Performance Optimization Limits

Trade-offs

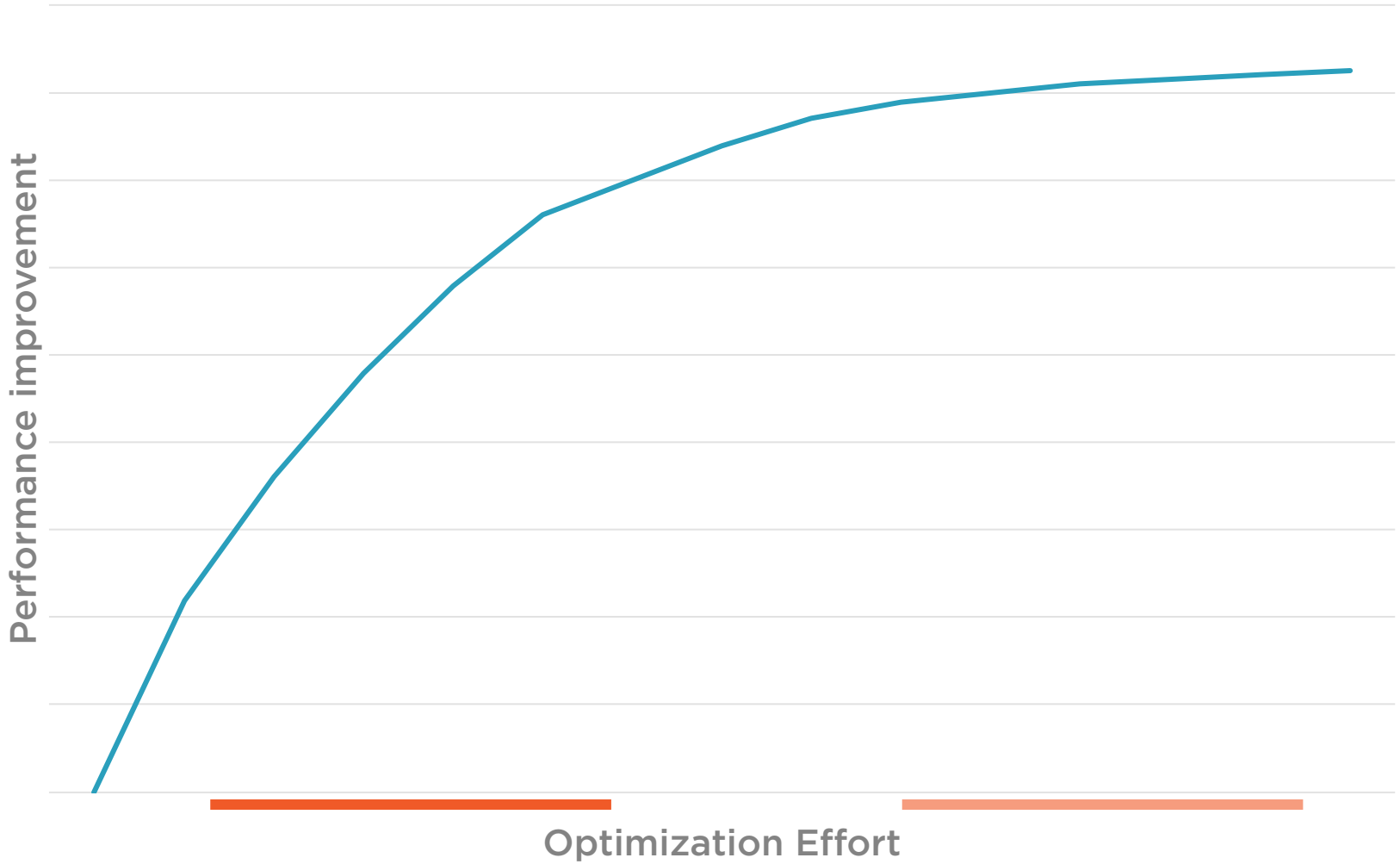
CPU cycles vs. memory

Modular design vs. tightly coupled

Simple vs. complex

Diminishing returns

Performance Optimization Diminishing Returns



Performance Optimization Limits

Trade-offs

Diminishing returns

Application Performance Metrics

Throughput

Latency (Response time)

