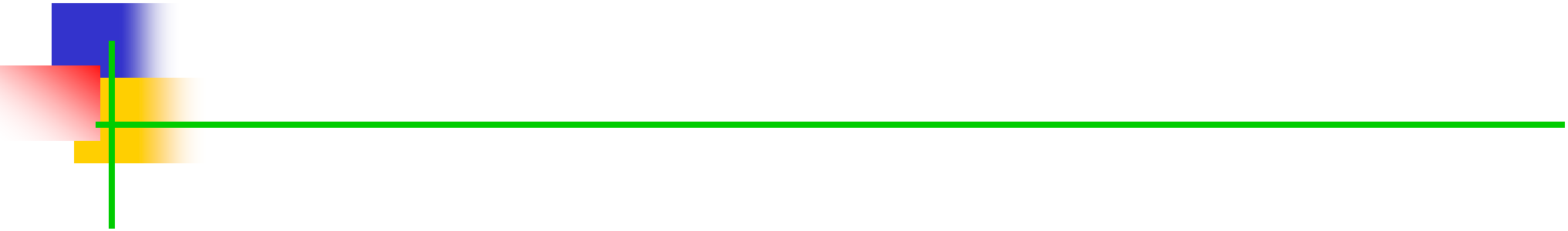
Service Oriented Architecture & Web services



Application Services Design Considerations

- State Management
- Application Servers
- **Load Balancing**

Load Balancing

- **An Overview of Load Balancing**
- Load Balancing Policies
- Elasticity
- Session Affinity

State Management (Cont.)
Example: A skier service API

- Horizontal scaling

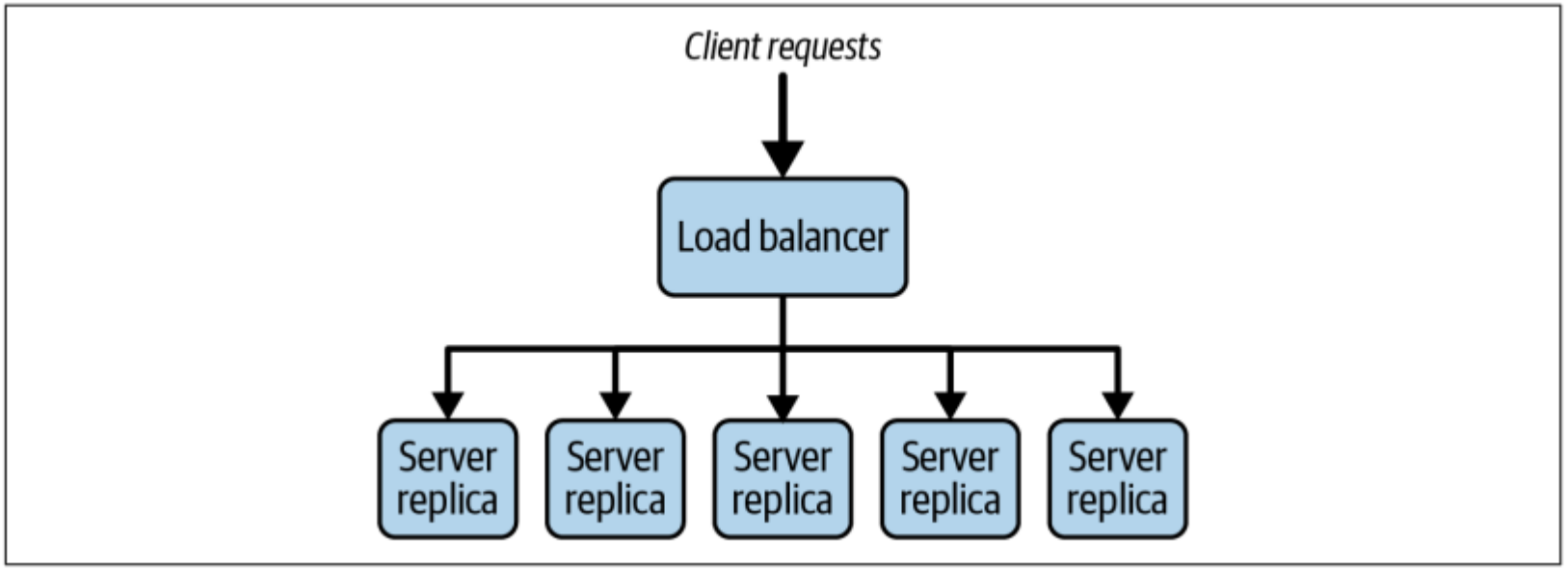


Figure 5-4. Simple load balancing example

- Load balancers may act at the *network level* or the *application level*.

Horizontal Scaling

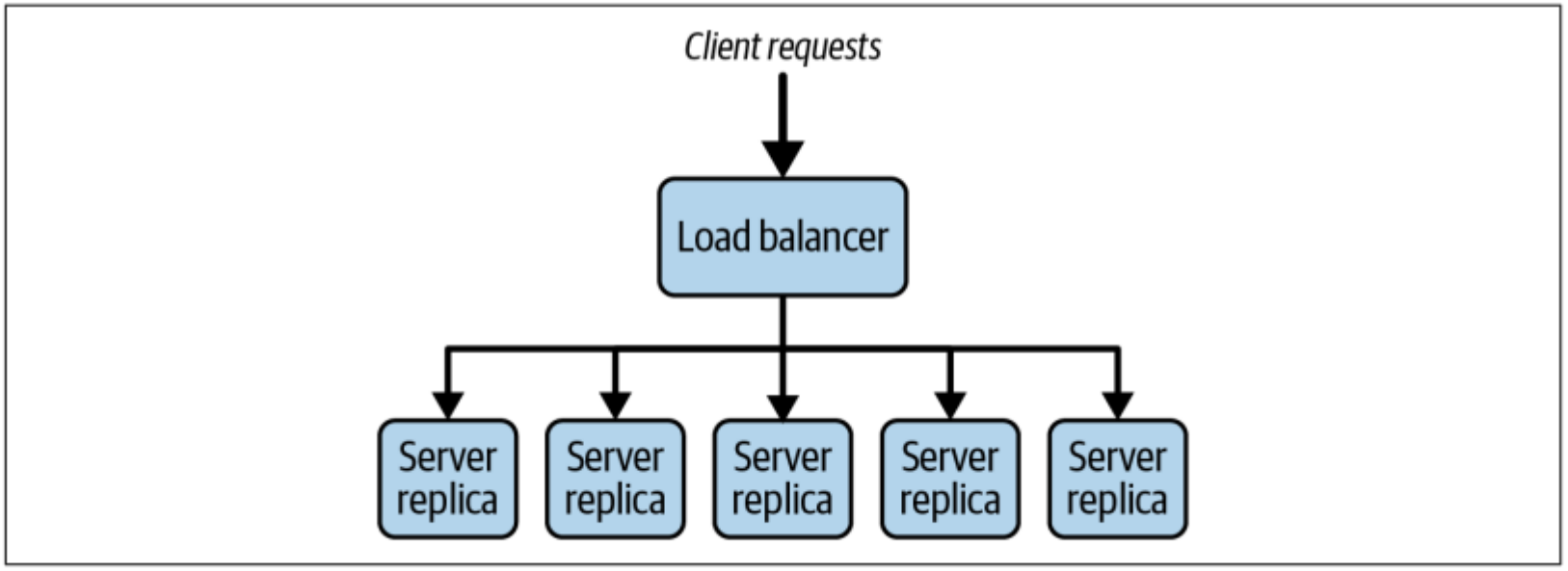


Figure 5-4. Simple load balancing example

- A core principle of scaling a system is being able to easily add new processing capacity to handle increased load.
- A simple and effective approach is deploying multiple instances of stateless server resources and using a load balancer to distribute the requests across these instances. (Horizontal scaling)

5

Load Balancing

- Load balancing aims to effectively utilize the capacity of a collection of services to optimize the response time for each request.
- This is achieved by distributing requests across the available services to ideally utilize the collective service capacity.
- The objective is to avoid overloading some services while underutilizing others.
- Load balancing acts at the network level or the application level.

6

Load Balancing

- Network-level load balancers distribute requests at the network connection level, operating on individual TCP or UDP packets.
  - Routing decisions are made on the basis of client IP addresses.
  - Once a target service is chosen, the load balancer uses a technique called network address translation (NAT).
  - This changes the destination IP address in the client request packet from that of the load balancer to that of the chosen target.
  - Network load balancers are relatively simple as they operate on the individual packet level
  - This means they are extremely fast (Why?)

