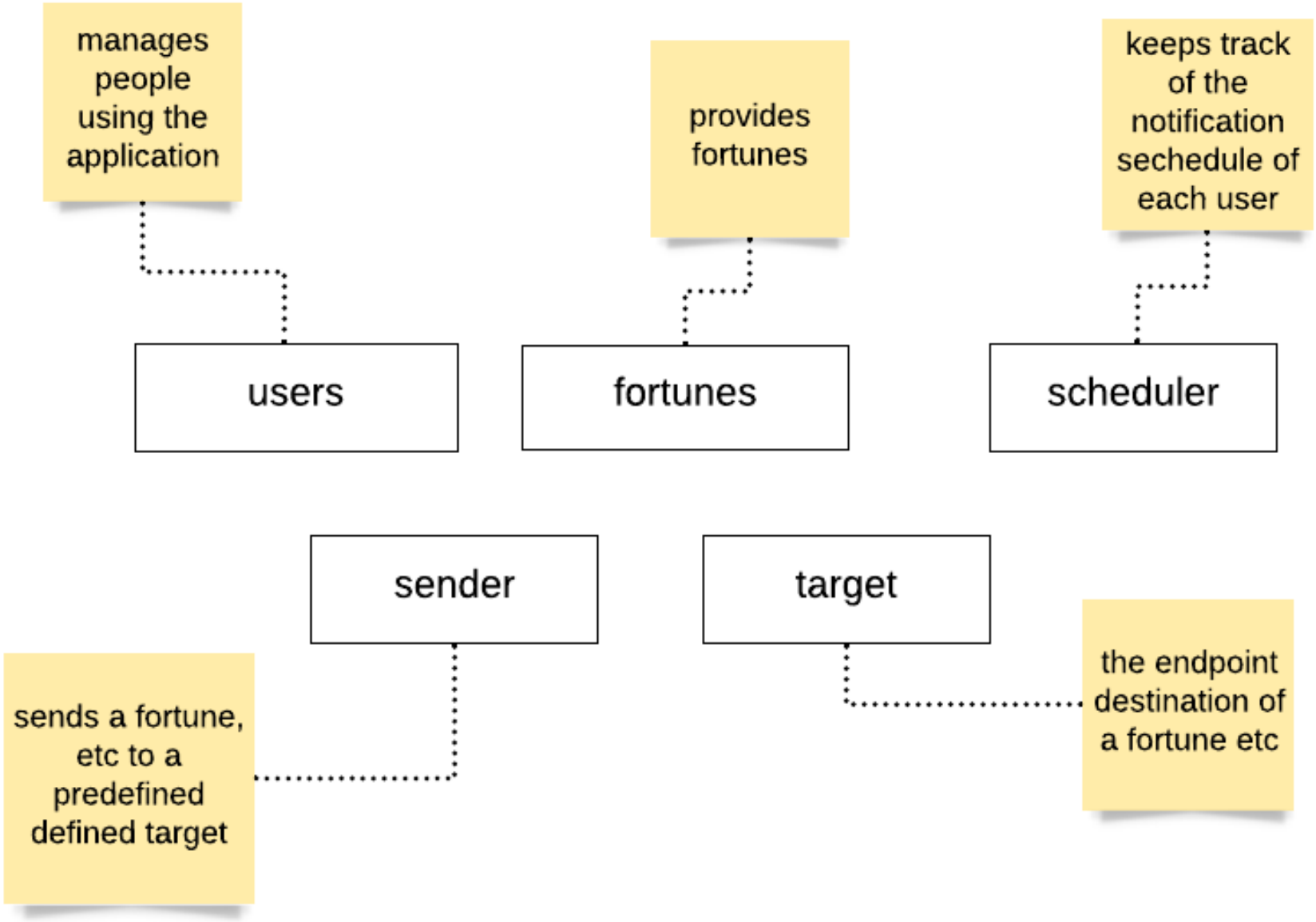
- Web services are applications that use web protocols
  - Usually REST but can also include other technologies like SOAP
  - Defined by how clients interact with them
- Microservices are an architectural pattern
  - Can use any technology, not just web protocols
  - Defined by its structure and behavior
- Microservices are often build as web services because the technology is easy to deploy as a microservice
- The 12 factor app principles are guidelines for converting web services to microservices

