**Escaping Monolith Hell**

* **The architecture of the FTGO application:**

**Diagram

Description automatically generated**

The FTGO application has a hexagonal architecture (which is an architectural style).

In hexagonal architecture, the core of the application consists of business logic. Surrounding the business logic are various adapters that implement UI and integrate with external systems.

The business module consists of modules, each of which is a collection of domain objects. Examples of modules include Order Management, Delivery Management, Billing and Payments.

There are several adapters that interface with the external systems. Some are inbound adapters, which handle requests by invoking the business logic, including the REST API and Web UI adapters.

Others are outbound adapters which enable the business logic to access the MySQL database and invoke cloud services such as Twilio and Stripe.

Despite having a logically modular architecture, the FTGO application is packaged as a single WAR file. The application is an example of widely used monolithic style of software architecture, which structures a system as a single executable or deployable component.

* **The benefits of the monolithic architecture:**

In the early days of FTGO, when the application was relatively small, the application’s monolithic architecture had lots of benefits.

Simple to develop:

IDES and other developer tools are focused on building a single application.

Easy to make radical changes to the application:

You can change the code and the database schema, build and deploy.

Straightforward to test:

The developers wrote end-to-end tests that launched the application, invoked the REST API and tested the UI with selenium.

Straightforward to deploy:

All a developer had to do was copy the WAR file to a server that had tomcat installed.

Easy to scale:

FTGO ran multiple instances of the application behind a load balancer.

Over time, though, development, testing, deployment and scaling become much more difficult.

* **Living in monolithic hell:**

Successful applications like the FTGO application have a habit of outgrowing the monolithic architecture.

Each sprint, the FTGO development team implemented a few more stories, which made the code base larger.

Moreover, as the company became more successful, the size of the development team steadily grew.

Not only did this increase the growth rate of the code base, it also increased the management overhead.

Diagram

Description automatically generated

Following are the list of management problems:

Complexity intimidates developer

As it is large and complex, it will be tough to understand fully by developer.

Development is slow.

Path from commit to deployment is long and arduous

Scaling is difficult.

Delivering a reliable monolith is challenging.

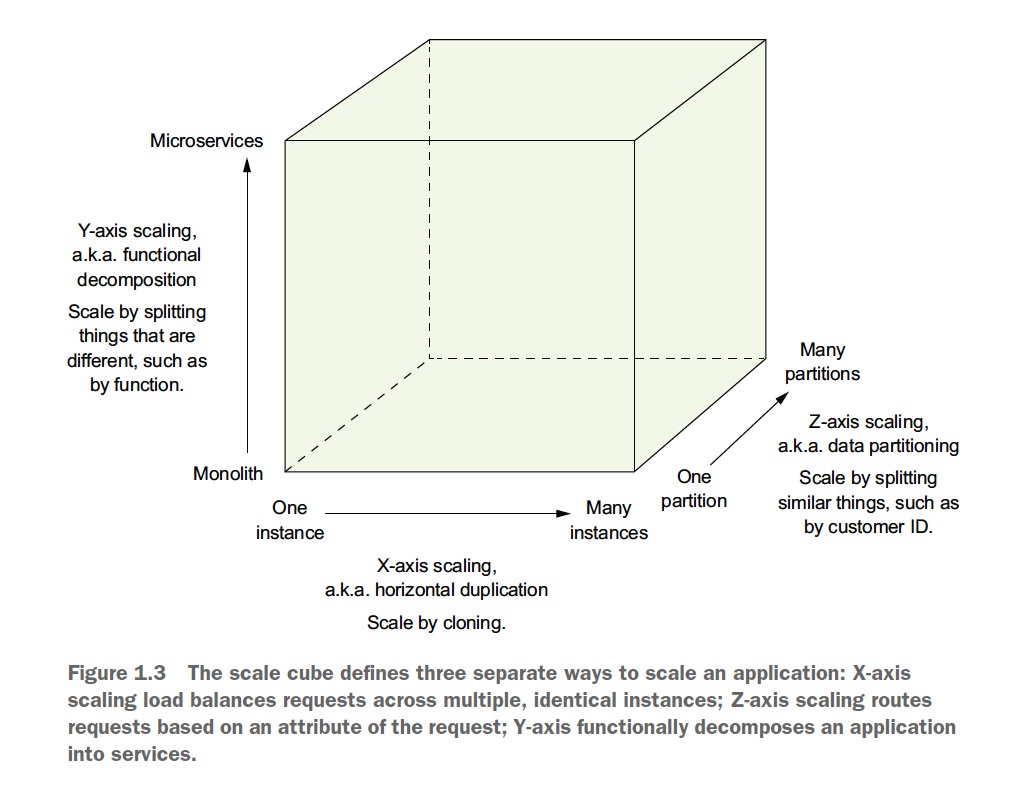
Locked into increasingly obsolete technology stack.

* **Scale cube and Microservices:**

My definition of the microservice architecture is inspired by Martin Abbot and Michael Fisher’s excellent book.

The Art of Scalability

This book describes a useful, three-dimensional scalability model. The “scale cube” shown below.



This model defines three ways to scale an application: X, Y, and Z.