7

Load Balancing

- Application-level load balancers reassemble the complete HTTP request and base their routing decisions on the values of the HTTP headers and on the actual contents of the message.
- For example, a load balancer can be configured to send all POST requests to a subset of available services, or distribute requests based on a query string in the URI.
- They are slower than network-level load balancers.
- Let us check the variance in performance between them…

8

Network load balancer delivers on average around 20% higher performance for the 32, 64, and 128 client tests

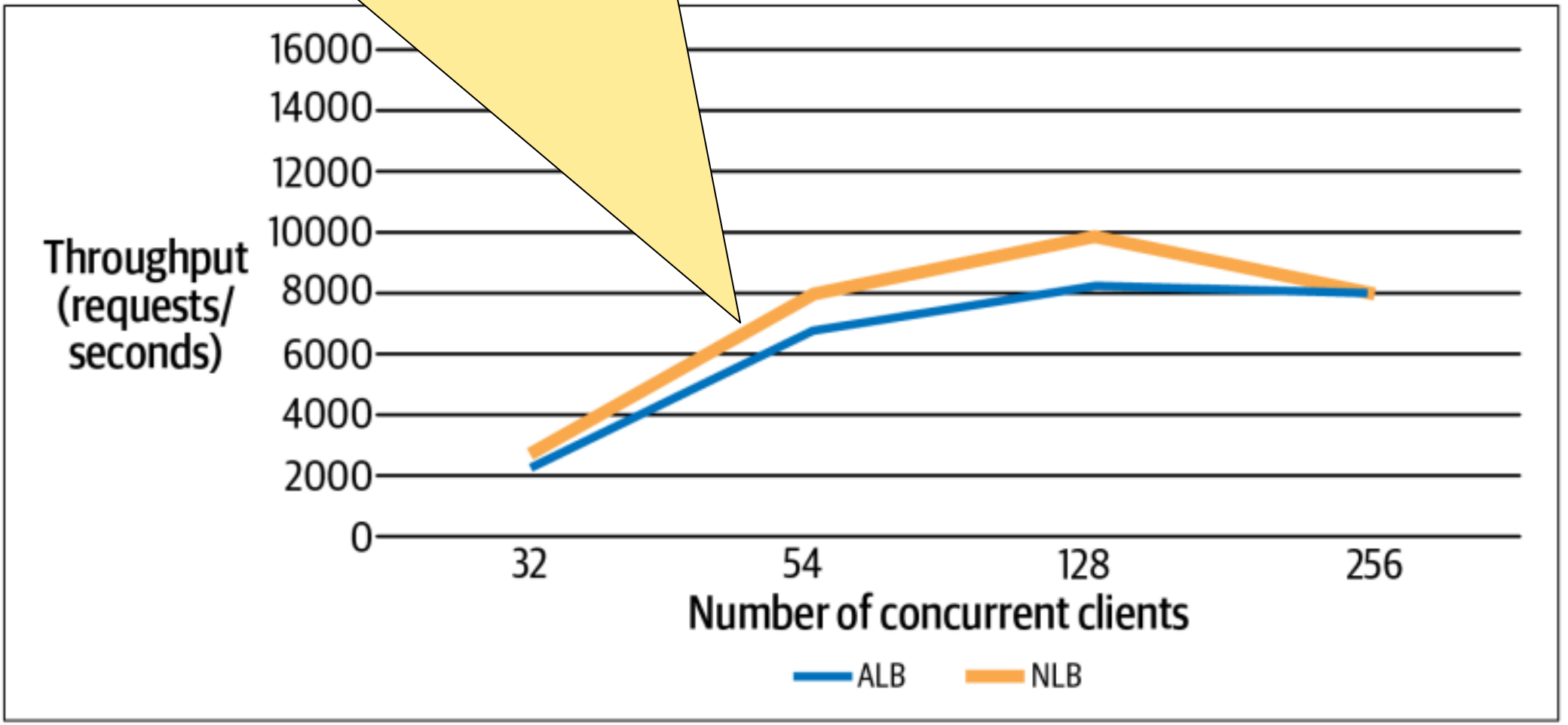


Figure 5-5. Comparing load balancer performance[4]

For 256 clients, the performance of the two load balancers is essentially the same. This is because the capacity of the 4 replicas is exceeded and the system has a bottleneck

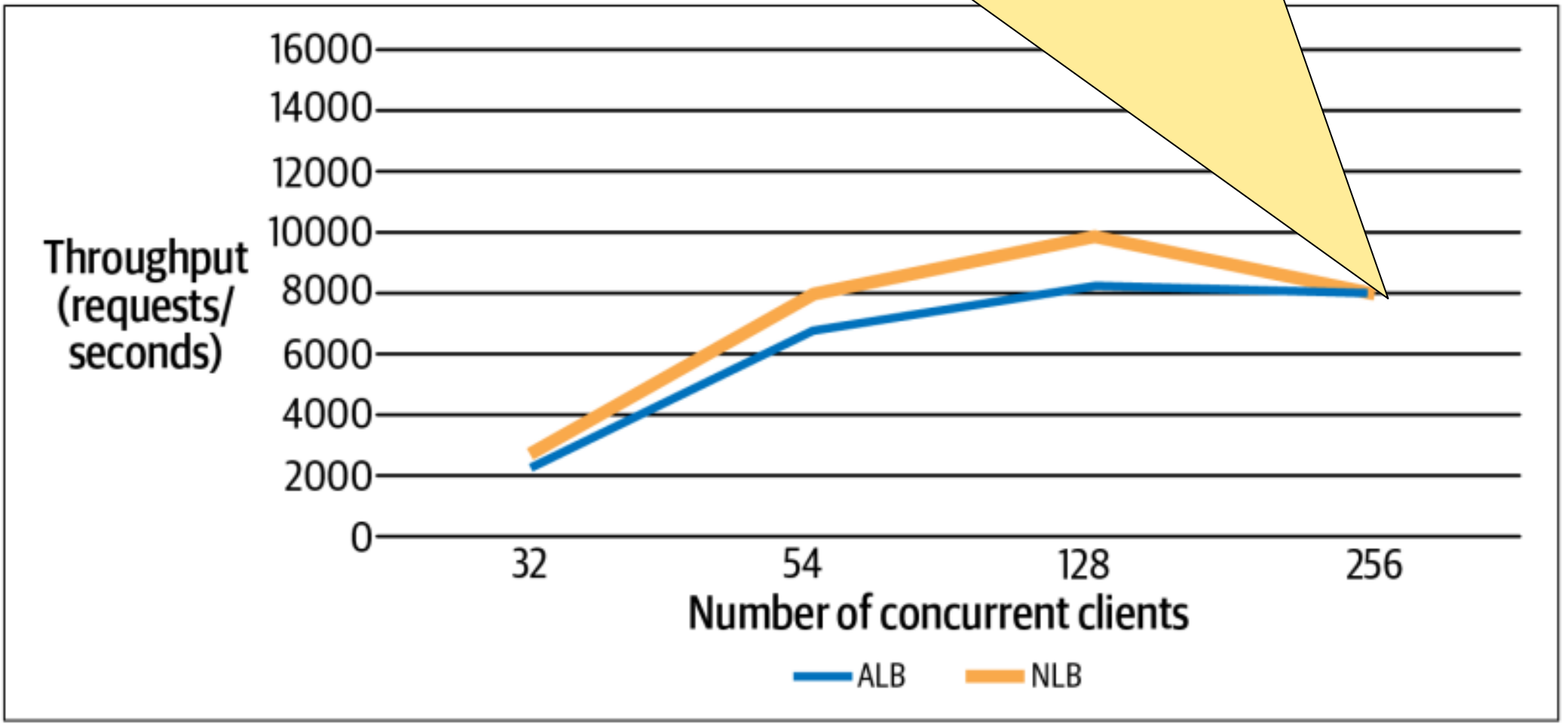


Figure 5-5. Comparing load balancer performance[4]

Load Balancing

- An Overview of Load Balancing
- **Load Balancing Policies**
- Elasticity
- Session Affinity

Load Balancing Policies

- Load distribution policies dictate how the load balancer chooses a target service to process a request.
- The following are four of the most commonly supported across all load balancers:
  - *Round robin:* The load balancer distributes requests to available servers in a round-robin fashion.
  - *Least connections:* The load balancer distributes new requests to the server with the least open connections.
  -

Load Balancing Policies

- The following are four of the most commonly supported across all load balancers:
  - *HTTP header field:* The load balancer directs requests based on the contents of a specific HTTP header field. For example, all requests with the header field X-ClientLocation:US,Seattle could be routed to a specific set of servers.
  - *HTTP operation:* The load balancer directs requests based on the HTTP verb in the request

Load Balancing

- An Overview of Load Balancing
- Load Balancing Policies
- **Elasticity**
- Session Affinity

Load Balancing: Elasticity

- *Elasticity* is the capability of an application to dynamically provision new service capacity to handle an increase in requests.
- As load increases, new replicas are started and the load balancer directs requests to these.
- As load decreases, the load balancer stops services that are no longer needed.
- Scaling policies can be defined to determine when to scale up and down.
- Example: “Capacity for a service should be increased when the average service CPU utilization across all instances is over 70%, and decreased when average CPU utilization is below 40%”.