Elapsed time

Examples of Performance Goals

Average latency for database queries should be 50ms (Latency)

Should be able to support 5,000 concurrent users (Throughput)

Performance Testing Guidelines

- Have a warm-up period
- Ensure the test setup and traffic being sent is representative of production
- Measure system performance during the test

Performance Tuning Tools

Load Testing Tools

Jmeter, Gatling, Commercial options

Application Instrumentation

Dropwizard, Micrometer, Spectator, Prometheus

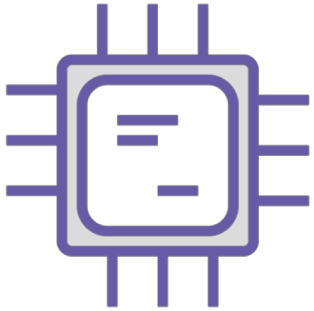
System Monitoring Tools

typeperf, vmstat, iostat, netstat

JVM Monitoring Tools

JMC, JConsole, jcmd

Java Profiling



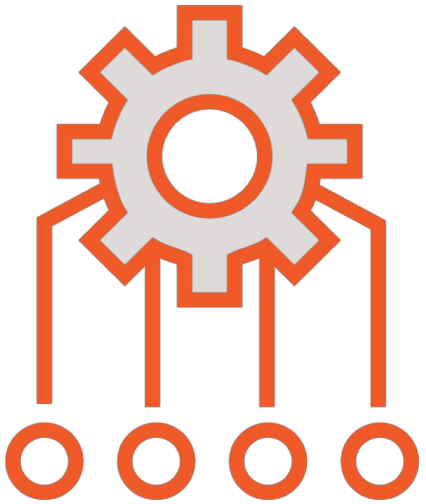
CPU Profiling

Find hot methods

Memory Profiling

View memory usage and allocations

View Garbage Collection



Thread Profiling

View thread states and lock contention

Additional Profiling Capabilities

Automated analysis

SQL profiling

I/O profiling

Exception analysis

Java Profiler Software

Java Flight Recorder

Open source:

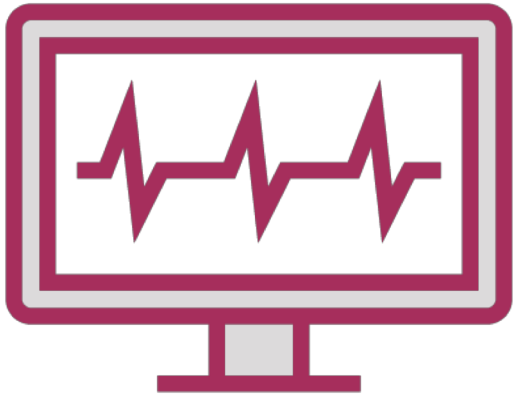
- Java VisualVM
- NetBeans Profiler

Commercial:

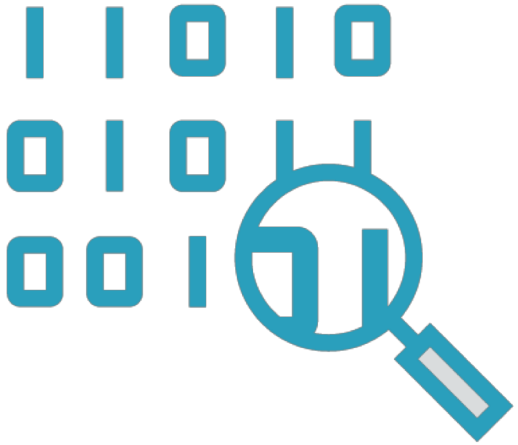
- JProfiler
- YourKit Java Profiler

Optimizing code that's not frequently executed will result in an almost negligible contribution to overall performance

Set Performance Targets



Setup Monitoring



Performance Analysis

Performance Optimization

Just-In-Time Compilation

Improves performance by identifying hot methods and compiling them to optimized machine code

Tuning:

- Choosing a compilation mode
- Choosing the compilation threshold
- Tuning the size of the code cache