**X-axis scaling load balances requests across multiple instances:**

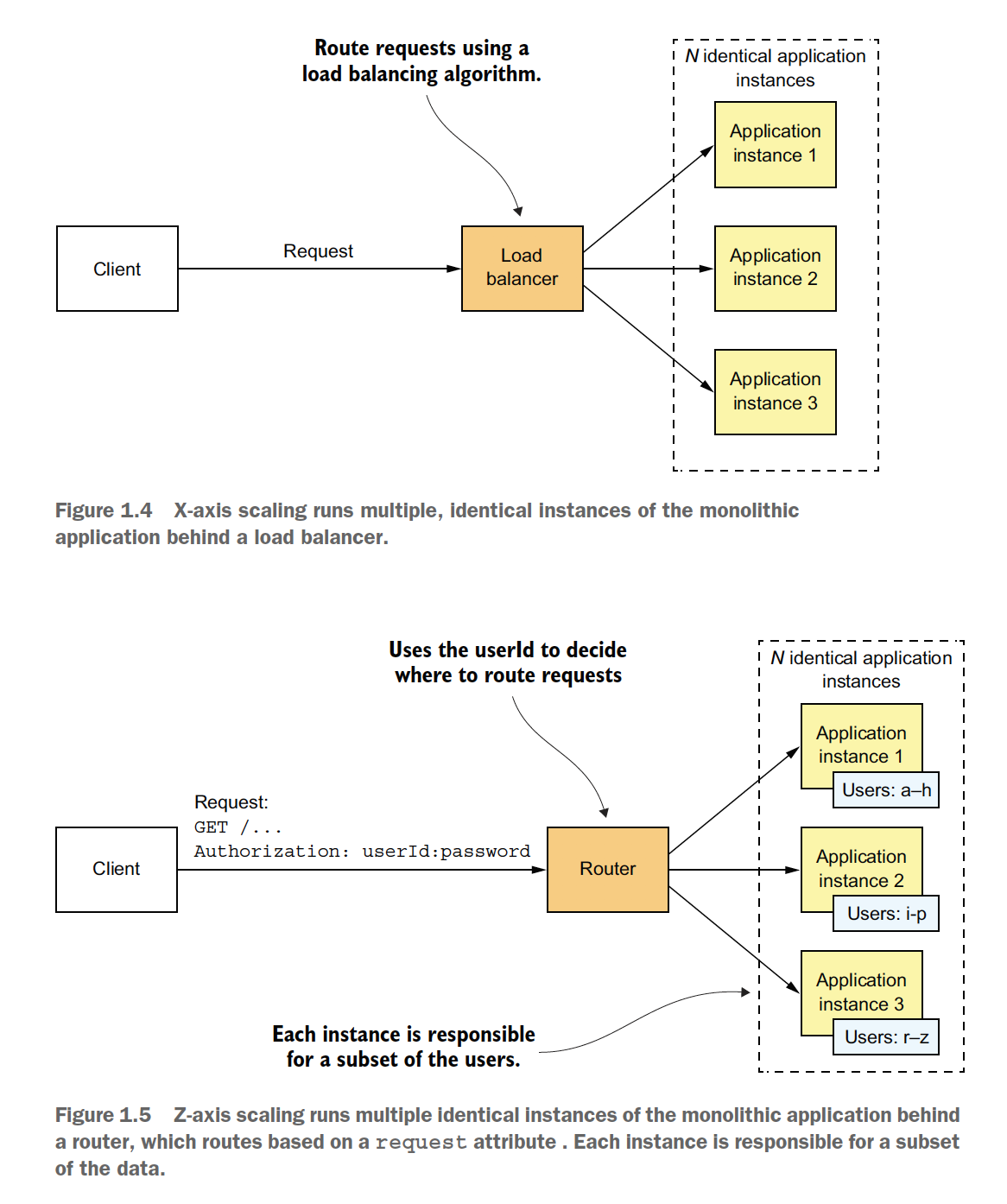
X-axis scaling is a common way to scale a monolithic application. You run multiple instances of the application behind the load balancer. The load balancer distributes requests among the N identical instances of the application. This is a great way of improving the capacity and availability of an application.

**Z-axis scaling routes requests based on an attribute of the request:**

Z-axis scaling also runs multiple instances of the monolith application, but unlike x-axis scaling, each instance is responsible for only a subset of the data.

The router in front of the instances uses a request attribute to route it to the appropriate instance. An application might, for example, route requests using userId.

In this example, each instance is responsible for a subset of users. The router uses the userId specified by the request Authorization header to select one of the N identical instances of the application. Z-axis scaling is a great way to scale an application to handle increasing transaction and data volumes.



**Y-axis scaling functionally decomposes an application into services:**

X-axis and z-axis scaling improve the application’s capacity and availability. “***But neither of the approach solves the problem of increasing development and application complexity***”.

To solve those, you need to apply Y-axis scaling, or “***functional decomposition***”.

Below figure shows how Y-axis scaling works: by splitting a monolithic application into a set of services.

Diagram

Description automatically generated

A service is a mini application that implements narrowly focused functionality, such as order management, customer management and so on. A service is scaled using X-axis scaling, though some services may also use Z-axis scaling. For example, the order service consists of a set of load-balanced service instances.

The high level definition of microservice architecture (microservices) is an architectural style that functionally decomposes an application into a set of services. Note, that this definition doesn’t say anything about size. Instead, what matters is that each service has a focused, cohesive set of responsibilities. Later we will discuss what this means.

Now let’s look at how the microservice architecture is a form of modularity.

* **Microservices as a form of modularity:**

Modularity is essential when developing large, complex applications. A modern application like FTGO is too large to develop by an individual. It’s also too complex to be understood by a single person.

Applications must be decomposed into modules that are developed and understood by different people. In monolithic application, modules are defined using a combination of programming language constructs (such as Java package) and build artifacts (such as Java JAR files). However, as the FTGO developers have discovered, this approach tends not to work well in practice. Long lived, monolithic applications usually degenerate into big balls of mud.

The microservice architecture uses services as the unit of modularity. A service has an API, which is an impermeable boundary that is difficult to violate. You can’t bypass the API and access an internal class as you can with Java package. As a result, it is much easier to preserve the modularity of the application over time. There are other benefits of using services as a building blocks, including the agility to deploy and scale them independently.