Load Balancing: Elasticity

AWS Auto Scaling groups

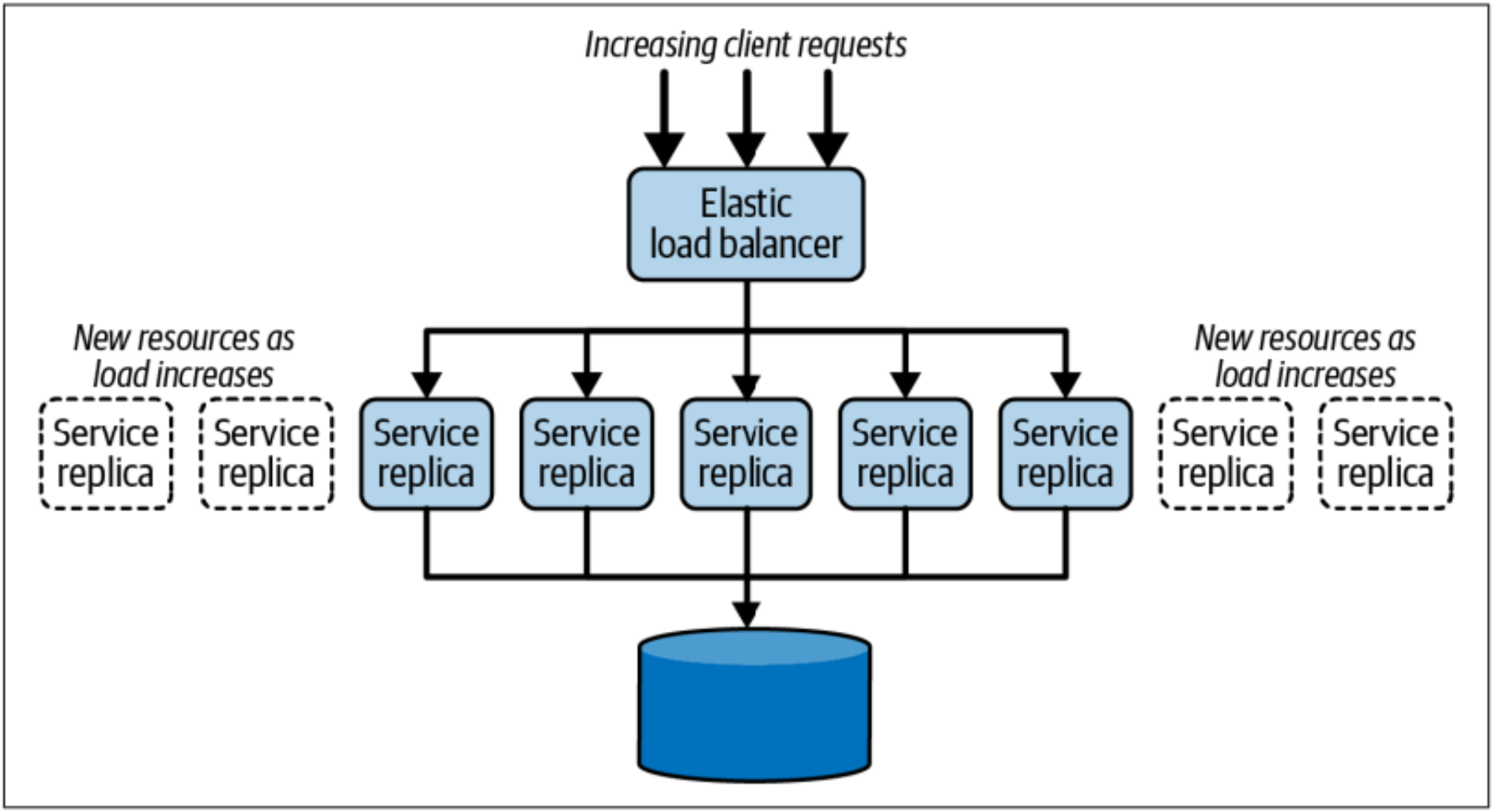
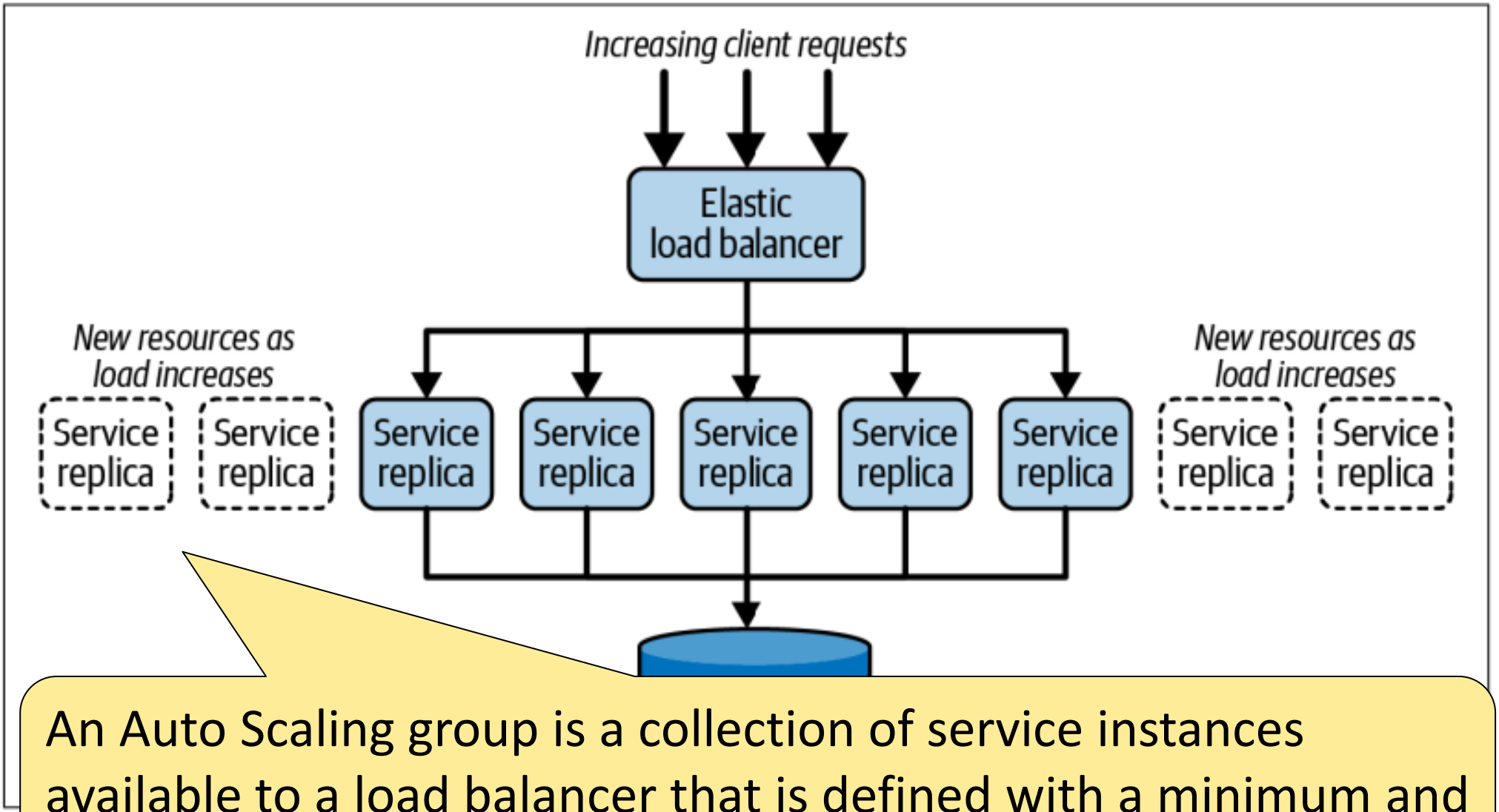


Figure 5-6. Elastic load balancing

Load Balancing: Elasticity

AWS Auto Scaling groups



An Auto Scaling group is a collection of service instances available to a load balancer that is defined with a minimum and maximum size.

