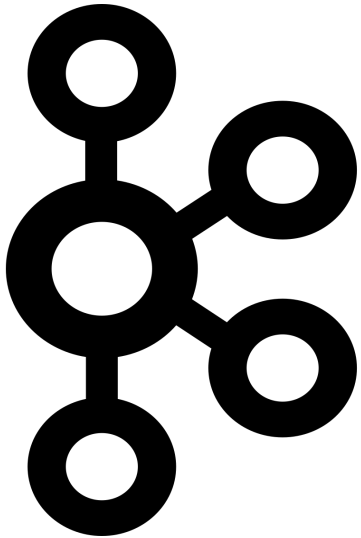


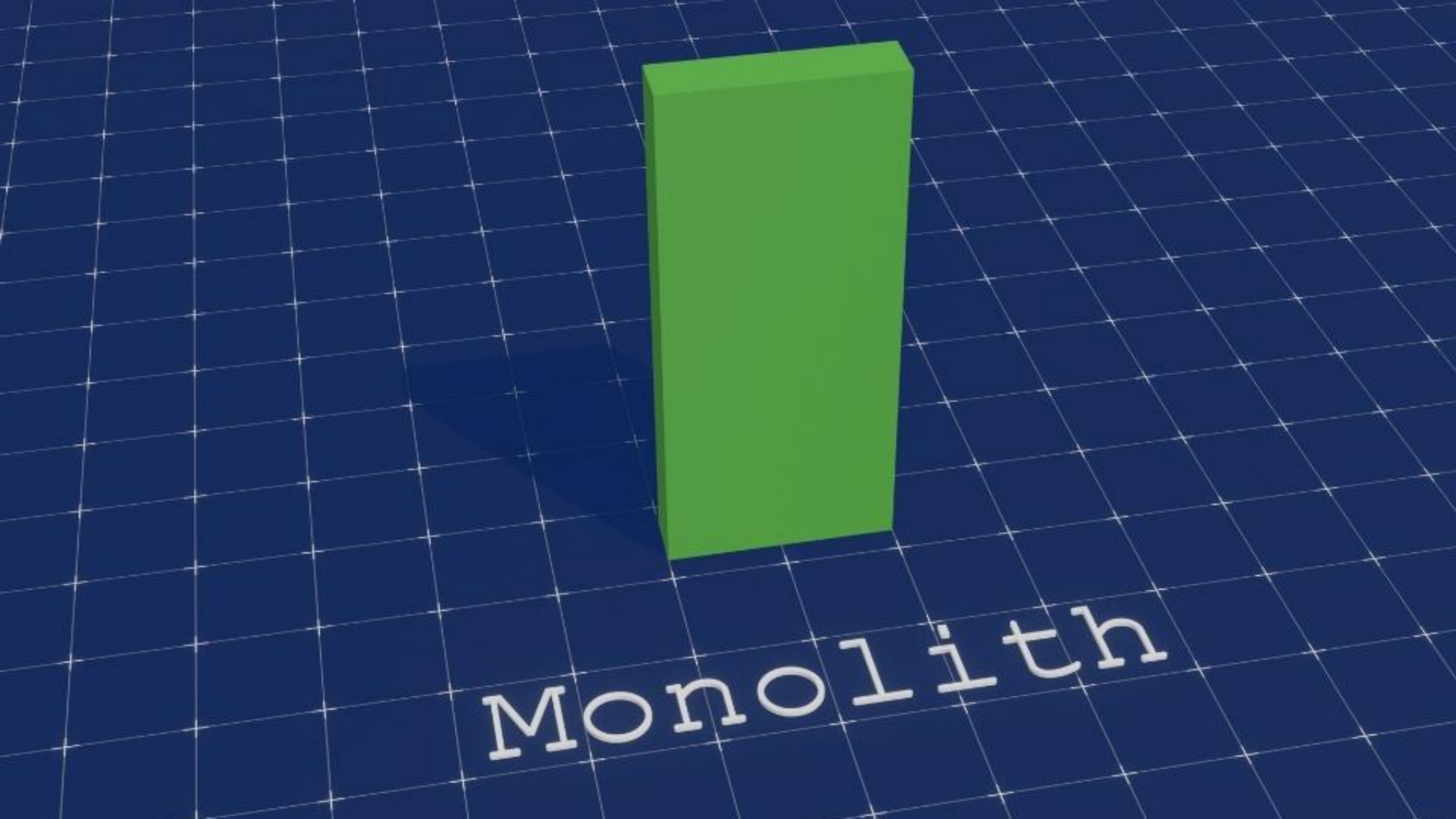
Apache Kafka

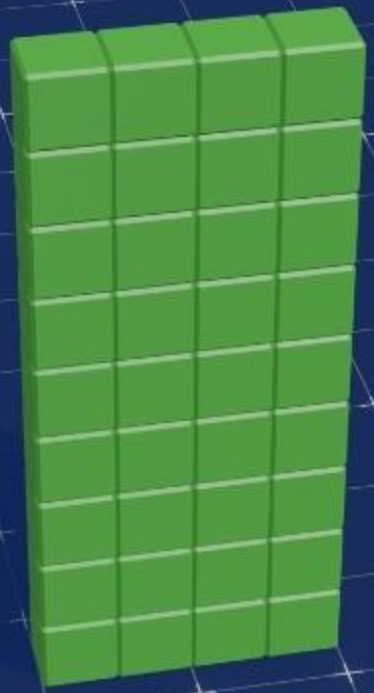
Introduction

An Evolving System will require change

Blocker:

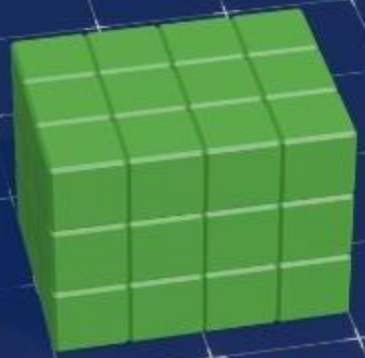
The Monolith Arch


Monolith



Modules

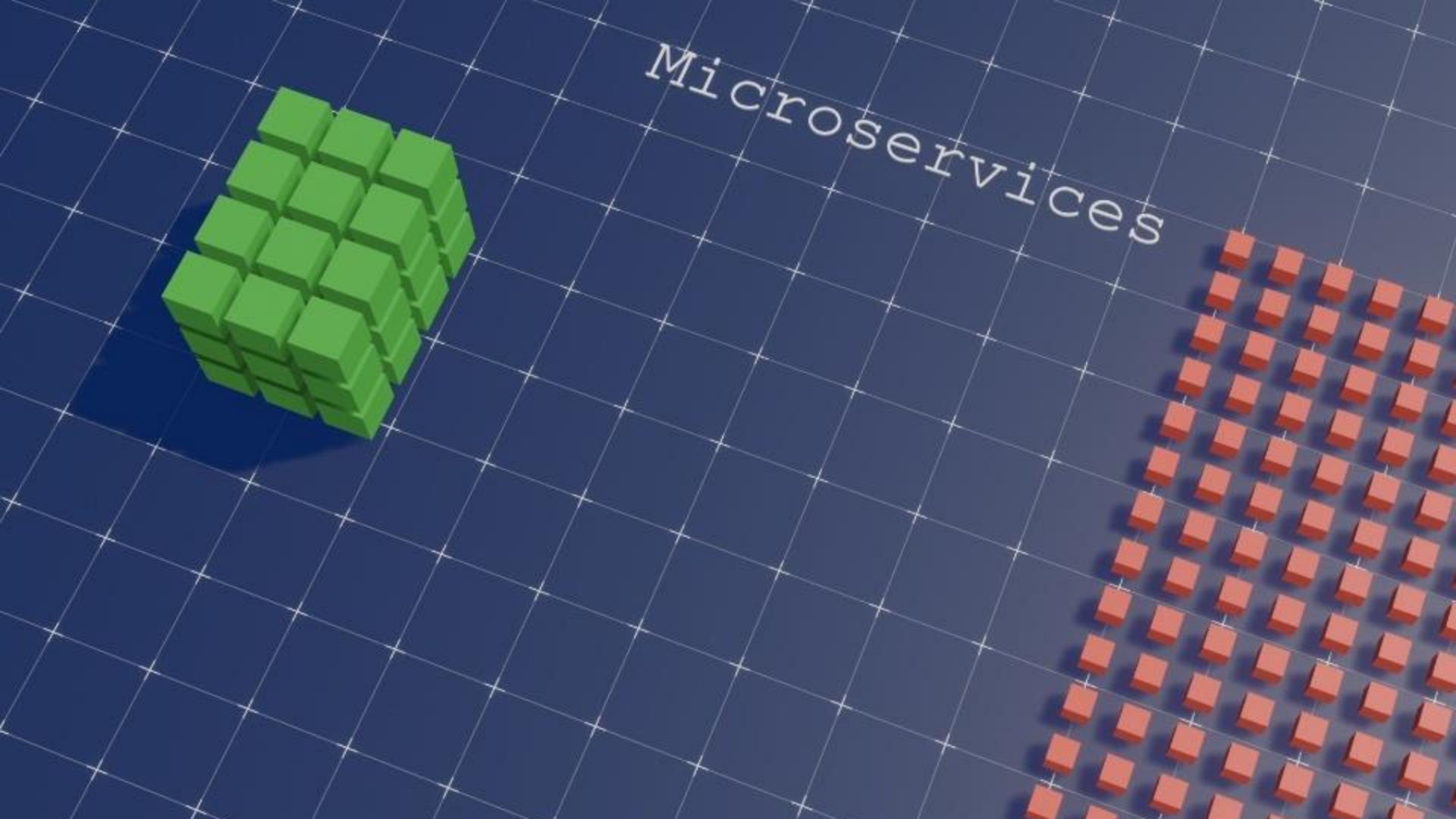
Monolith

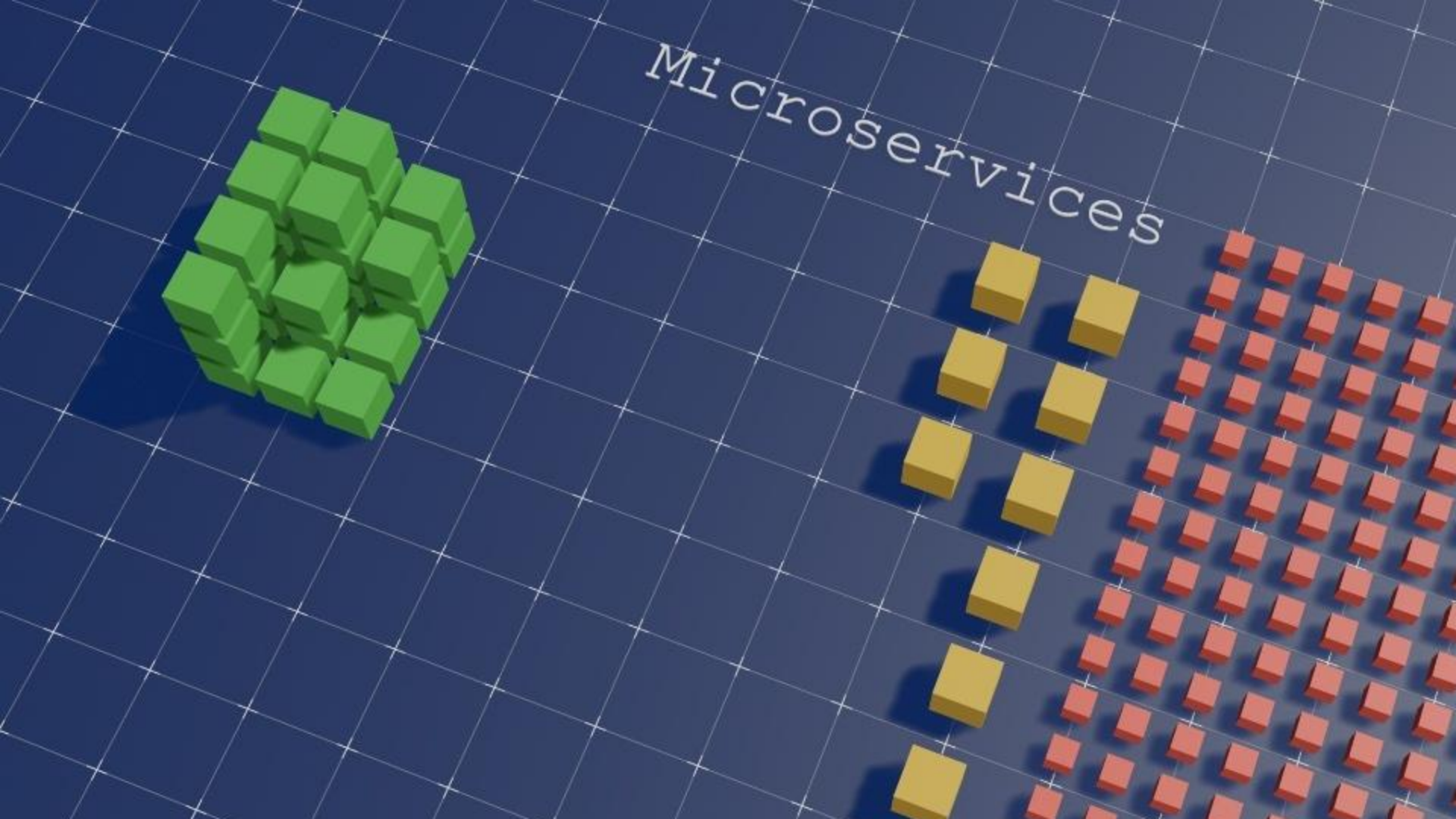


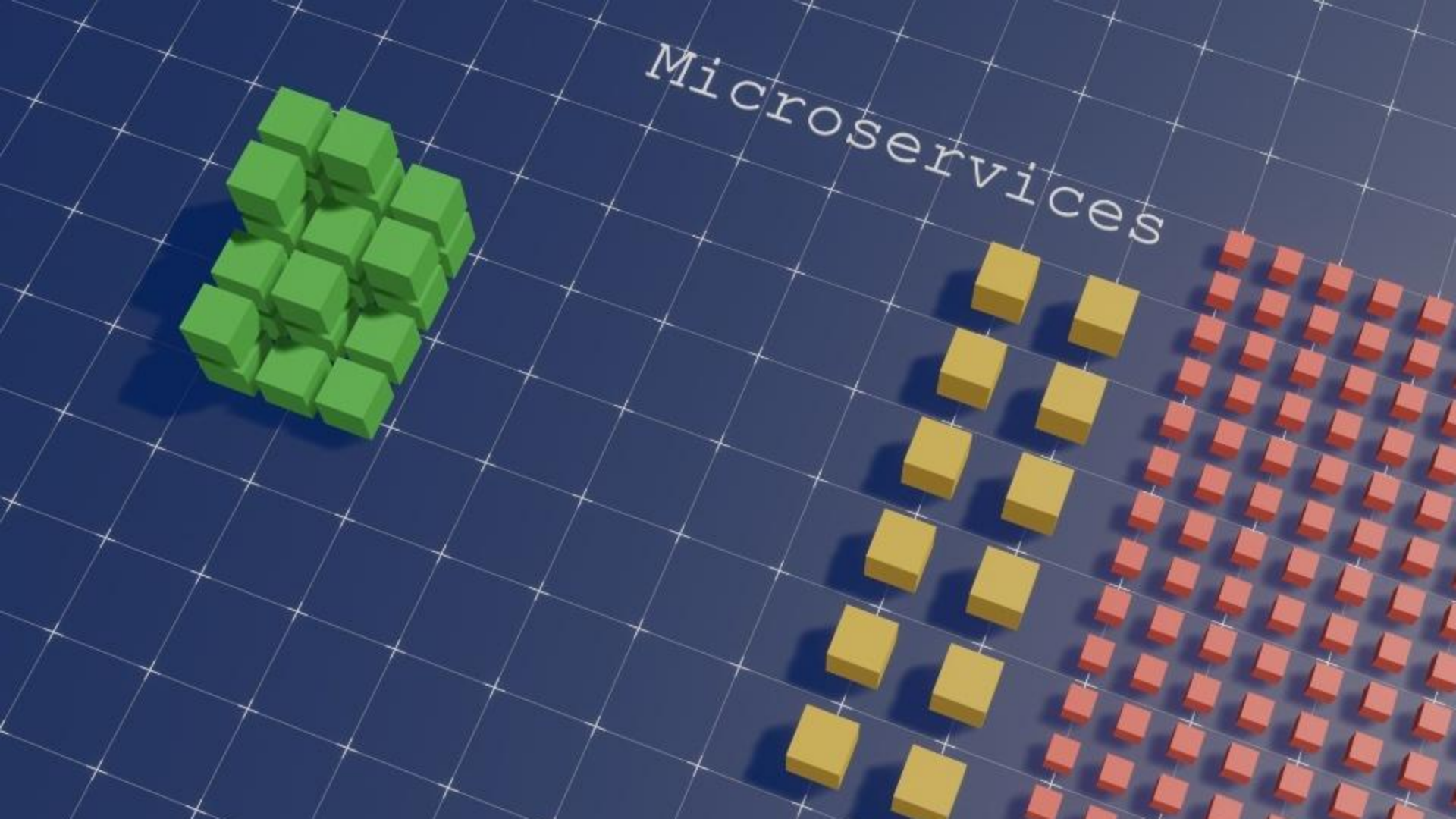
Modules

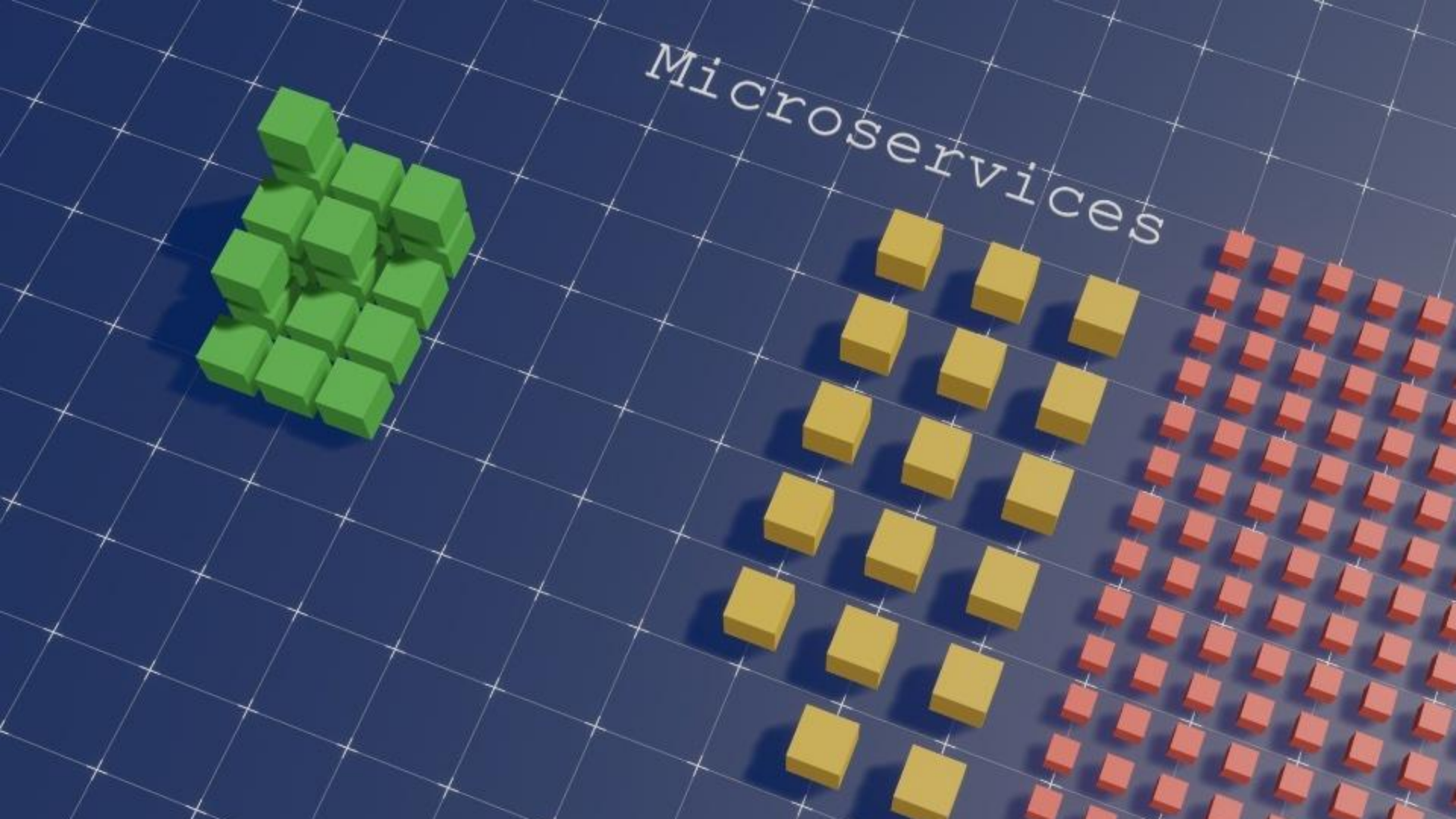
Monolith

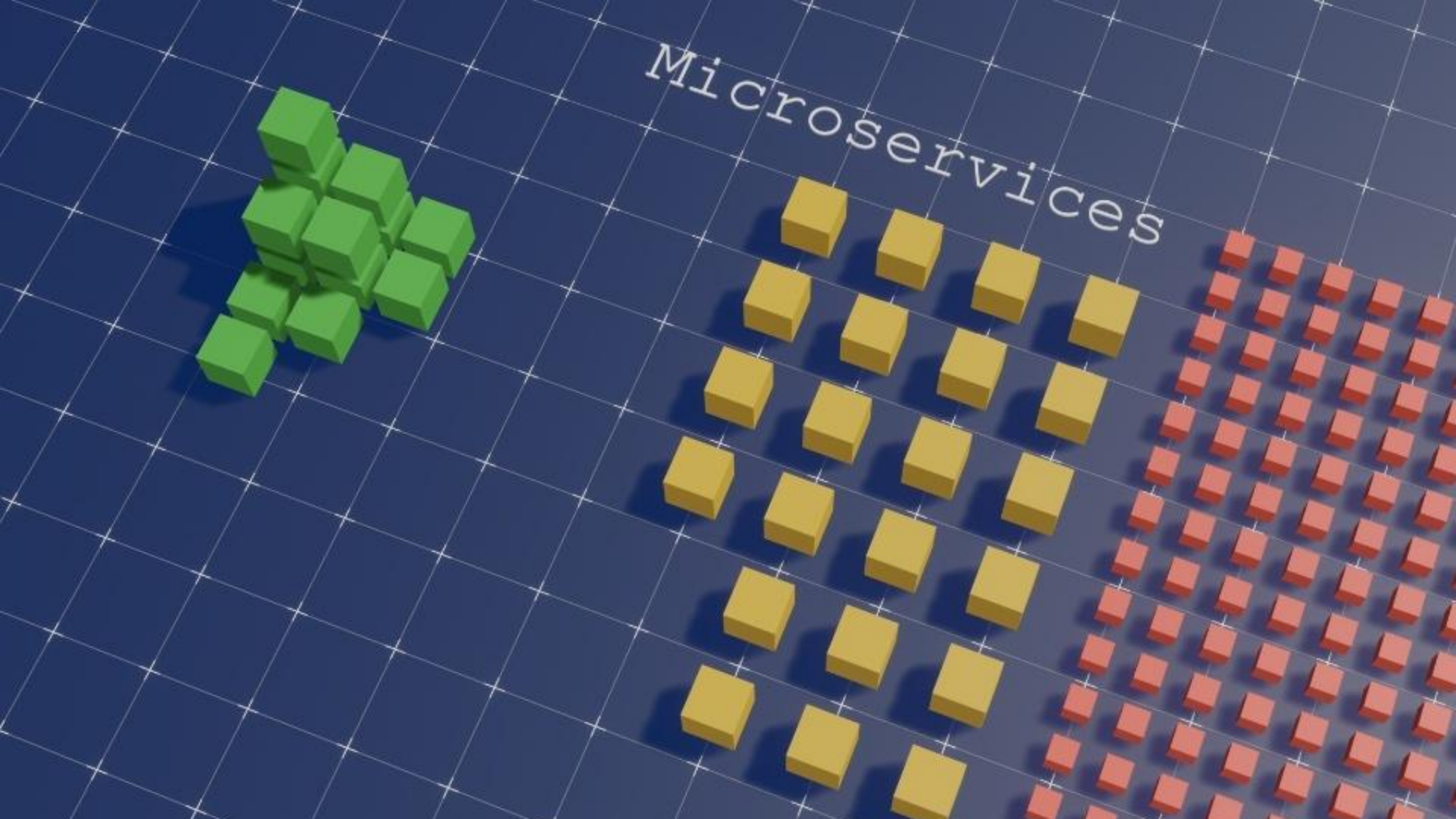
Solution:
The Distributed Systems

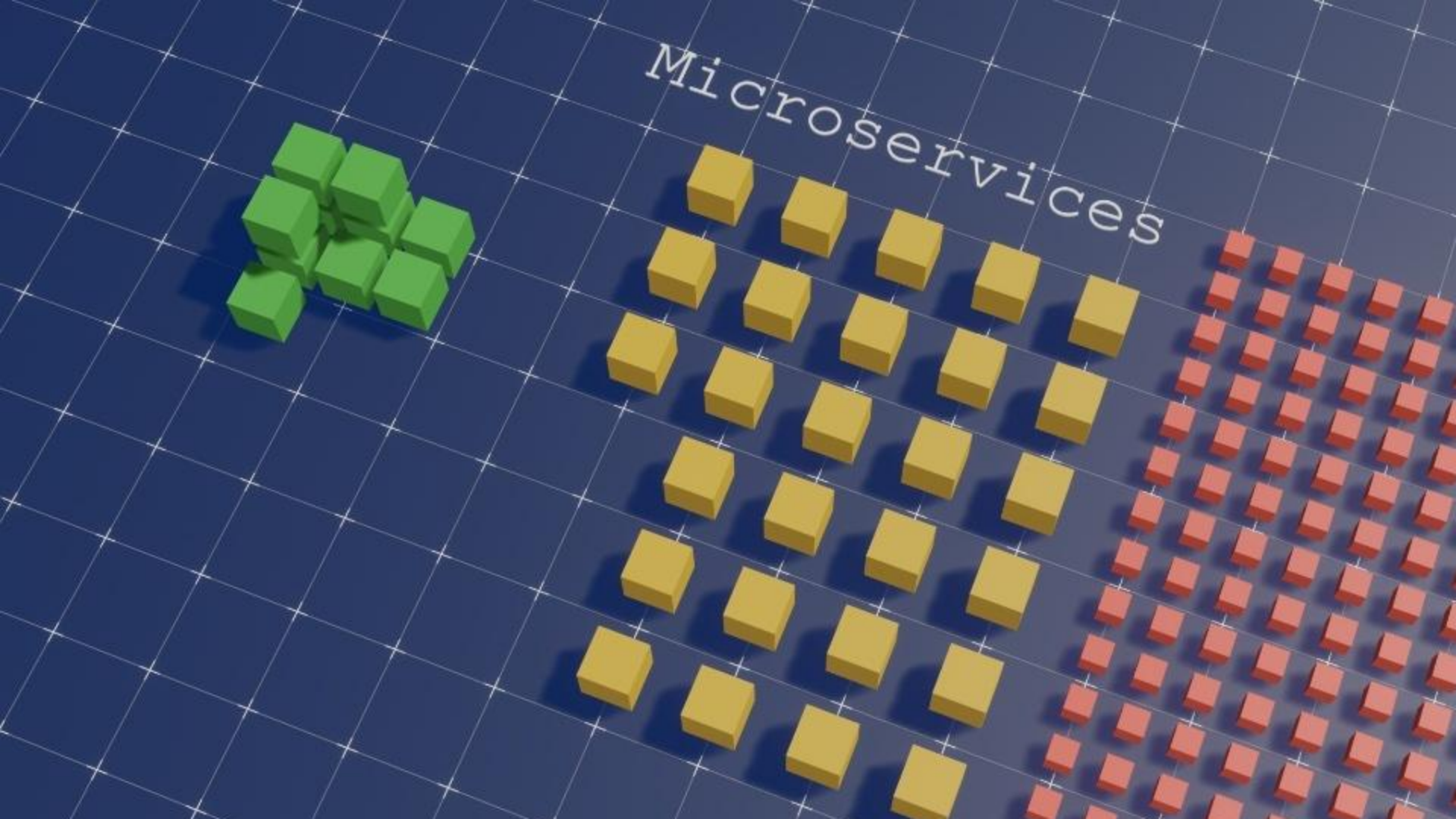

Microservices

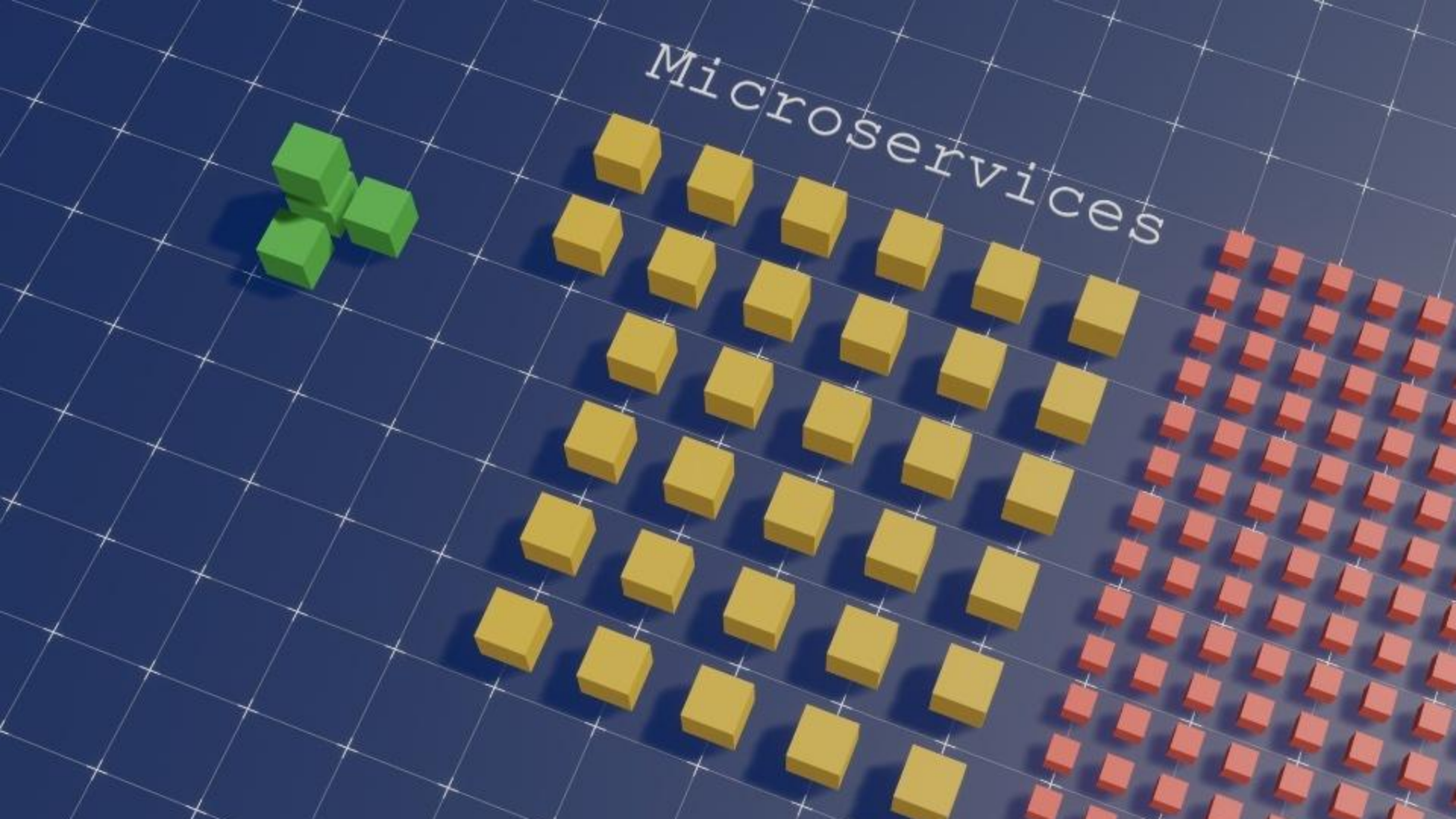

Microservices

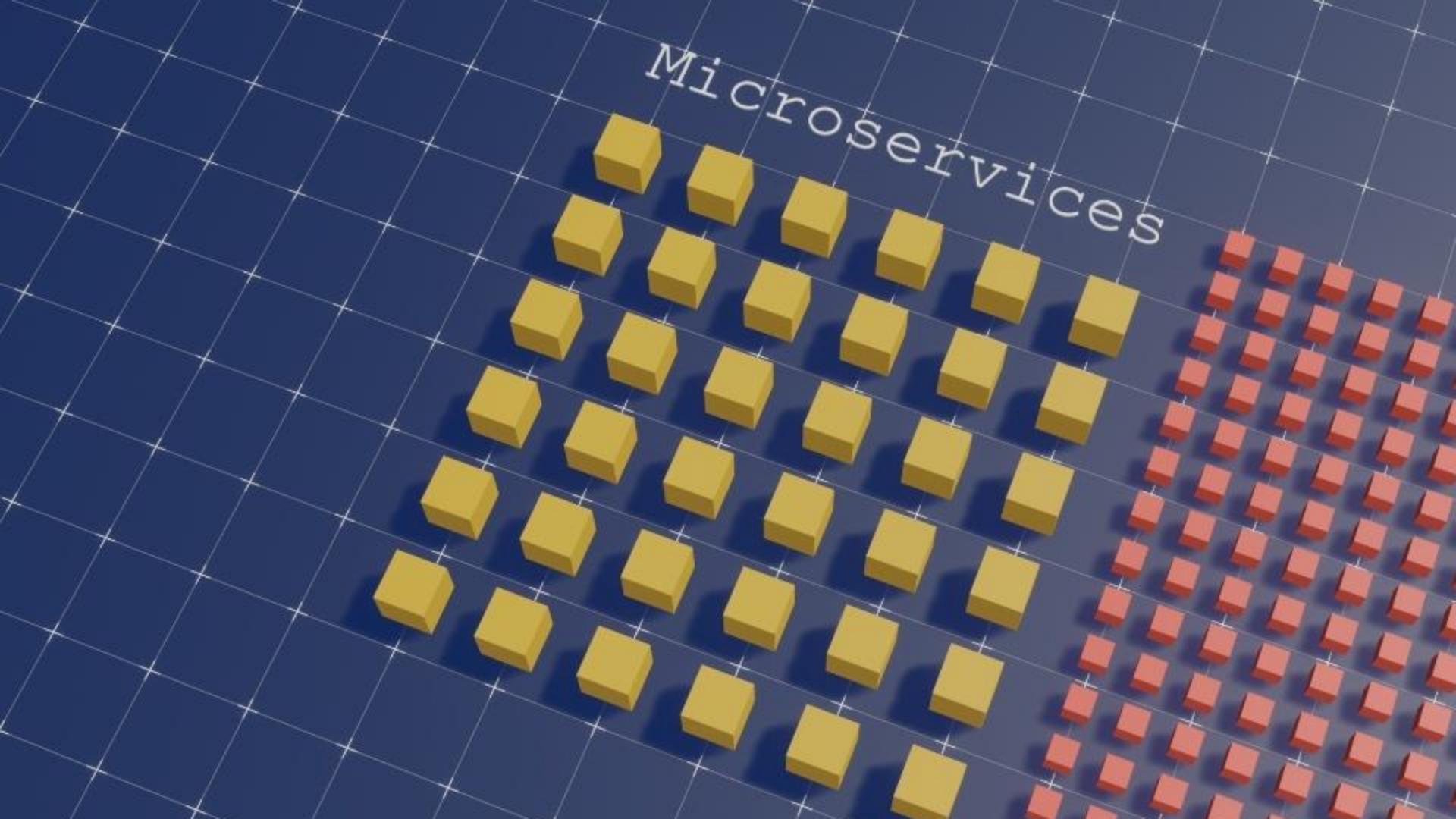

Microservices

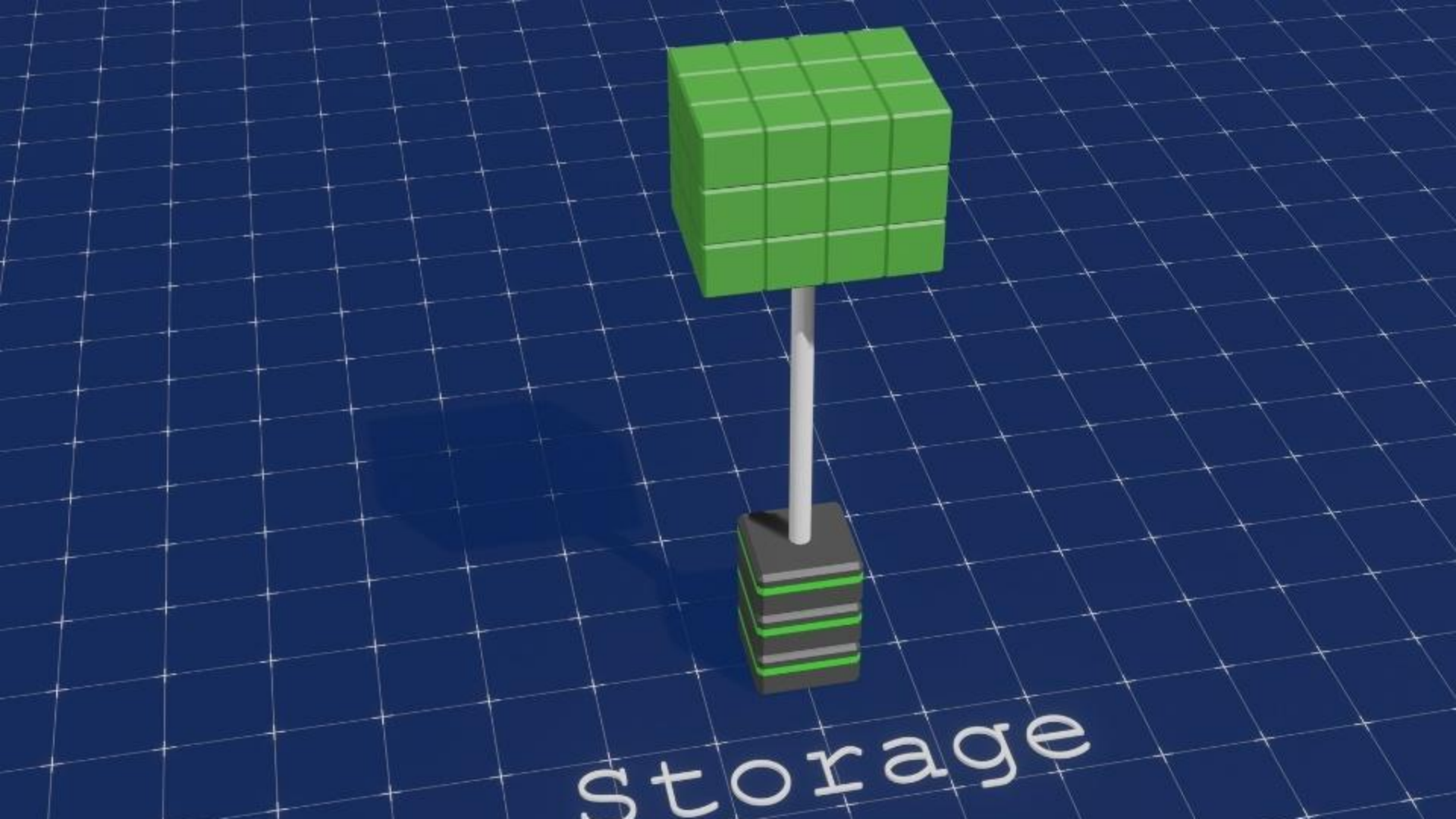

Microservices

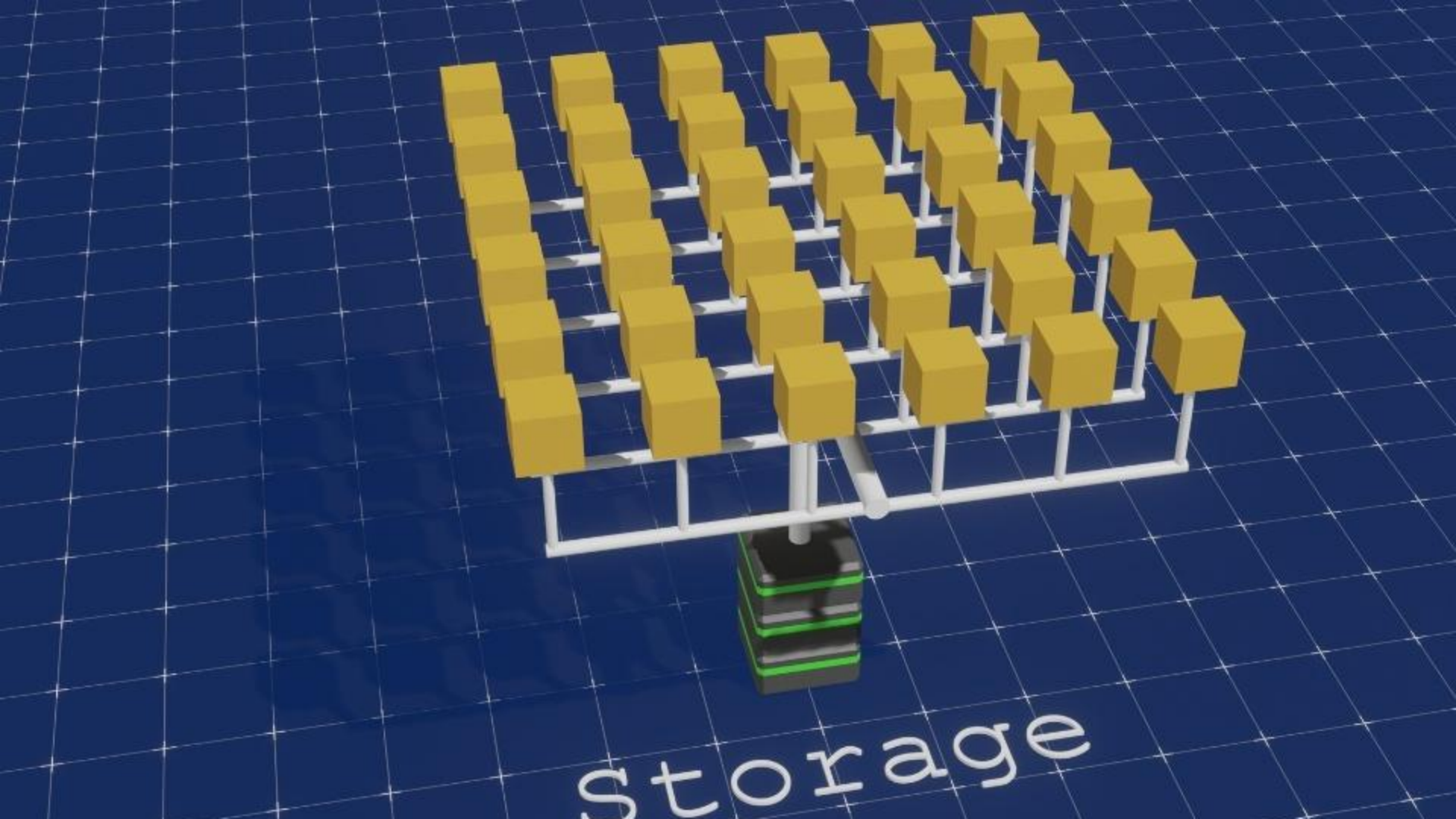

Microservices

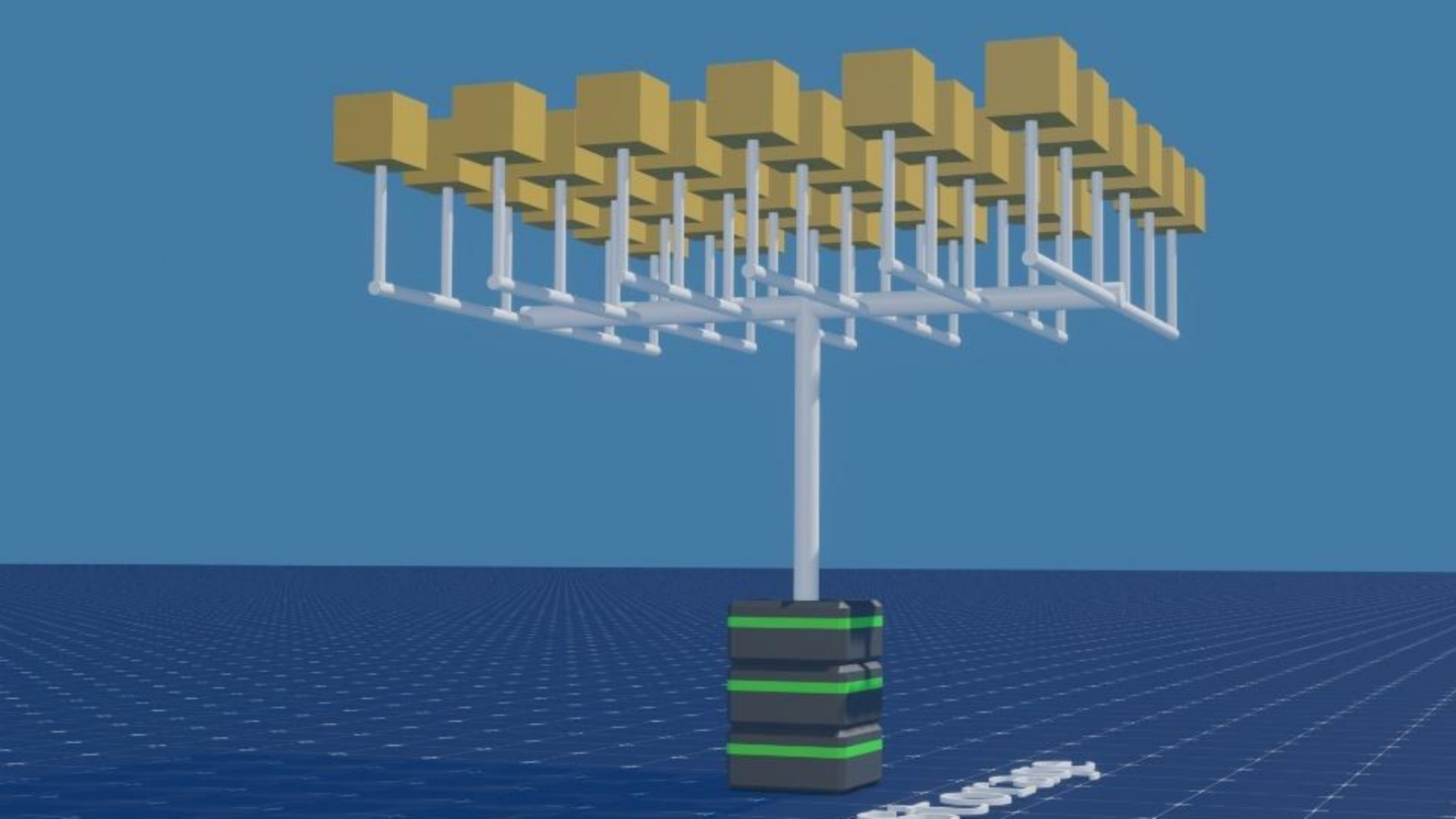

Microservices

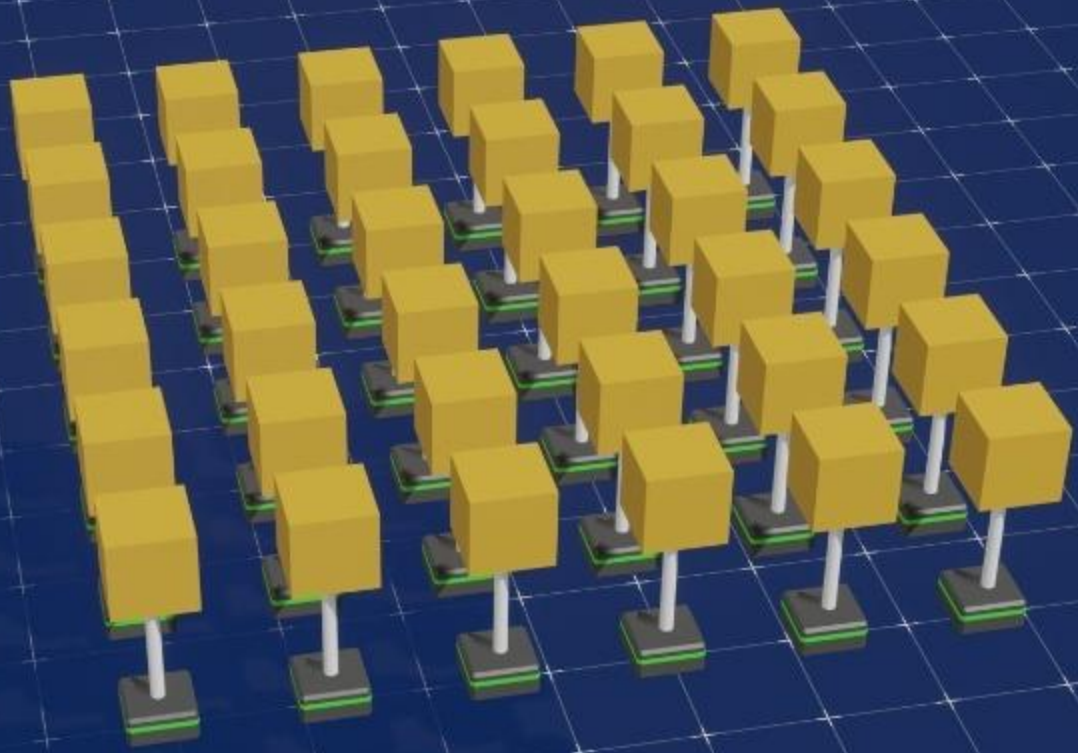

Microservices


Microservices

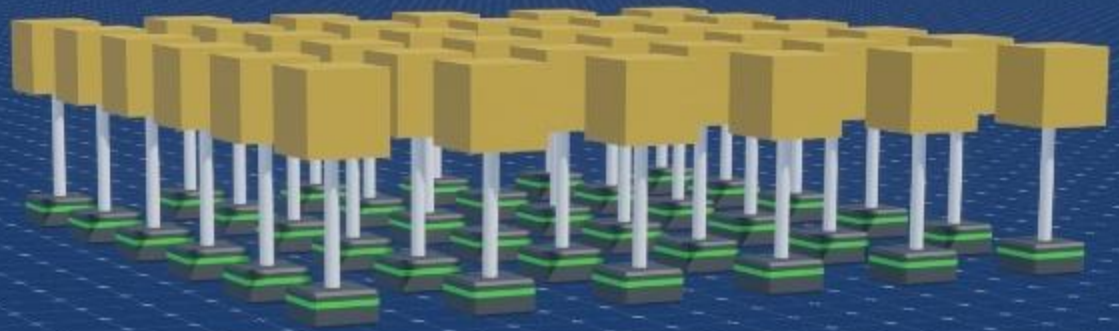

Storage


Storage

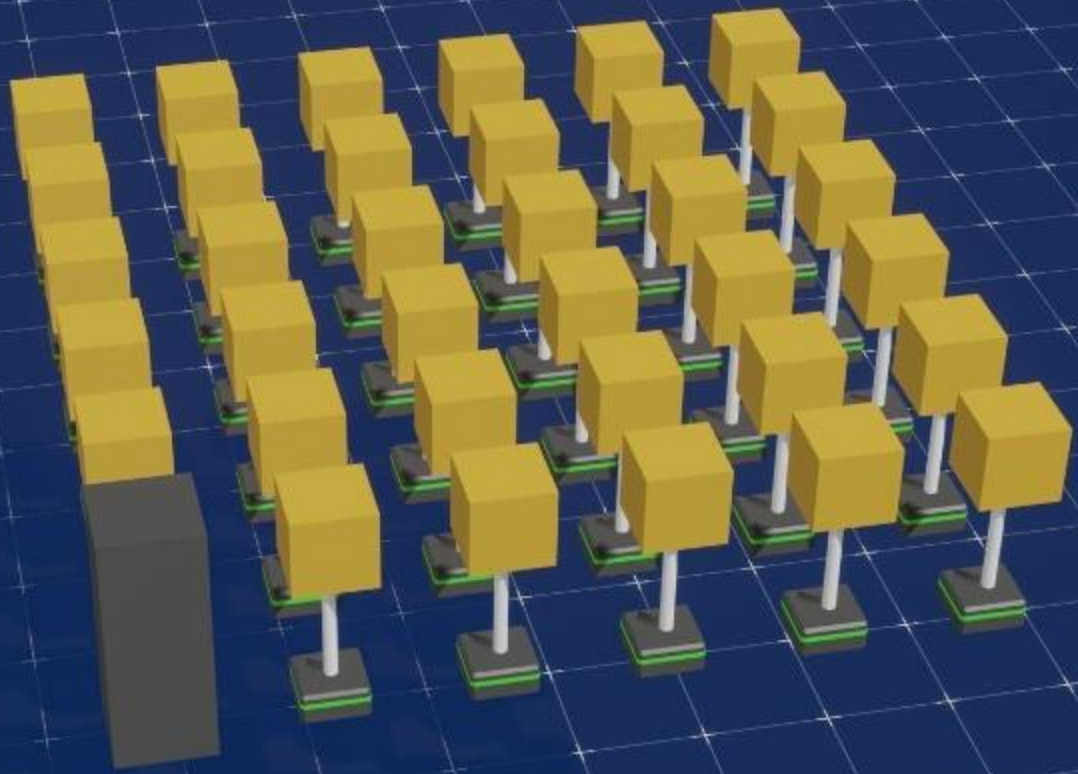




loosley coupled

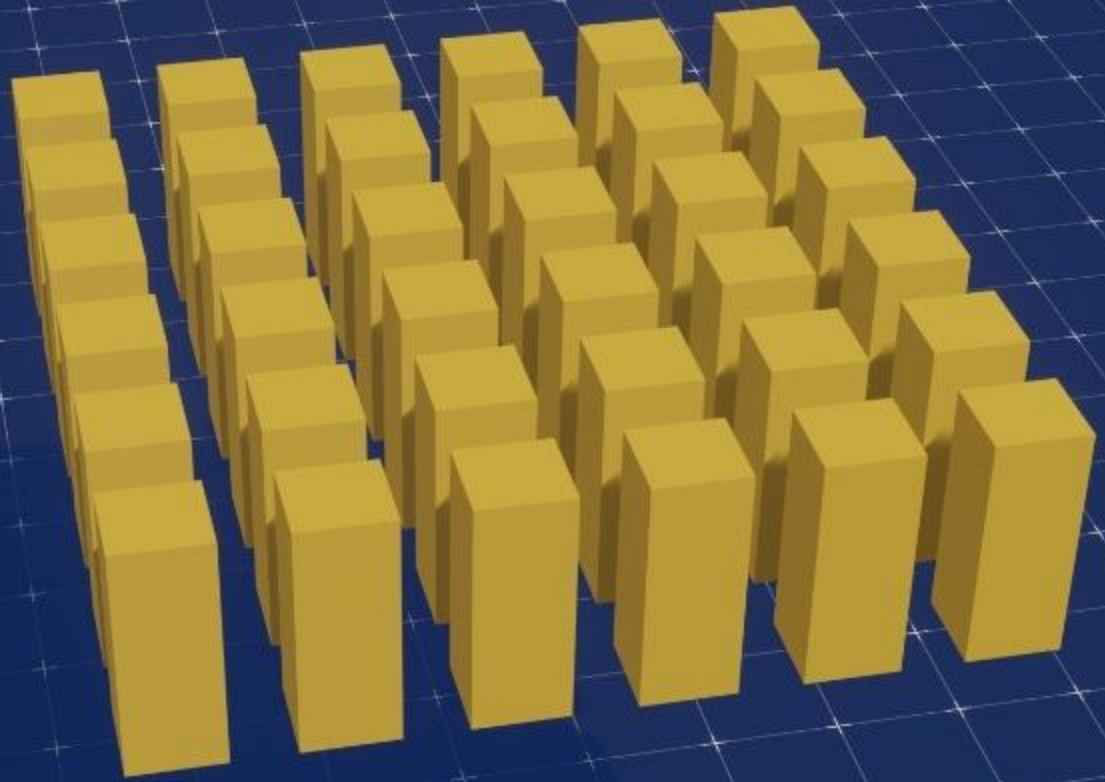


loosley coupled



loosley coupled

loosley coupled



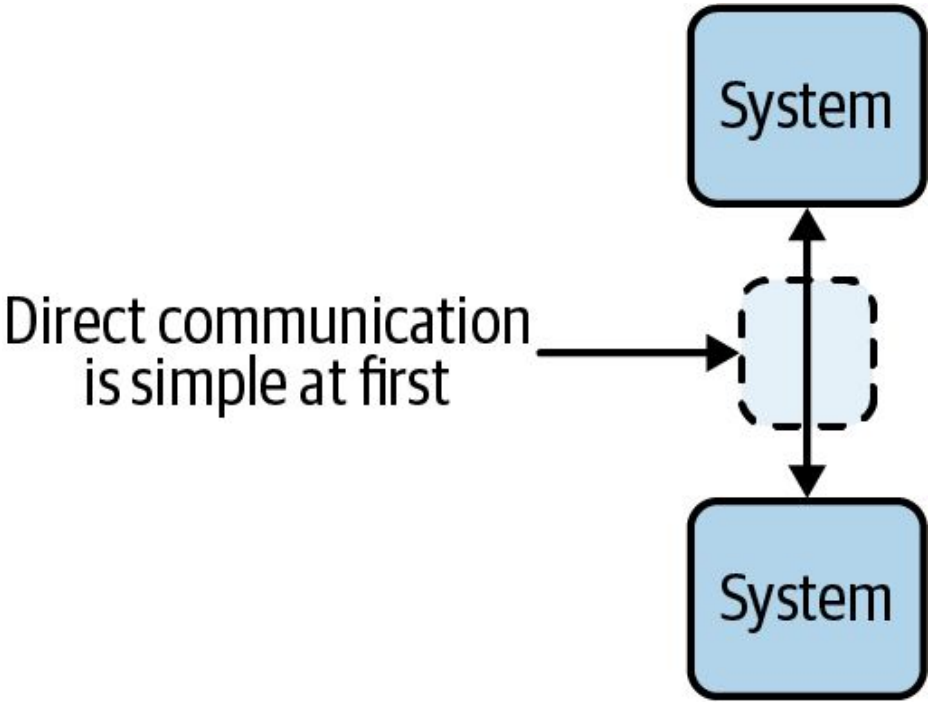
loosley coupled

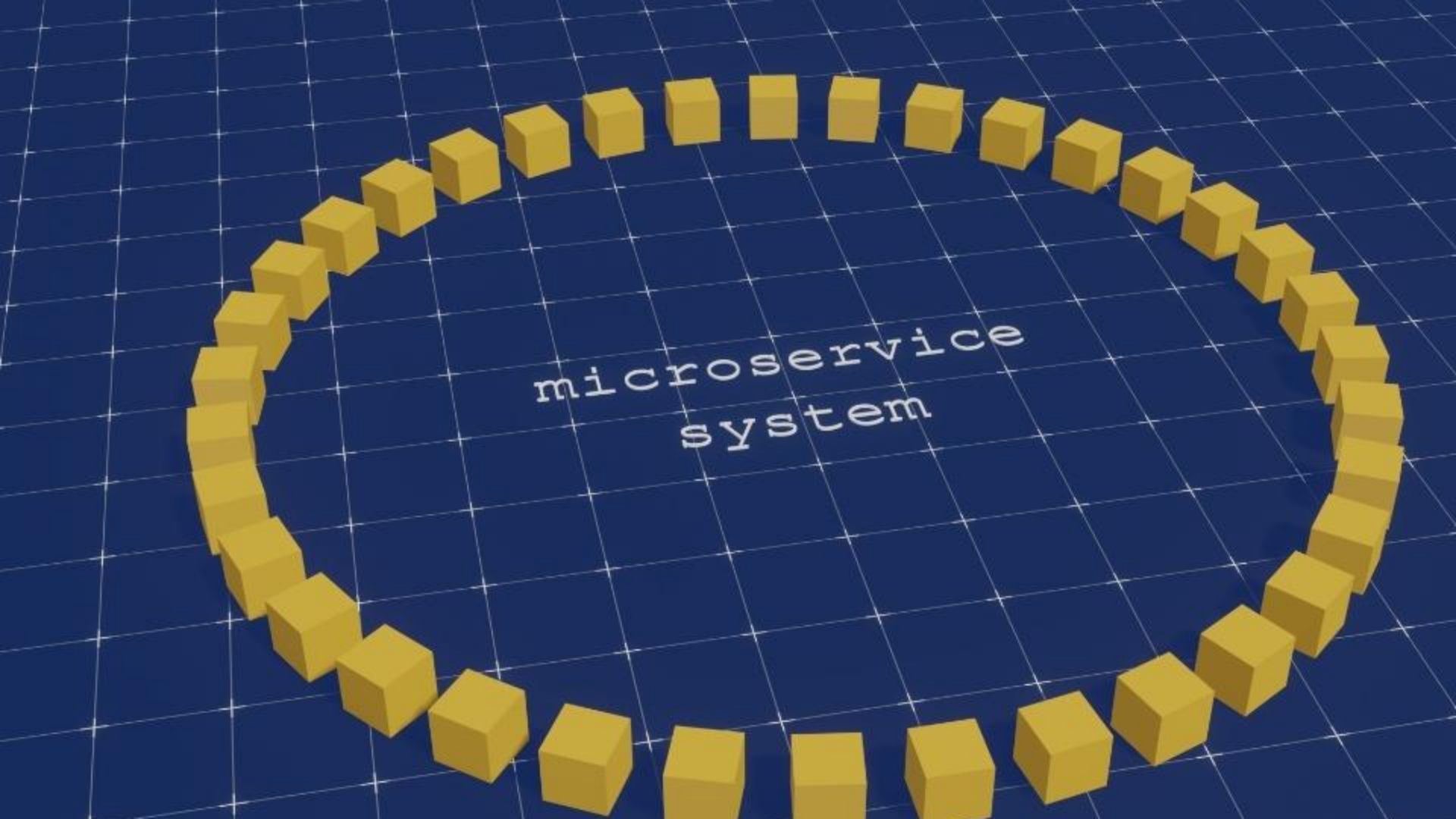
Traditional Communication Model

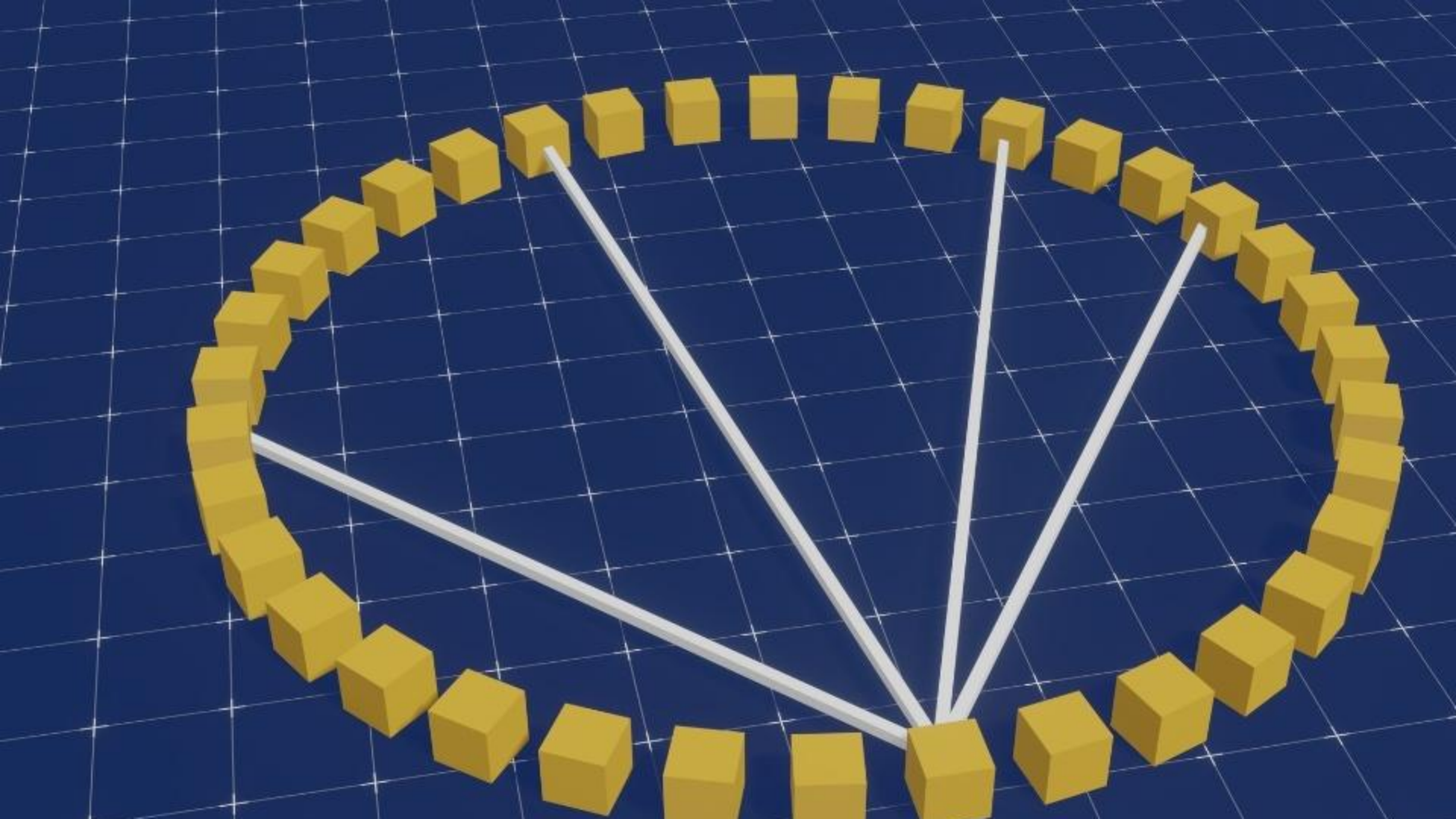
Client-Server Communication

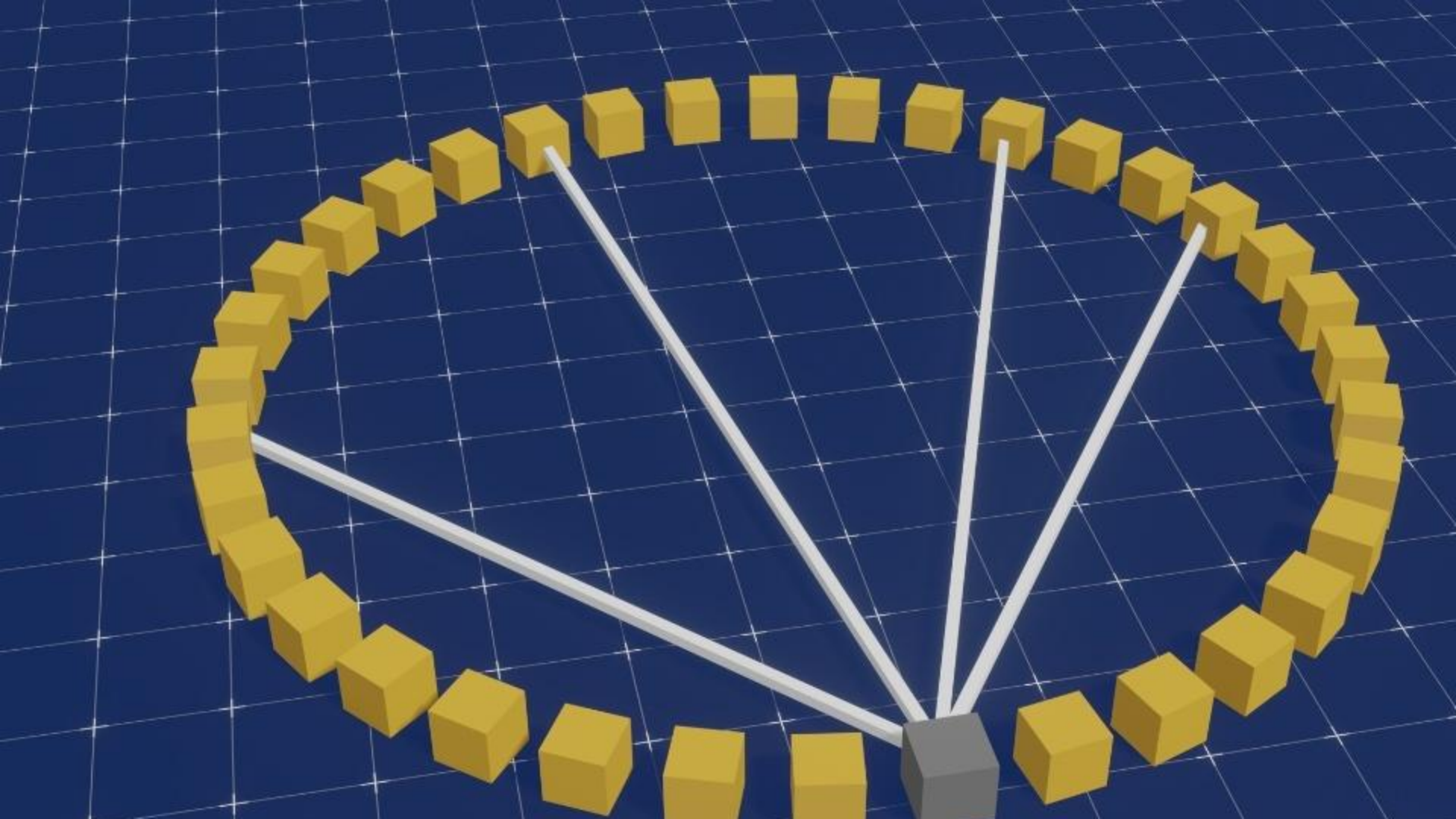
The Client-Server Communication Model

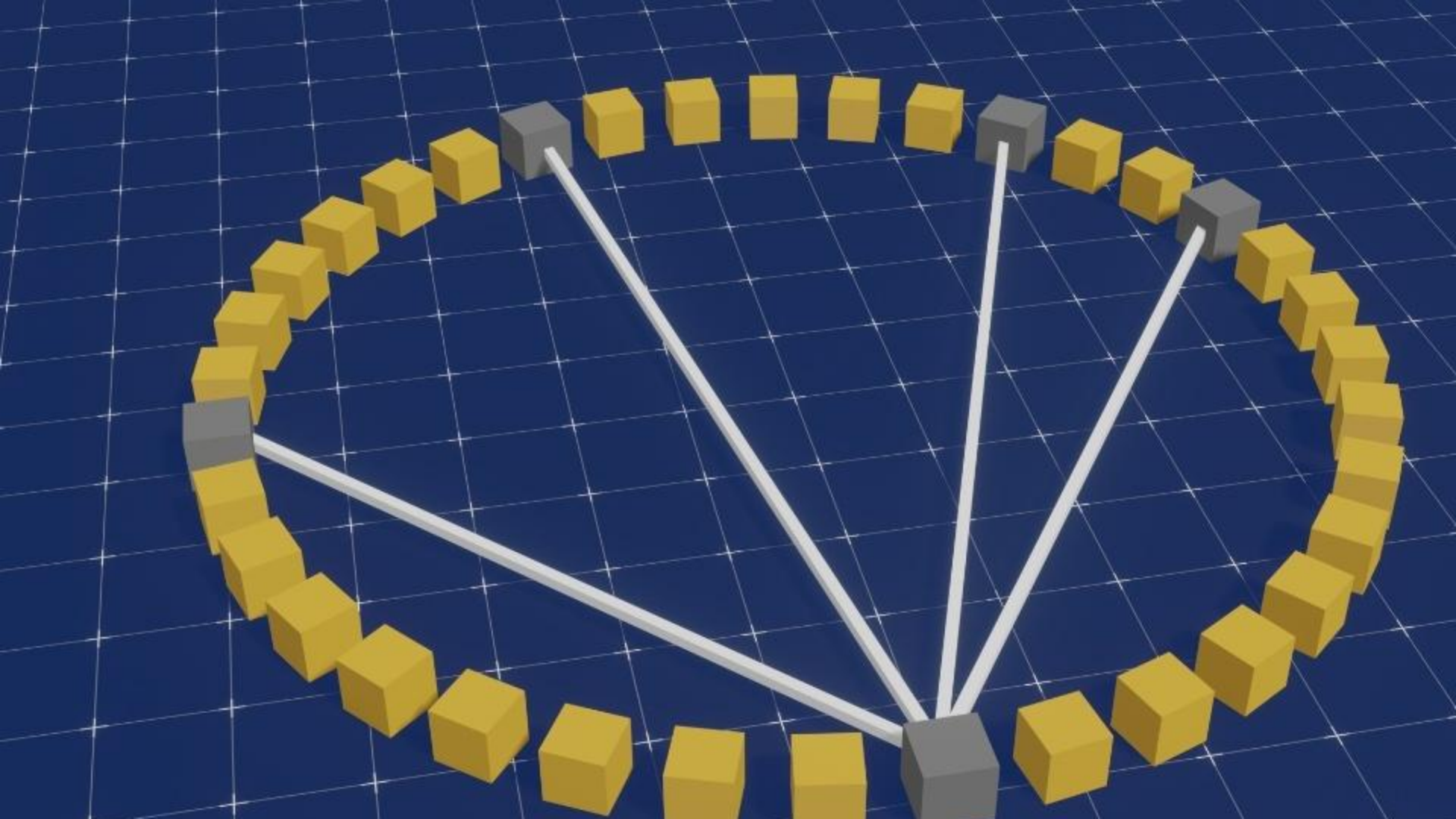
- Common in applications, microservices, and databases