* **Each service has its own database:**

A key characteristics of microservice architecture is that the services are loosely coupled and communicate via API’s. One way to achieve loose coupling is by each service having its own data store.

In the online store, for example Order Service has a database that includes the ORDERS table, and Customer Service has its database, which includes CUSTOMERS table.

At development time, developers can change a service’s schema without having to co-ordinate with developers working on other services.

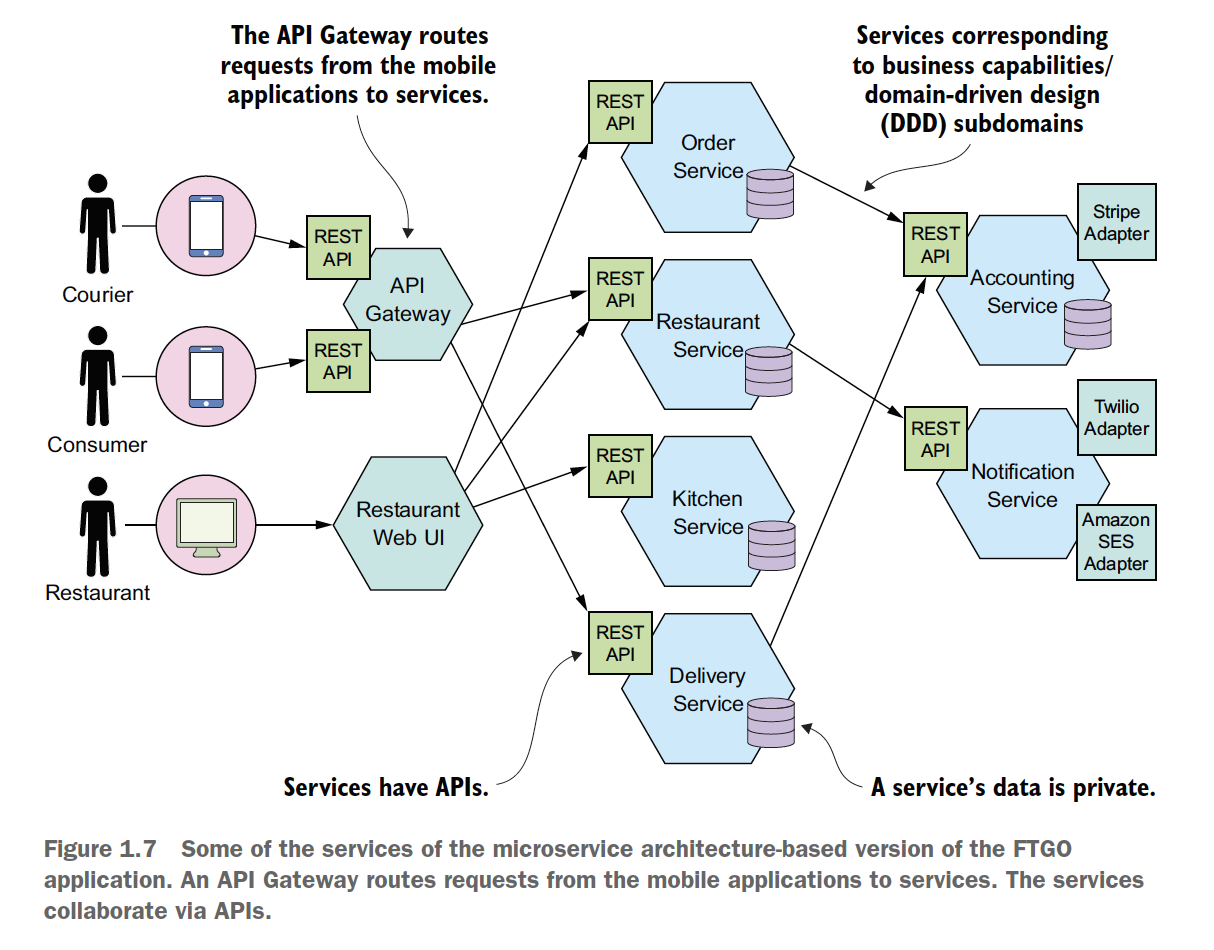
At runtime, the services are isolated from each other – one service will never be blocked because another service holds a database lock.

Note:

The requirement for each service to have its own database doesn’t mean it has its own database server. You don’t, for example, have to spend 10 times more on Oracle RDBMS licenses.

* **The FTGO microservice architecture:**

If we apply Y-axis decomposition to the FTGO application, we get the architecture shown in figure.



The decomposed application consist of numerous frontend and backend services. We would also apply X-axis and, possibly Z-axis scaling, so that at runtime there would be multiple instances of each service.

The frontend services include an API gateway and the restaurant web UI. The API gateway plays the role of a façade and is described detail in upcoming chapters provides the REST APIs that are used by consumer’s and courier’s mobile applications.

The restaurant web UI implements the web interface that’s used by the restaurants to manage menus and process orders.

The FTGO application’s business logic consists of numerous backend services. Each backend service has a REST API and its own private datastore.

The backend services include the following.

Order Service – Manages orders

Delivery Service – Manages delivery of orders from restaurants to

Consumers.

Restaurant Service – Maintains information about restaurants.

Kitchen Service – Manages the preparation of orders.

Accounting Service – Handles billing and payments.

Each service and its API are clearly defined. Each one can be independently developed, tested, deployed, and scaled. Also, this architecture does a good job of modularity.

* **Comparing Microservice architecture and SOA:**

Some critics of the microservices architecture claim it’s nothing new – its service oriented architecture. At a high level there are some similarities.