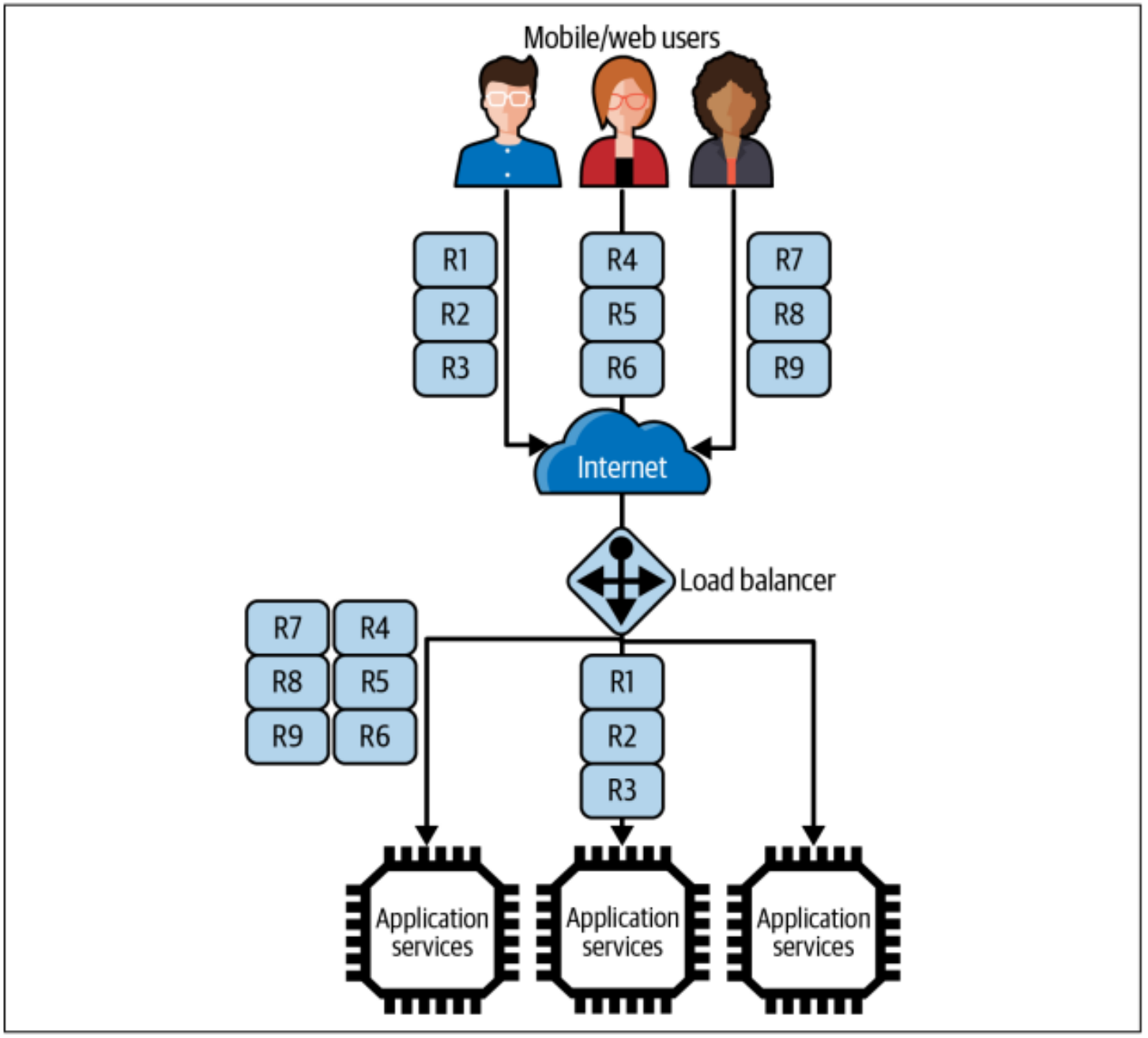
Load Balancing

- An Overview of Load Balancing
- Load Balancing Policies
- Elasticity
- **Session Affinity**

Load Balancing: Session Affinity

- *Session affinity* a load balancer feature for stateful services, where the load balancer sends all requests from the same client to the same service instance.
- This enables the service to maintain in-memory state about each specific client session.
- Sticky sessions can be problematic for highly scalable systems. They lead to a load imbalance problem, in which, over time, clients are not evenly distributed across services.

Load Balancing: Session Affinity



Required Reading

- Chapter 5: Application Services, from Ian Gorton's "Foundations of Scalable Systems", 2022.

Microservices

- Monolithic Applications
- Breaking Up the Monolith
- Deploying Microservices
- Principles of Microservices
- Resilience in Microservices

Microservices

- **Monolithic Applications**
- Breaking Up the Monolith
- Deploying Microservices
- Principles of Microservices
- Resilience in Microservices

Microservices

- The monolithic architectural style has dominated enterprise applications.
- This style decomposes an application into multiple logical modules or services, which are built and deployed as a single application.
- These services offer endpoints that can be called by external clients.
- Endpoints provide security and input validation and then delegate the requests to shared business logic, which in turn will access a persistent store through a data access objects (DAO) layer.

Monolithic Applications

- The monolithic architectural style has dominated enterprise applications.
- This style decomposes an application into multiple logical modules or services, which are built and deployed as a single application.
- These services offer endpoints that can be called by external clients.
- Endpoints provide security and input validation and then delegate the requests to shared business logic, which in turn will access a persistent store through a data access objects (DAO) layer.

Monolithic Applications

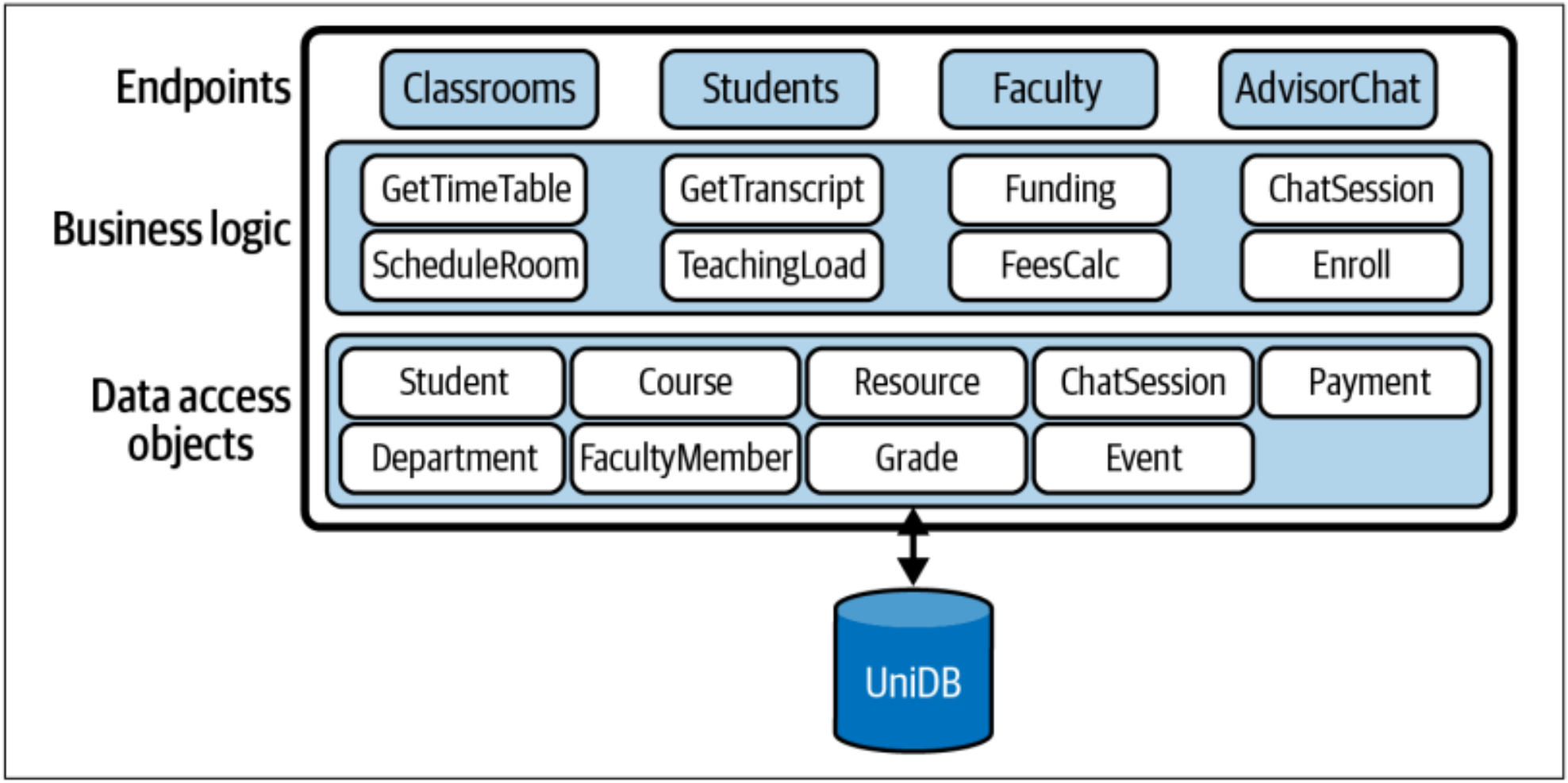


Figure 9-1. Example monolithic application

Monolithic Applications

- Popular platforms such as IBM Websphere and Microsoft .Net enable all the services to be built and deployed as a single executable package.
- This is where the term *monolith*—the complete application—originates. APIs, business logic, data access, and so forth are all wrapped up in a single deployment artifact
- Advantages of the monolith?
  - Architecture?
  - Testing?
  - Deployment?
  - Monitoring?