Garbage Collection Tuning

Trying to find the balance between minimizing GC pauses and background GC activity

Tuning factors:

- Size of the heap regions
- Number of background threads
- Threshold at which GC activities are triggered

Code Optimization

Performance tuning is understanding the cost of data structures and algorithms and figure out ways to pay less for the same functionality

Big-O notation

Application usage patterns matter

Cost Reduction

Choosing the right data structure

Minimizing memory footprint

Reducing lock contention / thread synchronization

Caching

Expensive Object Creation

Object creation may require extensive computation, I/O, or system resources

To avoid repeatedly incurring the cost, we can turn to caching

Caching

Mechanism by which data is stored so that future requests for the data can be served faster

Object Reuse

The object is created just once and then reused every time its needed afterwards

Suitable for stateless, thread-safe objects

```
public class QueryValidator {
  private static final String REGEX = ...;

  // Frequently called method
  public boolean validateQueryString(String queryString) {
    // Pattern.matches(REGEX, queryString) //same as below
    Pattern pattern = Pattern.compile(REGEX);
    Matcher matcher = pattern.matcher(queryString);
    return matcher.matches();
  }
}
```

```
public class QueryValidator {
  private static final String REGEX = ...;

  // Frequently called method
  public boolean validateQueryString(String queryString) {
    // Pattern.matches(REGEX, queryString) //same as below
    Pattern pattern = Pattern.compile(REGEX);
    Matcher matcher = pattern.matcher(queryString);
    return matcher.matches();
  }
}
```

```
public class QueryValidator {
  private static final String REGEX = ...;
  private static final Pattern QUERY_PATTERN =
                                        Pattern.compile(REGEX);

  // Frequently called method
  public boolean validateQueryString(String queryString) {
    Matcher matcher = QUERY_PATTERN.matcher(queryString);
    return matcher.matches();
  }
}
```

```
public class QueryValidator {
  private static final String REGEX = ...;
  private static final Pattern QUERY_PATTERN =
                                        Pattern.compile(REGEX);

  // Frequently called method
  public boolean validateQueryString(String queryString) {
    Matcher matcher = QUERY_PATTERN.matcher(queryString);
    return matcher.matches();
  }
}
```

String Methods That Use a Pattern Object Under the Hood

split()*

matches()

replace()

replaceFirst()

replaceAll()

String method	Alternative
String.matches()	Cached Pattern object
String.split()	StringUtils.split()
String.replace()	StringUtils.replace()

Object Pooling

Suitable for non-thread safe, stateful objects

Application creates a number of expensive objects ahead of time and leases them out

Avoids each requester having to create and destroy the expensive object

Most commonly pooled objects; database connections, socket connections.

Only objects that are expensive to create should be pooled

```
try {
  ExpensiveObject obj =
    pool.borrowObject();
} finally {
  pool.returnObject(obj);
}
```

```
ExpensiveObject obj =
    new ExpensiveObject();
```

Implementing Object Pooling

Many libraries that create expensive objects already have object pooling implemented

Use the Apache Commons Pool library

Results Caching

Storing the results of an operation and the request that generated the result so that identical requests can simply retrieve the result instead of re-calculating it

```
Result foo(Request request) {
  Result result = cache.get(request);

  if (result == null) {
    result = doFoo(request);
    cache.put(request, result);
  }

  return result;
}
```

```
Result foo(Request request) {
  Result result = cache.get(request);

  if (result == null) {
    result = doFoo(request);
    cache.put(request, result);
  }

  return result;
}
```

```
Result foo(Request request) {
  Result result = cache.get(request);

  if (result == null) {
    result = doFoo(request);
    cache.put(request, result);
  }

  return result;
}
```

```
Result foo(Request request) {
  Result result = cache.get(request);

  if (result == null) {
    result = doFoo(request);
    cache.put(request, result);
  }

  return result;
}
```

The Effectiveness of Caching

```
Cache hit ratio = Num hits / Num requests
```

If the cache hit ratio is high then we get a performance boost

If the cache hit ratio is low then we would be wasting CPU cycles and memory space