12 Factor App for Microservices

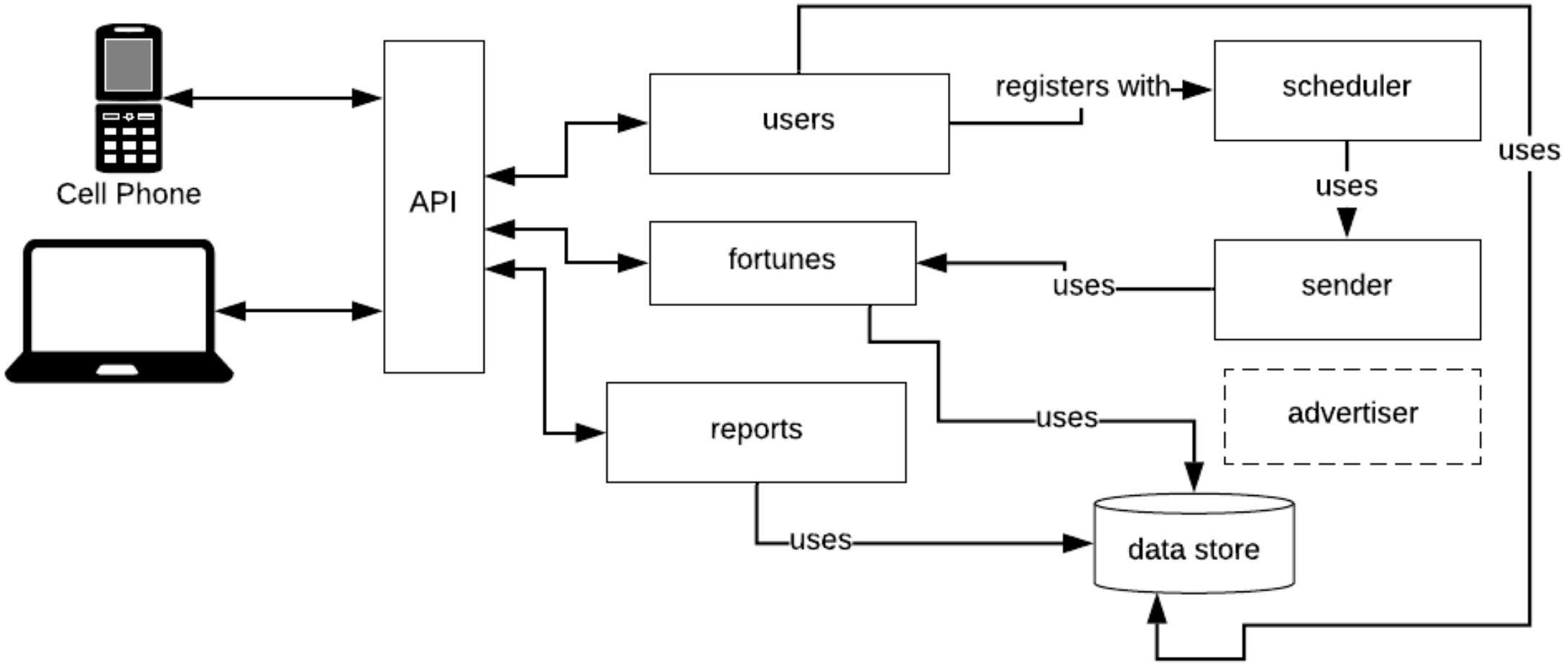
Factor	Description
I. Codebase	One codebase tracked in revision control, many deploys
II. Dependencies	Explicitly declare and isolate dependencies
III. Config	Store config in the environment
IV. Backing services	Treat backing services as attached resources
V. Build, release, run	Strictly separate build and run stages
VI. Processes	Execute the app as one or more stateless processes
VII. Port binding	Export services via port binding
VIII. Concurrency	Scale out via the process model
IX. Disposability	Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity	Keep development, staging, and production as similar as possible
XI. Logs	Treat logs as event streams
XII. Admin processes	Run admin/management tasks as one-off processes