SOA and microservice architecture are architectural styles that structure a system as a set of services.

Once you dig deep, you encounter significant differences.

|  |  |  |
| --- | --- | --- |
|  | SOA | Microservice |
| Inter-service communication | Smart pipes, such as Enterprise Service Bus, using heavy weight protocols, such as SOAP and other ws-\* standards. | Dumb pipes, such as a message broker, or direct service to service communication, using lightweight protocols such as REST or gRPC. |
| Data | Global data model and shared databases. | Data model and database per service. |
| Typical service | Larger monolithic application. | Smaller service. |

* **Benefits and drawbacks of the microservice architecture:**

**Benefits of microservice architecture:**

* + It enables the continuous delivery and deployment of large, complex applications.
  + Services are small and easily maintained.
  + Services are independently deployable.
  + Services are independently scalable.
  + The microservice architecture enables teams to be autonomous.
  + It allows easy experimenting and adoption of new technologies.
  + It has better fault isolation.

Diagram

Description automatically generated

**Drawbacks of the microservice architecture:**

Here are the major drawbacks and issues of the microservice architecture:

* + Finding the right set of services is challenging.
  + Distributed systems are complex, which makes development, testing and deployment difficult.
  + Deploying features that span multiple services require careful co-ordination.
  + Deciding when to adapt the microservice architecture is difficult.
* **The microservice architecture pattern language:**

Architecture and design are all about making decisions. You need to decide whether the monolithic or microservice architecture is best fit for your application. When making these decisions you have lots of trade-offs to consider. If you pick microservices architecture, you’ll need to address lots of issues.

A good way to describe the various architectural and design options and improve decision making is to use a pattern language.

* **Microservice is not a silver bullet:**

Back in 1986, Fred Brooks, author of The Mythical Man-Month, said that in software engineering, there are no silver bullets. That means there are no techniques or technologies that if adopted would give you tenfold boost in productivity. Yet decade years later, developers are still arguing passionately about their favourite silver bullets.

A lot of arguments follow the suck/rock dichotomy

<http://nealford.com/memeagora/2009/08/05/suck-rock-dichotomy.html>

a term coined by Neal Ford that describes how everything in software world either sucks or rocks, with no middle ground. These arguments have this structure: if you do X, then a puppy will die, so therefore you must do Y.

For example, synchronous vs reactive programming, object oriented vs functional, Java vs JavaScript, REST vs messaging. Of course, reality is much more nuanced.

Every technology has drawbacks and limitations that are often overlooked by its advocates.

As a result, the adoption of a technology usually follows the ***Gartner hype cycle*** <https://en.wikipedia.org/wiki/Hype_cycle>

In which emerging technology goes through five phases, including the “***peak of inflated expectations***” (it rocks), followed by the “***through of disillusionment***” (it sucks). And ending with the “plateau of productivity”. (We now understand the trade-offs and when to use it).

Microservices are not immune to the silver bullet phenomenon. Whether this architecture is appropriate for your application depends on many factor.

* **Patterns and pattern languages:**

A pattern is a re-usable solution to a problem that occurs in a particular context. It is an idea that has its origins in real-world architecture and that has proven to be useful in software architecture and design.

The concept of pattern was created by Christopher Alexander, a real-world architect. He also created the concept of a pattern language, a collection of related patterns that solves problems within a particular domain.

His book A Pattern Language describes a pattern language for architecture that consists of 253 patterns.

The patterns range from solutions to high-level problems, such as where to locate a city (“Access to water”), to low-level problems, such as how to design a room.

Christopher Alexander’s writing inspired the software community to adapt the concept of patterns and pattern languages. The book Design Patterns: Elements of Reusable Object-Oriented Software is a collection of object-oriented design patterns.

A commonly used pattern structure includes three especially valuable sections:

Forces

Resulting Context

Related Patterns

Forces: THE ISSUES THAT YOU MUST ADDRESS WHEN SOLVING A PROBLEM

The ***forces*** section of a pattern describes the force (issues) that you must address when solving a problem in given context. Forces can conflict, so it might not be possible to solve all of them. Which forces are more important depends on the context. You have to prioritize solving forces over others. For example, code must be easy to understand and have good performance. Code written in reactive style has better performance than synchronous code. Explicitly listing the forces is useful because it makes clear which issues need to be solved.

Resulting Context: THE CONSEQUENCES OF APPLYING A PATTERN

The resulting context section of a pattern describes the consequences of applying a pattern. It consists of three parts.

Benefits:

The benefits of the pattern, including the forces that have been resolved.

Drawbacks:

The drawbacks of the pattern, including the unresolved forces.

Issues:

The new problems that have been introduced by applying the pattern.

RELATED PATTERNS: THE FIVE DIFFERENT TYPES OF RELATIONSHIPS

The related patterns section of a pattern describes the relationship between the pattern and other patterns. There are five types of relationships between patterns.