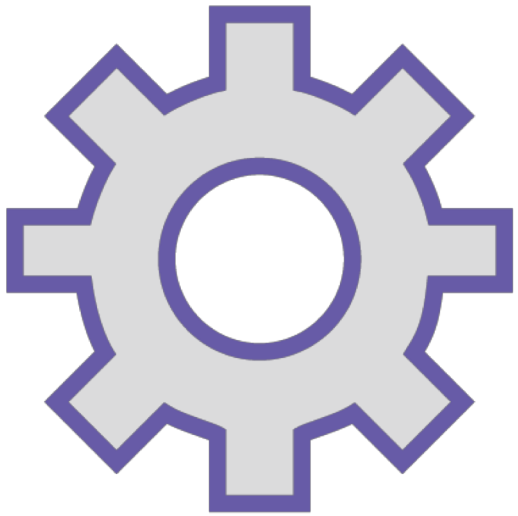
Caching is only effective if we have hot data or the data has low variance

In-memory Cache Implementations

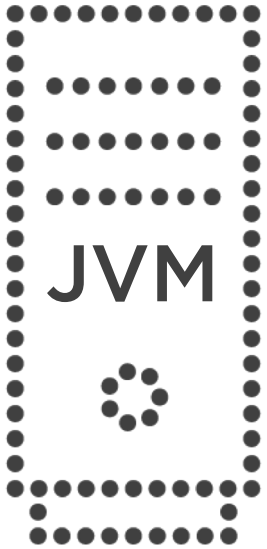
Use implementation from Guava, triava, or Apache Commons

Architecture Level Performance Optimizations

Performance Optimization Levels



System



JVM

Code

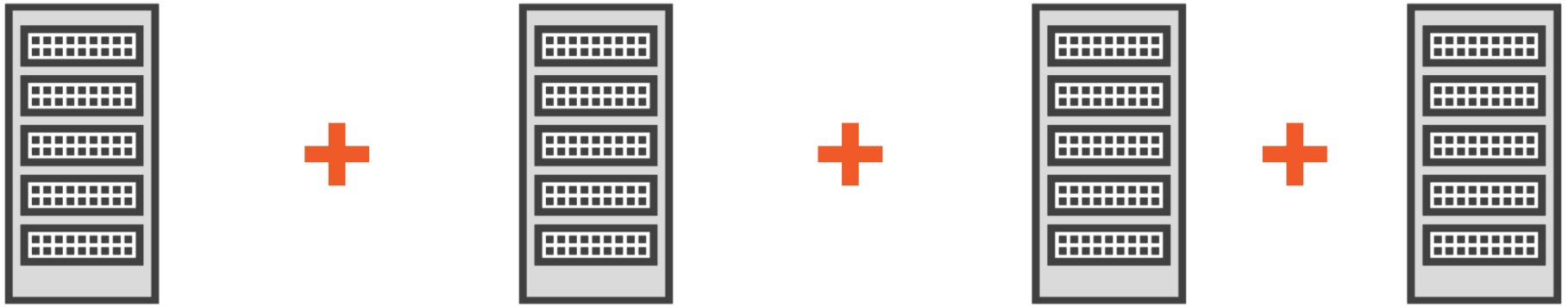
Architecture

Scale Up

Scale Out

Scale Up

Scale Out



Scale Out Tips

It is still important to correctly size each machine based on the application workload

- Requirement: 500 MB of RAM per core
- Machine: 16 GB of RAM with 8 cores

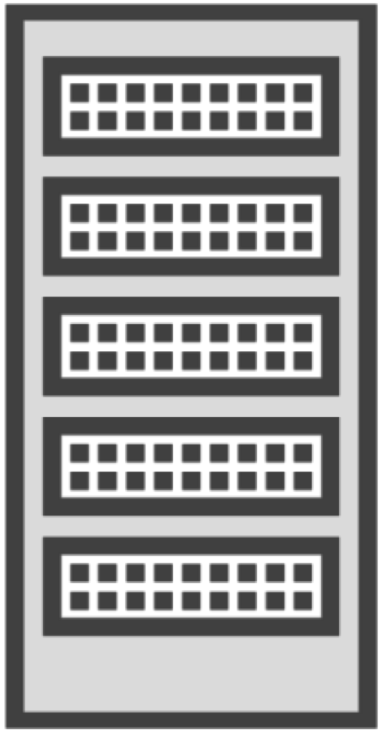
You may need to scale the database as well