- Simple and direct but becomes complex with scale



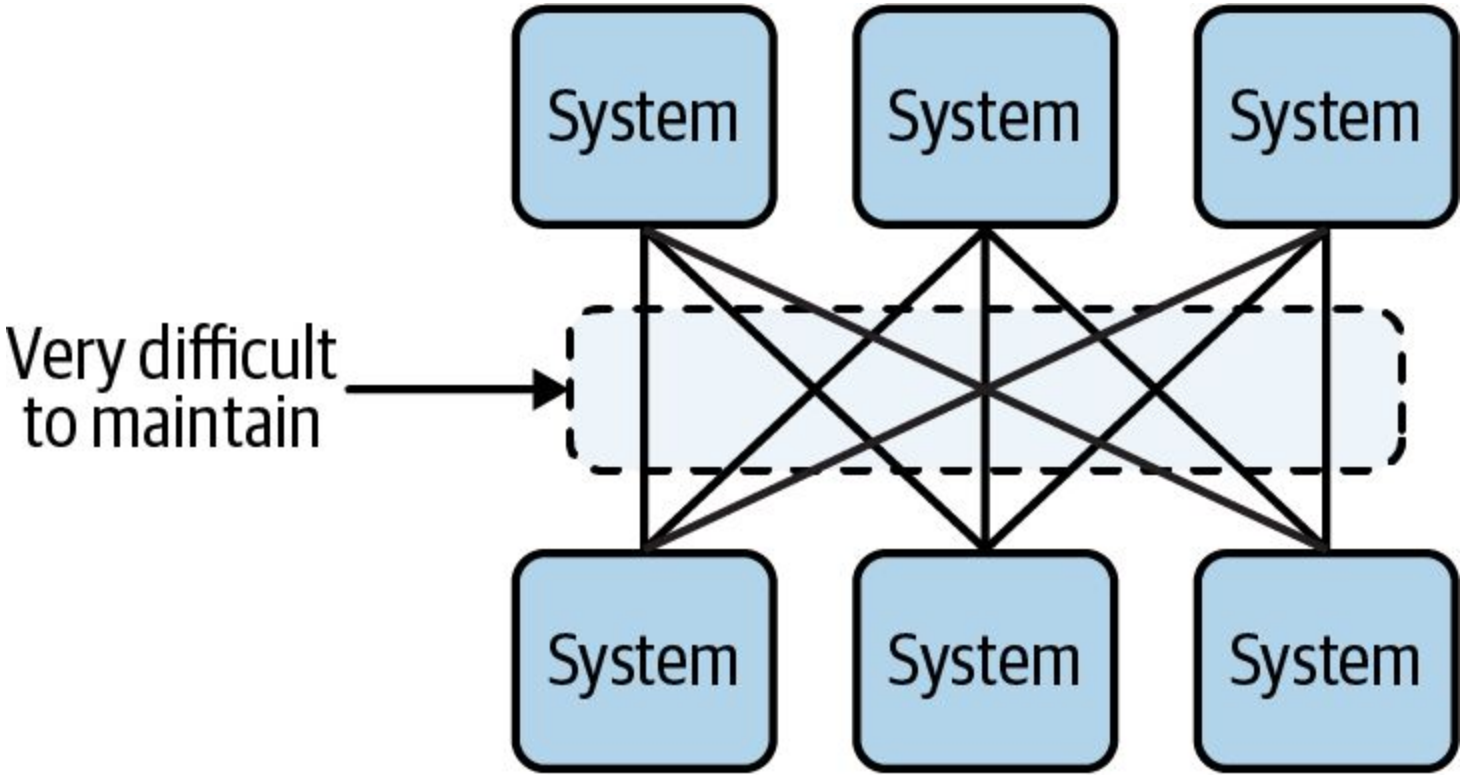

microservice
system







Challenges of scaling point-to-point communications



Drawbacks of Client-Server Model

- Tightly coupled systems, difficult maintenance and updates
- Synchronous pitfalls: No delivery guarantees if a system goes offline
- Communication inconsistencies: Different protocols and strategies
- Data flow issues: Overwhelming receiving systems, lack of replayability

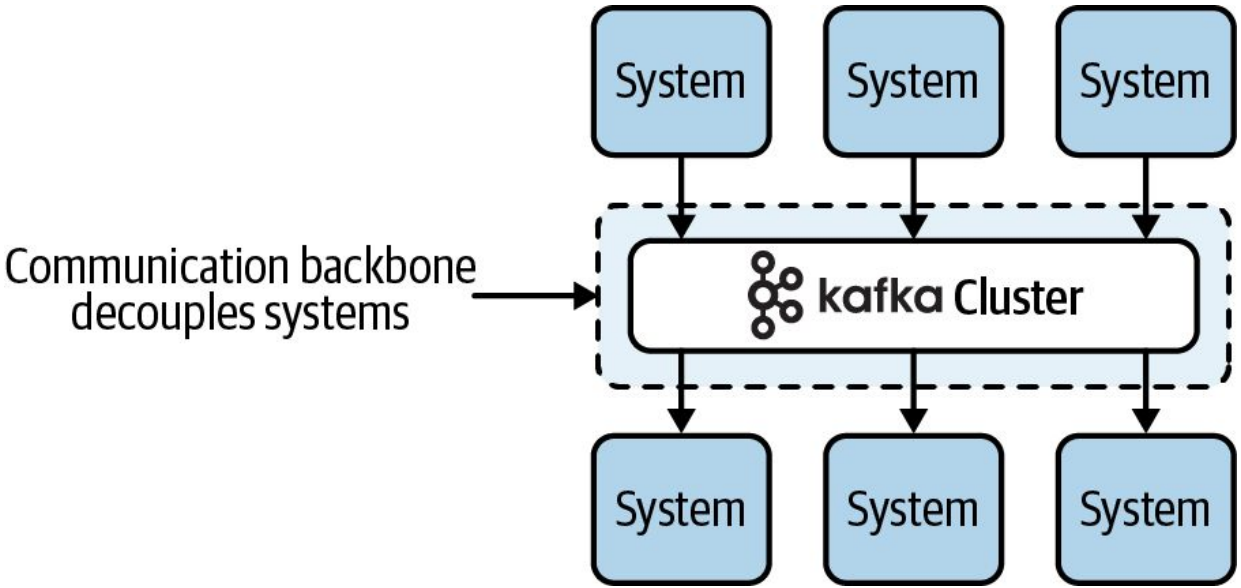
Client-Server Model - challenges

1. Cross language communication
2. Service-discovery
3. Load-balancing
4. Network-latency
5. Reliability
6. Versioning & Compatibility

Introducing Kafka's Communication Model

Kafka as a Central Communication Hub

- Decentralizes and simplifies system communication
- Publish-subscribe model replaces direct interactions

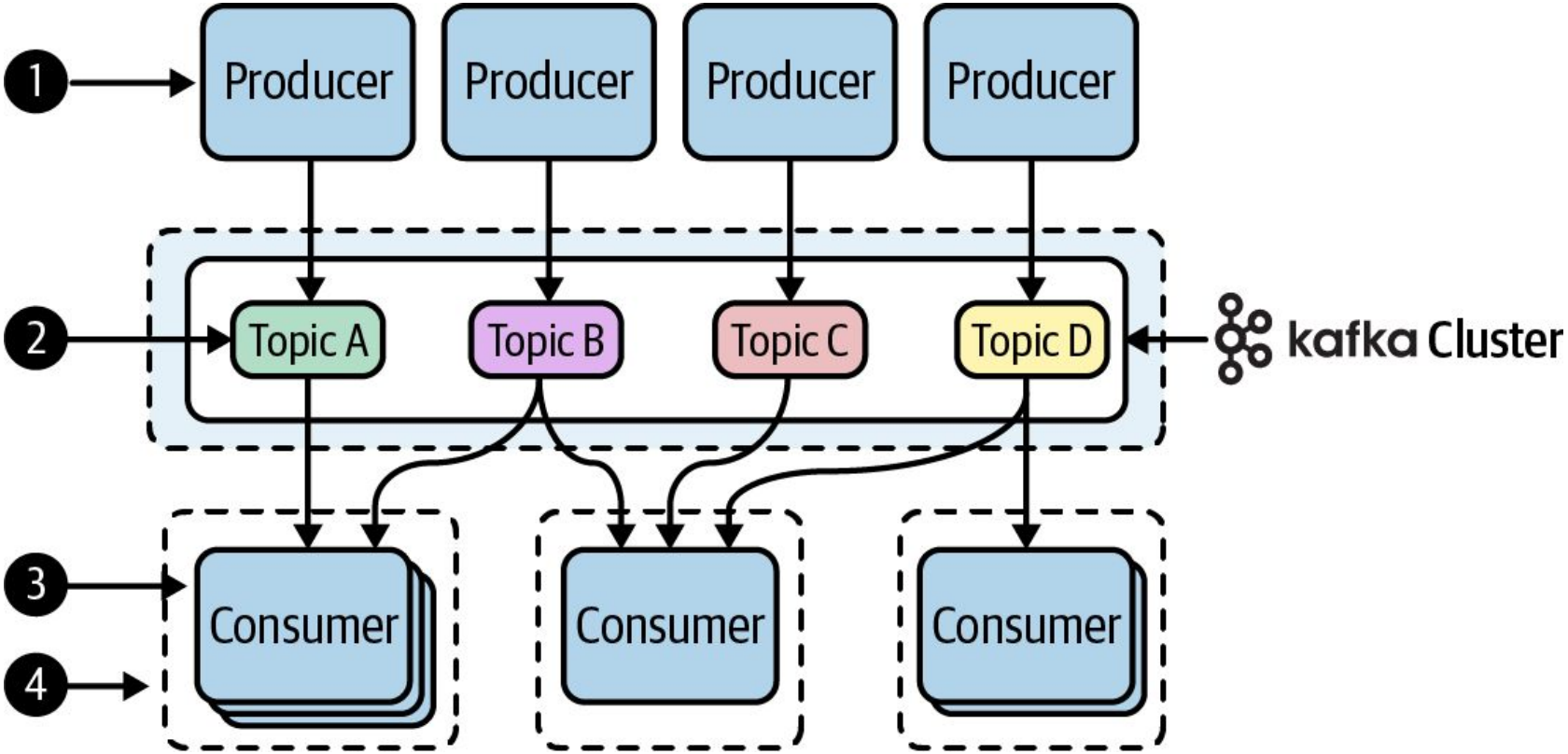


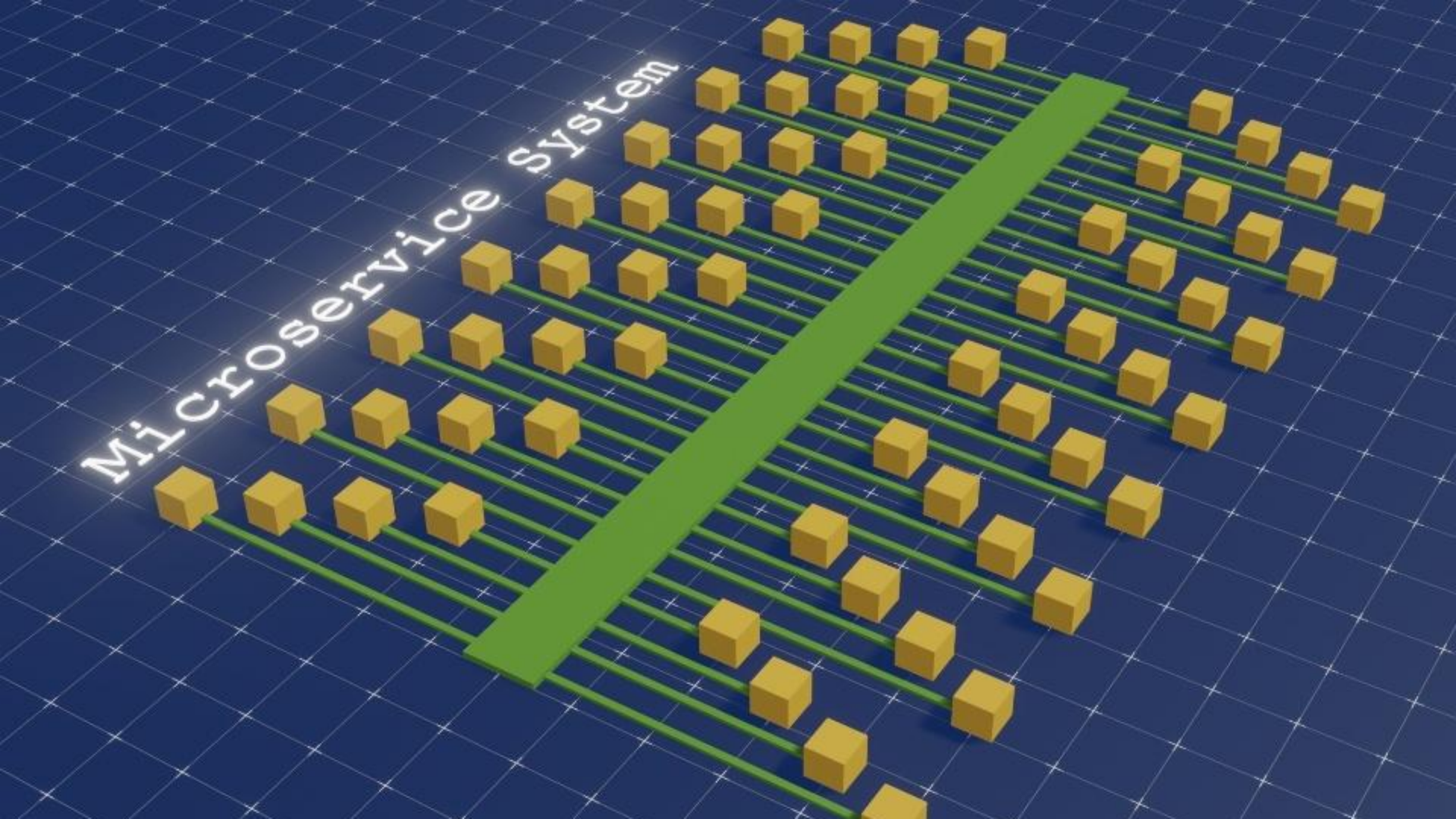
Kafka’s Publish-Subscribe Mechanics

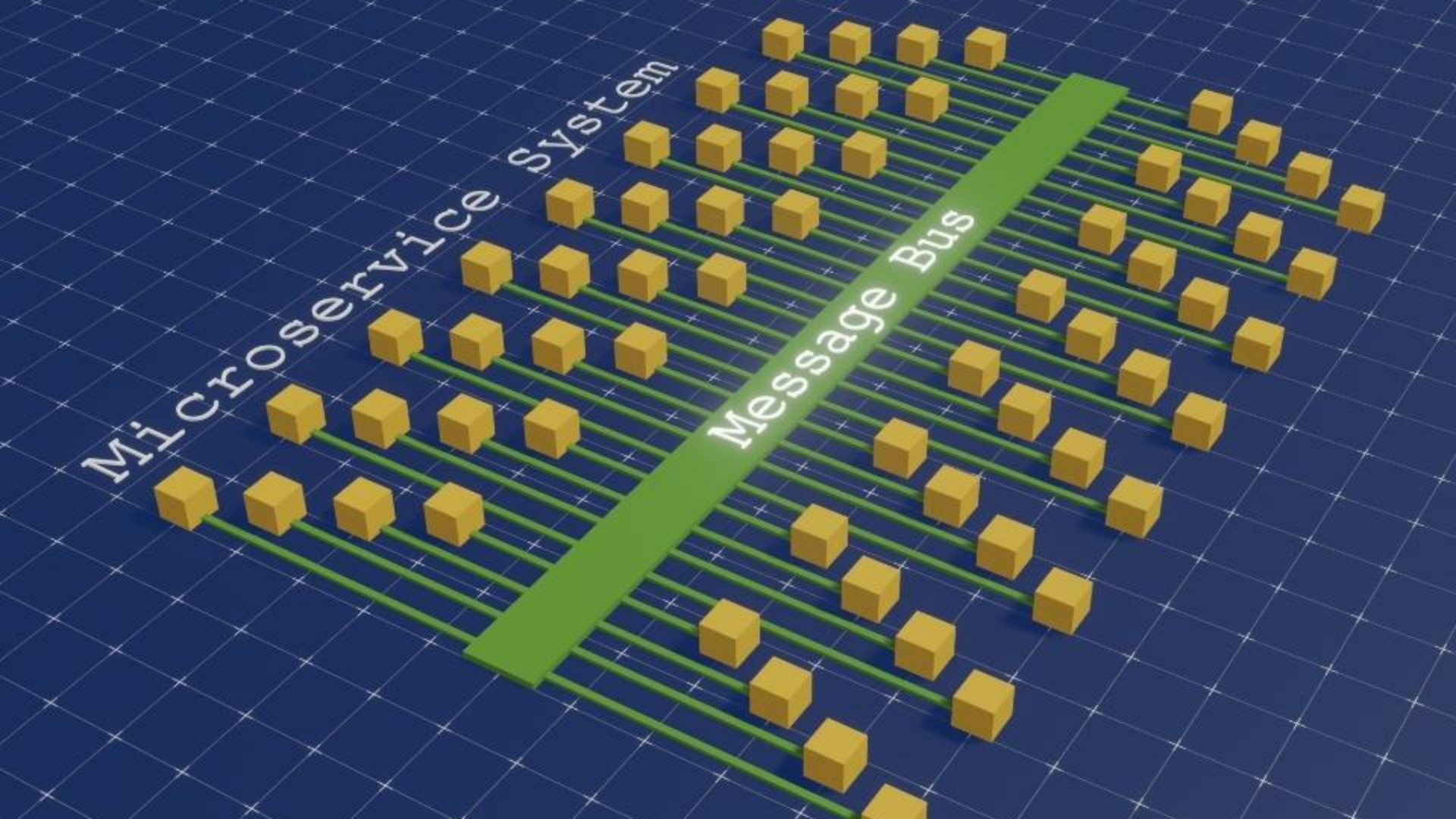
Understanding Kafka's Pub/Sub Model

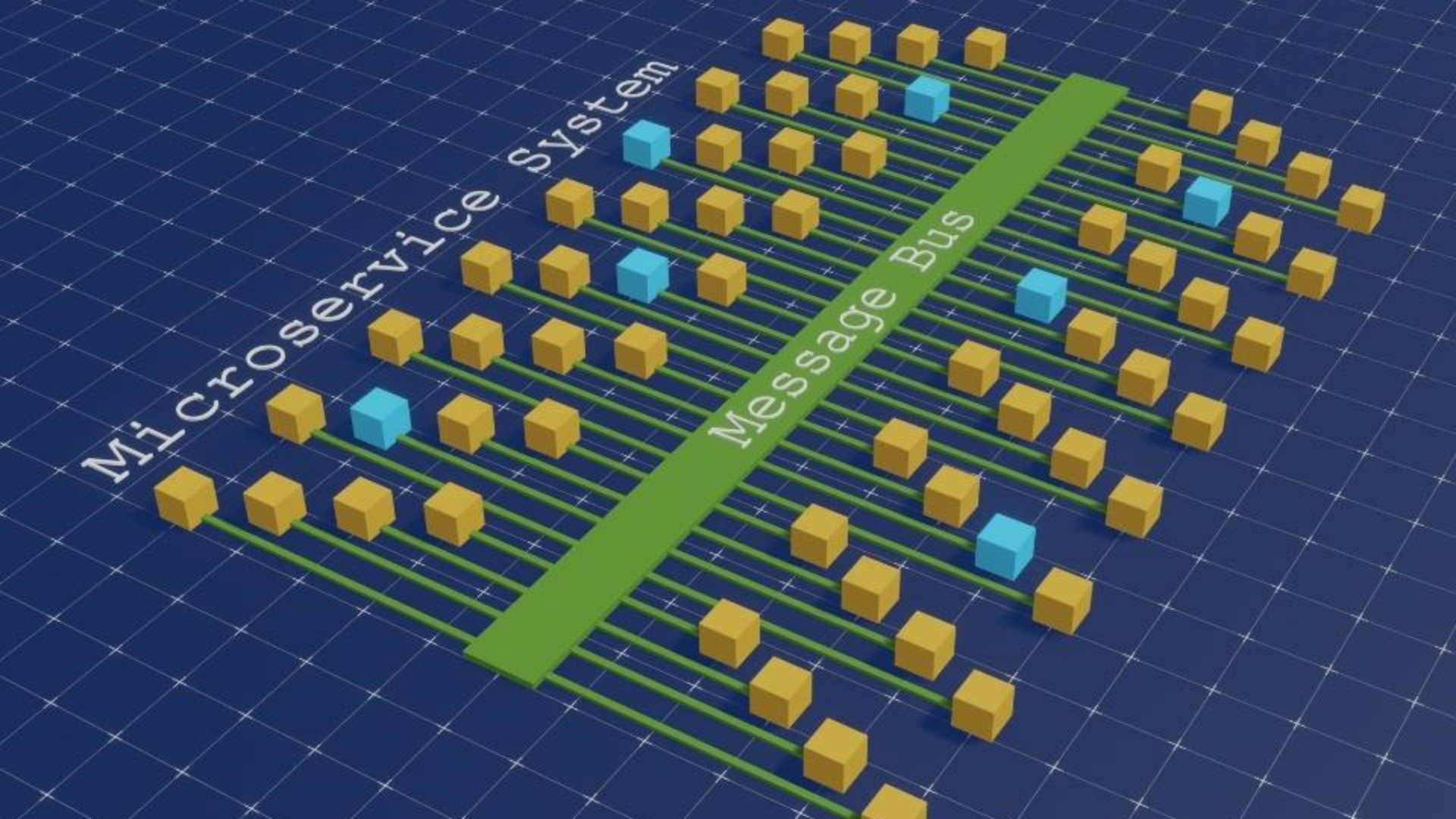
- Producers publish data to topics without knowing the consumers
- Consumers subscribe to topics of interest, decoupling from producers
- Consumer groups for distributed processing and fault tolerance