Predecessor

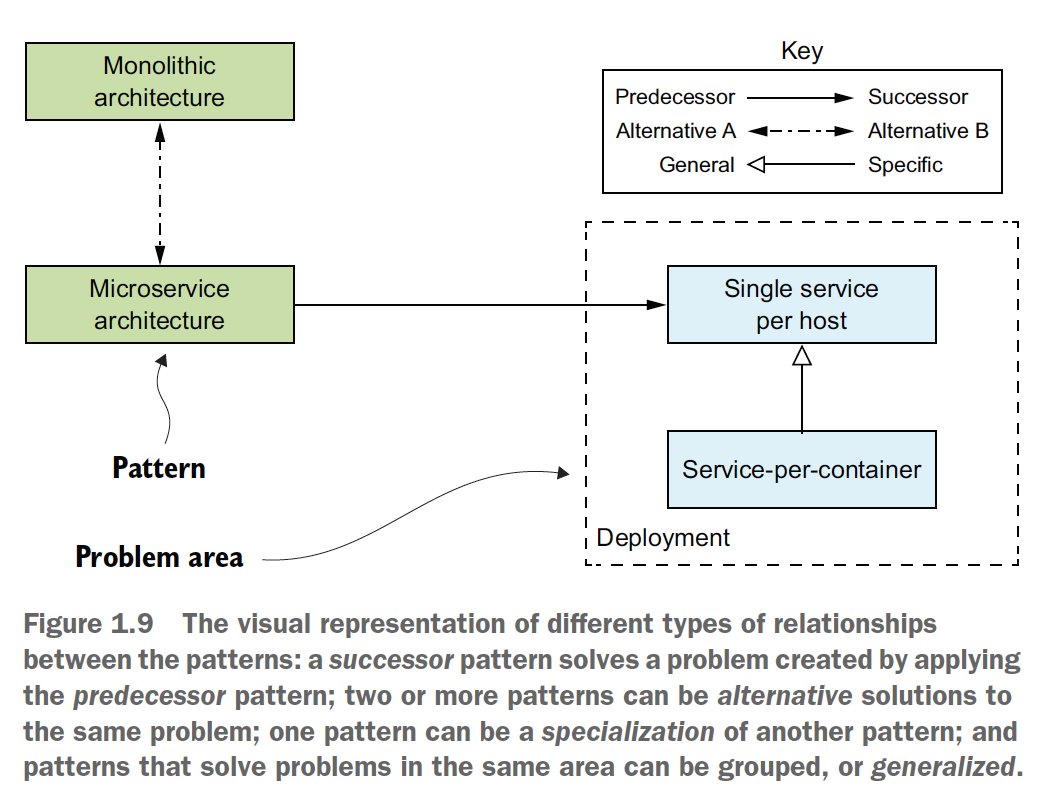
Successor

Alternative

Generalization

Specialization

In addition, you can organize patterns that tackle issues in a particular problem area into groups.

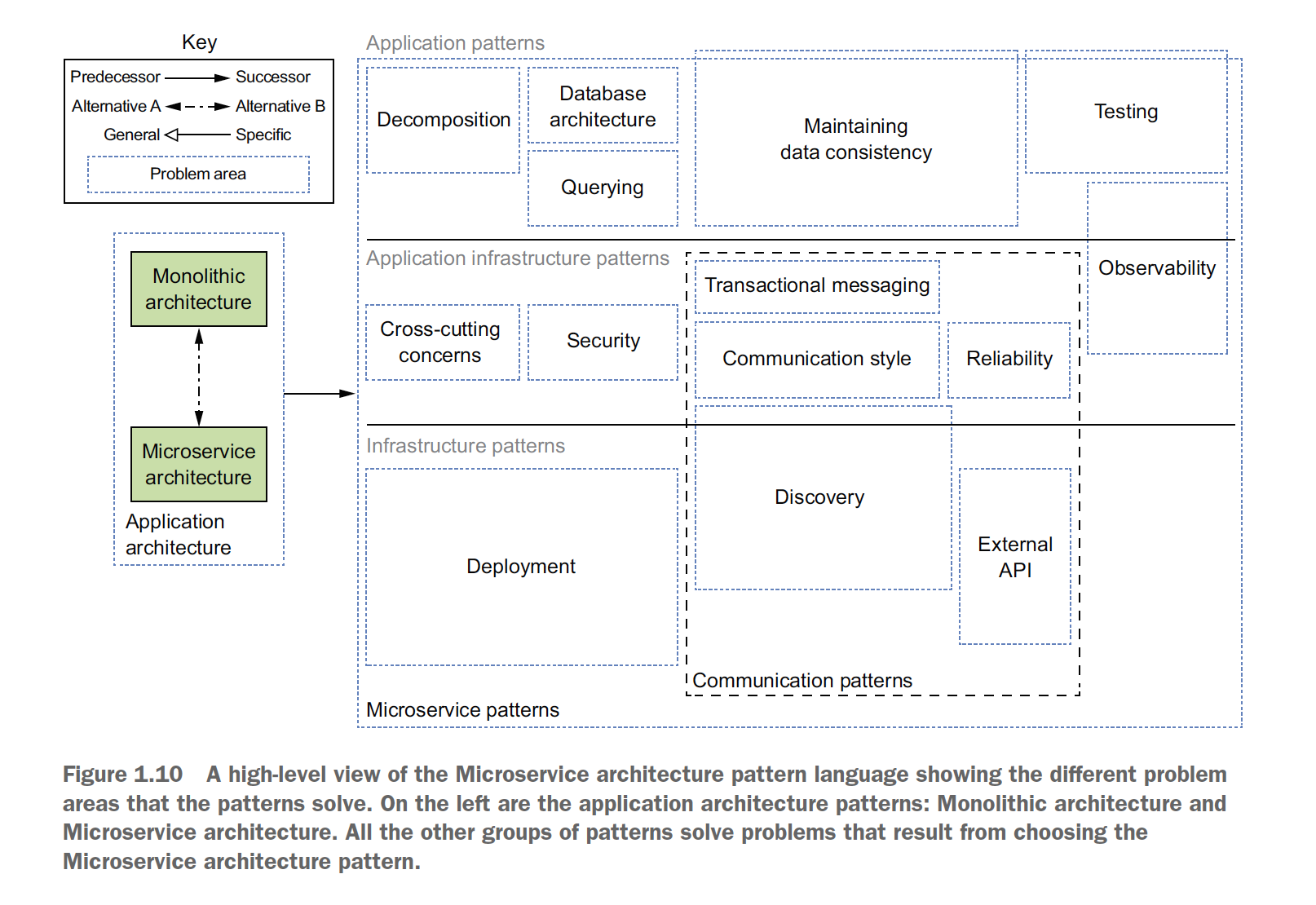


A collection of patterns related through these relationships sometimes form what is known as a pattern language. The patterns in a pattern language work together to solve problems in a particular domain. In particular Chris Richardson, created the Microservice architecture pattern language. It’s a collection of interrelated software architecture and design patterns for microservices.