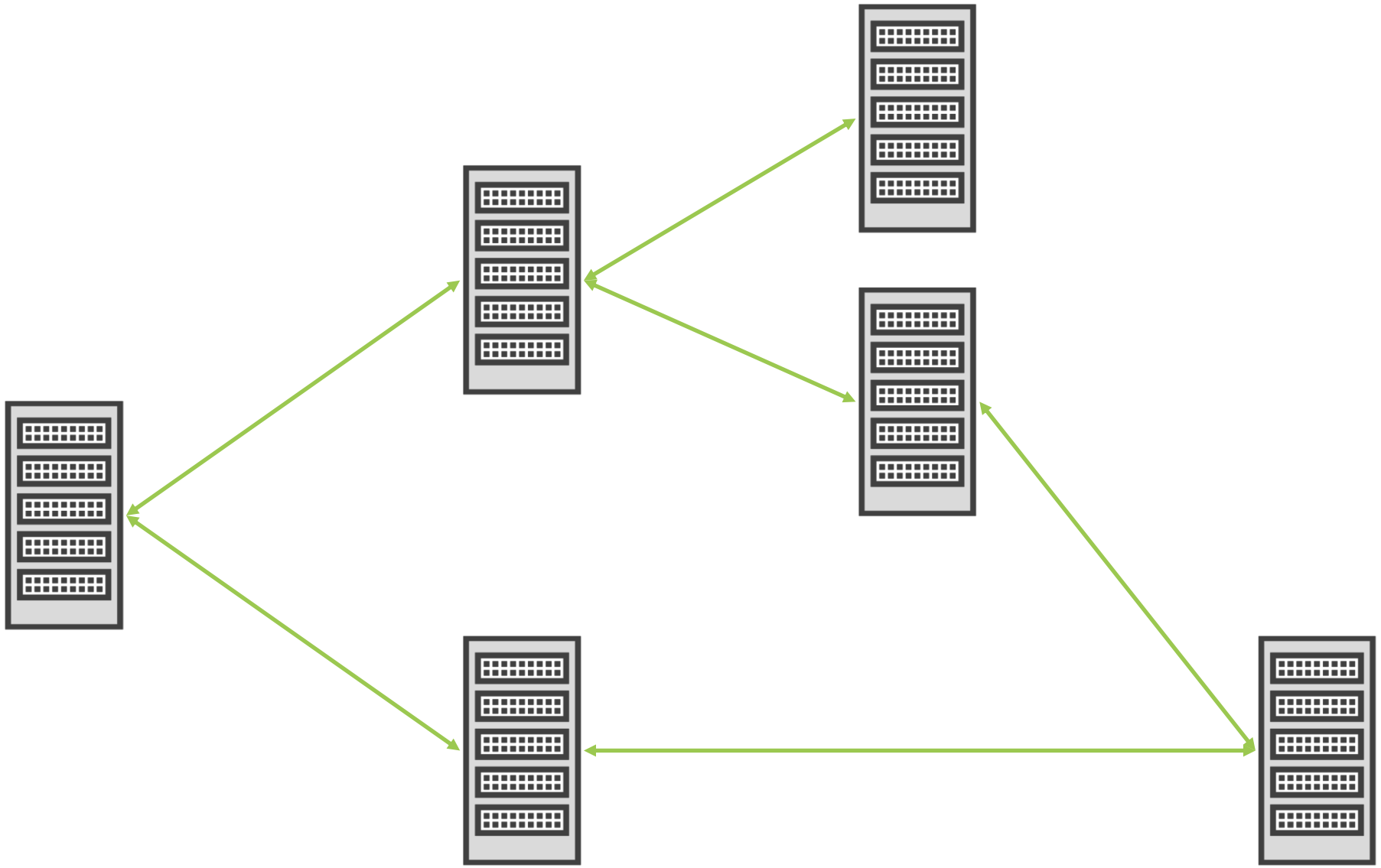
Client-side Database Performance Optimizations