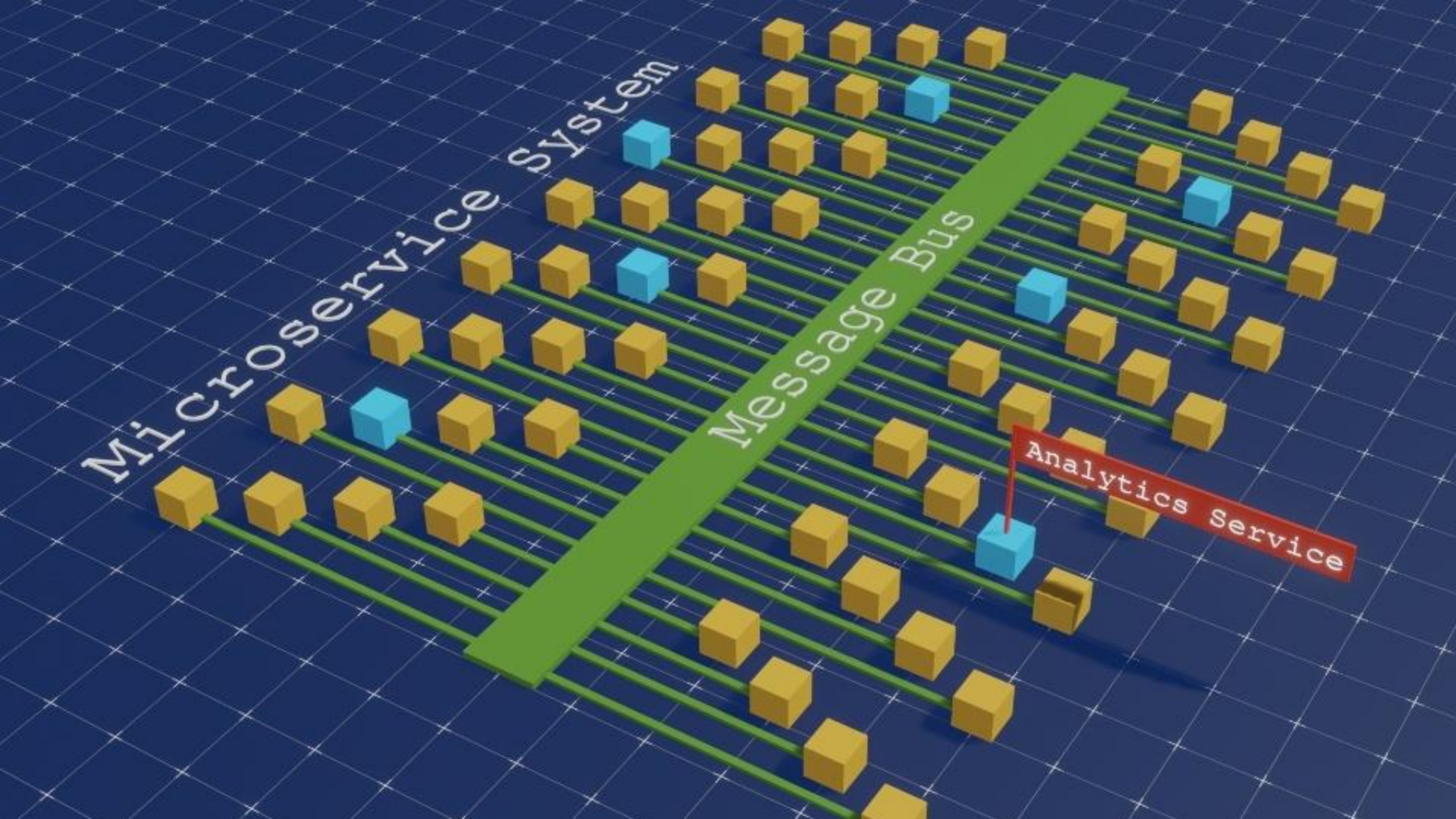
Understanding Kafka's Pub/Sub Model

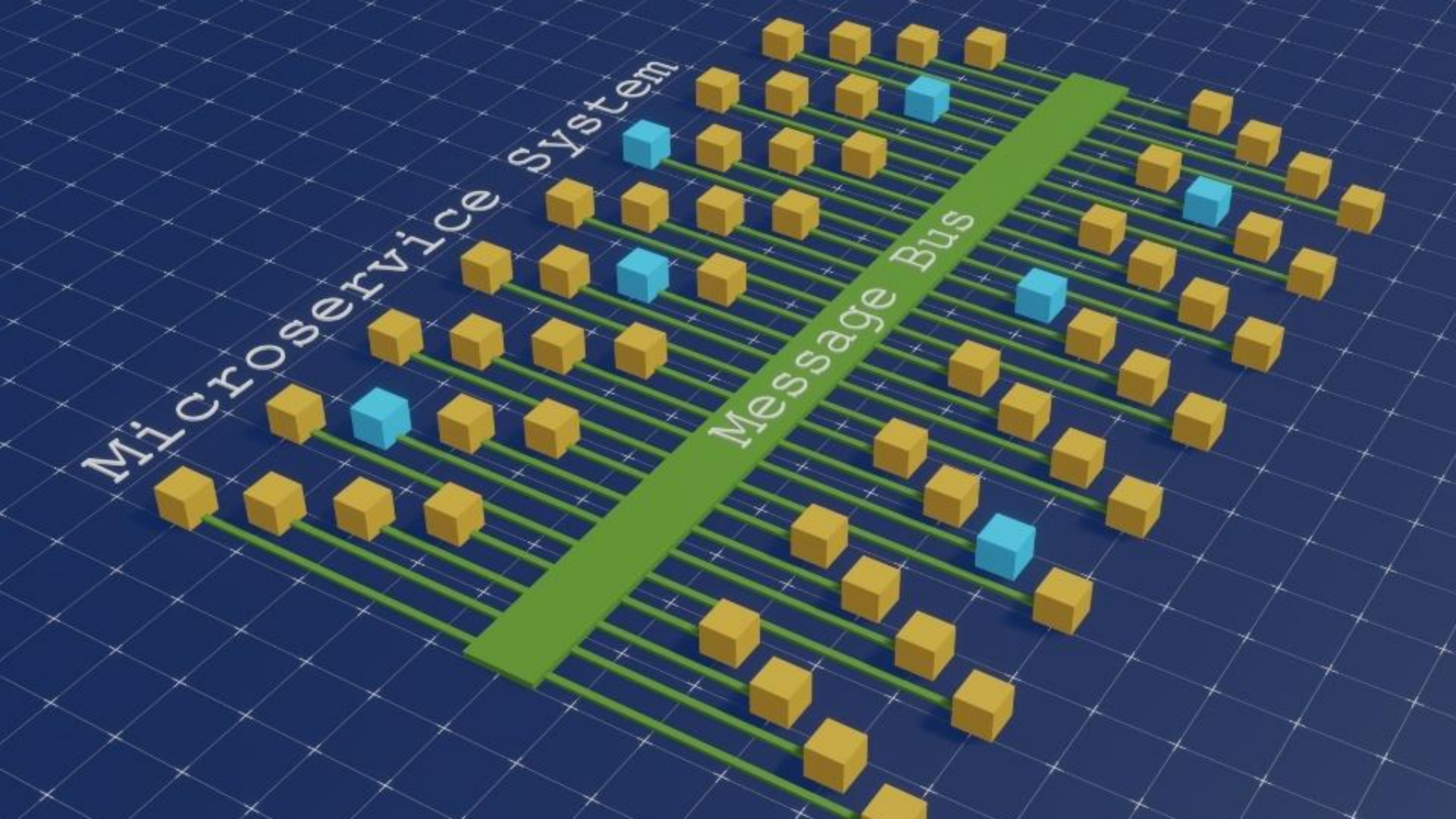


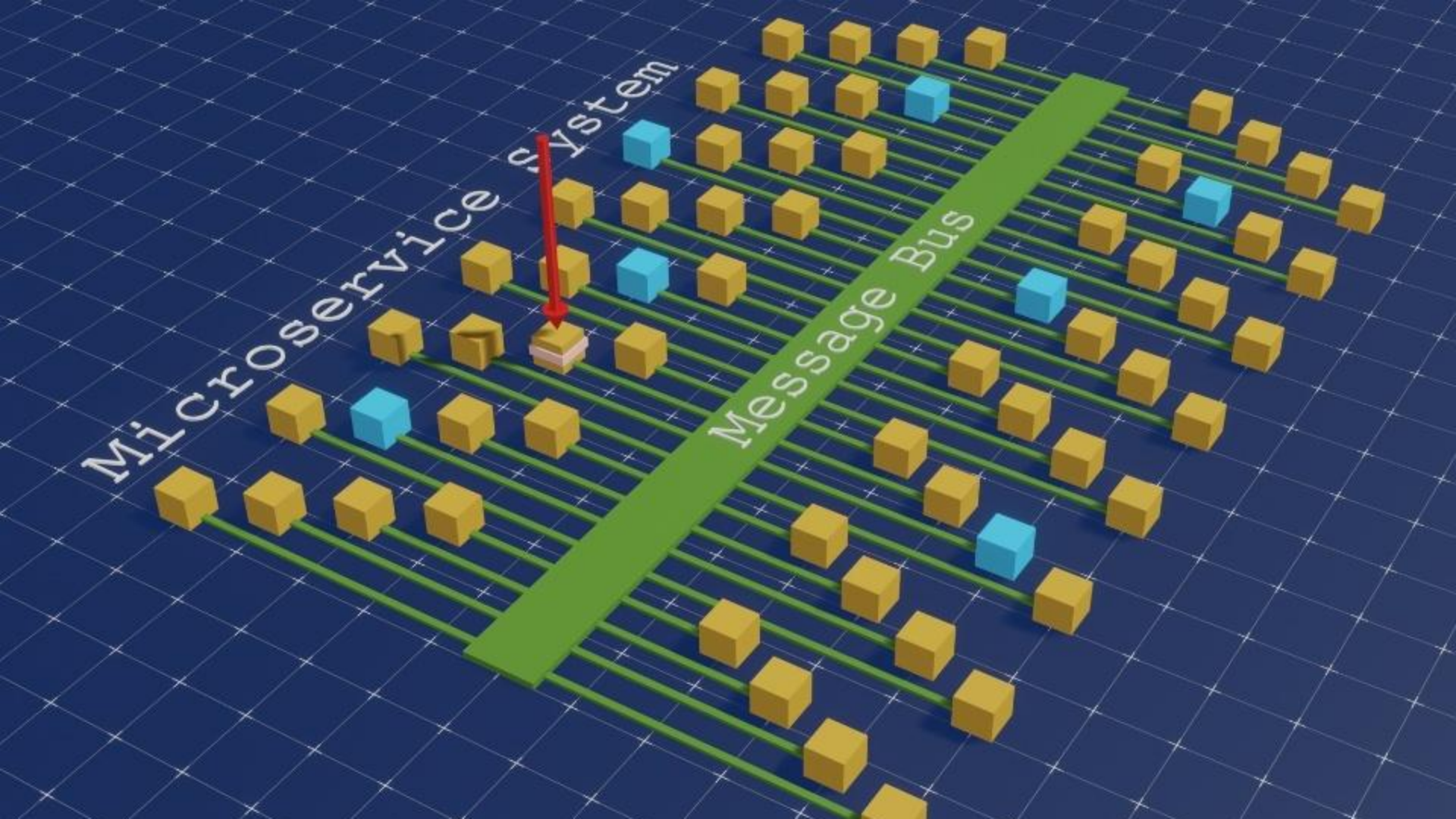

Microservice System

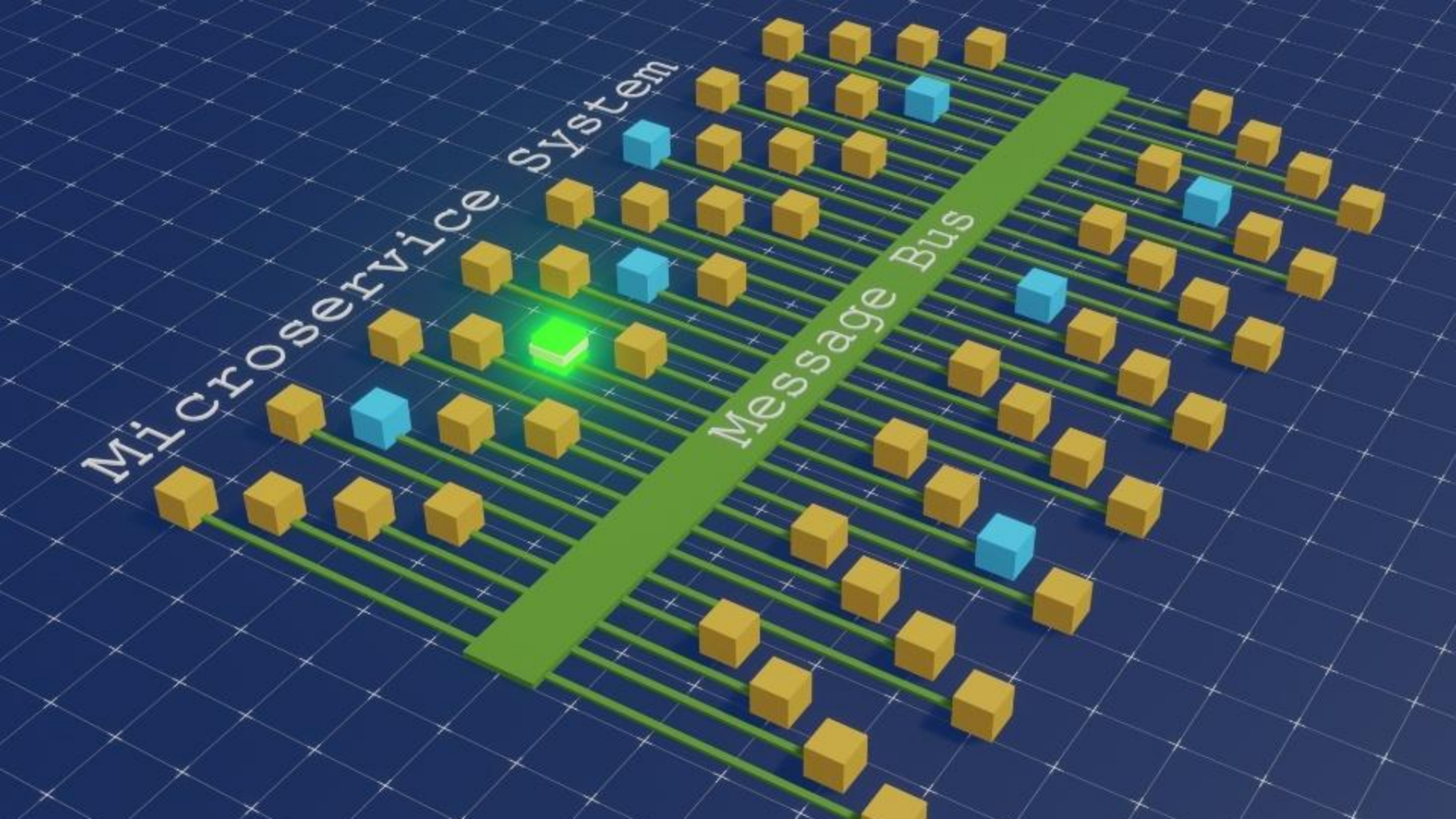

Microservice System
Message Bus

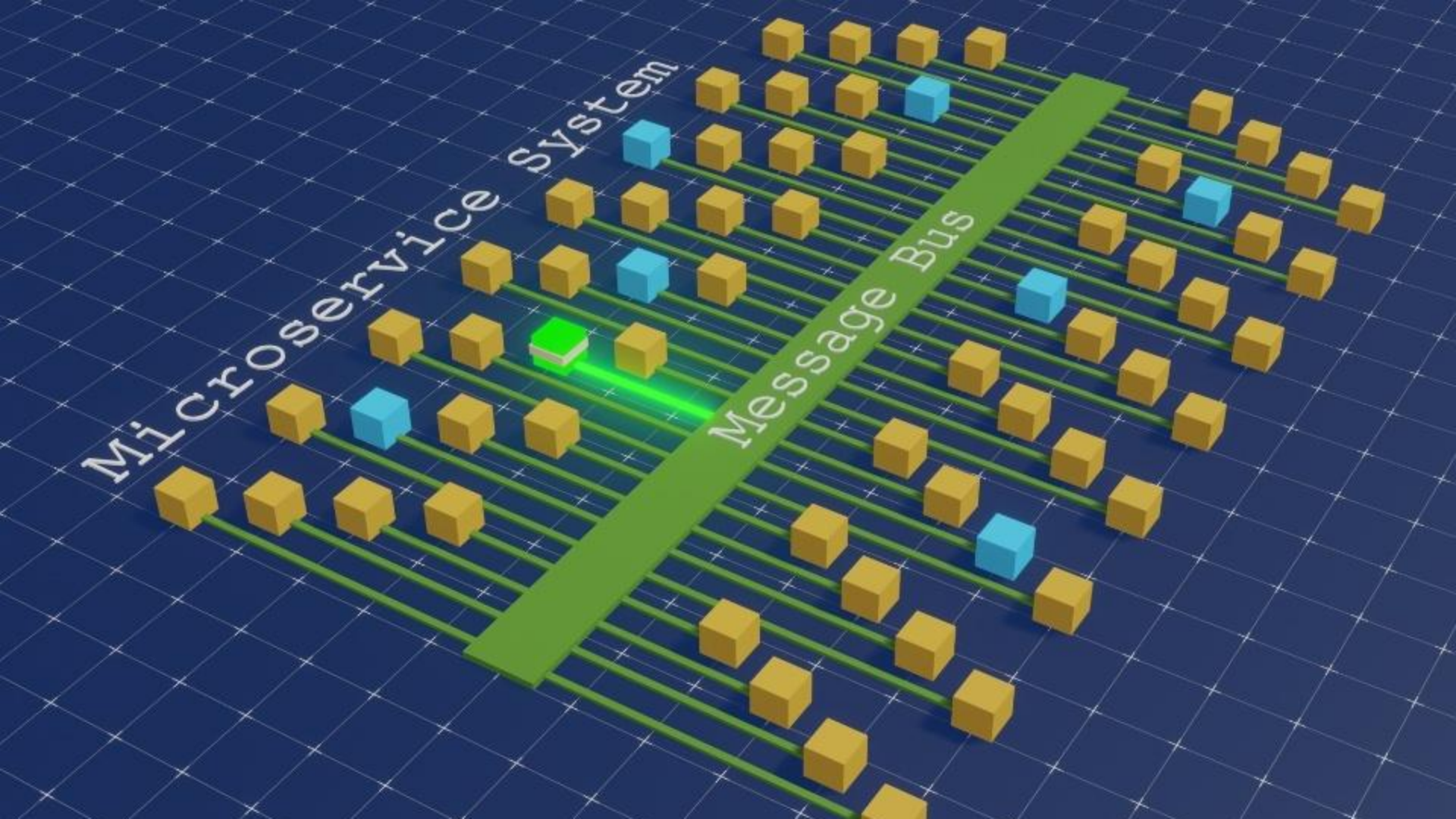

Microservice System
Message Bus

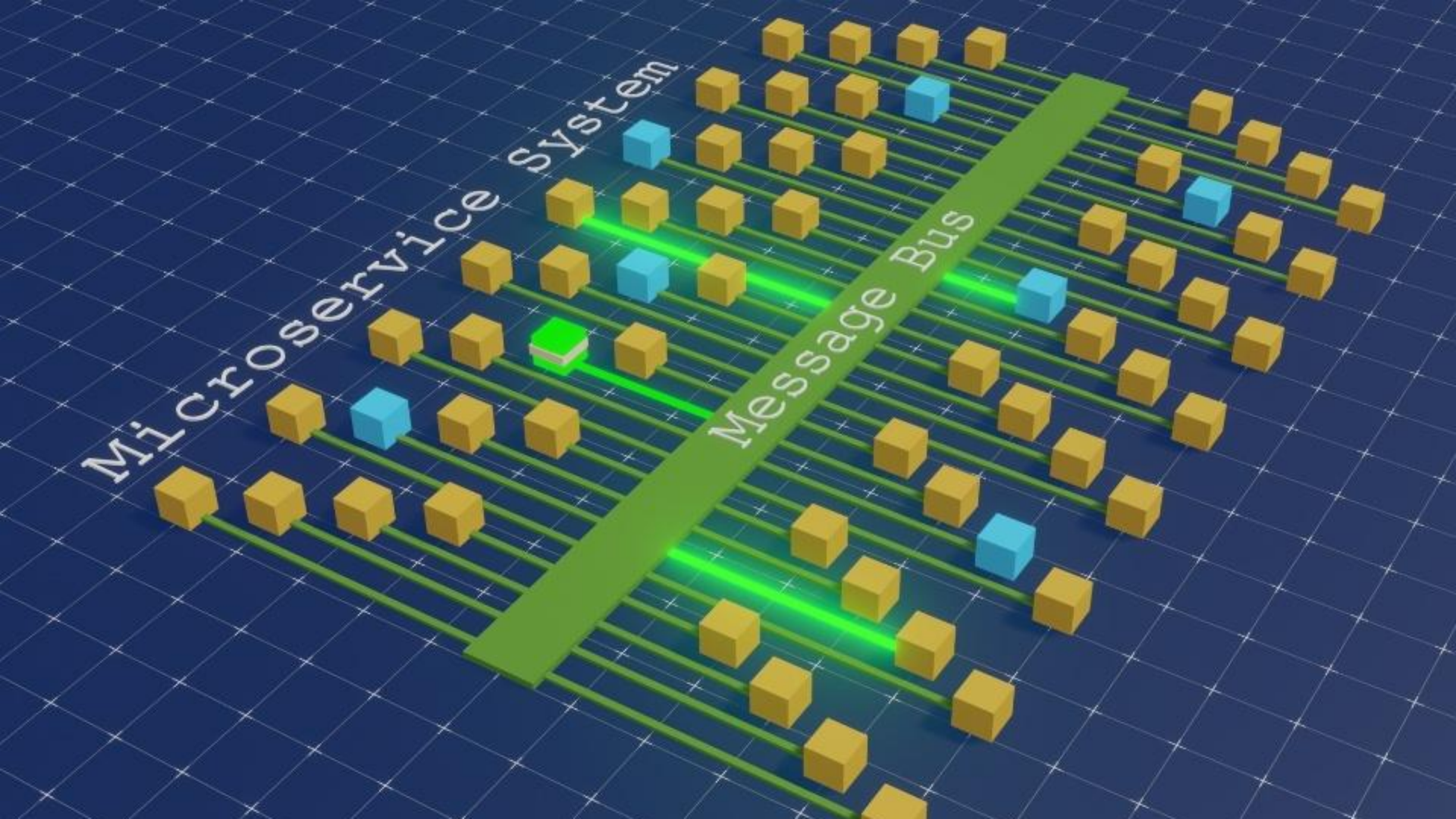

Microservice System
Message Bus
Analytics Service

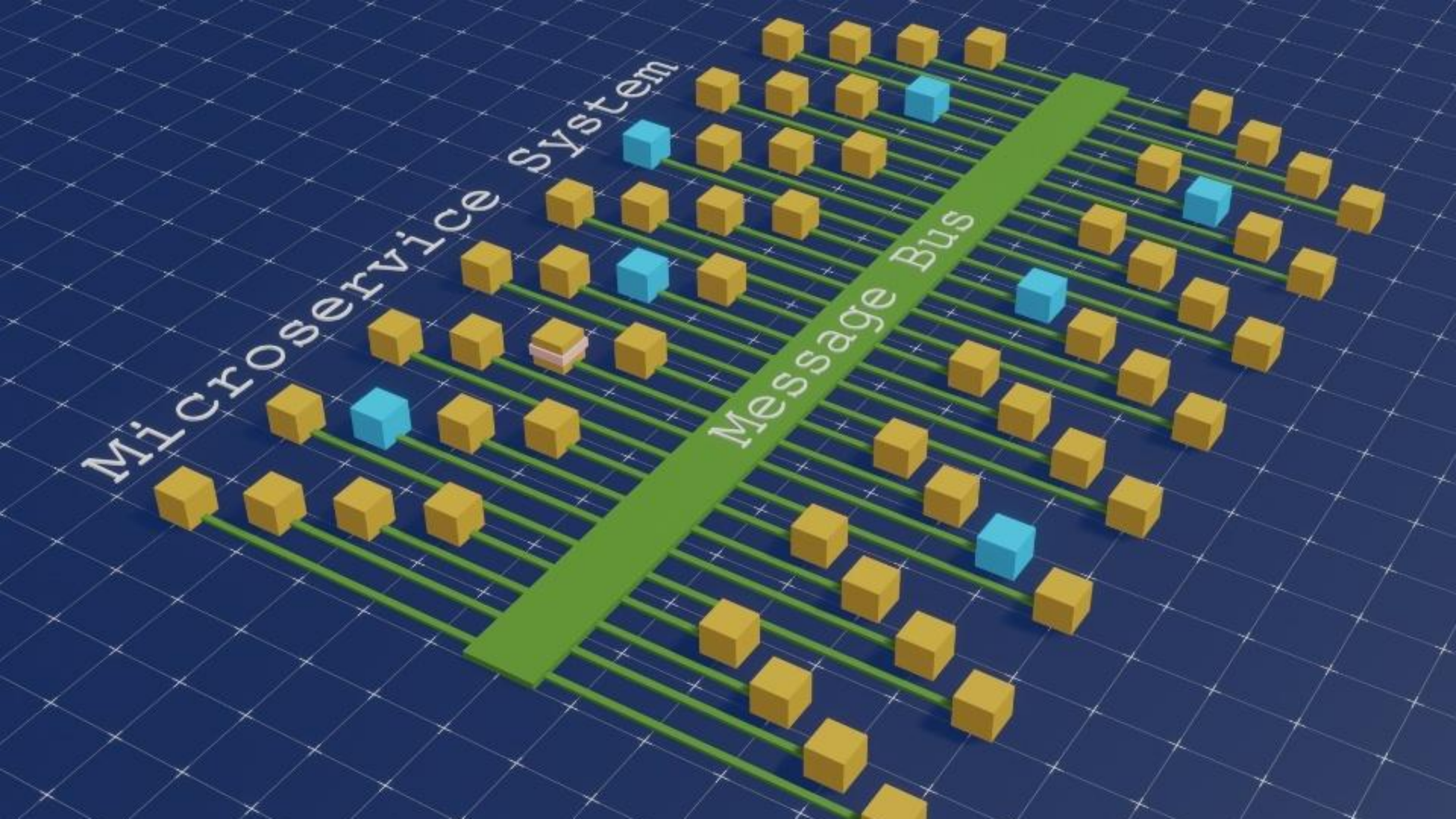

Microservice System
Message Bus

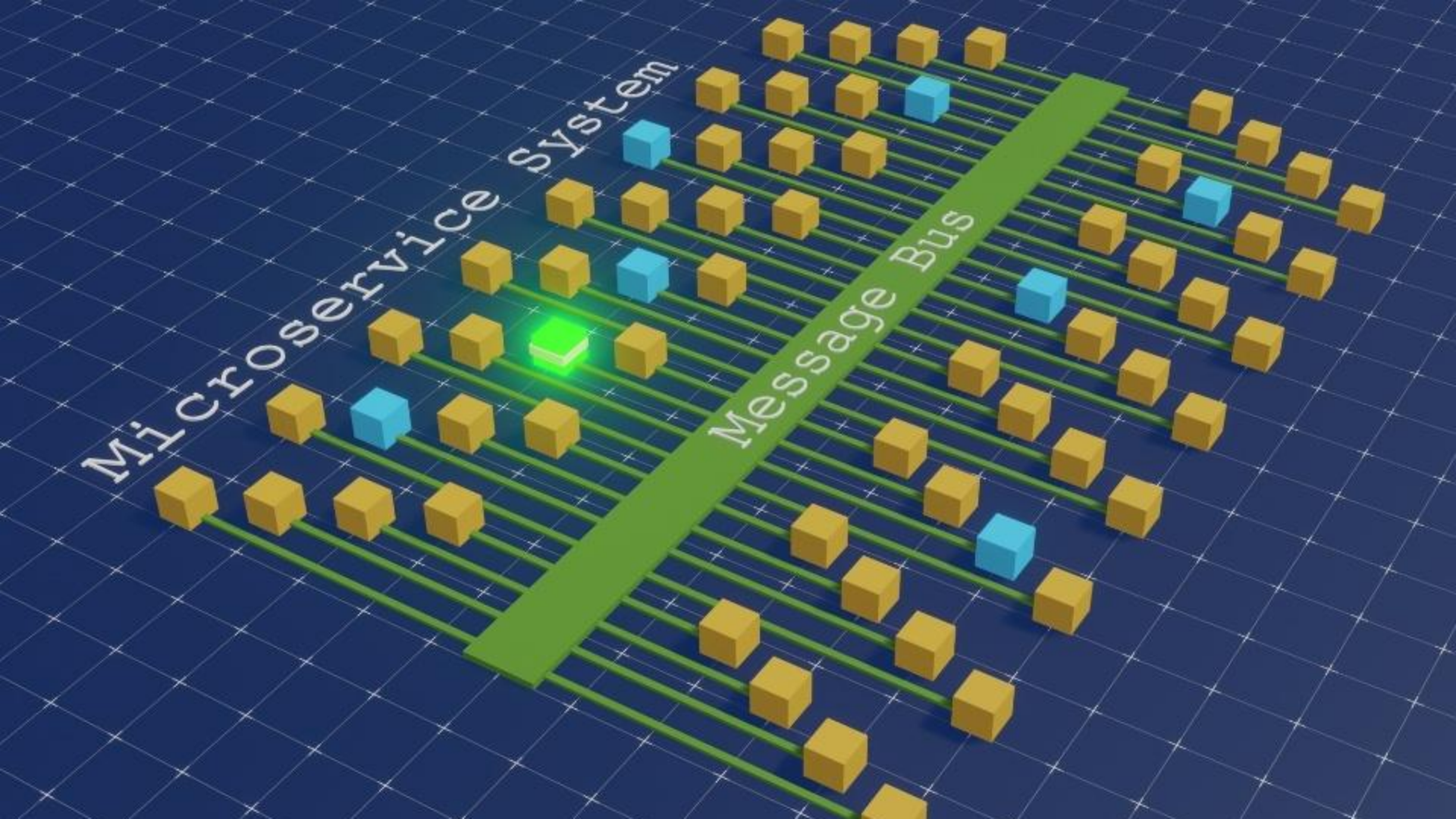

Microservice system
Message Bus

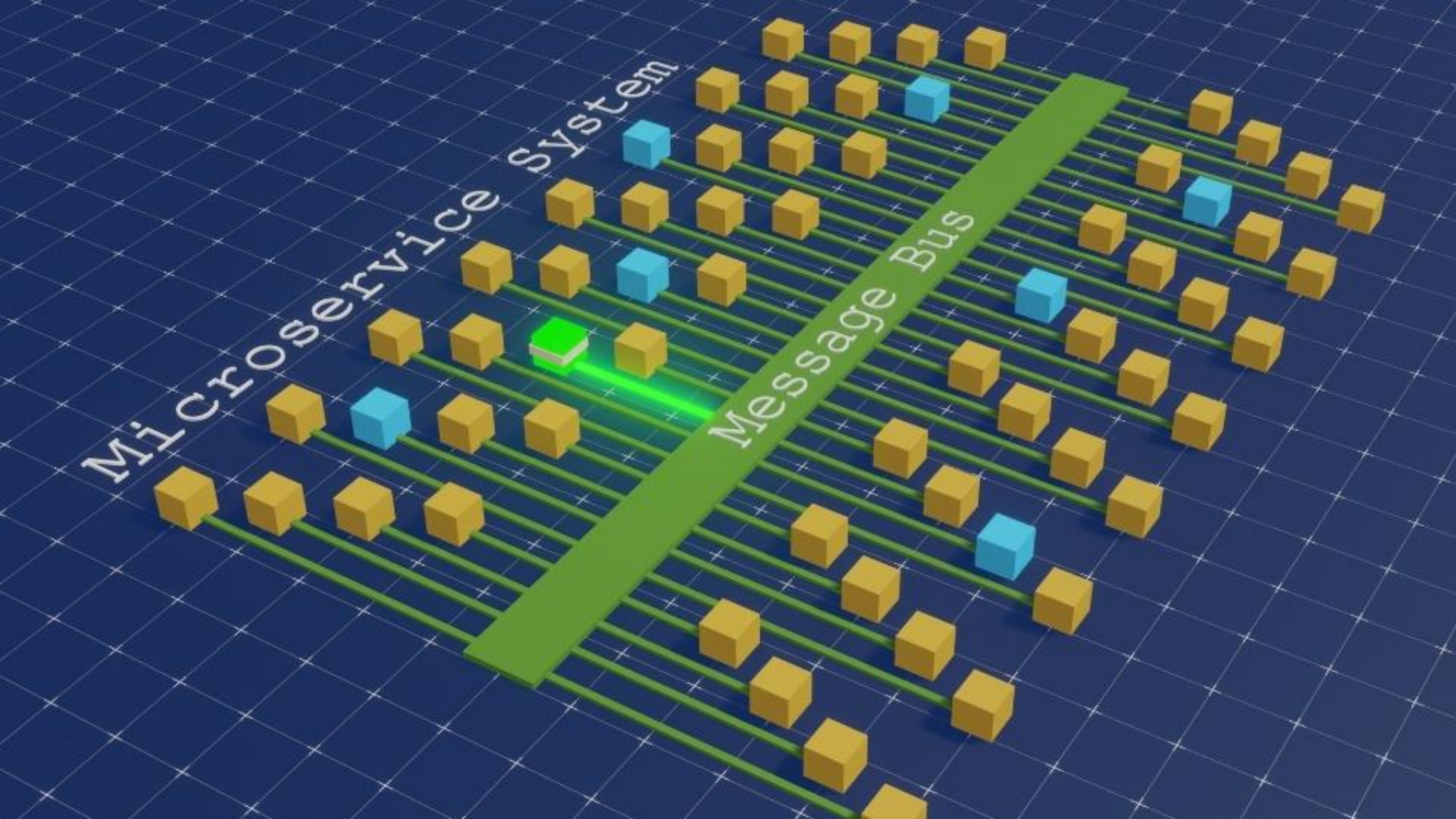

Microservice System
Message Bus

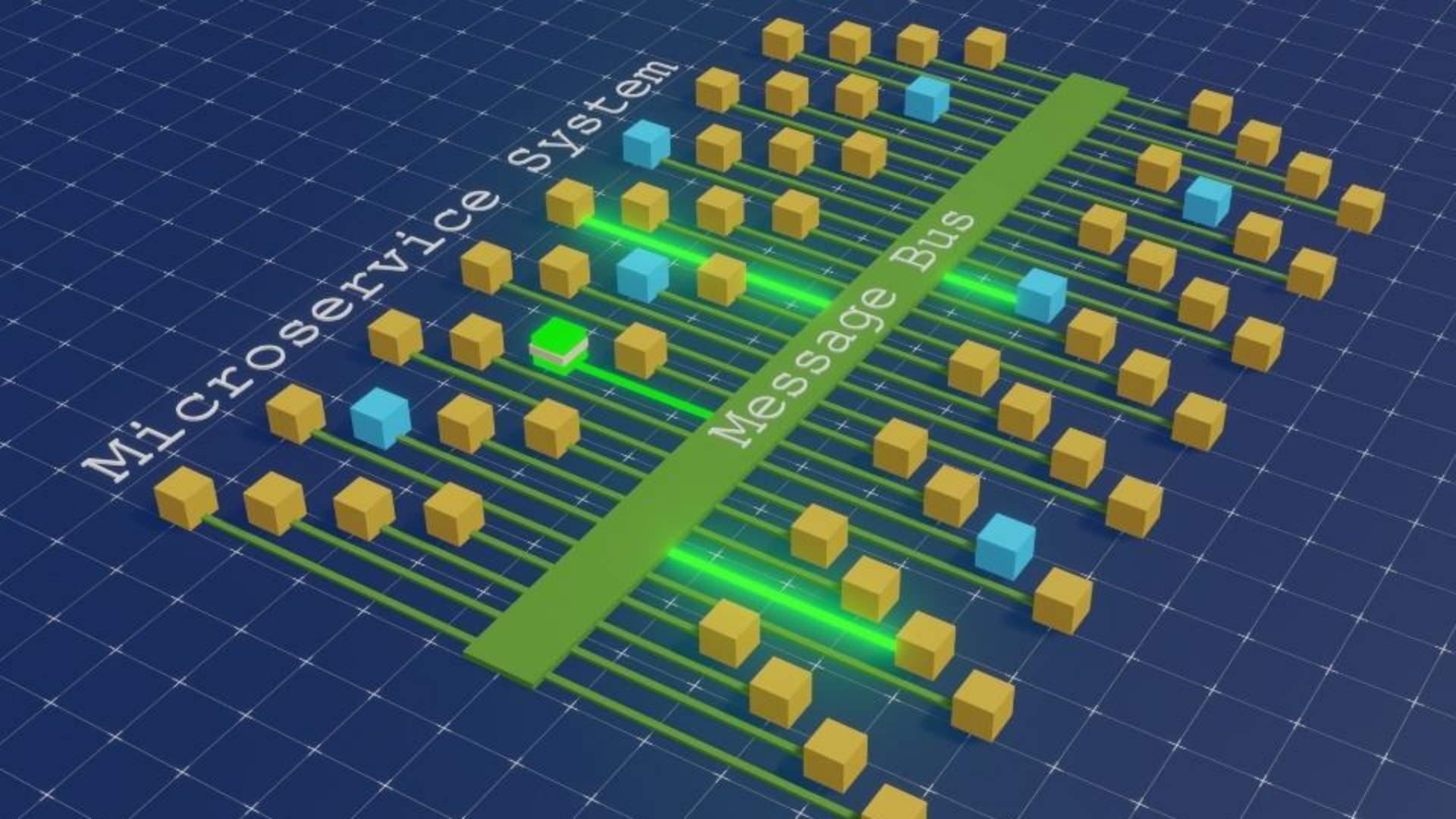

Microservice System
Message Bus

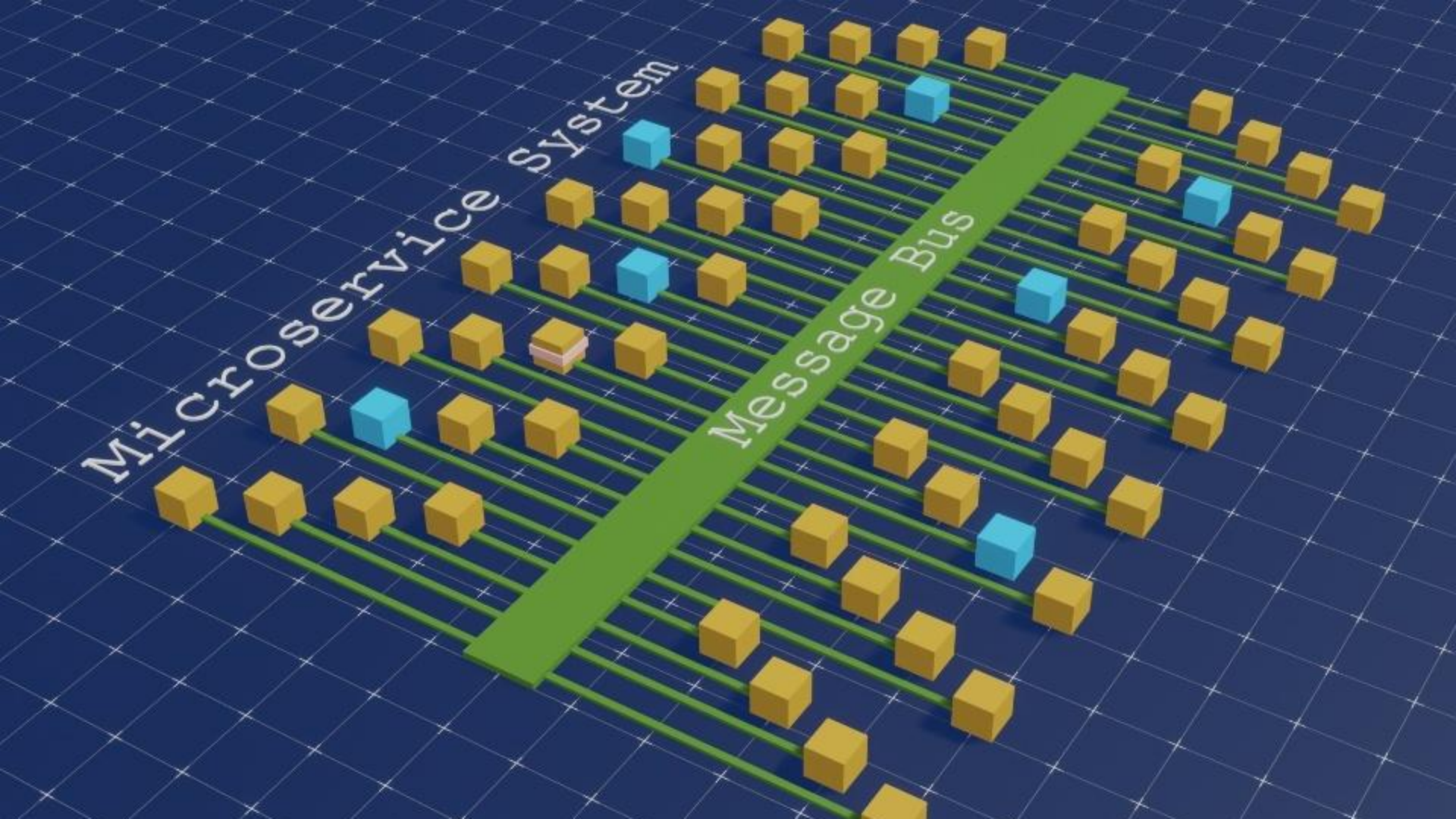

Microservice System
Message Bus

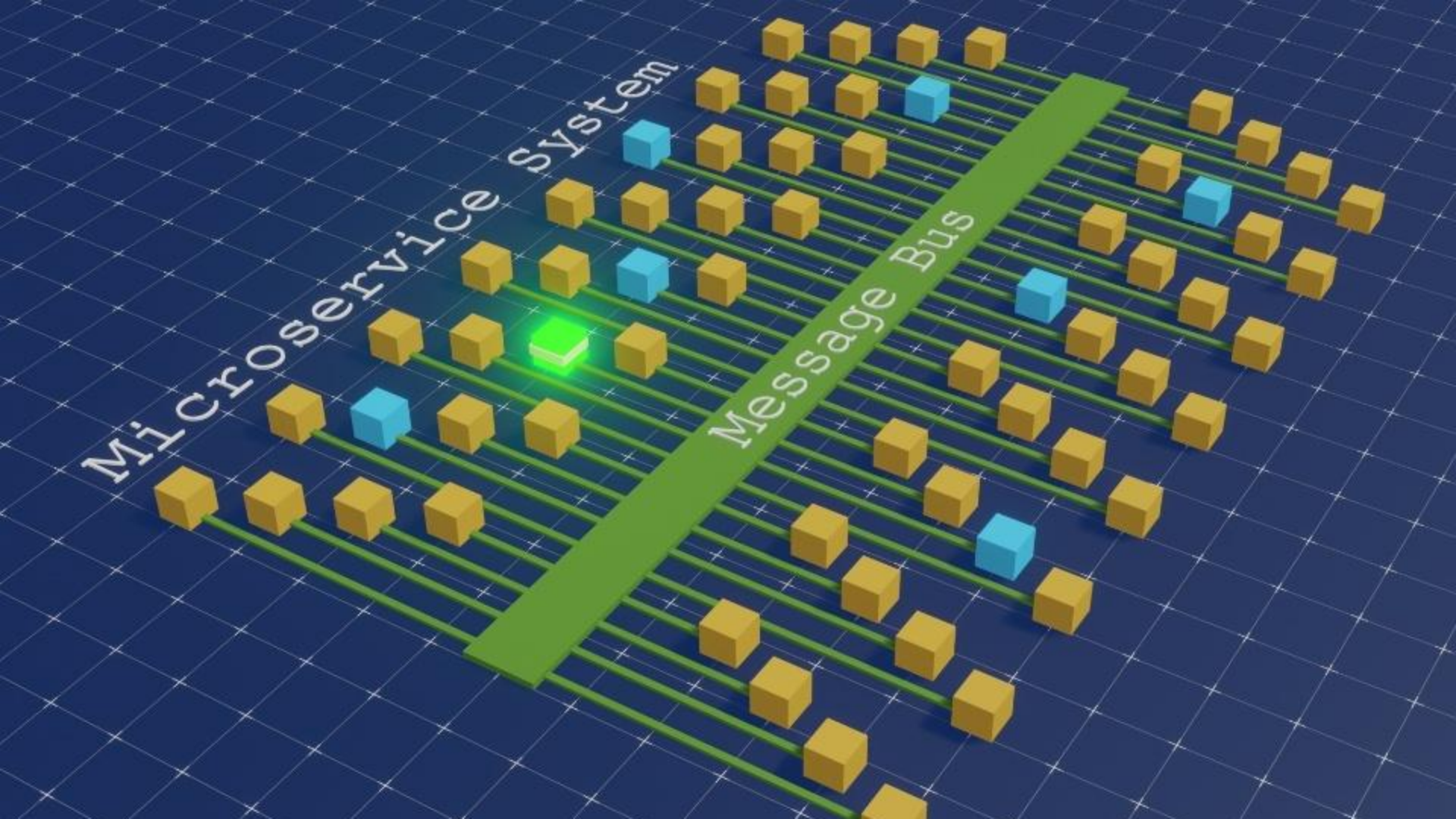

Microservice System
Message Bus

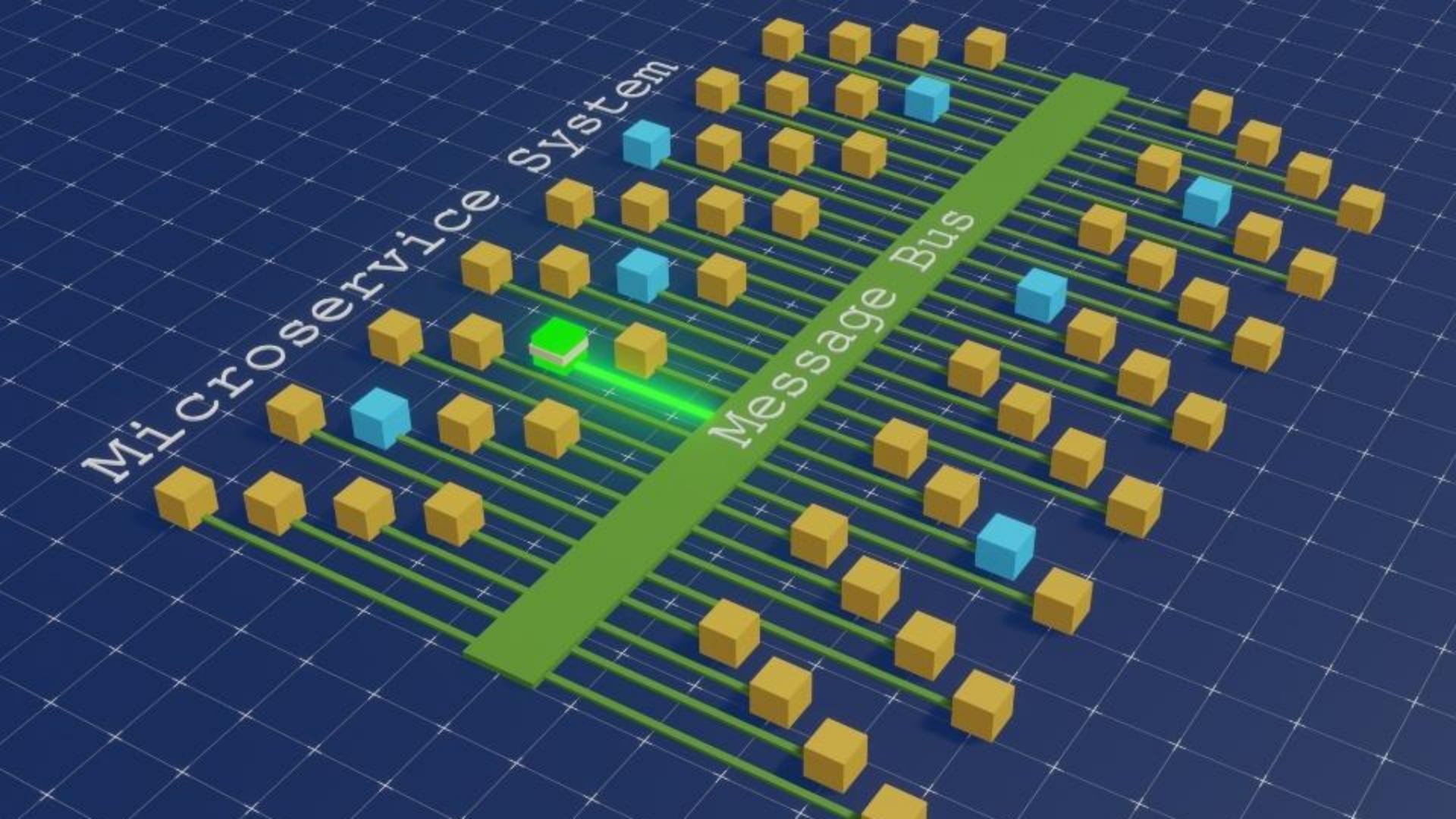

Microservice System
Message Bus