Monolithic Applications

- Monoliths become problematic as the system features or request volumes increase.
  - Code complexity
  - Scaling out→ Assume a sudden spike in the use of the AdvisorChat service occurs, how would you handle that?

Microservices

- Monolithic Applications
- **Breaking Up the Monolith**
- Deploying Microservices
- Principles of Microservices
- Resilience in Microservices

Breaking Up the Monolith

- A microservice architecture decomposes the application functionality into multiple independent services that communicate and coordinate when necessary
- Each microservice is totally self-contained, encapsulating its own data storage where needed, and offers an API for communications.

Breaking Up the Monolith

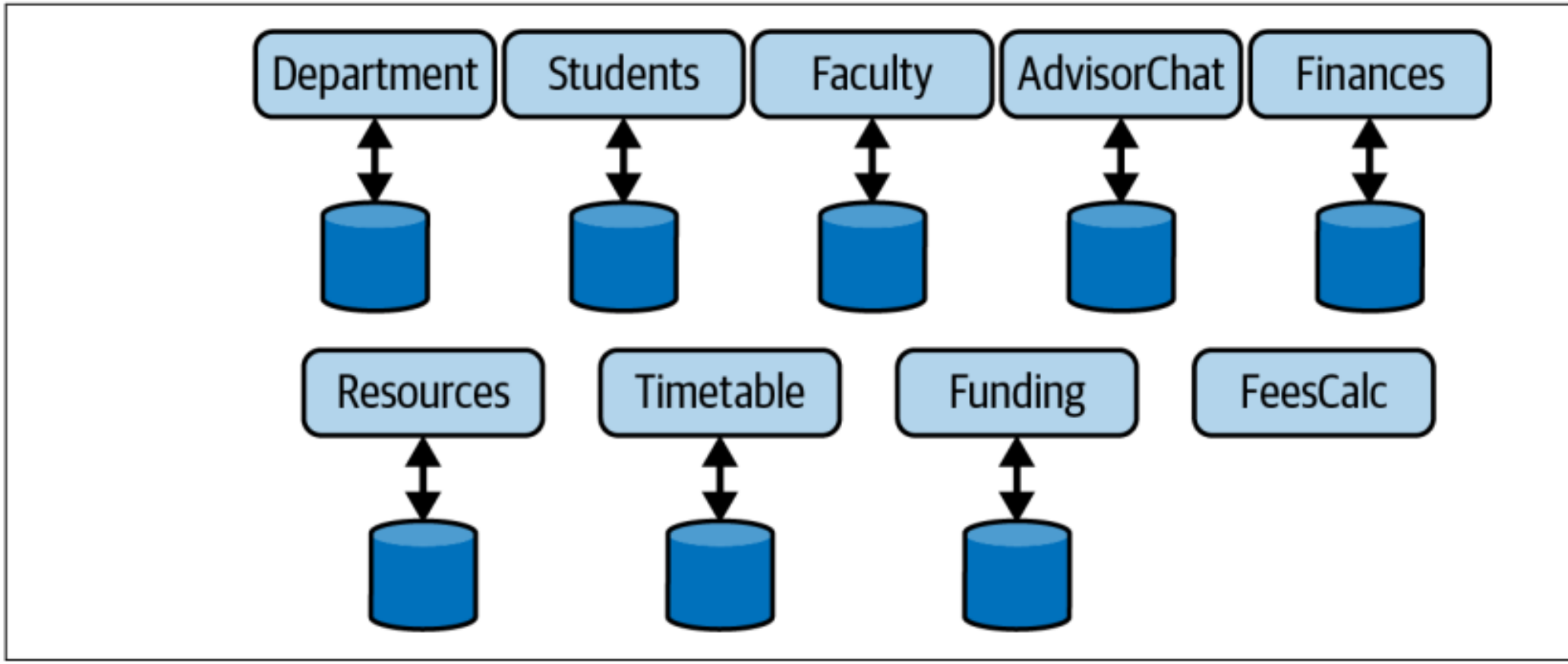


Figure 9-2. A microservice architecture example

Breaking Up the Monolith

- Microservices offer the following advantages as systems grow in code size and request load:
  - Code base:
    - An individual service should not be more complex than a small team size can build, evolve, and manage.
    - The team has full autonomy to choose their own development stack and data management platform.
    - Revisions of the microservice can be independently deployed as needed.
  - Scaling out

Breaking Up the Monolith

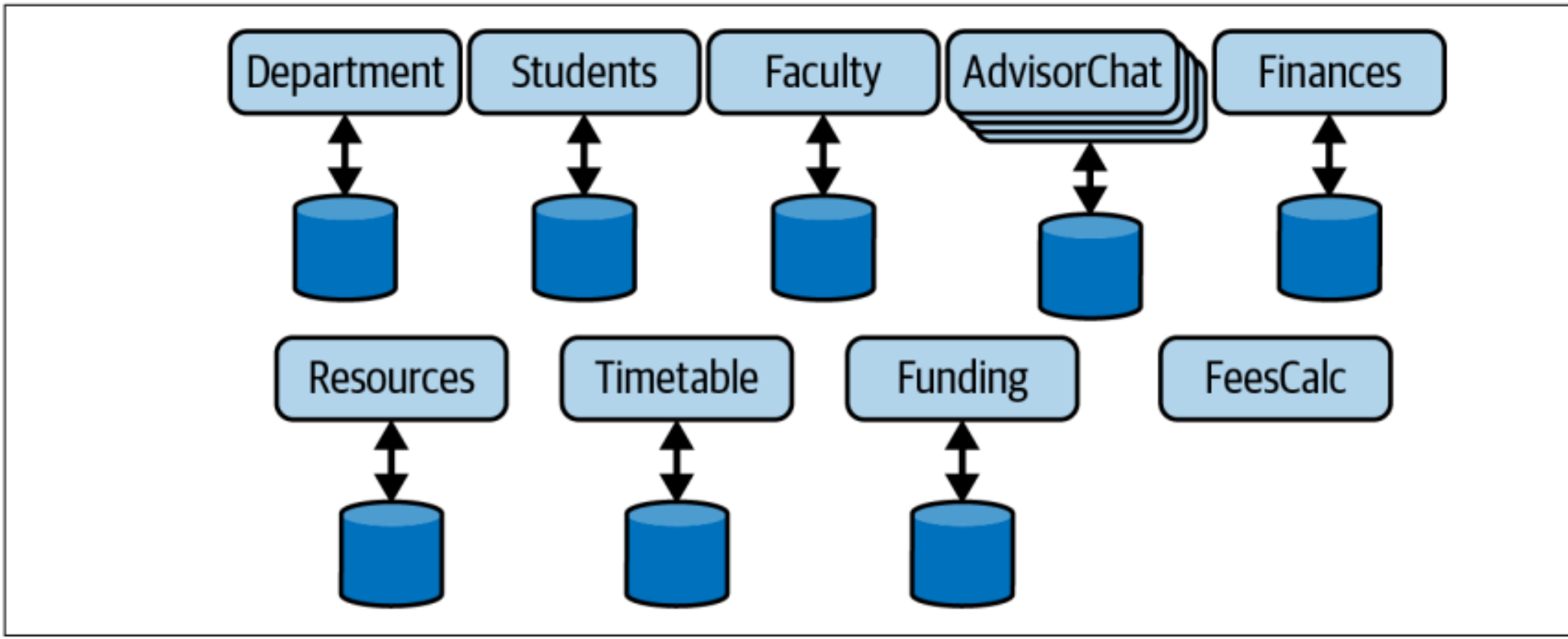


Figure 9-3. Independently scaling a microservice

Breaking Up the Monolith

- But…How can you decompose the system functionality into individual services?
- The domain model needs to be analyzed and adjusted to meet the reality of the costs of distributed communications and the complexity of system management and monitoring.
- You need to factor in request loads and the interactions needed to serve these requests, so that excessive latencies aren't incurred by multiple interactions between microservices.
- **Let us see an example**