* **Overview of Microservice architecture pattern language:**

The Microservice architecture pattern language is a collection of patterns that help you architect an application using the microservice architecture. Below figure shows the high-level structure of the pattern language.

The pattern language first helps you decide whether to use microservice architecture. It describes the monolithic and microservice architecture, along with their benefits and drawbacks. Then, if the microservice architecture is a good fit for your application, the pattern language helps you use it effectively by solving various architecture and design issue.



The patterns are also divided into three layers:

Infrastructure patterns:

These solves problems that are mostly infrastructure issues outside of development.

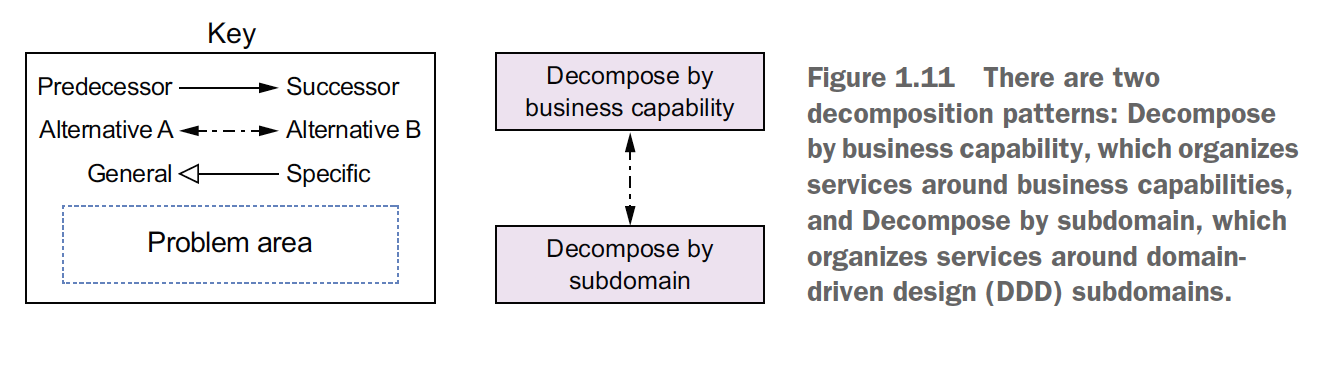
Application Infrastructure:

These are for infrastructure issues that also impact development.

Application Patterns:

These solves problems faced by developers.

**PATTERNS FOR DECOMPOSING AN APPLICATION INTO SERVICES:**



**COMMUNICATION PATTERNS:**

An application built using the microservice architecture is a distributed system. Consequently, inter-process communication is an important part of the microservice architecture.

Following are the communication patterns which are organized into five groups:

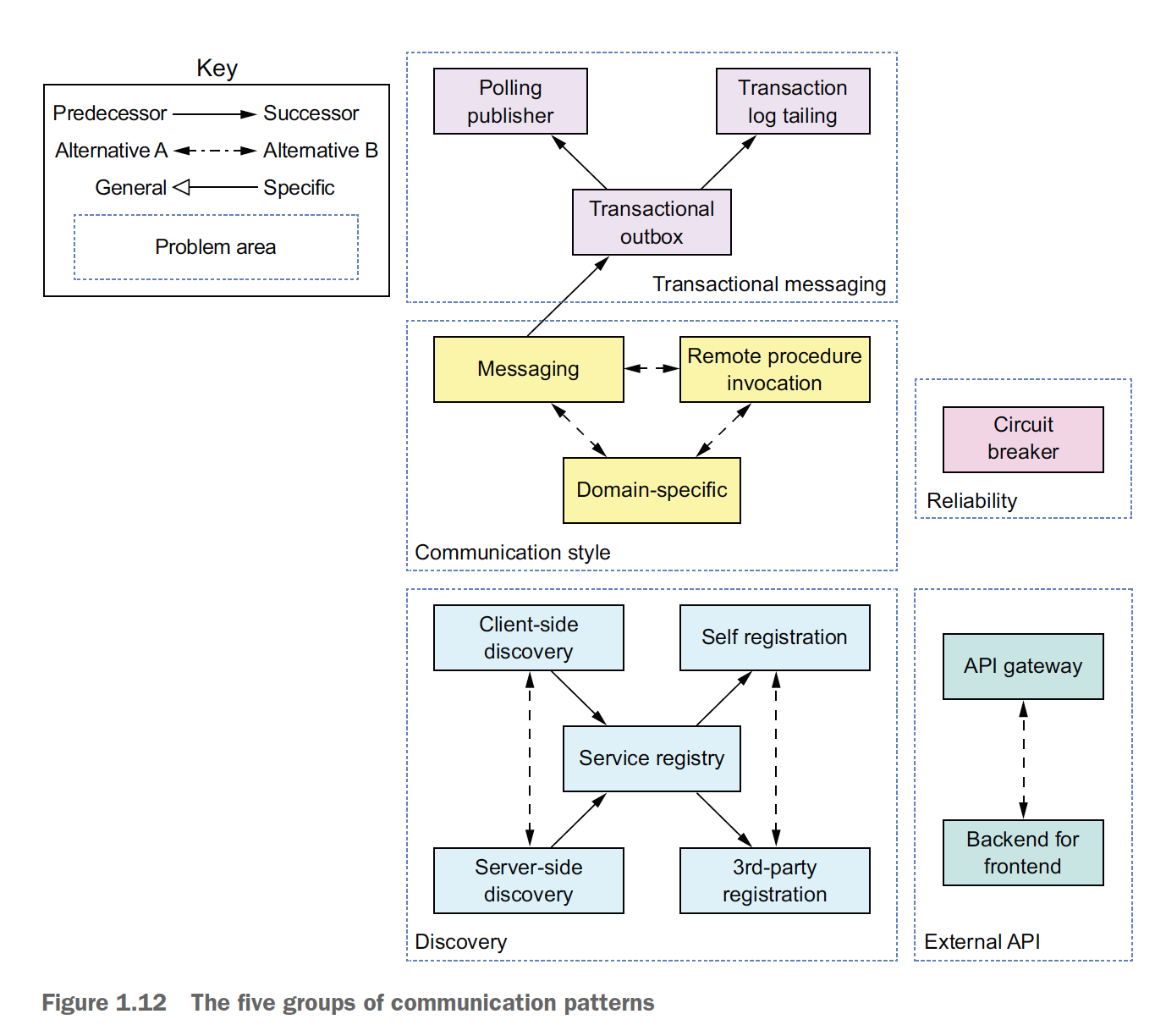
***Communication style*** – What kind of IPC mechanism should you use?