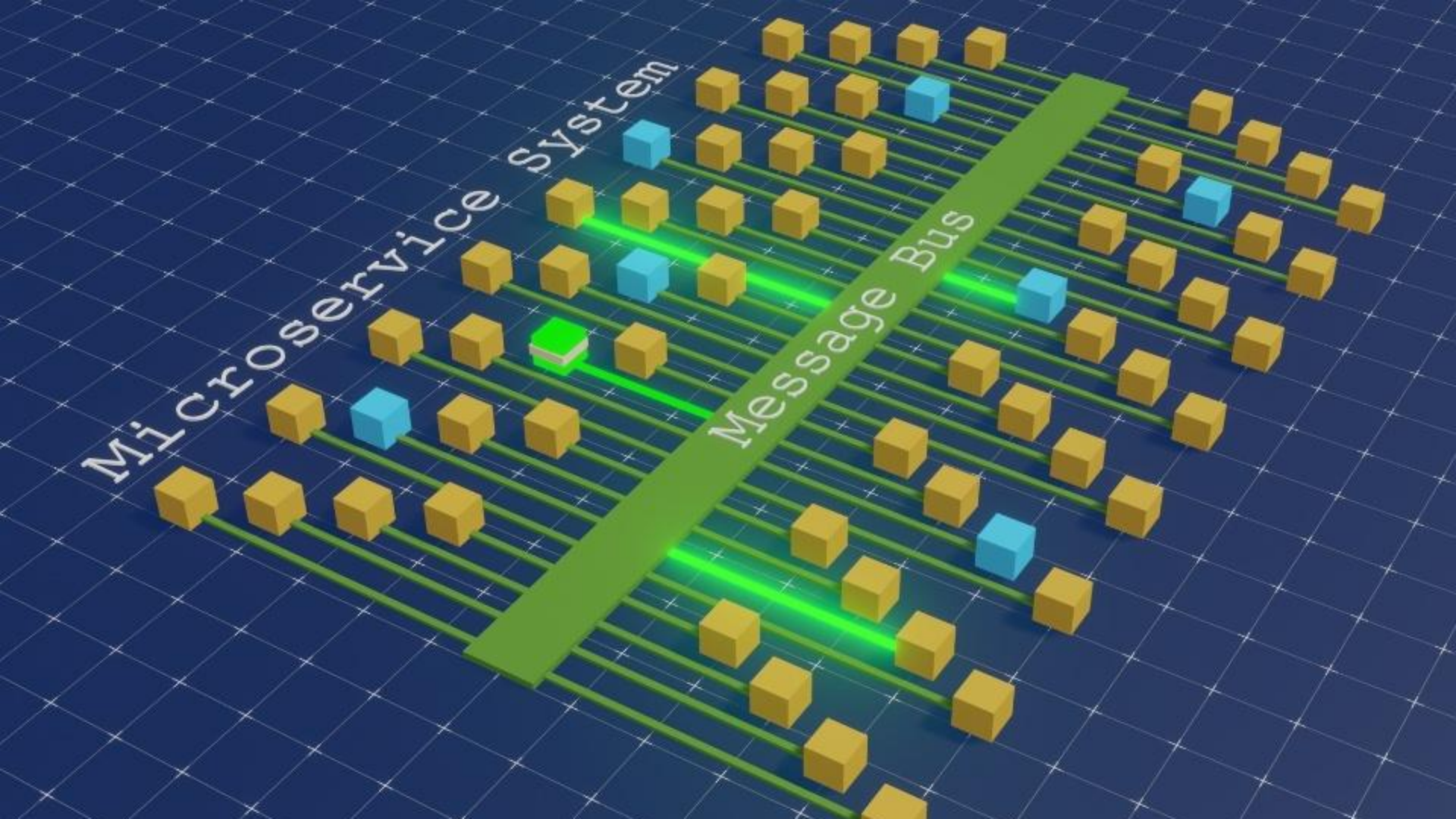

Microservice System
Message Bus

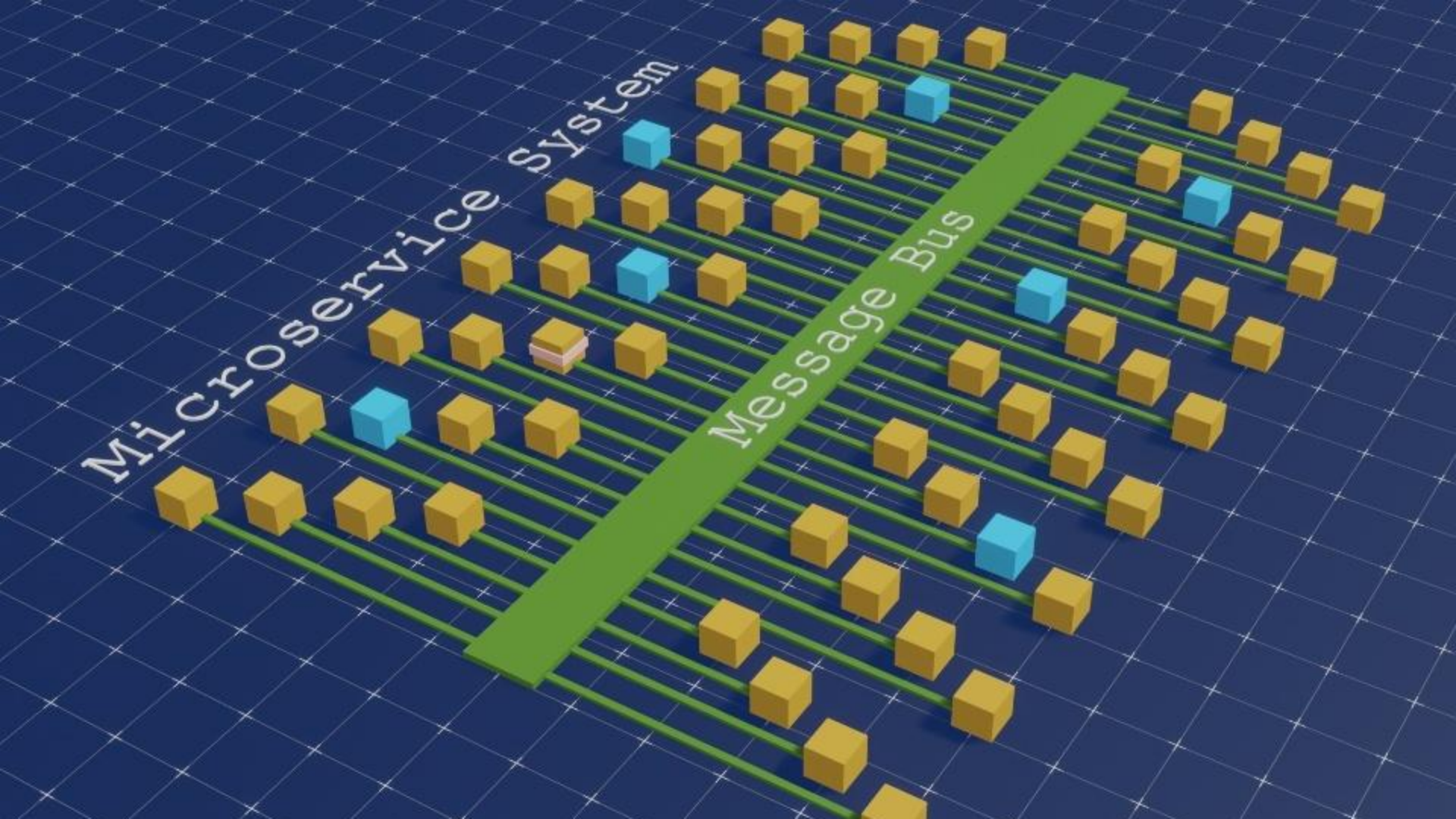

Microservice System
Message Bus

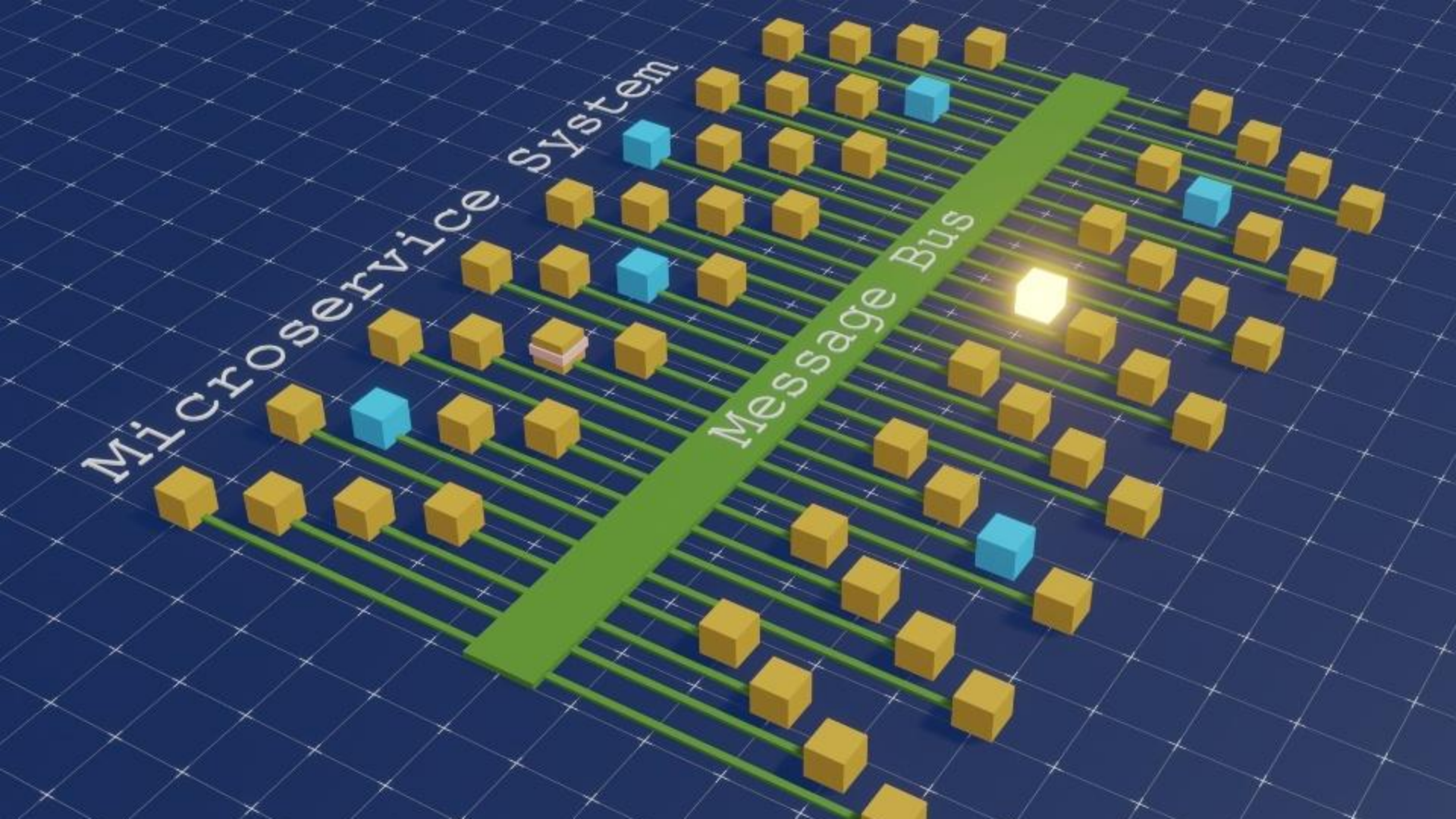

Microservice System
Message Bus

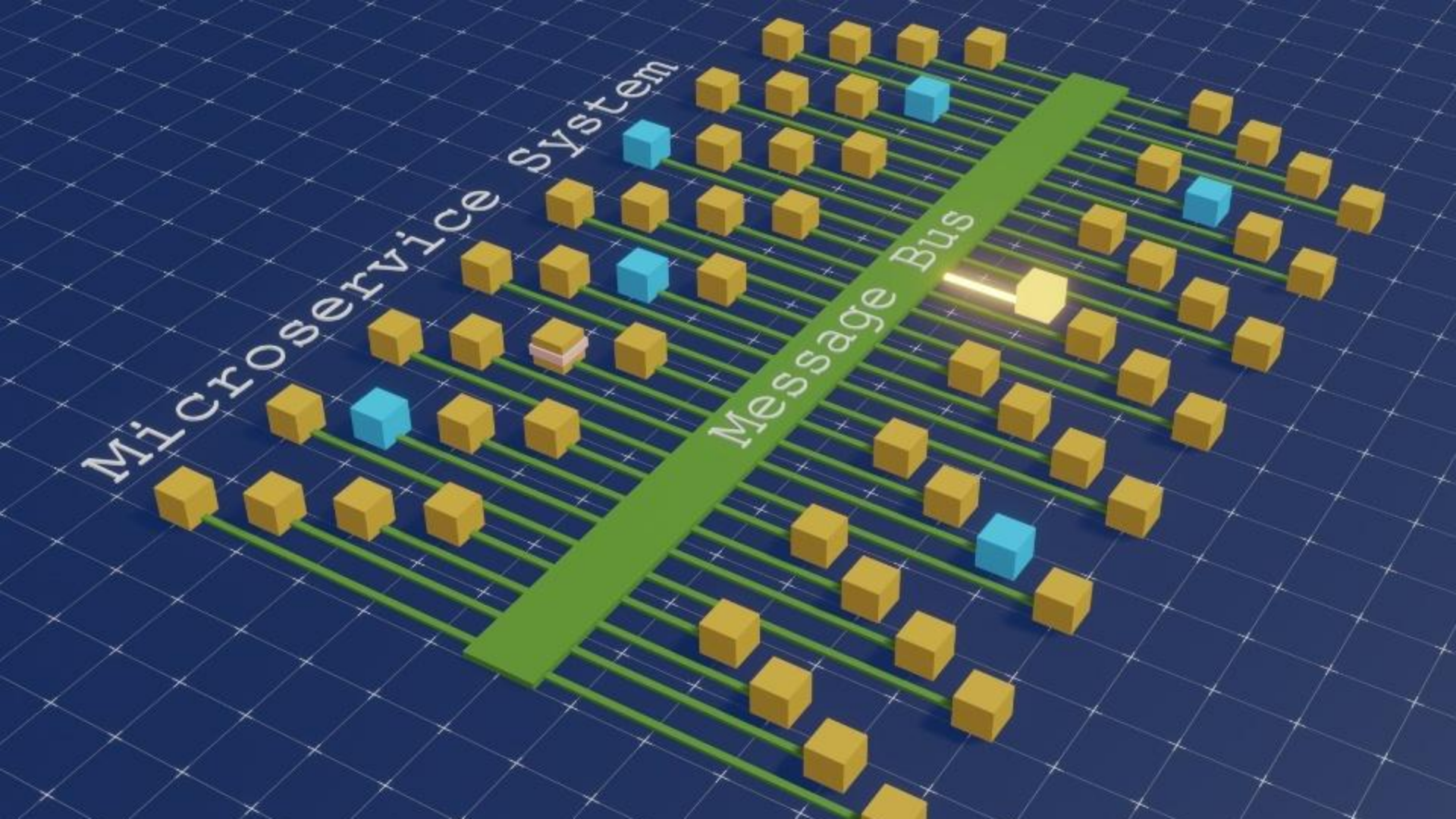

Microservice System
Message Bus

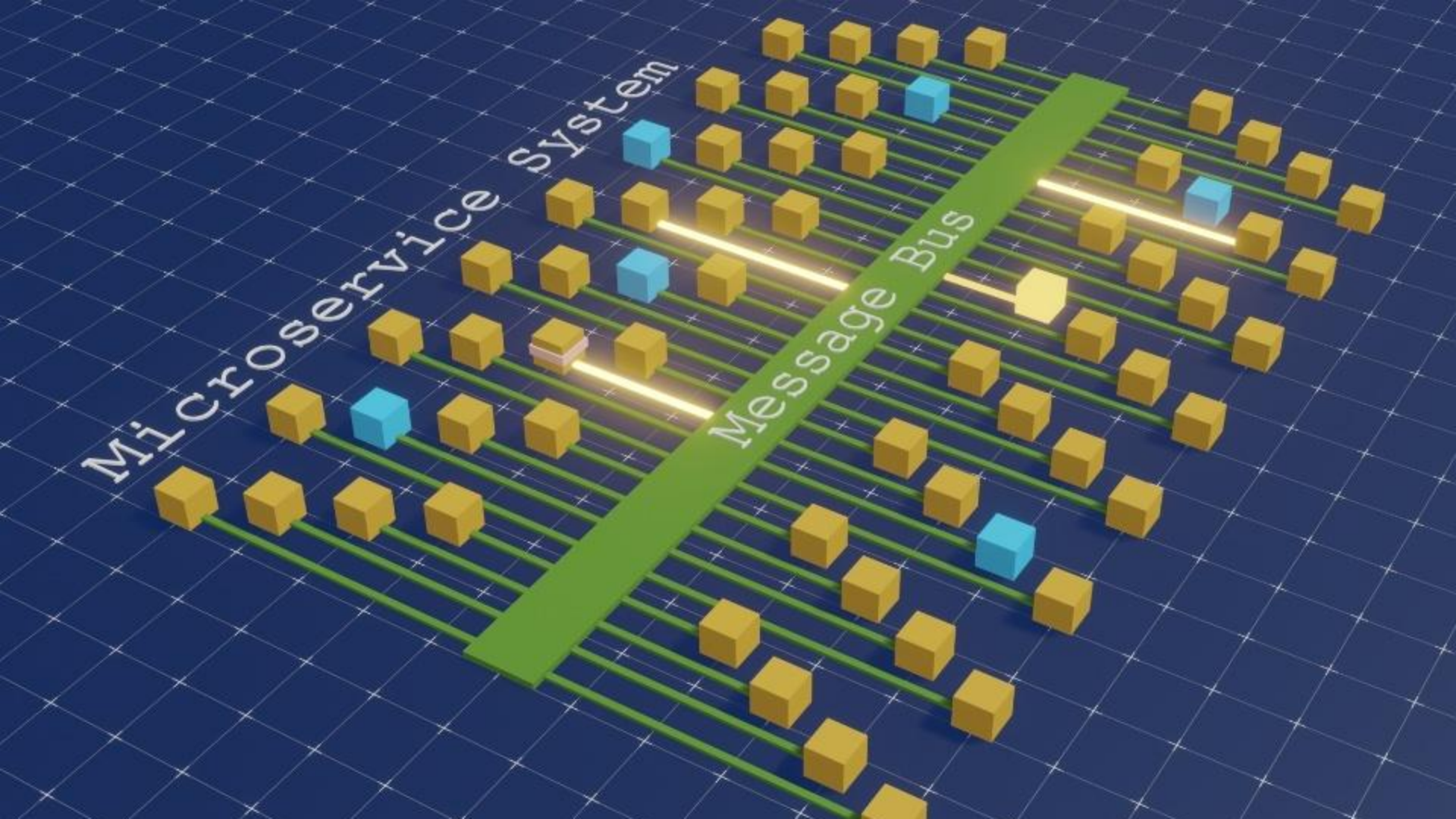

Microservice System
Message Bus

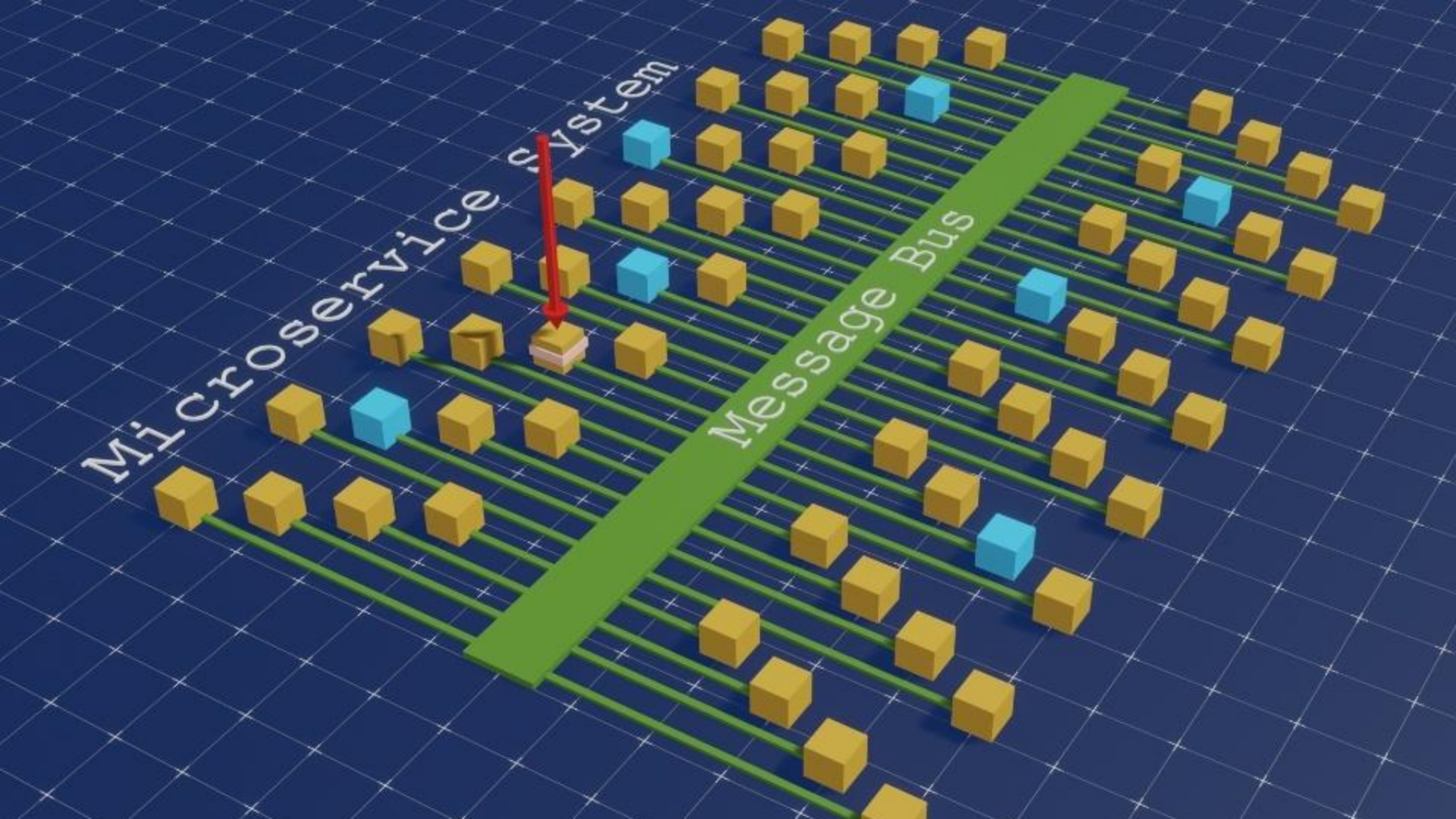

Microservice System
Message Bus

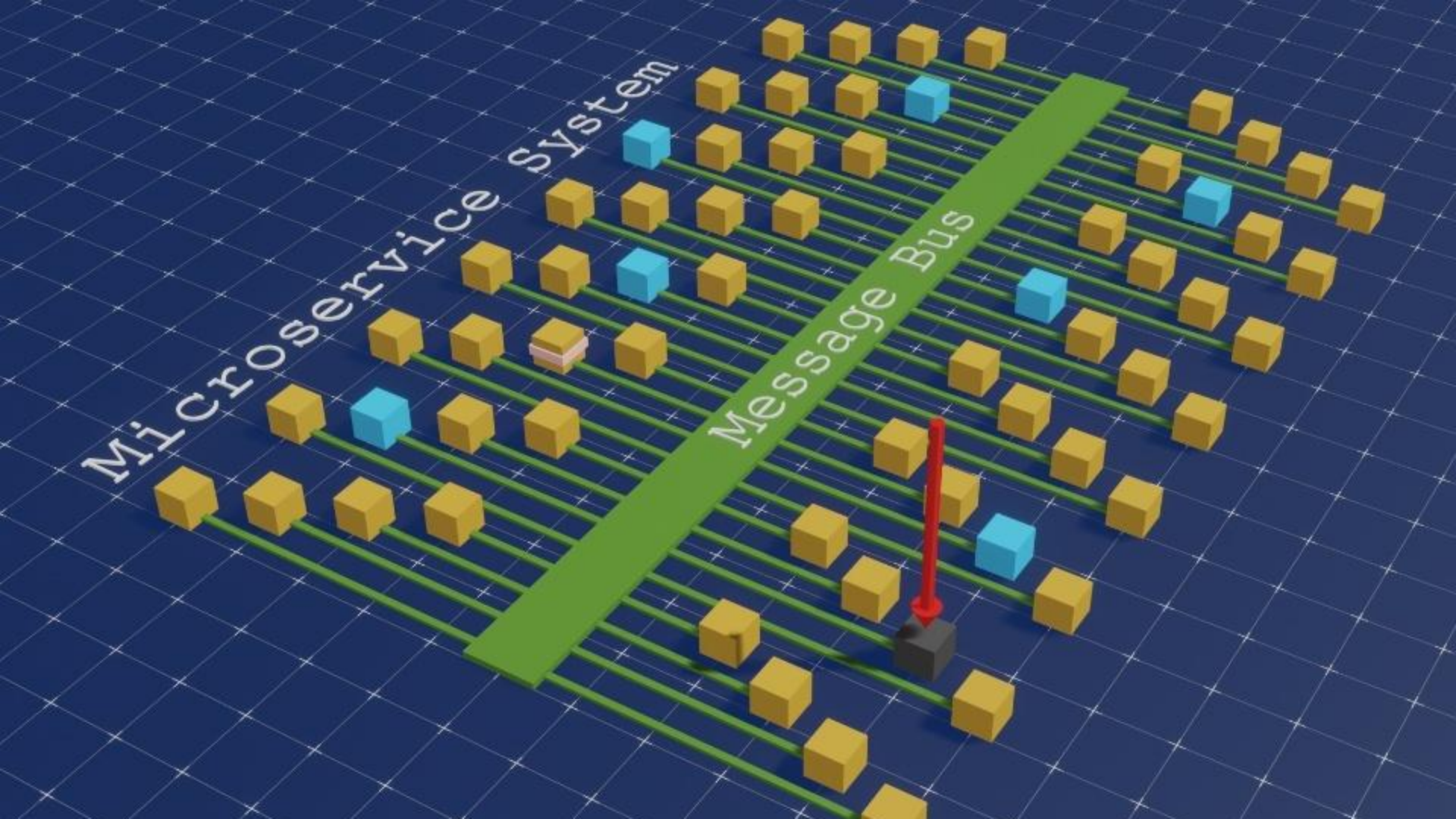

Microservice System
Message Bus

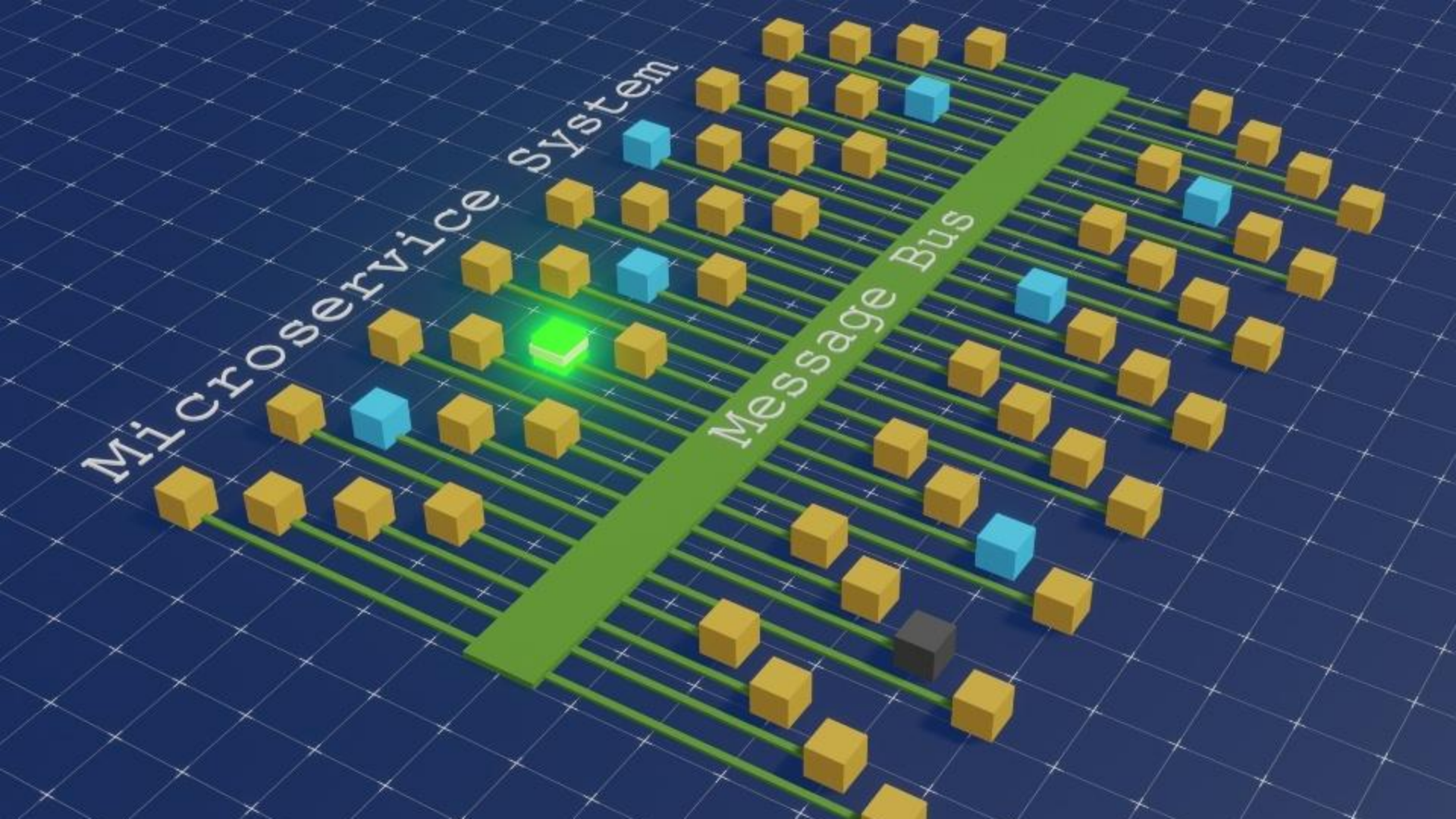

Microservice System
Message Bus

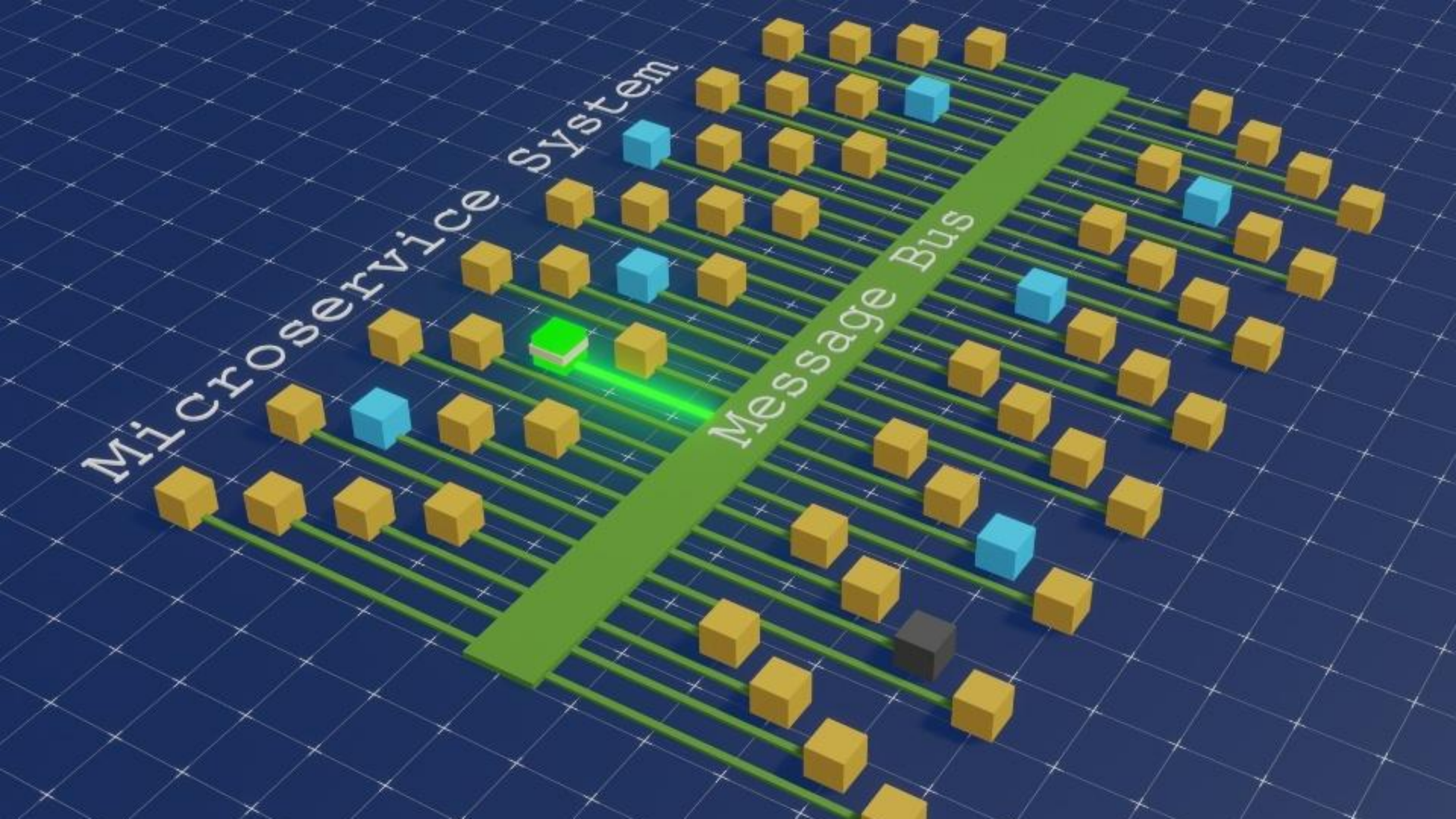

Microservice System
Message Bus

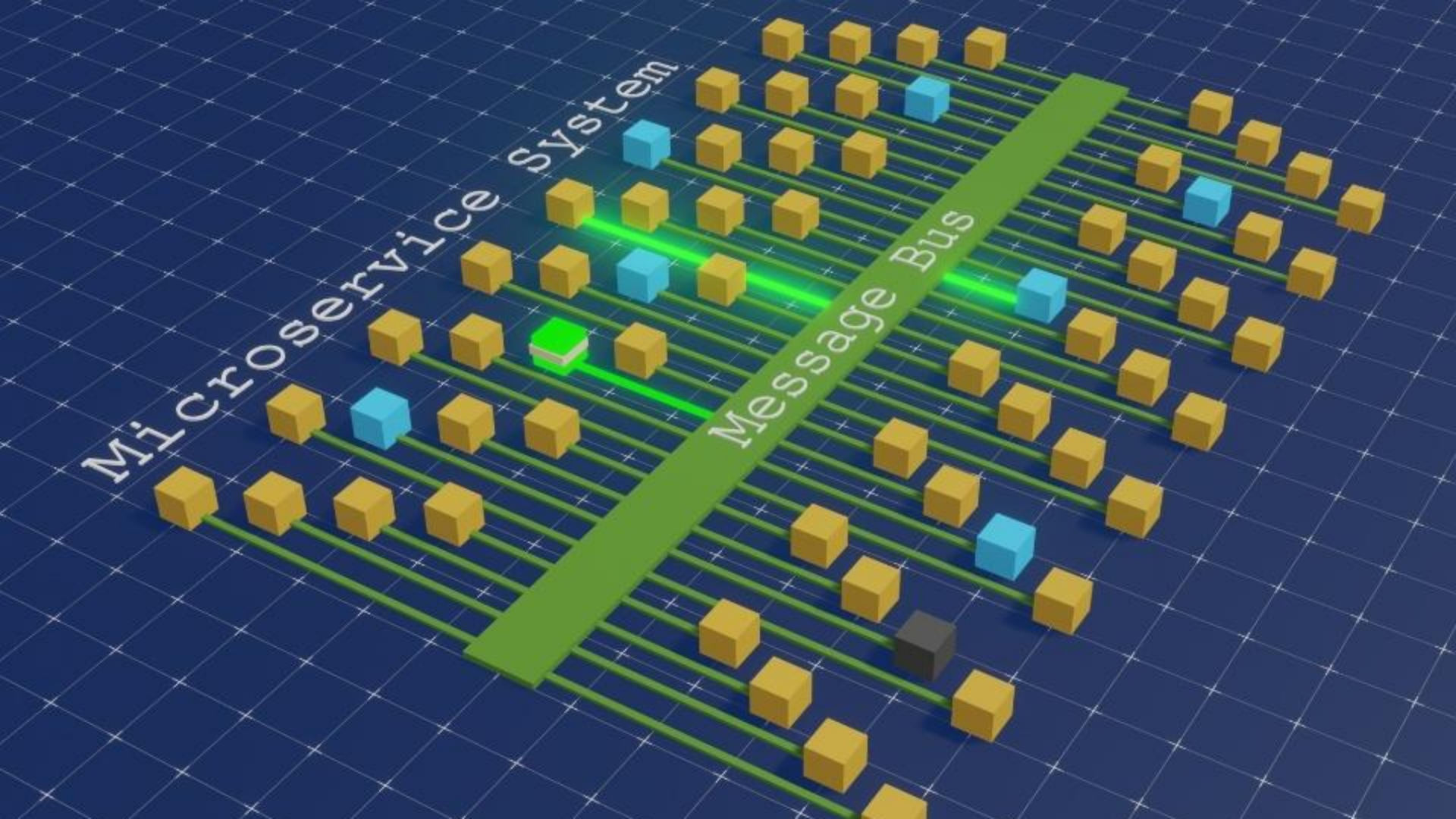

Microservice System
Message Bus

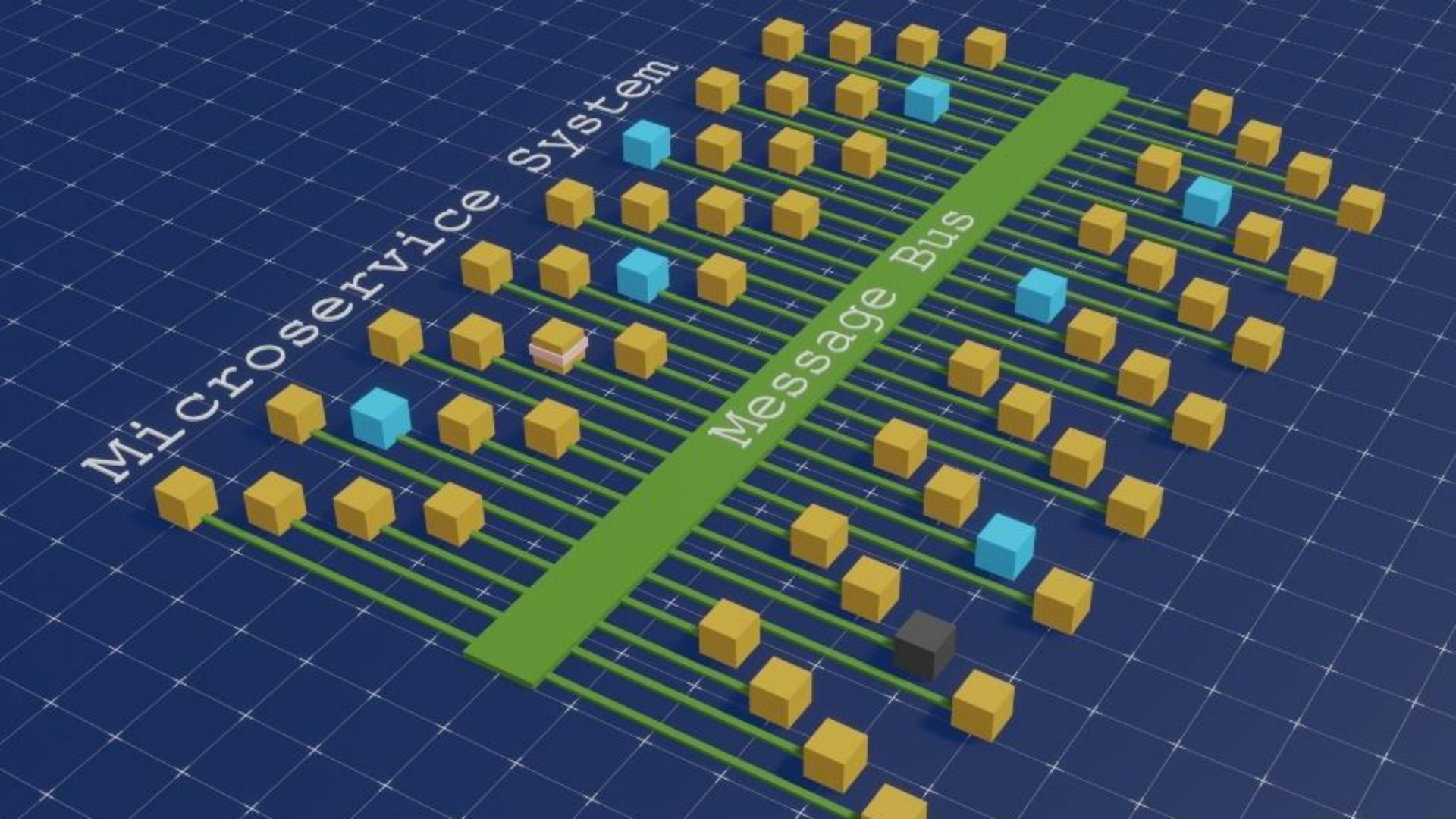

Microservice system
Message Bus

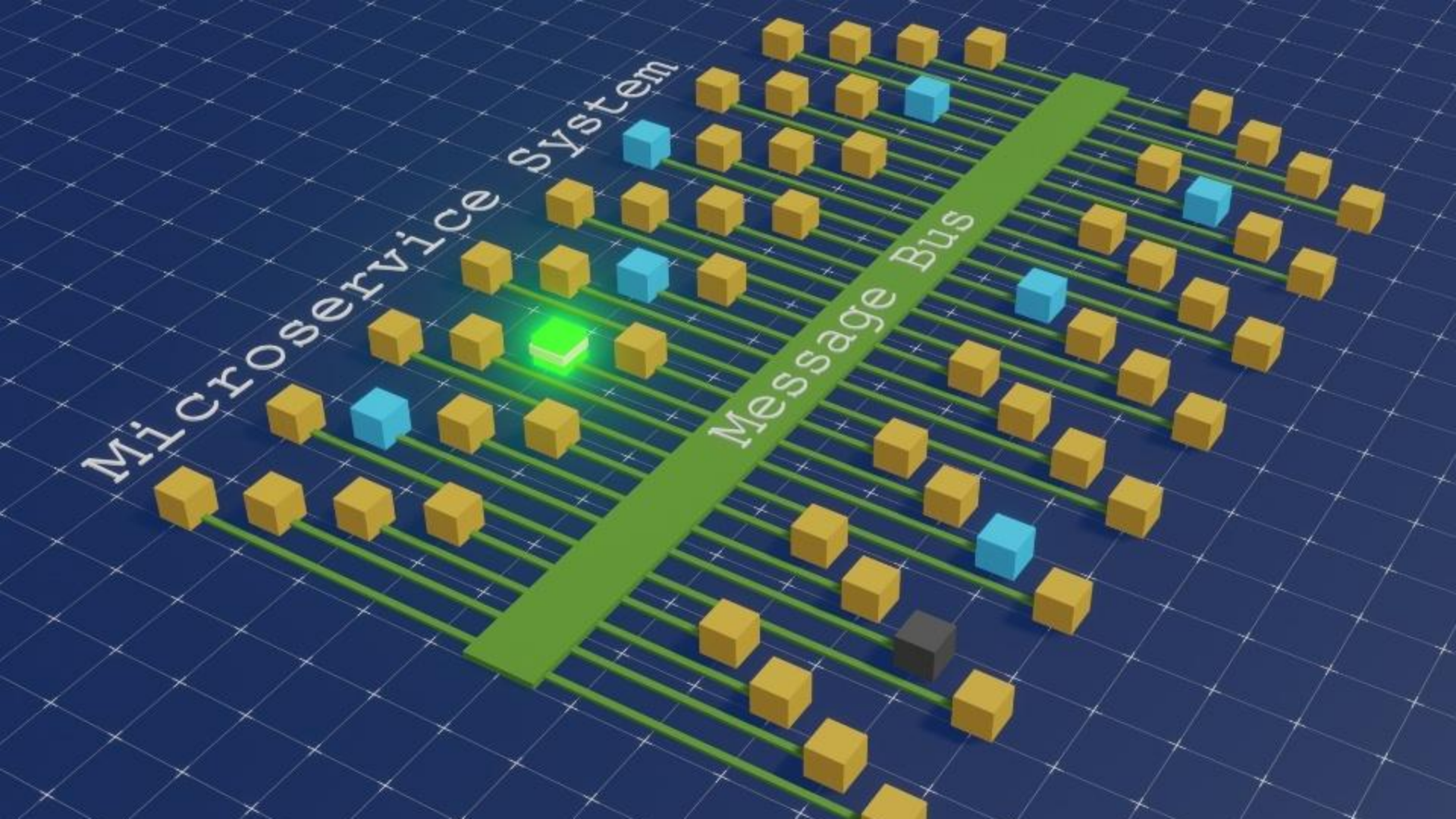

Microservice System
Message Bus

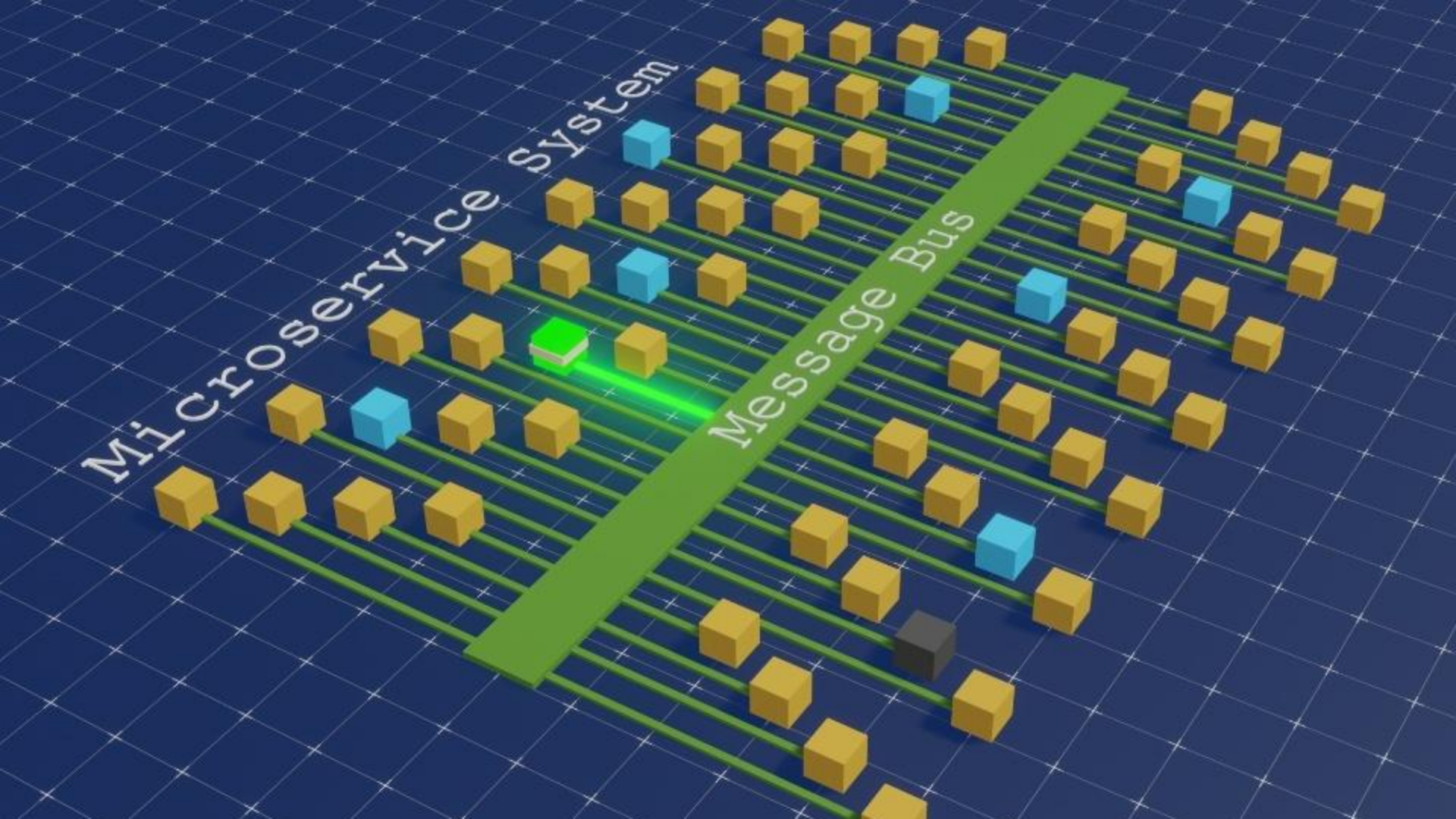

Microservice System
Message Bus