Breaking Up the Monolith

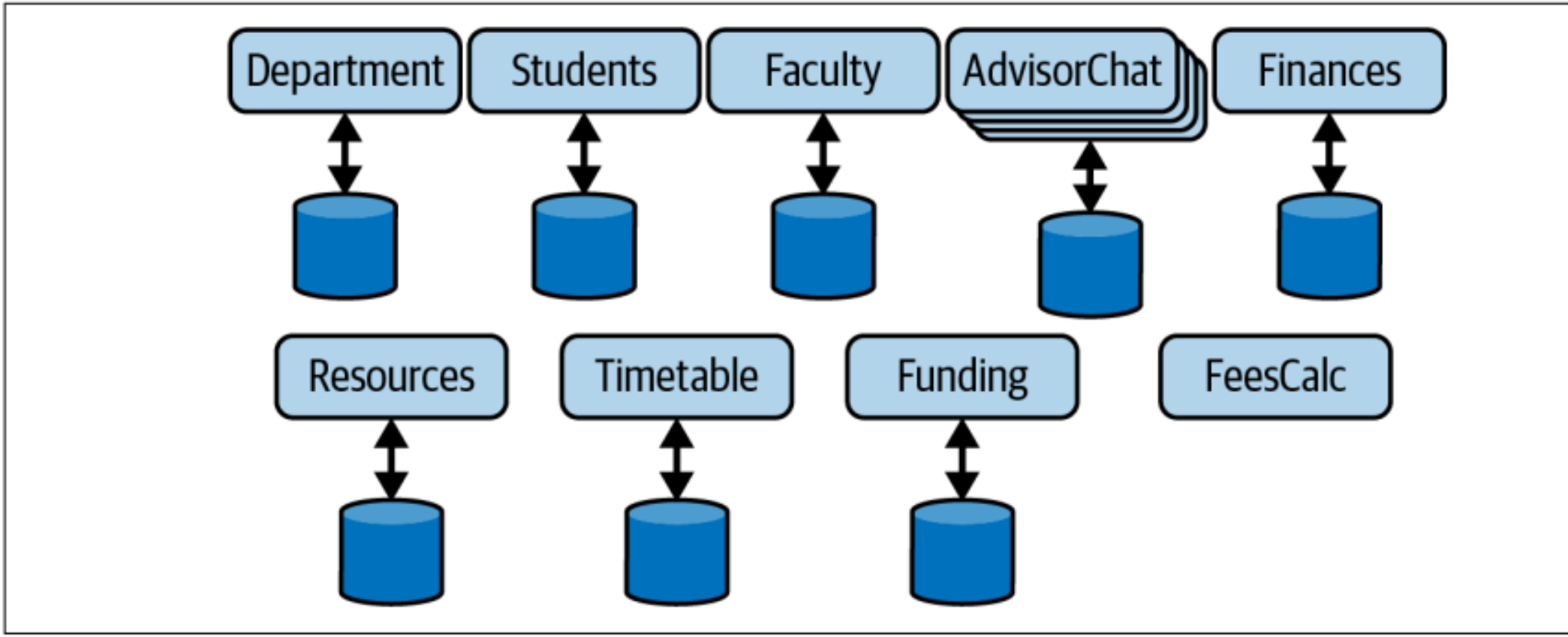


Figure 9-3. Independently scaling a microservice

- Should we merge Faculty and Funding services?
- Should we duplicate the data? Side effect?

Microservices

- Monolithic Applications
- Breaking Up the Monolith
- **Deploying Microservices**
- Principles of Microservices
- Resilience in Microservices

Deploying Microservices

- One common approach to deploying microservices is serverless processing platform.
- *Serverless* platforms, do not require any compute resources to be statically provisioned.
- Using technologies such as AWS Lambda or Google App Engine (GAE), the application code is loaded and executed on demand, when requests arrive.
- If there are no active requests, there are essentially no resources in use and no charges to meet

Deploying Microservices

- Serverless technologies feature automatic scaling, built-in high availability, and a pay-for-use billing model to increase agility and optimize costs.
- These technologies also eliminate infrastructure management tasks like capacity provisioning and, so you can focus on writing code that serves your customers.
- Serverless applications start with AWS Lambda, an event-driven compute service natively integrated with over 200 AWS services and software as a service (SaaS) applications.

Deploying Microservices

- A microservice can be built to expose its API on the serverless platform of your choice.
- The serverless option has three advantages:
  - *Deployment is simple (How?)*
  - *Pay by usage (How?)*
  - *Ease of Scaling (How?)*

Deploying Microservices

- How about exposing multiple endpoints?
  - When you deploy all your microservices on a serverless platform, you expose multiple endpoints that clients need to invoke.
  - What exactly would the client need to discover in such a case?
- What if you decide to refactor your microservices by perhaps combining two in or order to eliminate network calls?
  - Or move an API implementation from one microservice to another?
  - Or even change the endpoint (IP address and port) of an API?

Exposing backend changes

Deploying Microservices

- Recall the Façade pattern of the Gang of Four?
- In microservices, you can exploit an analogous approach using the API gateway pattern.
- An API gateway essentially acts as a single entry point for all client requests
- It insulates clients from the underlying architecture of the microservices that implement the application functionality

Deploying Microservices

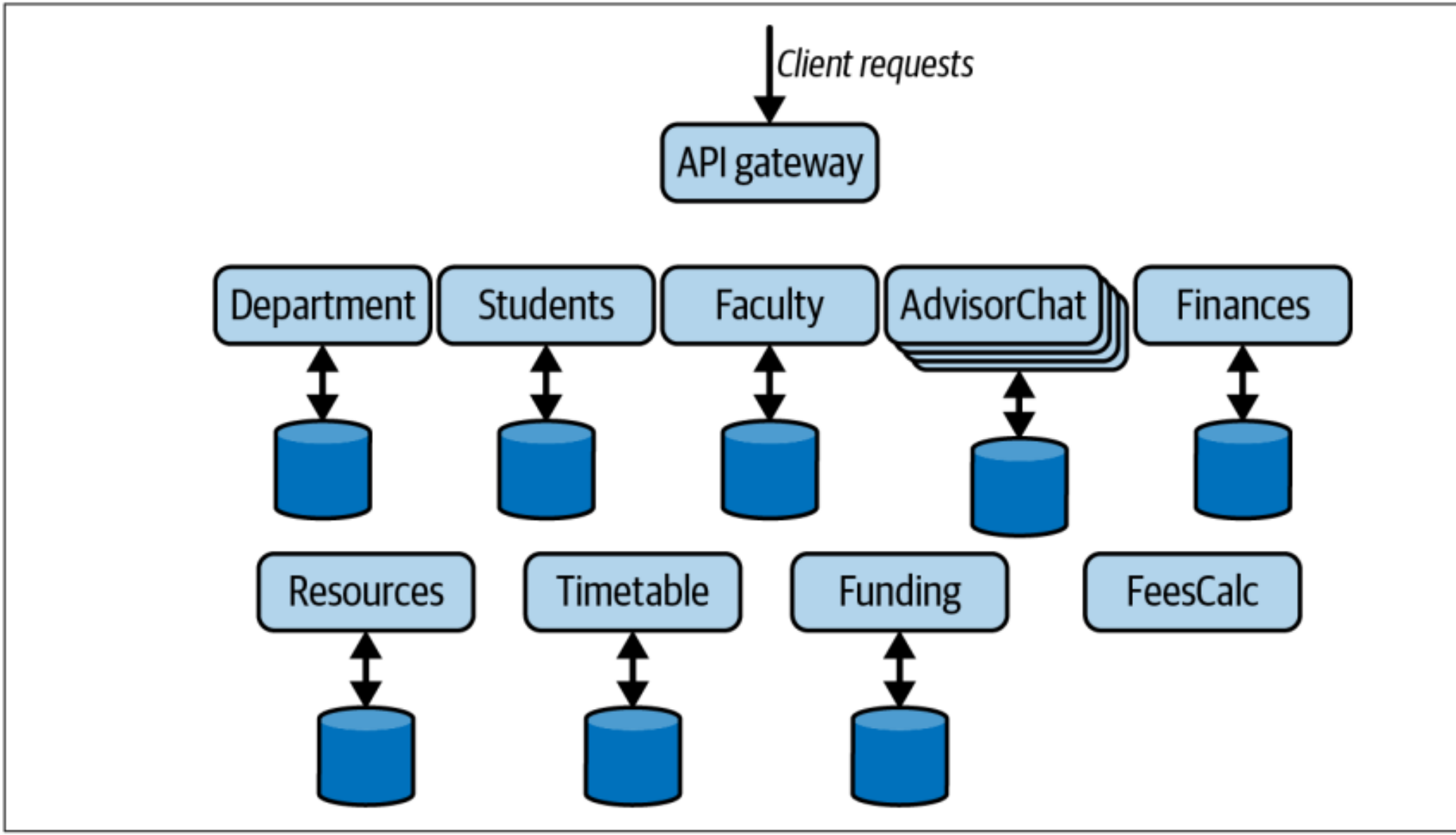


Figure 9-4. The API gateway pattern

Deploying Microservices

- There are multiple API gateway implementations you can exploit in your systems.
- These range from powerful open source solutions such as the NGINX Plus and Kong API gateways to cloud vendor–specific managed offerings.
- What functions should an API gateway implementation provide?