***Discovery*** – How does a client of a service determine the IP address of a service instance so that, for example, it makes an HTTP request?

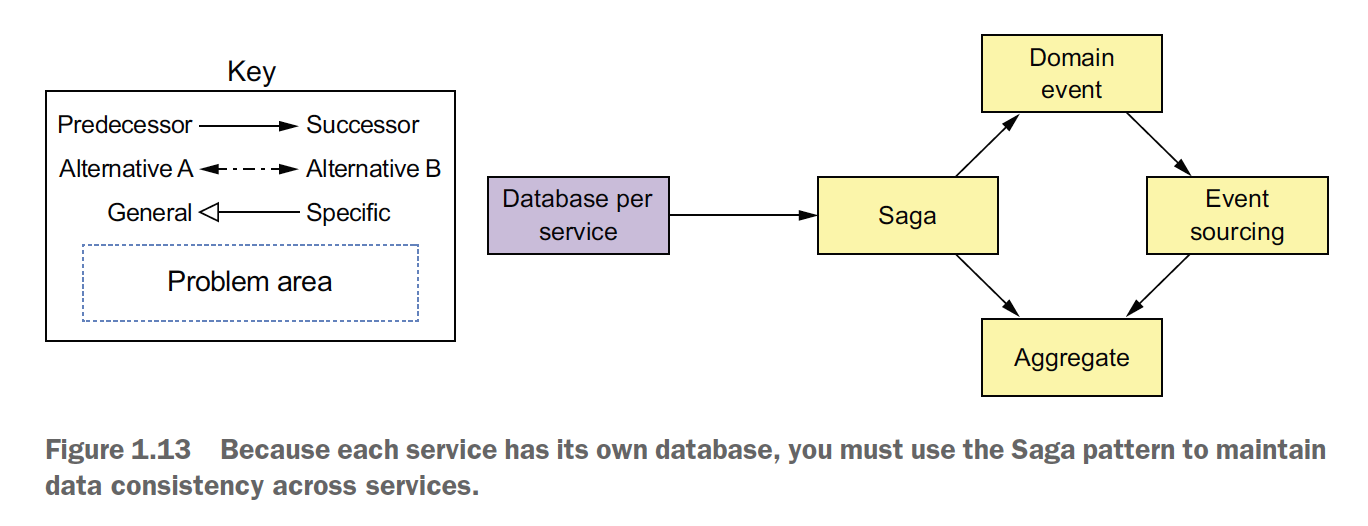
***Reliability*** – How can you ensure that communication between services is reliable even though services can be unavailable?

***Transactional Messaging*** – How should you integrate the sending of messages and publishing of events with database transactions that update business data?

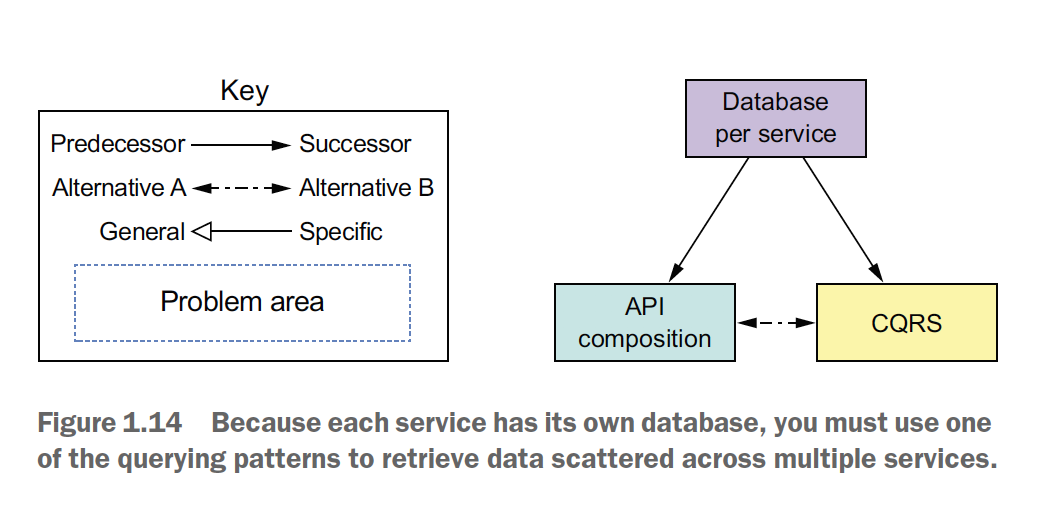
***External API*** – How do clients of your application communicate with services?