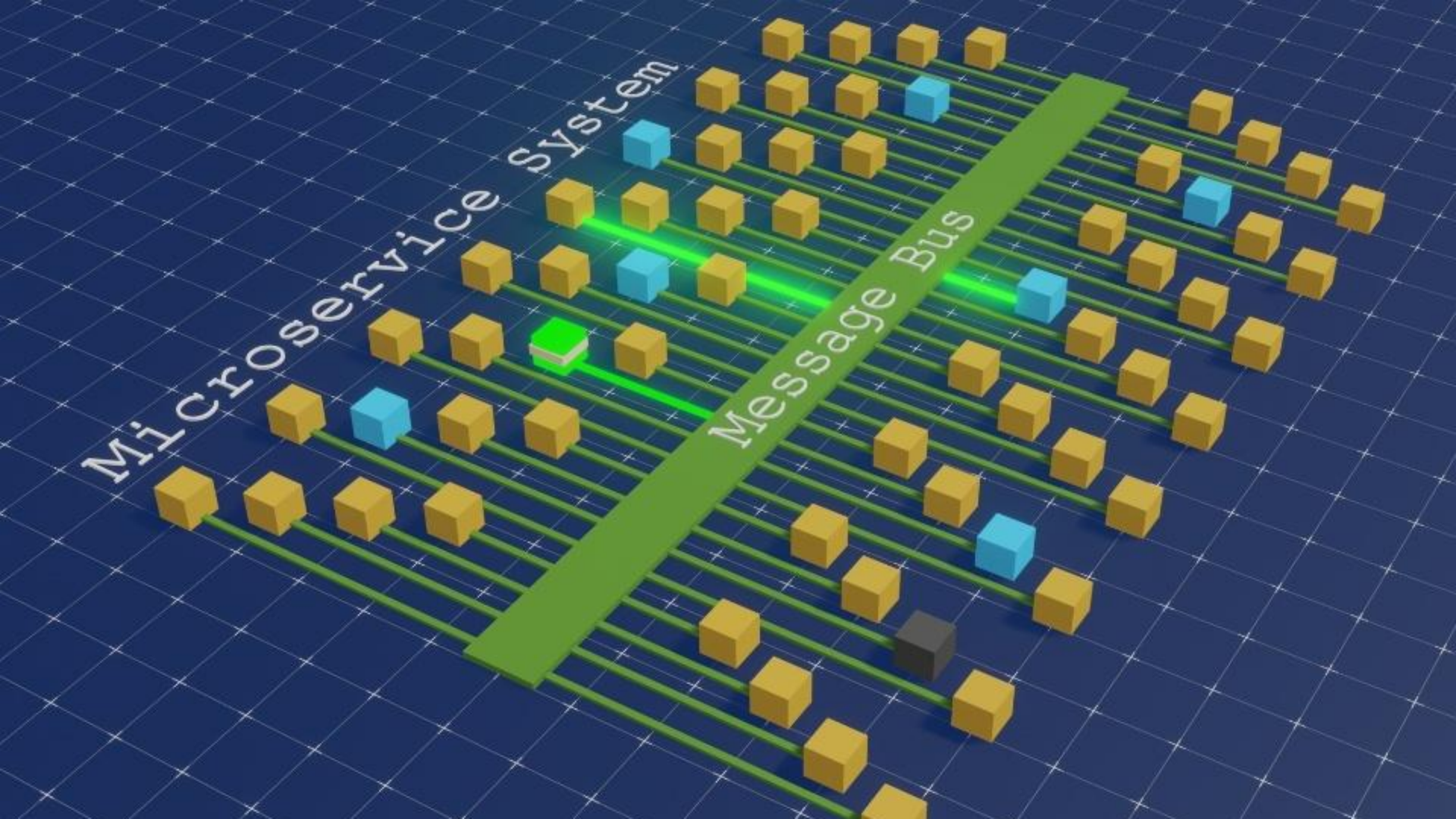

Microservice System
Message Bus

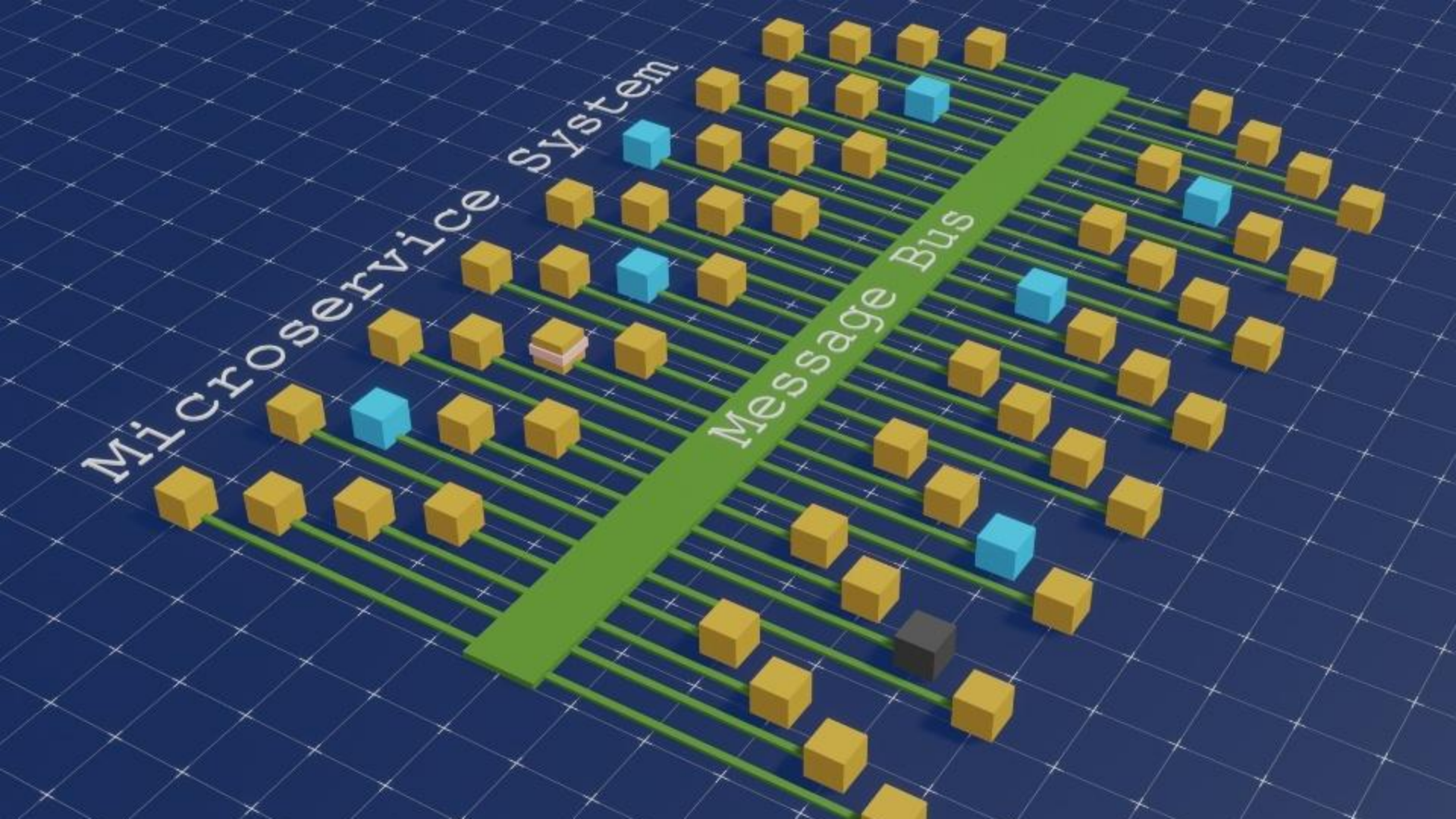

Microservice System
Message Bus

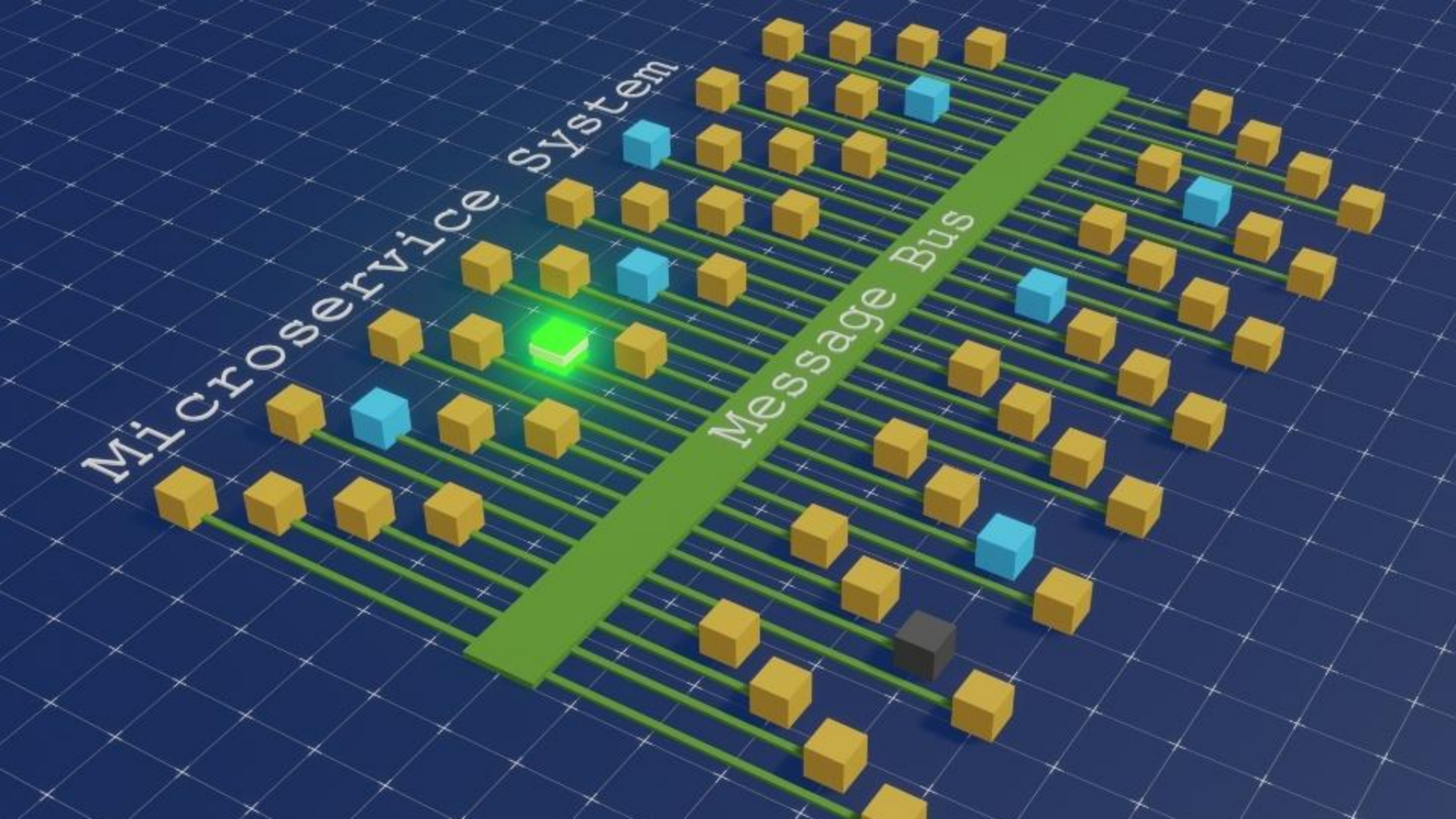

Microservice System
Message Bus

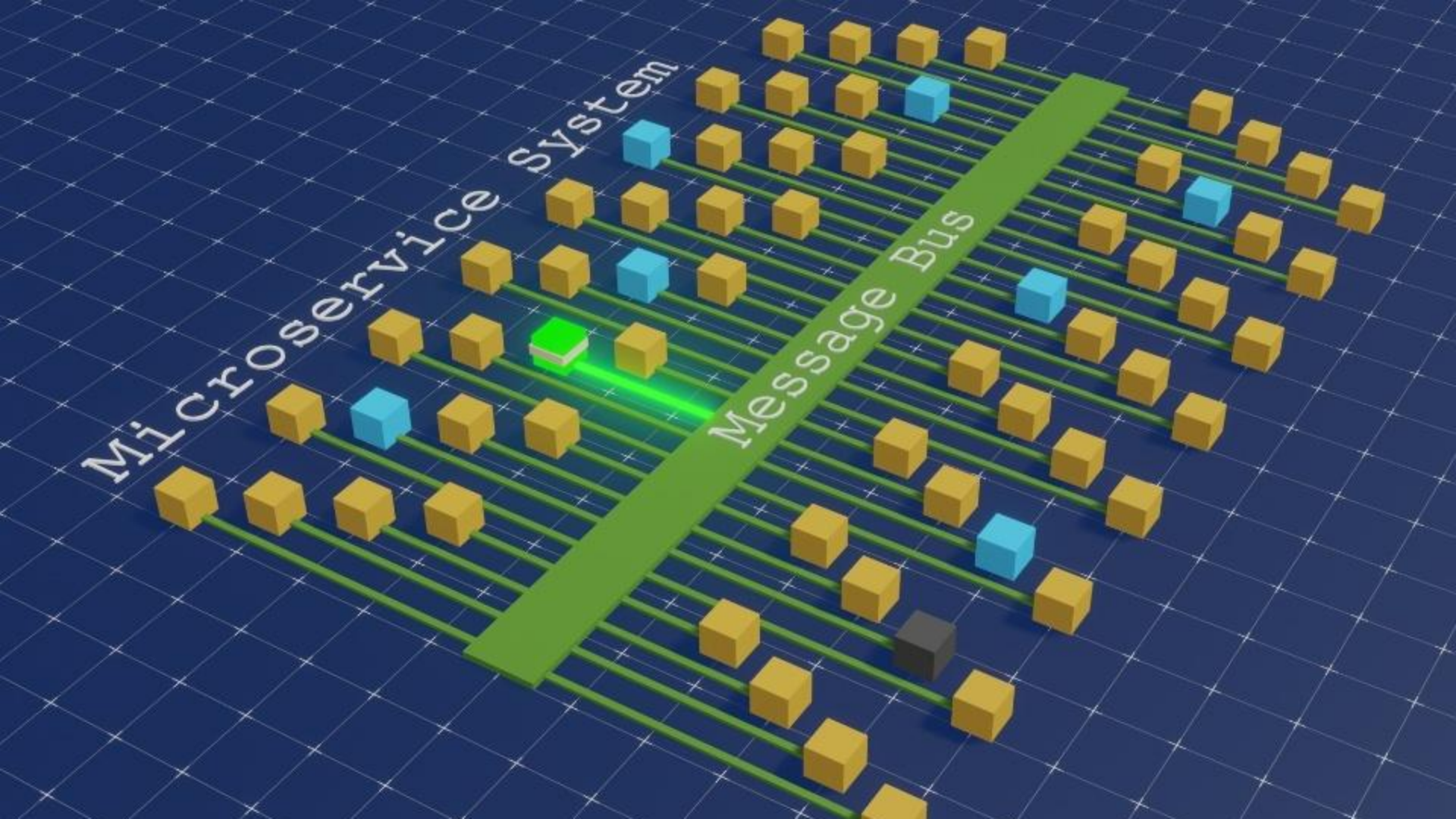

Microservice System
Message Bus

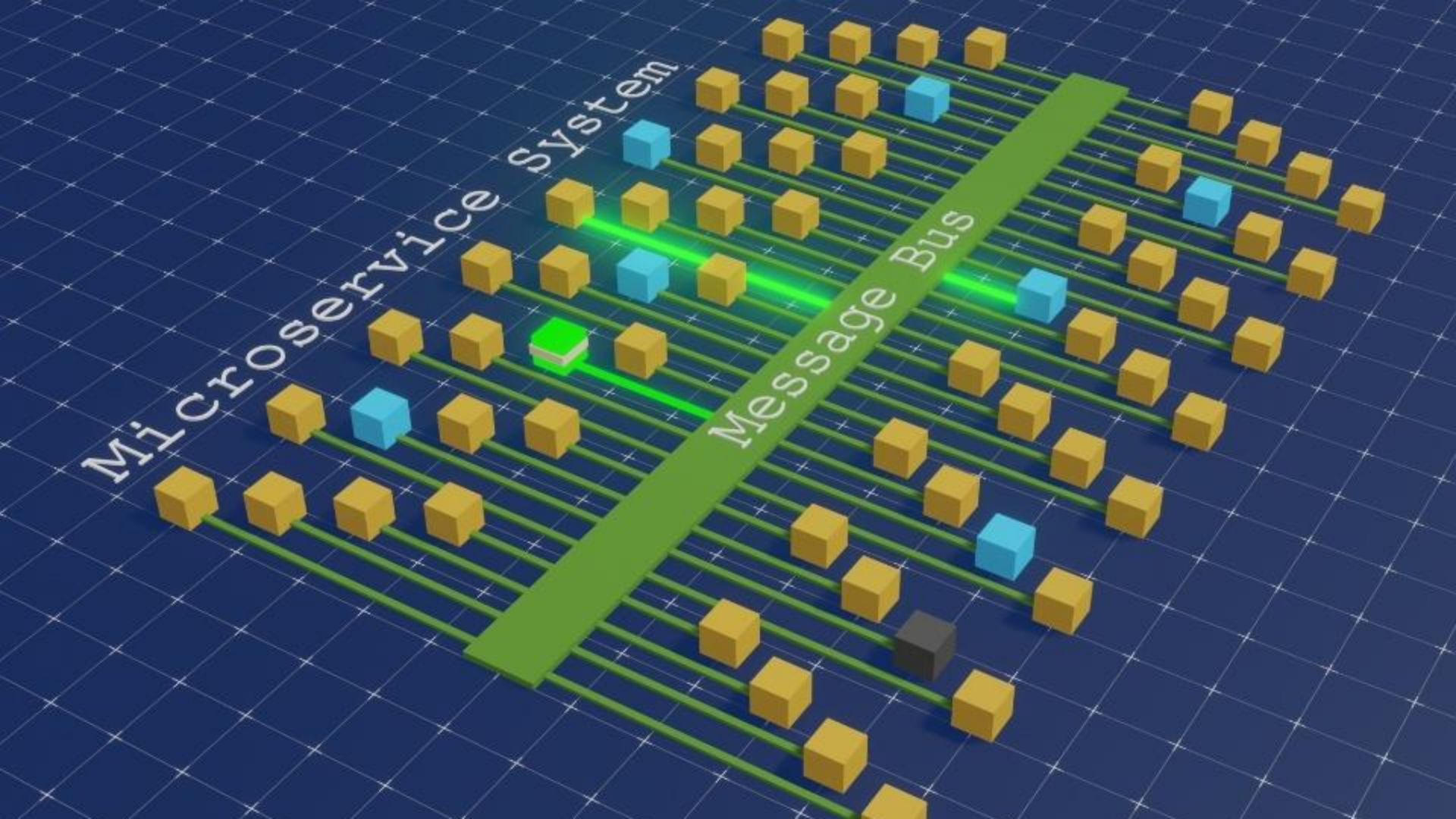

Microservice System
Message Bus

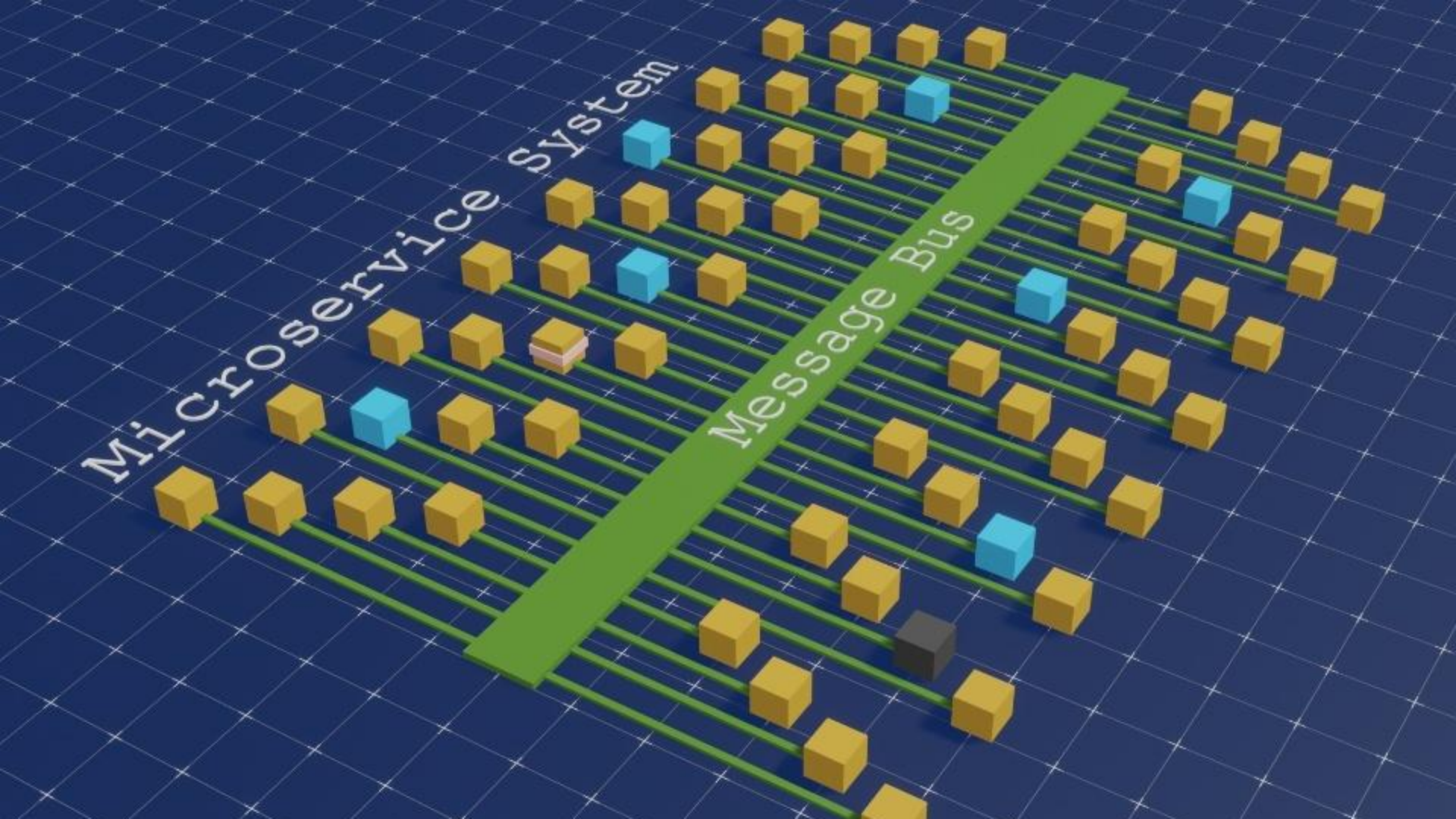

Microservice System
Message Bus

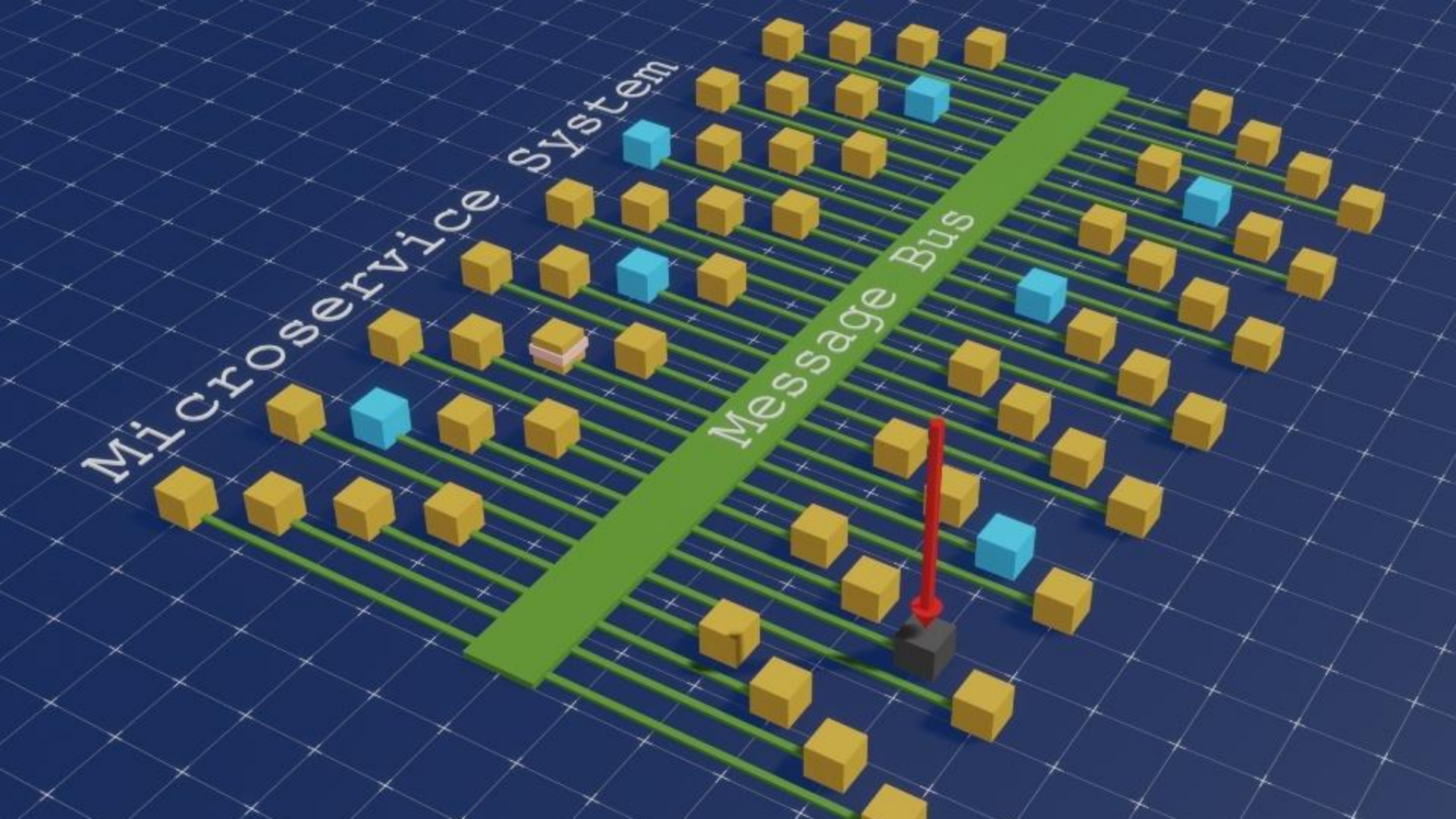

Microservice System
Message Bus

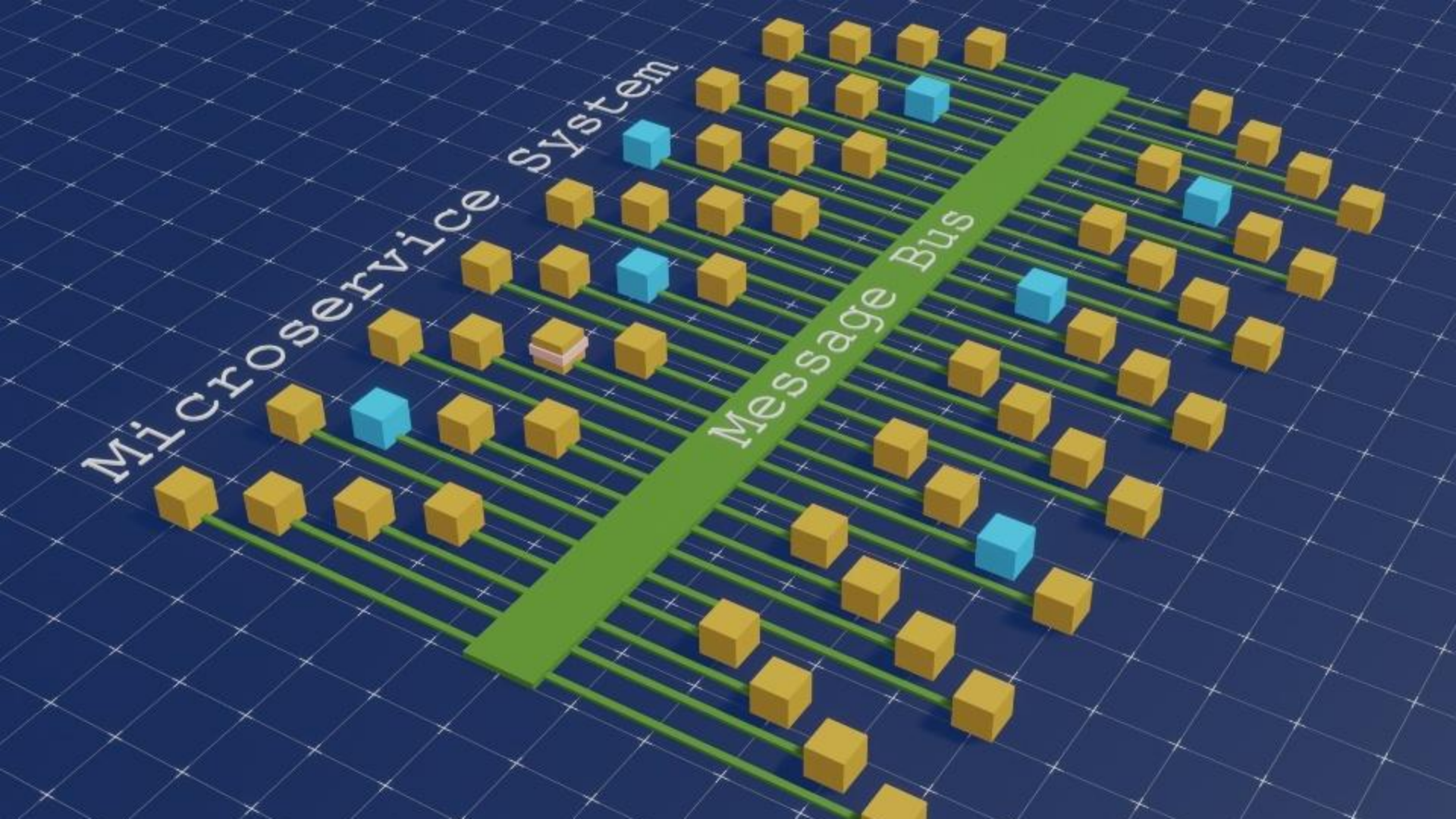

Microservice System
Message Bus

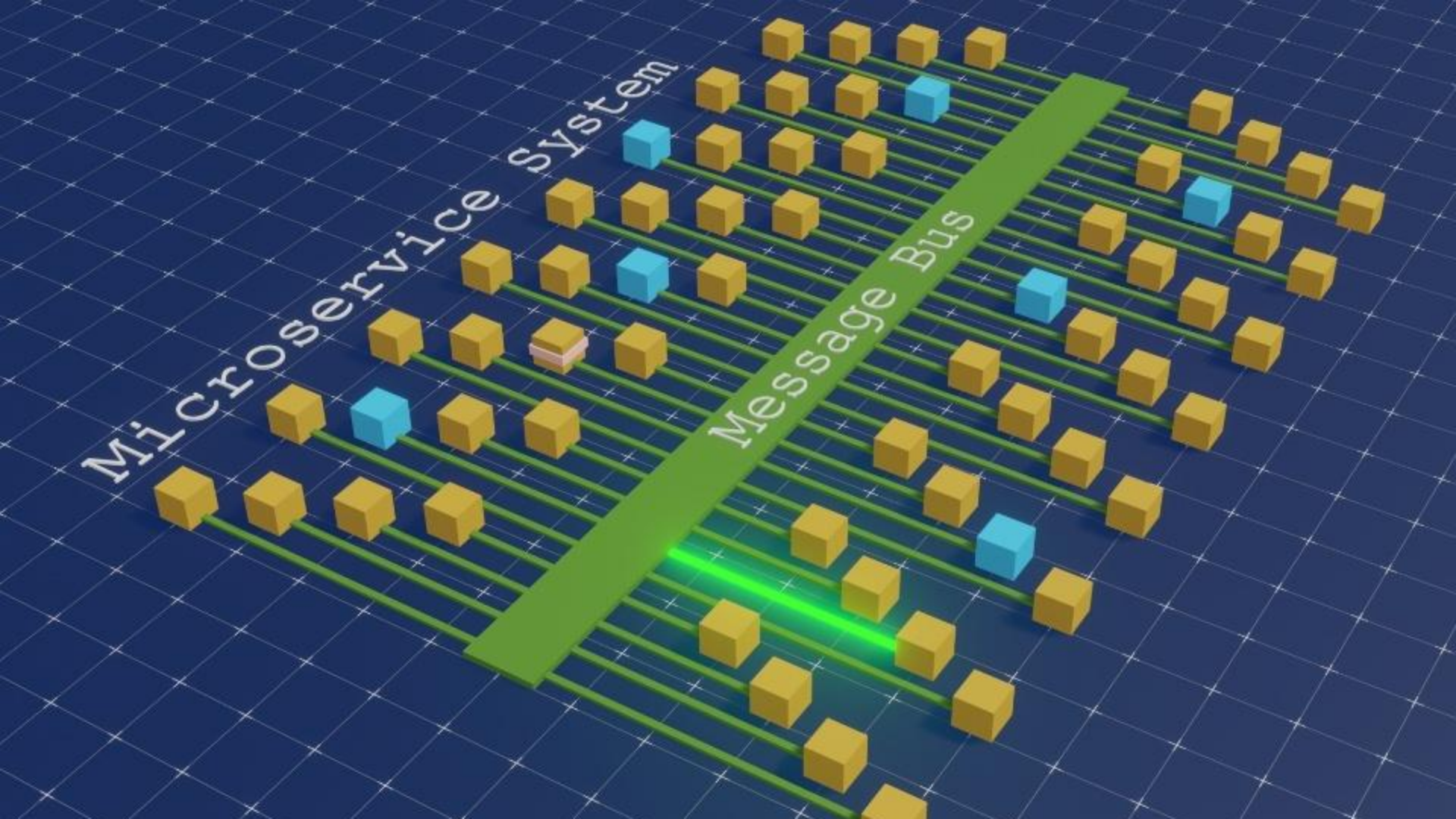

Microservice System
Message Bus

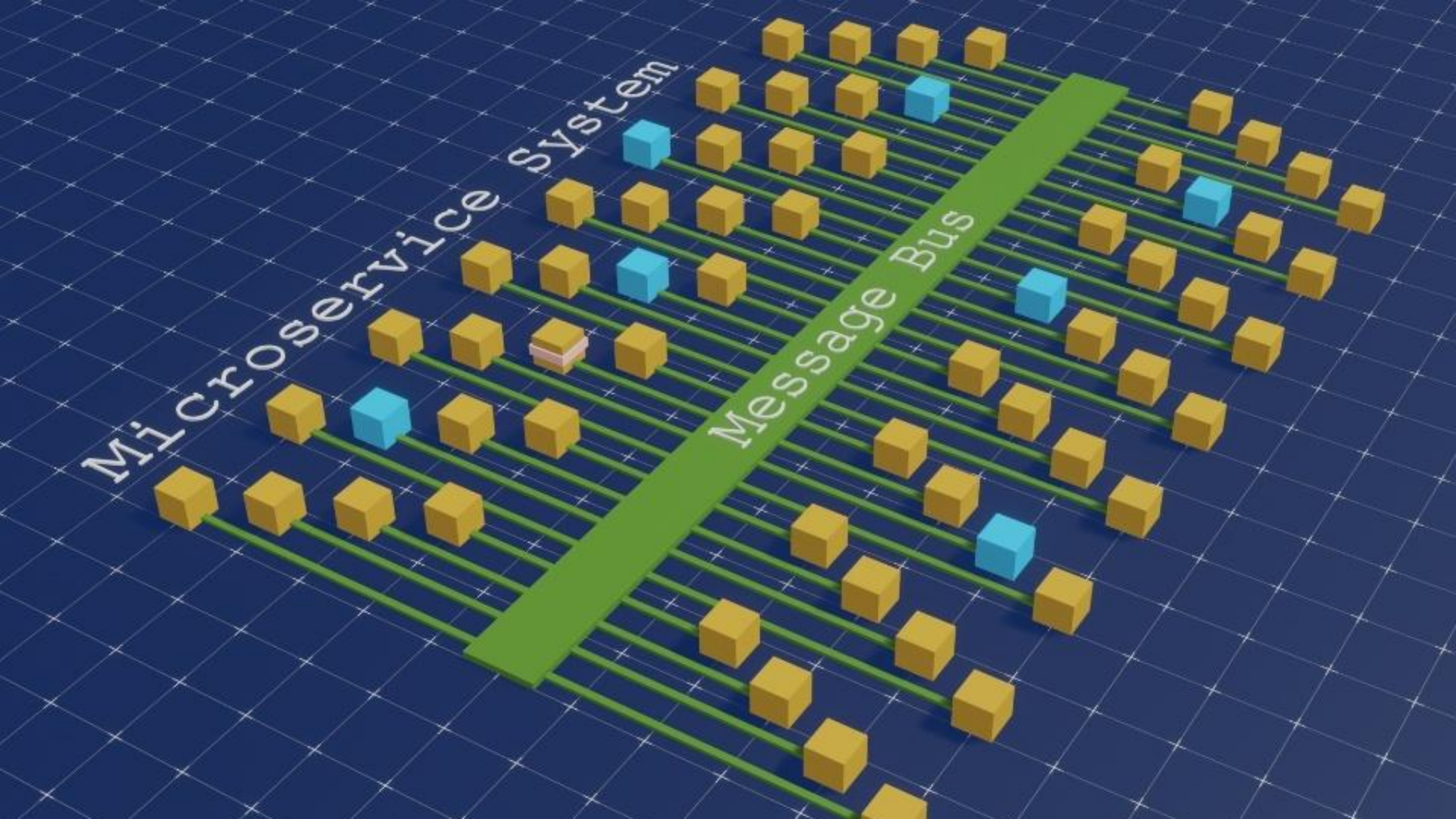

Microservice System
Message Bus

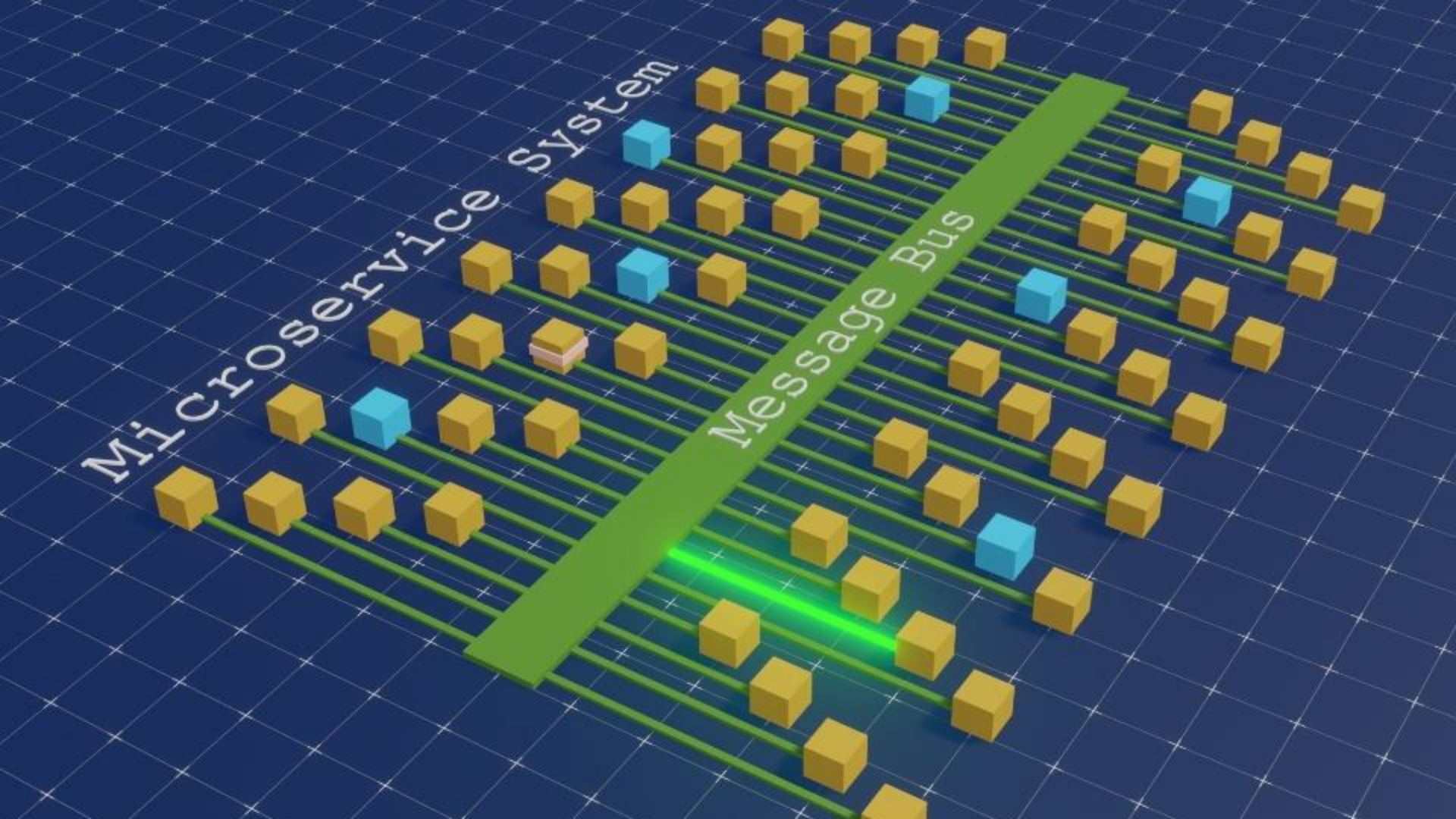

Microservice System
Message Bus

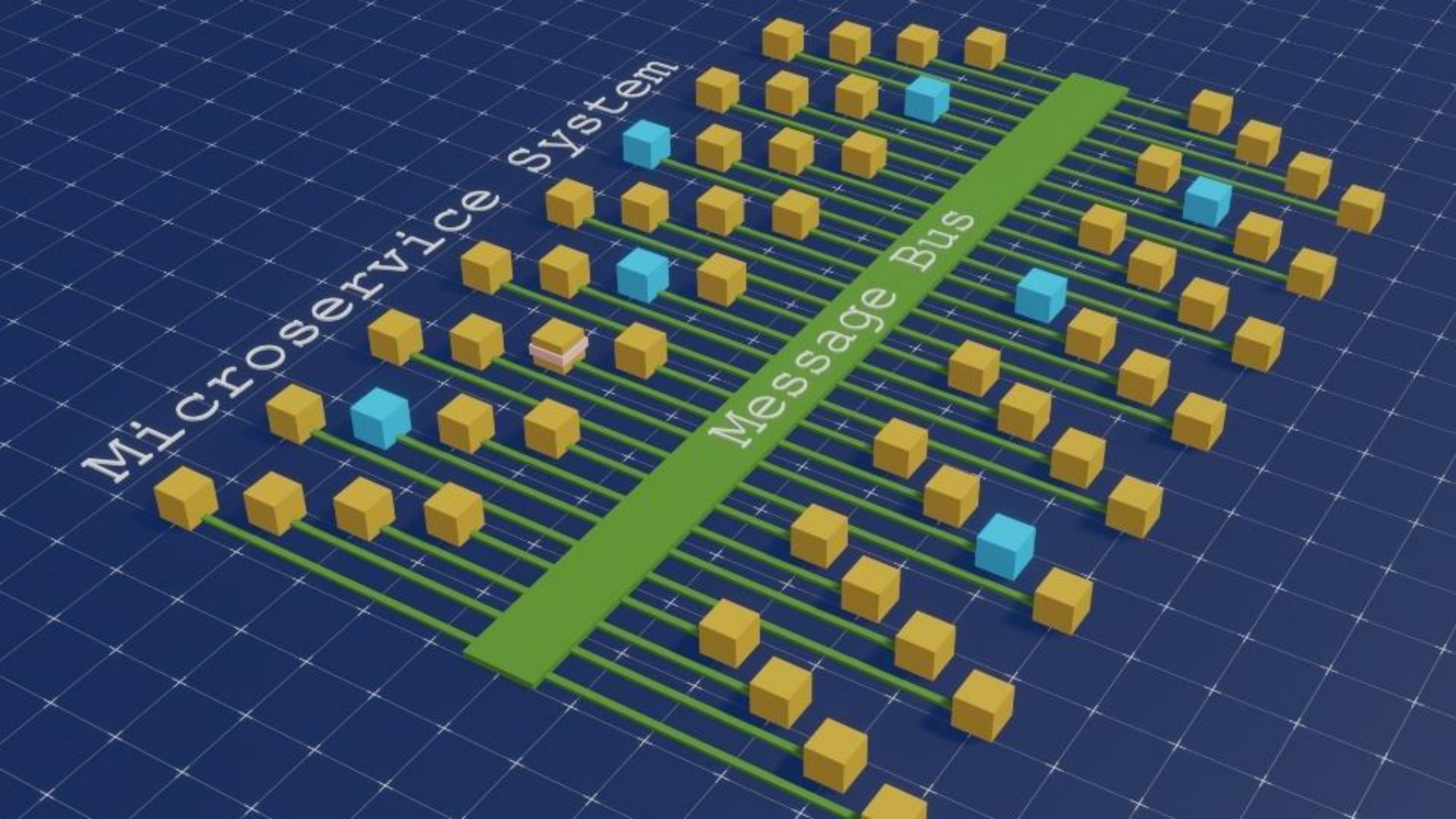

Microservice System
Message Bus

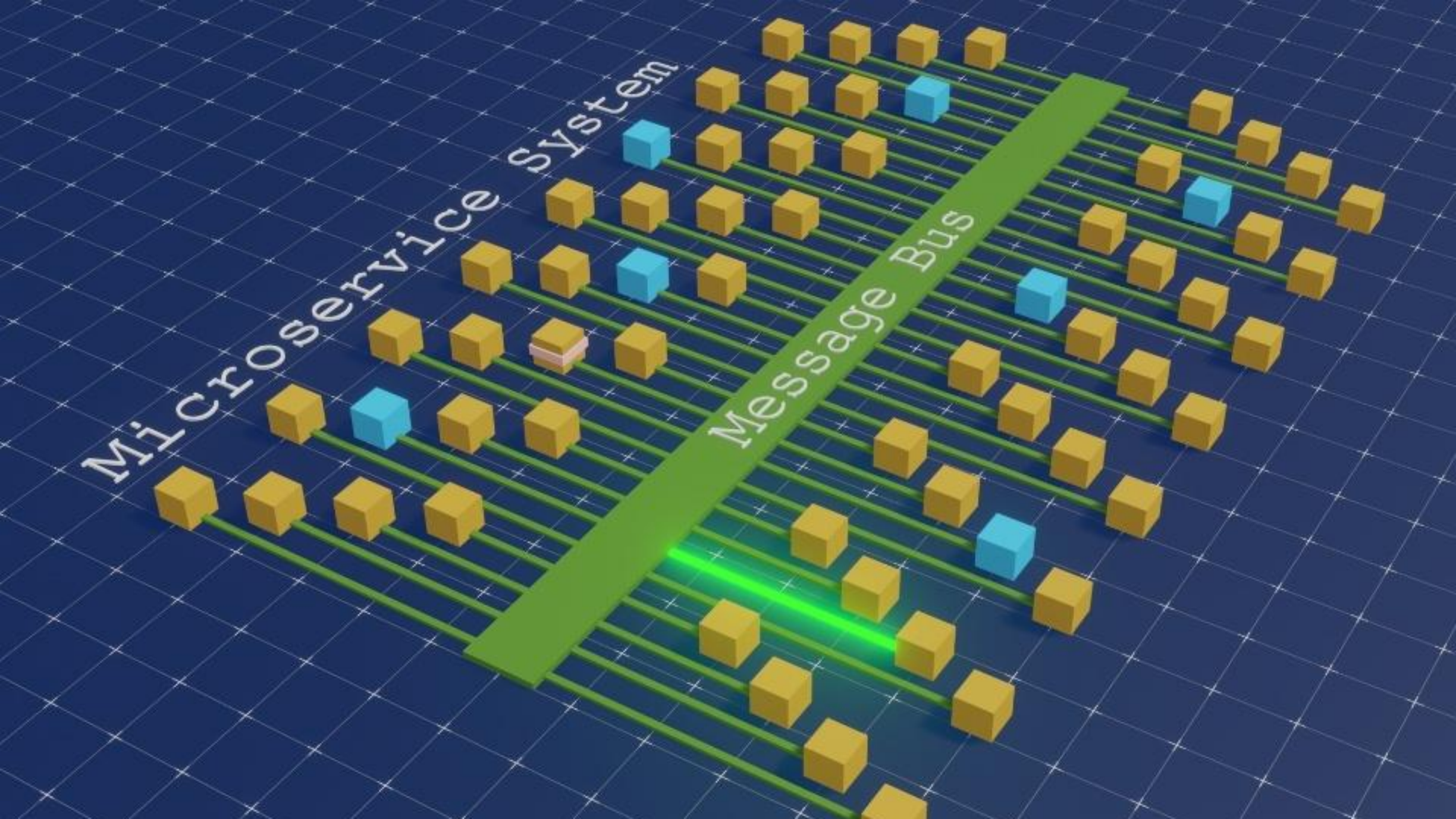

Microservice System