Connection pooling

Statement caching

Batching





Microservices Pros and Cons

Pros

Can scale services independently

Can scale teams

Cons

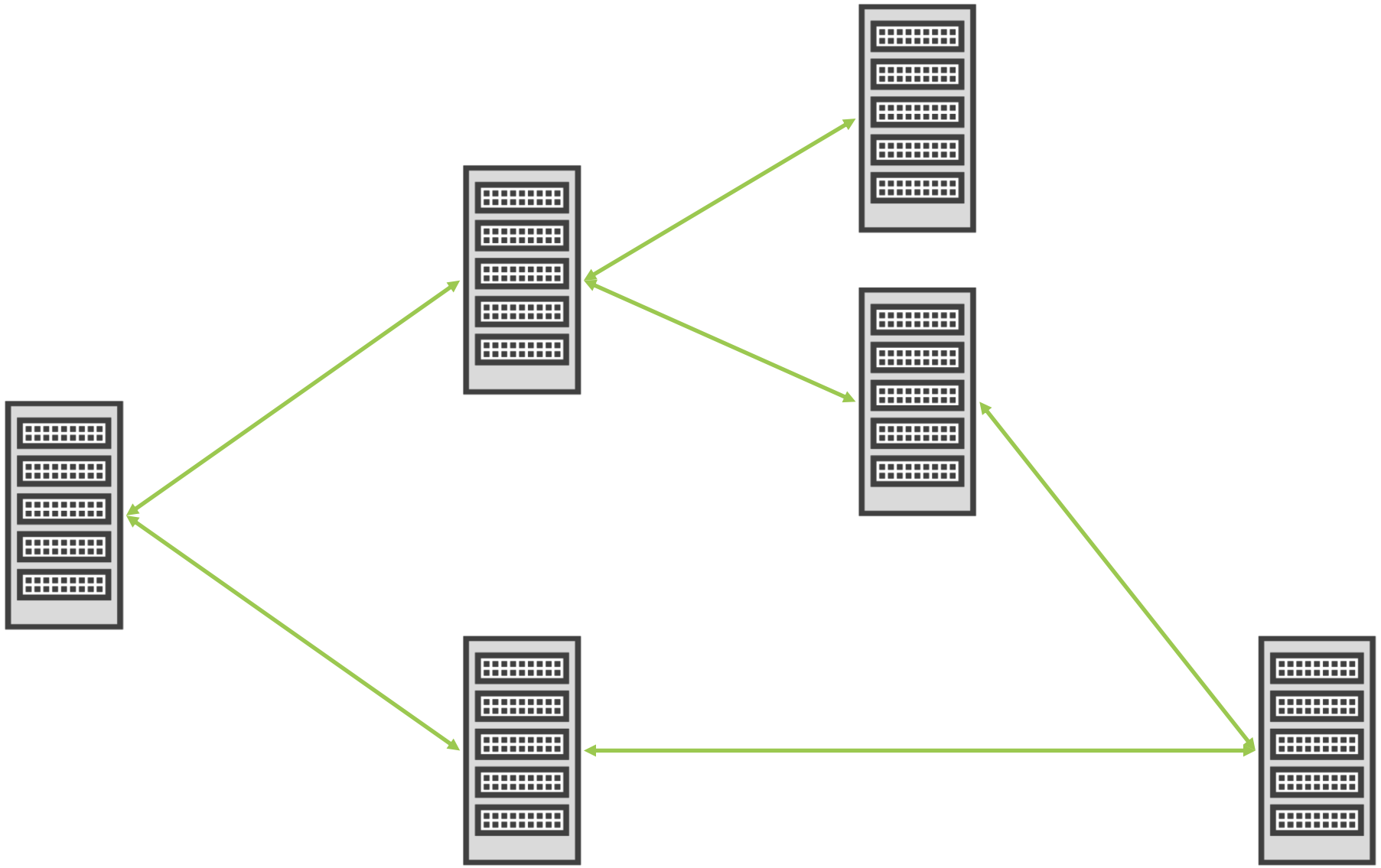
Performance overhead of inter-service communication