Microservice Styles

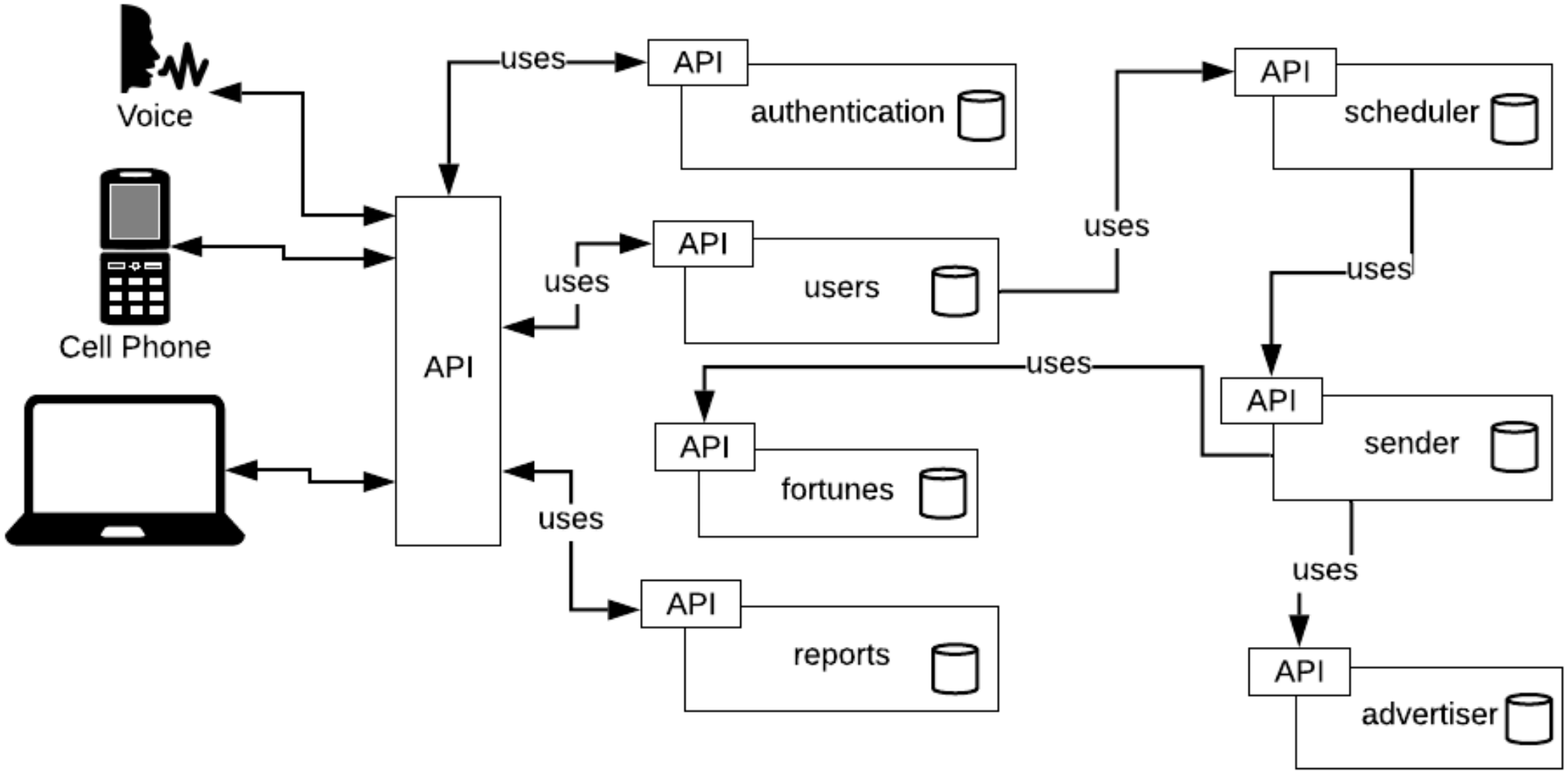
- Synchronous
- Asynchronous
- Hybrid



Monolithic Application



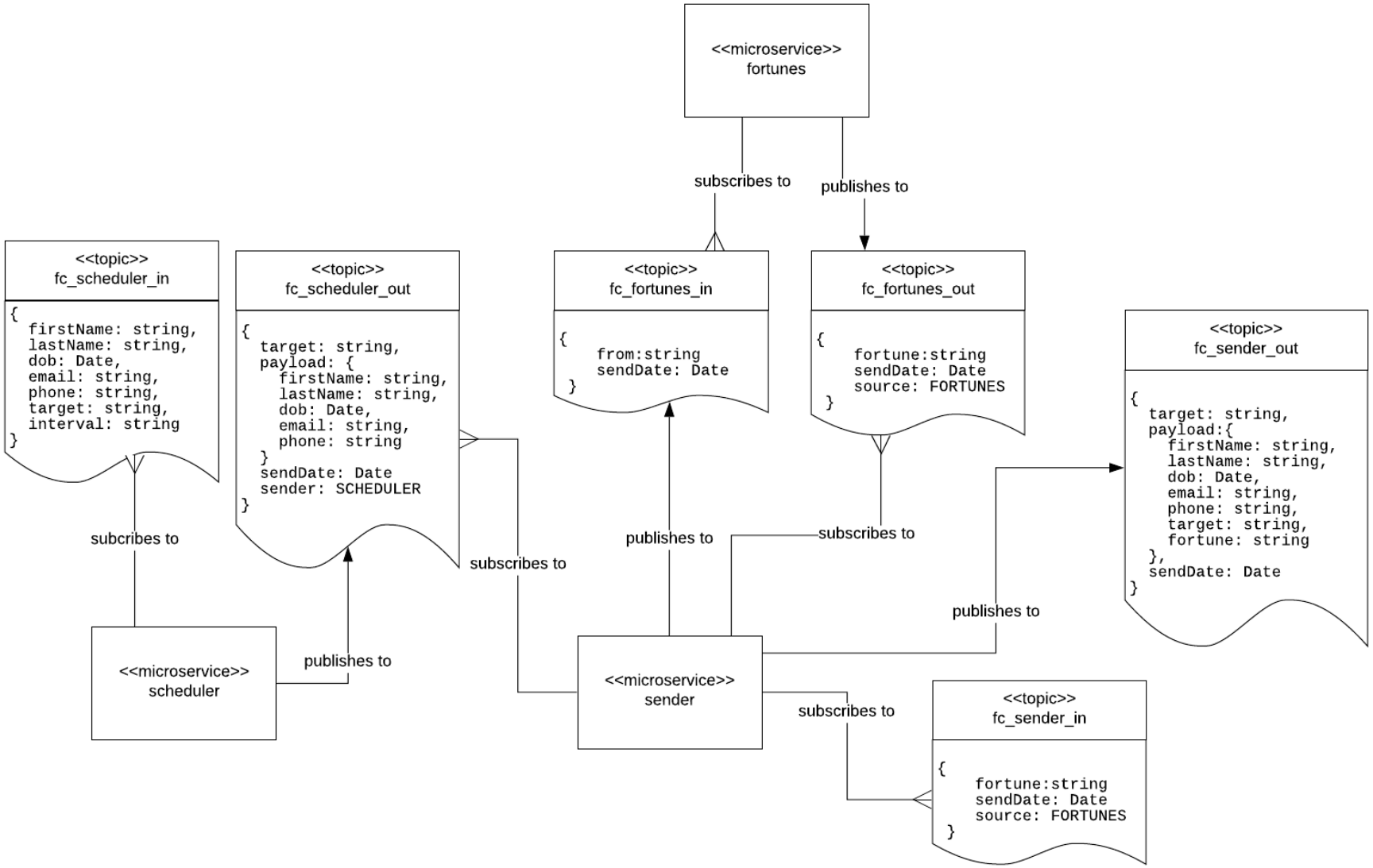
Microservices: Synchronous



Microservices: Synchronous

- Advantages:
  - Service is stateless (relative to caller)
  - Allows many active instances
  - Lower management overhead since no message server
  - Immediate and direct feedback (success or failure)
- Disadvantages:
  - Caller must wait, potentially limiting caller scalability
  - Can add hard dependencies between services if chained

Microservices: Asynchronous



Microservices: Asynchronous

- Advantages:
  - Improves scalability since no coupling between caller and service instance
  - Queueing allows for degree of inherent load balancing
- Disadvantages:
  - Dependent on external message server
  - Additional tuning and monitoring is required

Microservices: Hybrid

- Advantages:
  - Best of both worlds
  - Flexibility in terms of developer experience
  - Easy to implement as an API
- Disadvantages:
  - Hard to implement as a microservices
  - There’s no free lunch, you must contend with more latency issues
- Most system are hybrid
  - Some components should be synchronous (user login, payment)
  - Others should be queues (shipping requests, order fulfillment)
  - Part of the optimal design is getting the mix right

The 12 Factor App Methodology

- Methodology for building SaaS apps (software as a service)
  - Drafted by developers at Heroku
  - First presented by Adam Wiggins circa 2011
- Applies to building SaaS apps that:
  - Use declarative formats for setup automation
  - Have a clean contract with the underlying operating system (portable)
  - Are suitable for deployment on modern cloud platforms
  - Minimize divergence between development and production (DevOps)
  - Scale up without significant changes to tooling, architecture, or development practices

The 12 Factor App Methodology

- Methodology for building SaaS apps (software as a service)
  - Drafted by developers at Heroku
  - First presented by Adam Wiggins circa 2011
- Applies to building SaaS apps that:
  - Use declarative formats for setup automation
  - Have a clean contract with the underlying operating system (portable)
  - Are suitable for deployment on modern cloud platforms
  - Minimize divergence between development and production (DevOps)
  - Scale up without significant changes to tooling, architecture, or development practices


End Module