Deploying Microservices

- The general range of functions of an API gateway implementation, is listed as follow:
  - Proxy incoming client API requests with low millisecond latencies to backend microservices that implement the API. (What does that mean?)
    - **Proxy** → Mapping between client-facing APIs, handled by the API gateway, and backend microservice APIs is performed through admin tools or configuration files.
  - Provide authentication and authorization for requests

Deploying Microservices

- The general range of functions of an API gateway implementation, is listed as follow:
  - Define rules for throttling each API. (what is API throttling?)
    - API throttling is the process of limiting the number of API requests a user can make in a certain period.
  - Support a cache for API results so that requests can be handled without invoking backend services.
  - Integrate with monitoring tools to support analysis of API usage, latencies, and error metrics.
- Possible issue with API gateways?

Principles of Microservice Design

- Microservices should be
  - Highly observable: Monitoring of each service is essential to ensure they are behaving as expected
  - Hide implementation details: Their API is a contract which they are guaranteed to support, but how this is carried out is not exposed externally
  - Decentralize all the things:
    - One thing to decentralize is the processing of client requests that require multiple calls to downstream microservices. These are often called workflows.
    - There are two basic approaches to achieving this, namely orchestration and choreography

Principles of Microservice Design

- Microservices should
  - Isolate failure: The failure of one microservice should not propagate to others and bring down the application.
  - Deploy independently : Every microservice should be independently deployable, to enable teams to roll out enhancements and modifications without any dependency on the progress of other teams.

Principles of Microservice Design

- Microservices Workflows:
  - Orchestration and choreography are commonly used for implementing use cases that require access to more than one microservice.
  - Example: A faculty member may wish to get a list of the classes they are teaching in a semester and the resources available for audio-visual within each classroom they have been allocated to.
  - Implementing this use case requires access to the *Faculty*, *Timetable*, and *Resources* microservices.

Principles of Microservice Design

- There are two basic approaches to implement this workflow:
- *Peer-to-peer choreography:* The required microservices communicate directly to satisfy the request. This shares the responsibility and knowledge of processing the workflow across each autonomous microservice. Communications may be synchronous or asynchronous
- *Centralized orchestration:* The logic to implement the workflow is embedded in a single component, often a dedicated microservice. This communicates with the domain services and sends the results back to the user.
- Tradeoffs in terms of ease of monitoring and request load?

Resilience in Microservices

- Things start to get really fun when request frequencies and volumes increase. Threads contend for processing time, memory becomes scarce, network connections become saturated, and latencies increase.
- To ensure your systems don’t fail suddenly as loads increase, there are a number of necessary precautions you need to take.

Resilience in Microservices
Cascading Failures

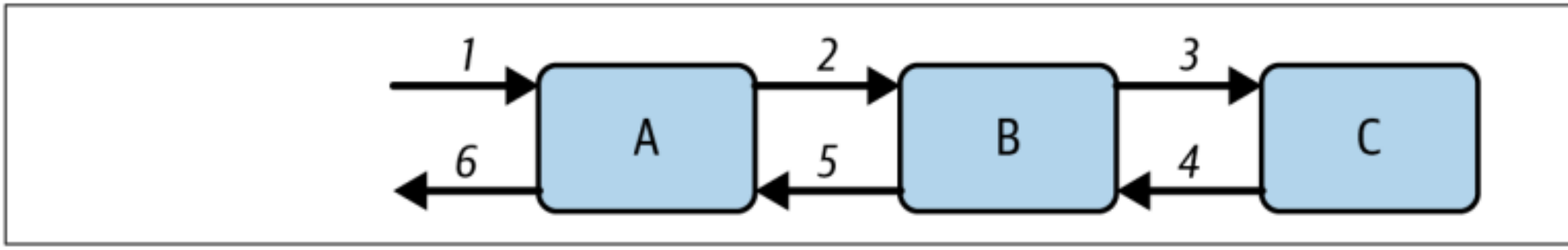


Figure 9-5. Microservices with dependencies

What is the flow when a request arrives at service A?

Resilience in Microservices
Cascading Failures

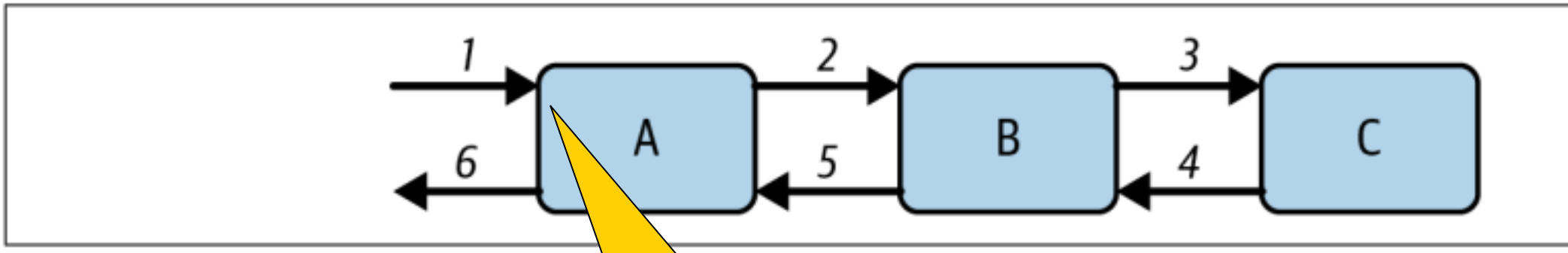


Figure 9-5. Microservices wit[illegible]pendencies

What if the request load on microservice A increases?

Resilience in Microservices
Cascading Failures

- Recall that application servers have fixed-size thread pools.
- Once all threads in B are occupied making calls to C, if requests continue to arrive at high volumes, they will be queued until a thread is available.
- Response times from B to A start to grow, and in an instant all of A's threads will be blocked waiting for B to respond.
- The slow responses from C can cause requests to A and B to fail.

Resilience in Microservices
Cascading Failures

- How do **cascading failures** differ from **downstream services**?
- If a **downstream service** simply fails or is unavailable, the caller gets an error immediately and can respond accordingly.
- **Cascading failures** are triggered by slow response times of dependent services.
  - Requests return results, just with longer response times.
  - If the overwhelmed component continues to be bombarded with requests, it has no time to recover and response times continue to grow.

Resilience in Microservices
Cascading Failures

- This situation is often exacerbated by clients that, upon request failure, immediately retry the operation
- Immediate retries simply maintain the load on the overwhelmed microservice, with very predictable results, namely another exception.

```
int retries = RETRY_COUNT;
while (retries > 0) {
    try {
        callDependentService();
        return true;
     } catch (RemoteCallException ex) {
         logError(e);
         retries = retries - 1;
}
return false;
```

What does the above code do?

Resilience in Microservices
Cascading Failures

- To address cascading failures, several patterns can be used:
  - Fail fast pattern
  - Circuit breaker pattern
  - Bulkhead pattern

Resilience in Microservices
Cascading Failures – Fail Fast Pattern

- The core problem with slow services is that they utilize system resources for requests for extended periods.
- A requesting thread is stalled until it receives a response.
- Consider that we have an API that normally responds within 50 ms:
  - How many requests can a thread process within 1 second?
  - If one request is stalled for 3 seconds, how many requests would be delayed?