Reducing Intercommunication Overhead

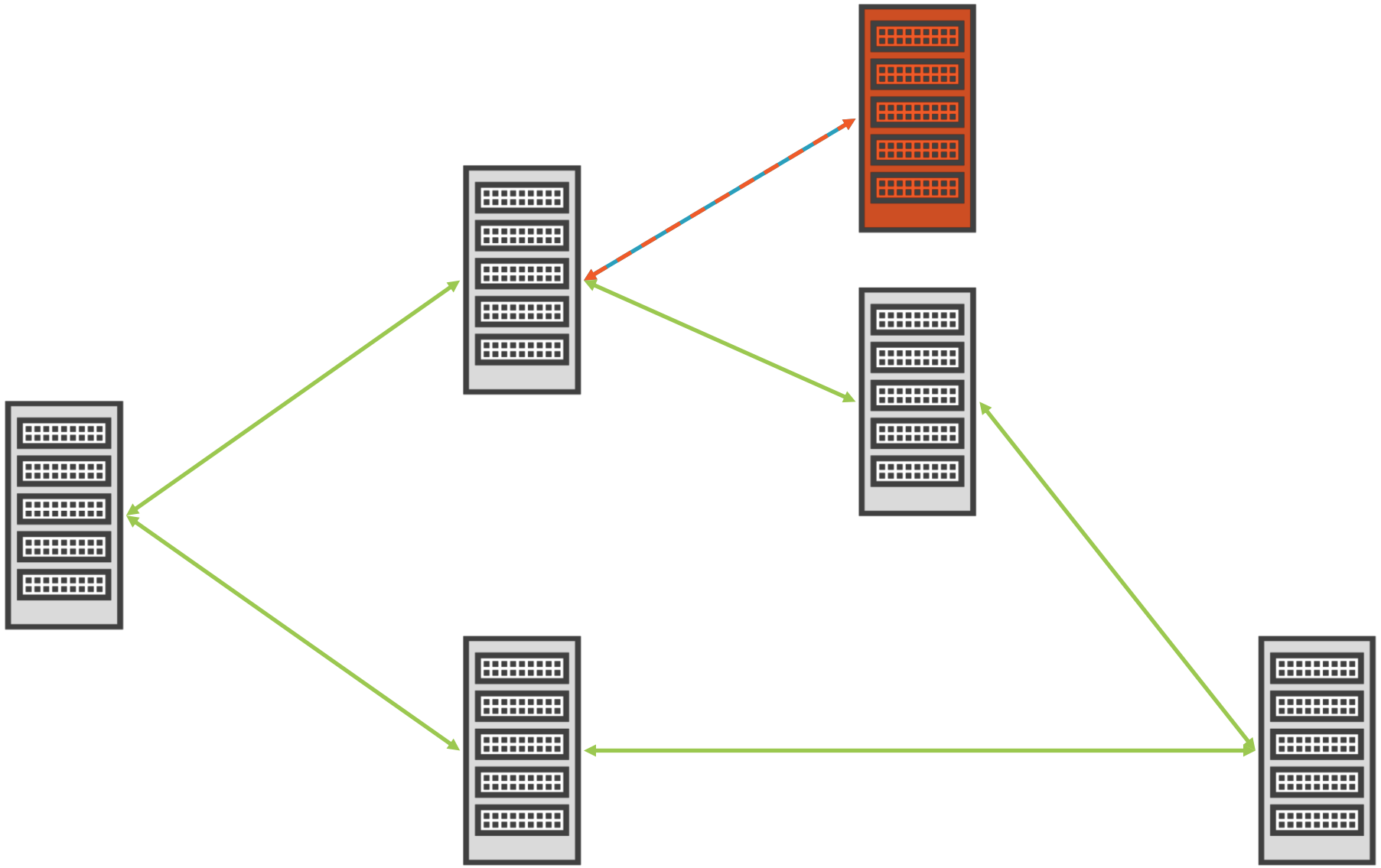
Use a binary format (e.g. Protobuf, Thrift, Avro, MessagePack)