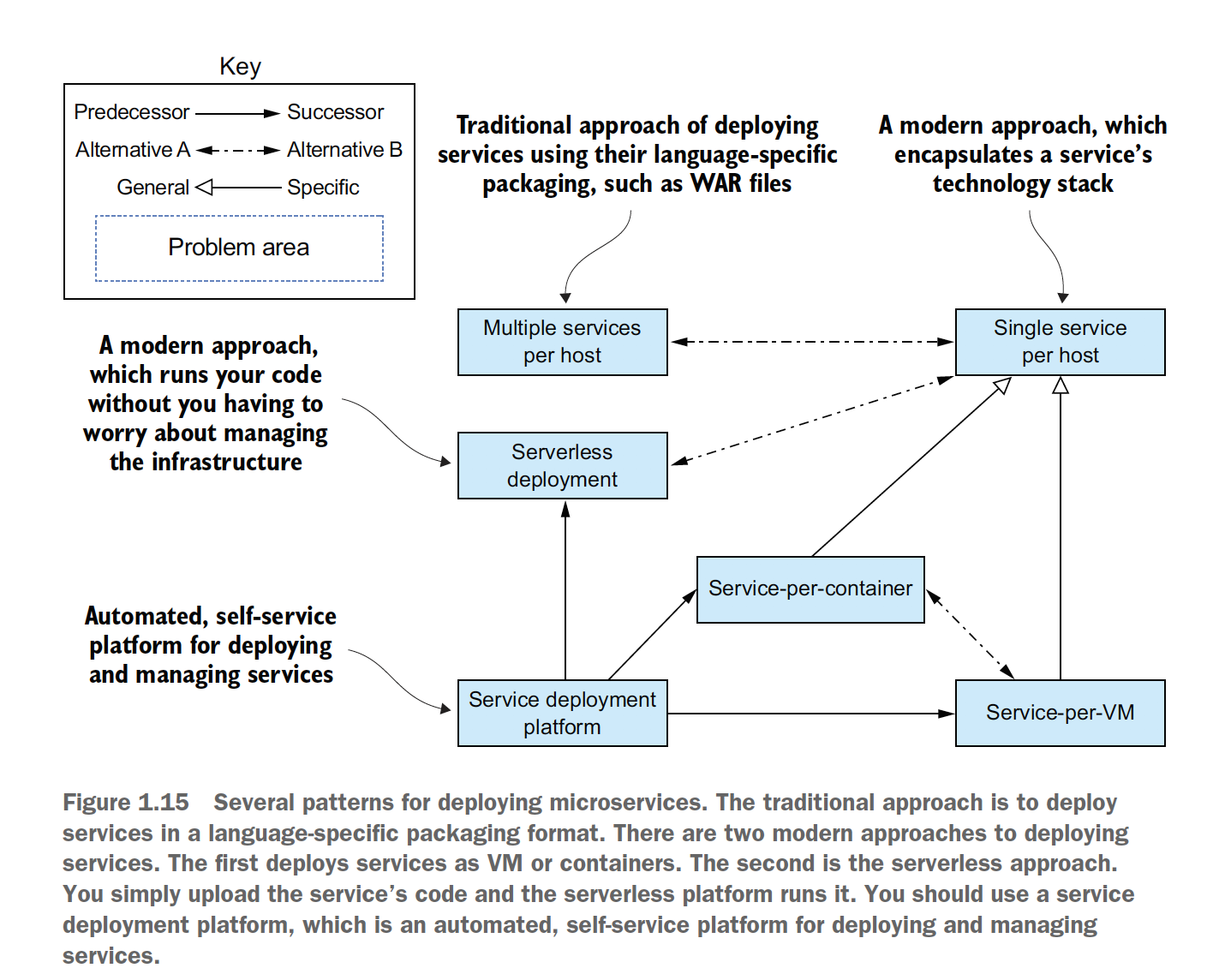
**DATA CONSISTENCY PATTERN FOR IMPLEMENTING TRANSACTION MANAGEMENT:**



**PATTERNS FOR QUERYING DATA IN A MICROSERVICE ARCHITECTURE**

****

**SERVICE DEPLOYMENT PATTERNS:**

****

**OBSERVABILITY PATTERNS PROVIDE INSIGHT INTO APPLICATION BEHAVIOR:**

You can use the following patterns to design observable services:

***Health check API*** – Expose an endpoint that returns the health of the service

***Log aggregation*** – Log service activity and write logs into a centralized logging server, which provides searching and altering.

***Distributed tracing*** – Assign each external request a unique ID and trace requests as they flow between services.

***Exception tracking*** – Report exception to an exception tracking service, which deduplicates exceptions, alerts developers, and tracks the resolution of each exception.

***Application metrics*** – Maintain metrics, such as counters and gauges, and expose them to a metrics server.

***Audit logging*** – Log user actions.

**PATTERNS FOR THE AUTOMATED TESTING OF SERVICES**

***Consumer-driven contract test*** – Verify that a service meets the expectations of its client.

***Consumer-side contract test*** – Verify that a client of a service can communicate with the service.

***Service component test*** – Test a service in isolation.

**PATTERNS FOR HANDLING CROSS-CUTTING CONCERNS**

**SECURITY PATTERNS**