Resilience in Microservices
Cascading Failures – Fail Fast Pattern

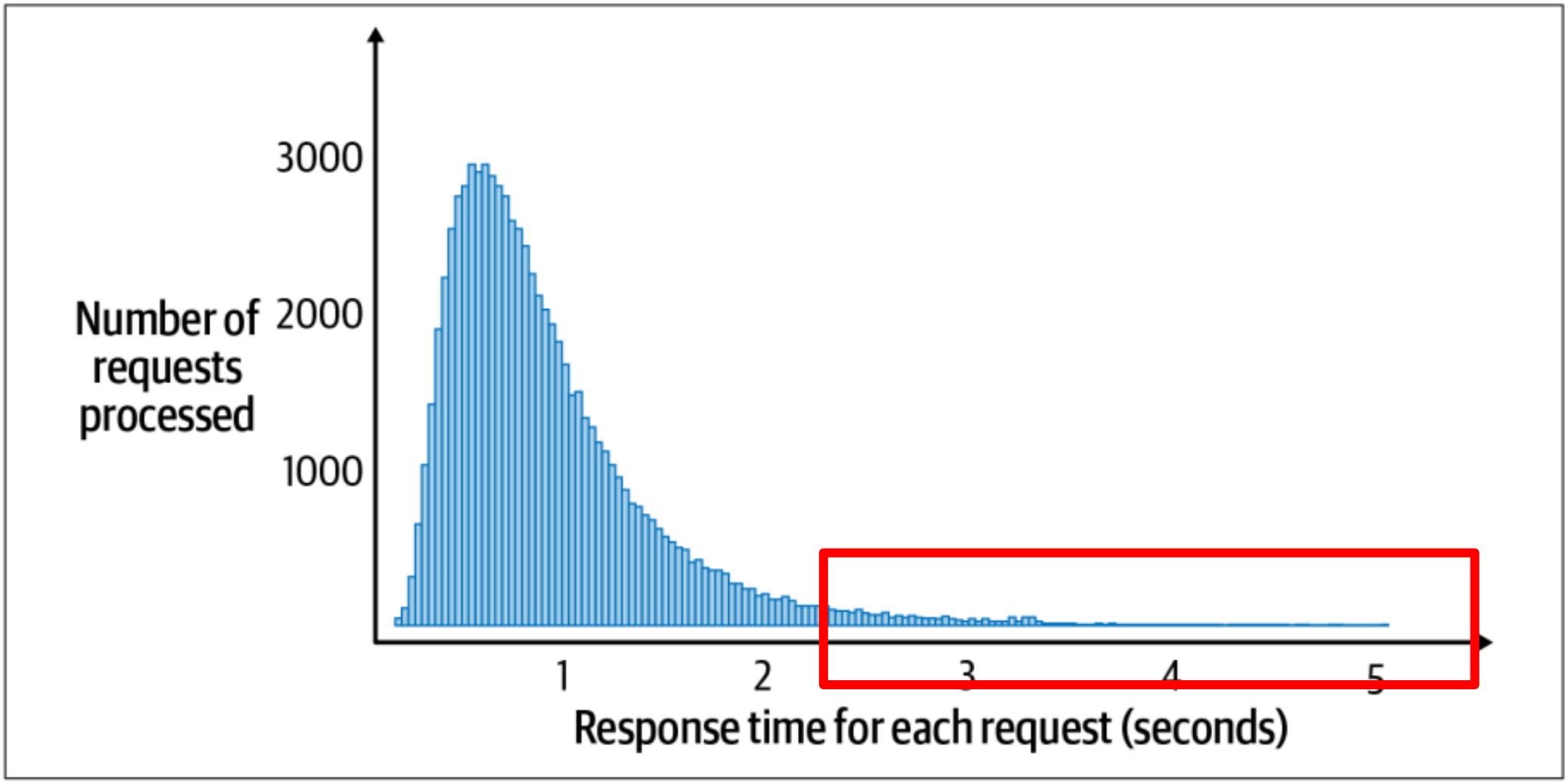


Figure 9-6. Typical long-tail response time

Resilience in Microservices
Cascading Failures – Fail Fast Pattern

- A common way to eliminate long response times is to **fail fast**. There are two main ways to achieve this:
  - When a request takes longer than some predefined time limit, instead of waiting for it to complete, the client returns an error to its caller. This releases the thread and other resources associated with the request.
  - Enable throttling on a server. If the request load exceeds some threshold, immediately fail the request with an HTTP 503 error. This indicates to the client that the service is unavailable.

Resilience in Microservices
Cascading Failures – Circuit Breaker Pattern

- To back off immediately from sending further requests and allow some time for the error situation to resolve.
- This can be achieved using the circuit breaker pattern, which protects remote endpoints from being overwhelmed when some error conditions occur
- The circuit breaker is configured to monitor some condition, such as error response rates from an endpoint, or the number of requests sent per second.

Resilience in Microservices
Cascading Failures – Circuit Breaker Pattern

1. If the configured threshold is reached—for example, 25% of requests are throwing errors—the circuit breaker is triggered.
2. This moves the circuit breaker into an OPEN state, in which all calls return with an error immediately, and no attempt is made to call the unstable or unavailable endpoint
3. The circuit breaker then rejects all calls until some suitably configured timeout period expires. At that stage, the circuit breaker moves to the HALF_OPEN state.

Resilience in Microservices
Cascading Failures – Circuit Breaker Pattern

4. Now, the circuit breaker allows client calls to be issued to the protected endpoint.
5. If the requests still fail, the timeout period is reset and the circuit breaker stays open.
6. If the request succeeds, the circuit breaker transitions to the CLOSED state and requests start to flow to the target endpoint

Resilience in Microservices
Cascading Failures – Circuit Breaker Pattern

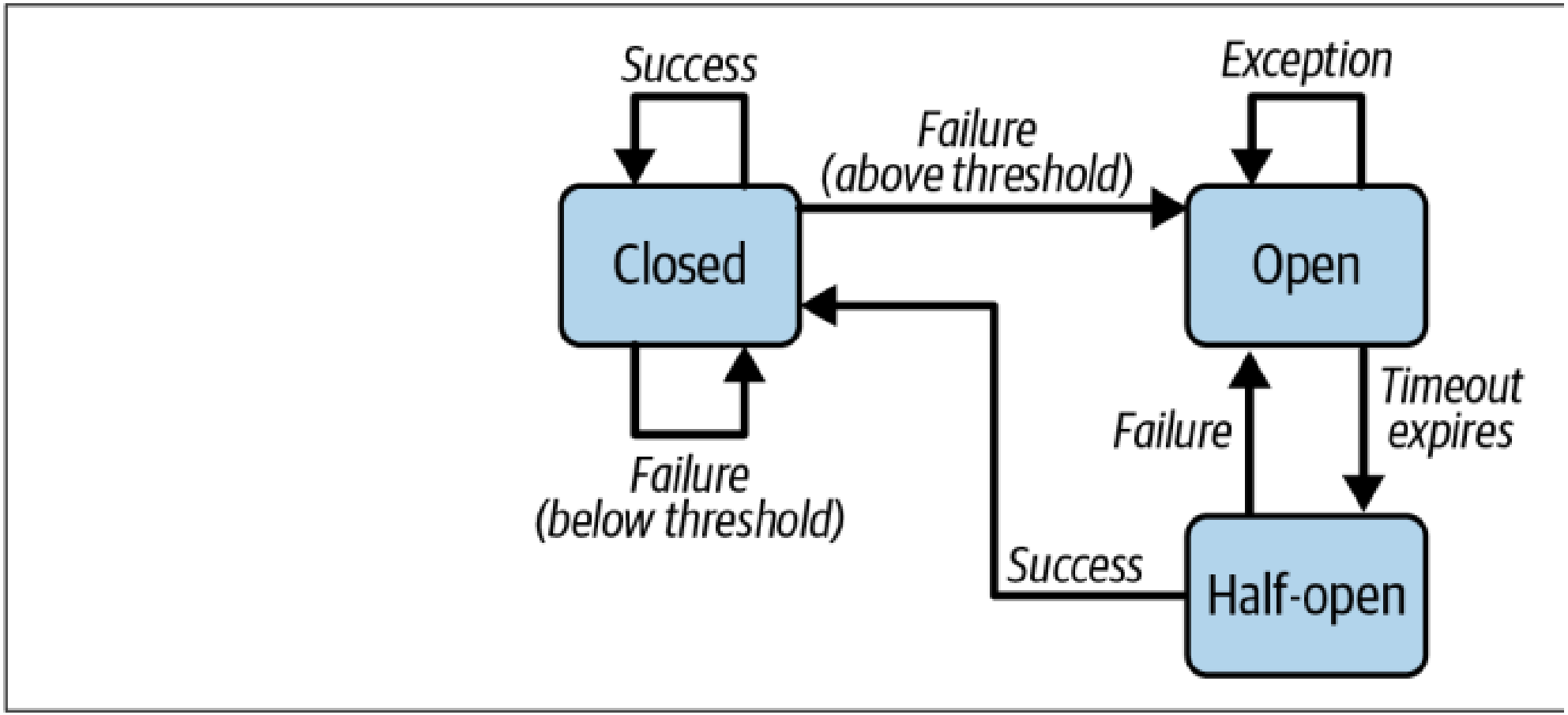


Figure 9-7. Circuit breaker pattern

Resilience in Microservices
Cascading Failures – Circuit Breaker Pattern

- There are numerous libraries available for incorporating circuit breakers into your applications.
- One popular library for Python, CircuitBreaker, is illustrated in the following code example.

```
from circuitbreaker import circuit

@circuit(failure_threshold=20,expected_exception=RequestException,
         recovery_timeout=5)
def api_call():
```

Resilience in Microservices
Cascading Failures – Bulkhead Pattern

- The term bulkhead is inspired by large shipbuilding practices.
- A ship is divided into several physical partitions, ensuring if a leak occurs in one part of the boat's hull, only a single partition is flooded and the boat, rather importantly, continues to float.
- Consider a microservice with two end points:
  - One end point enables clients to request the status of their currently placed orders
  - One end point enables clients to create new orders for products
- Where do you expect most of the requests to be?

Resilience in Microservices
Cascading Failures – Bulkhead Pattern

- What if a popular product is on Sale?
- Any effect on thread pools?
- Using Bulkhead, We can reserve a number of threads in a microservice to handle specific requests (e.g., a maximum of 150 threads of the shared thread pool for its exclusive use.
- The bulkhead pattern segregates remote resource calls in their own thread pools so that a single overloaded or failing service does not consume all threads available in the application server

Required Reading

- Chapter 9: Microservices, from Ian Gorton's "Foundations of Scalable Systems", 2022.