Use the circuit breaker pattern to limit cascading failures

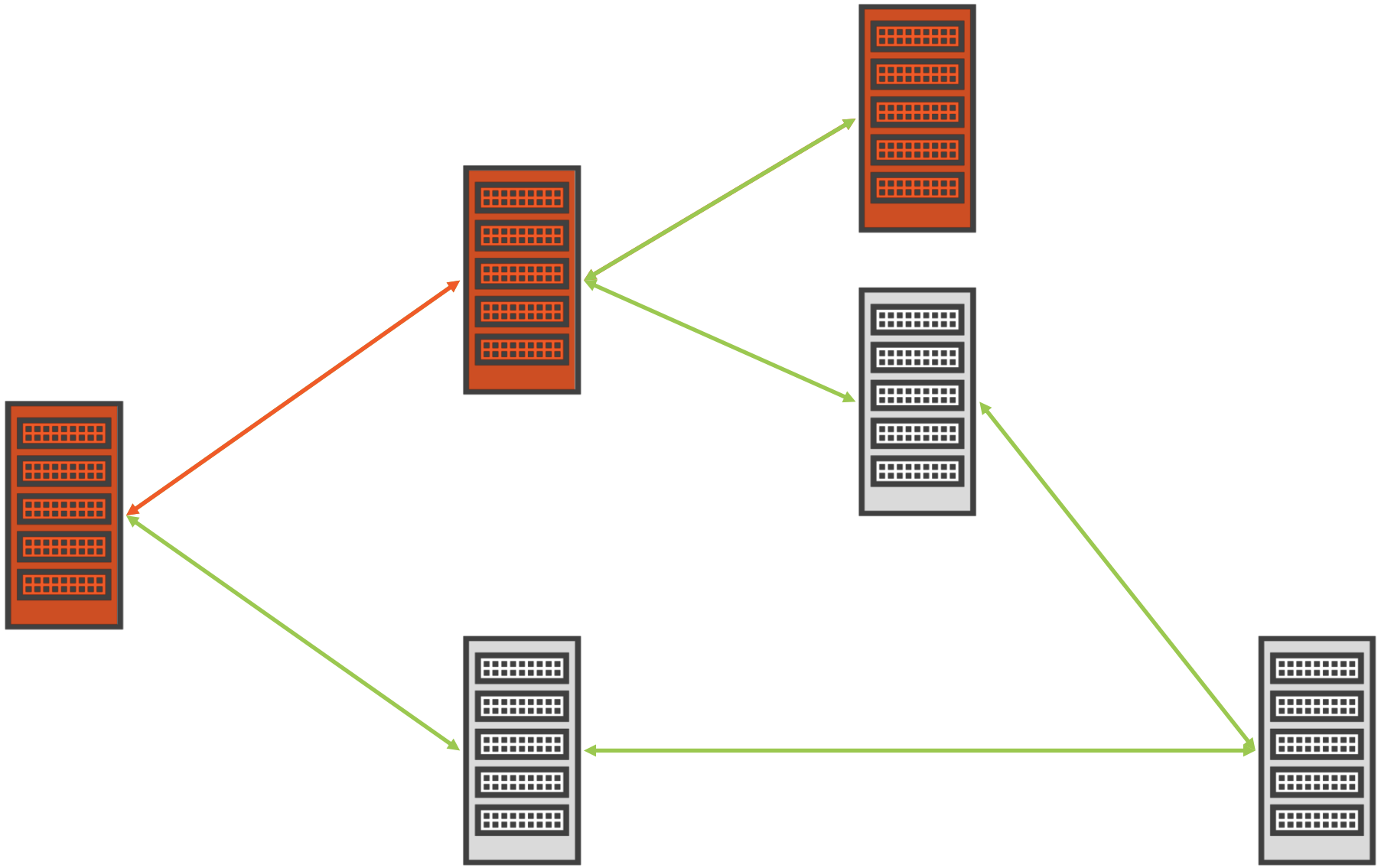
Use asynchronous communication between services



With Circuit Breaking



Without Circuit Breaking



Reducing Intercommunication Overhead

Use a binary format (e.g. Protobuf, Thrift, Avro, MessagePack)

Use the circuit breaker pattern to limit cascading failures

Use asynchronous communication between services