Message Bus

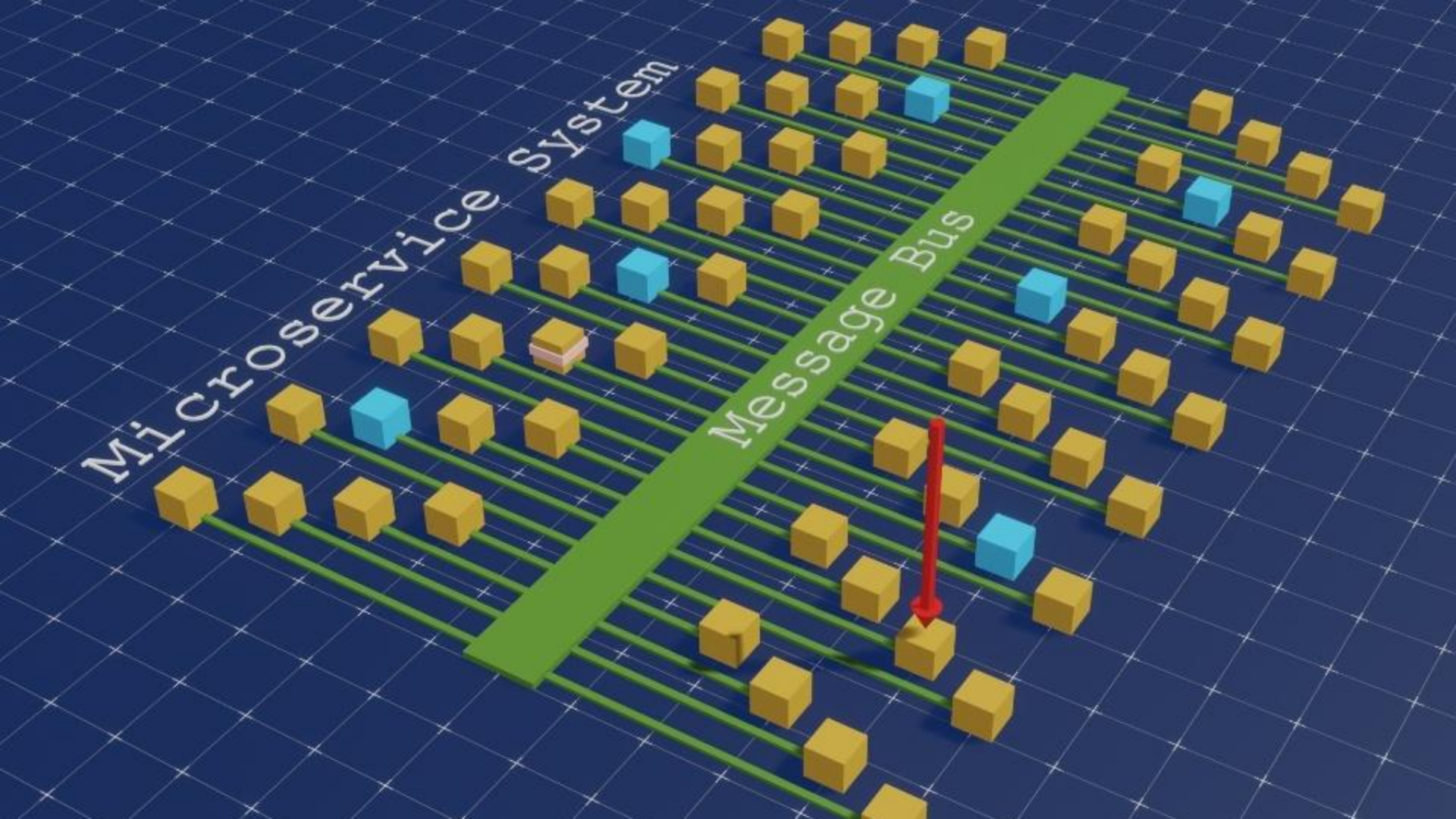

Microservice System
Message Bus


Microservice System
Message Bus


Microservice System
Message Bus


Microservice System
Message Bus

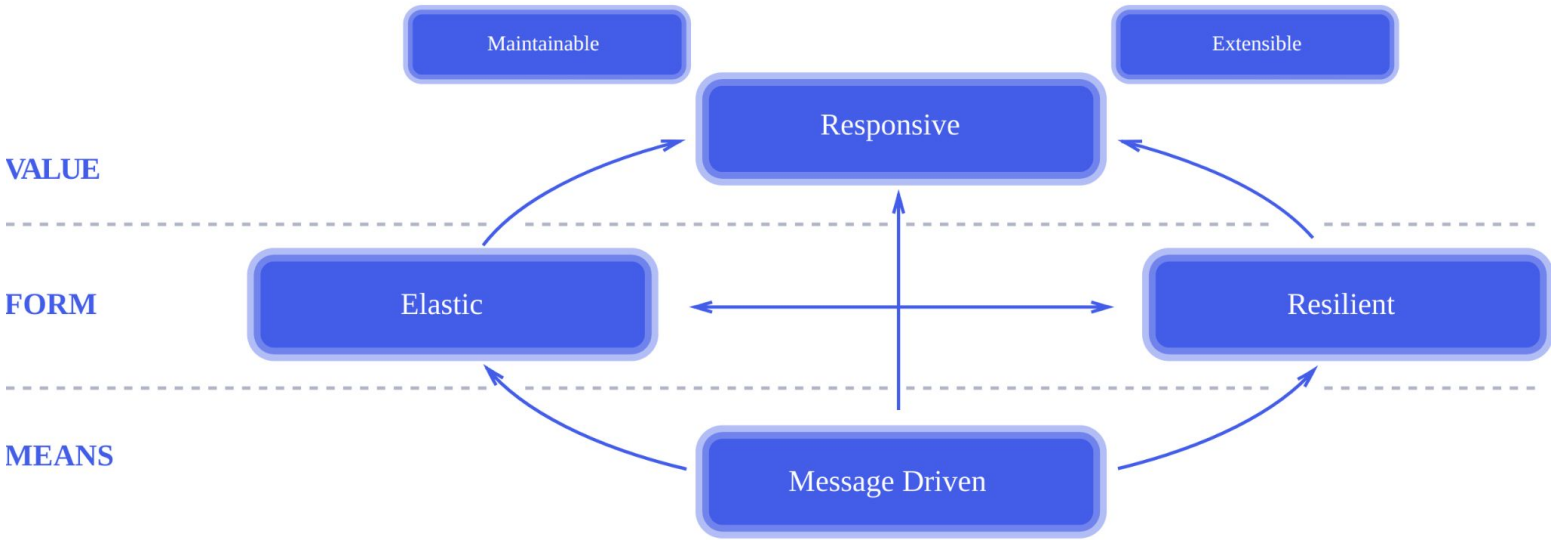

Microservice System
Message Bus


Microservice System
Message Bus

Advantages of Kafka's Communication Approach

- **Decoupled systems:** Easier maintenance and flexibility
- Asynchronous communication with robust delivery guarantees
- Standardized protocols enhance uniformity across systems
- Consumers manage their own data processing pace

The Reactive Manifesto



https://www.reactivemanifesto.org/

Event-Driven - Advantages

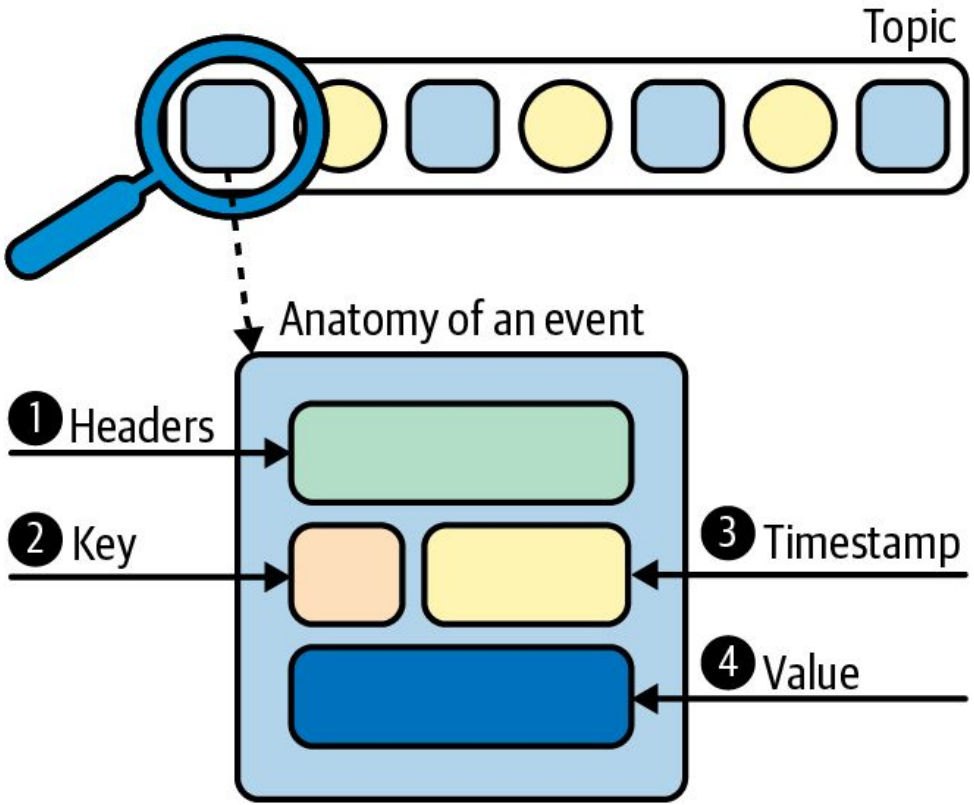
1. Decoupling of services
2. Scalability
3. Resilience and Fault Tolerance
4. Improved Responsiveness & Performance
5. Ease of Integration
6. Enables Reactive Programming Models

Event

Introduction to Data in Kafka Topics

- Kafka uses terms like messages, records, and events interchangeably.
- Preferred term in this context: Event.
- Definition of an event: A timestamped key-value pair that records an occurrence.

Anatomy of an Event in Kafka



Anatomy of an Event in Kafka

- **Application-level Headers:** Optional metadata, not commonly focused on in this text.
- **Keys**: Optional but vital for data distribution across partitions; help in correlating related records.
- **Timestamp**: Associates each event with the time of its occurrence; detailed exploration in later chapters.
- **Value**: Contains the event's actual data, stored as a byte array; requires deserialization by client applications.

Kafka Internal Arch

https://docs.google.com/presentation/d/1oZU7PEcszM1C1HmR7u7EI5-DhRiQJNKi/edit?usp=sharing&ouid=116008769779639876320&rtpof=true&sd=true

Zookeeper

Zookeeper is essentially a service for distributed systems offering a hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems.

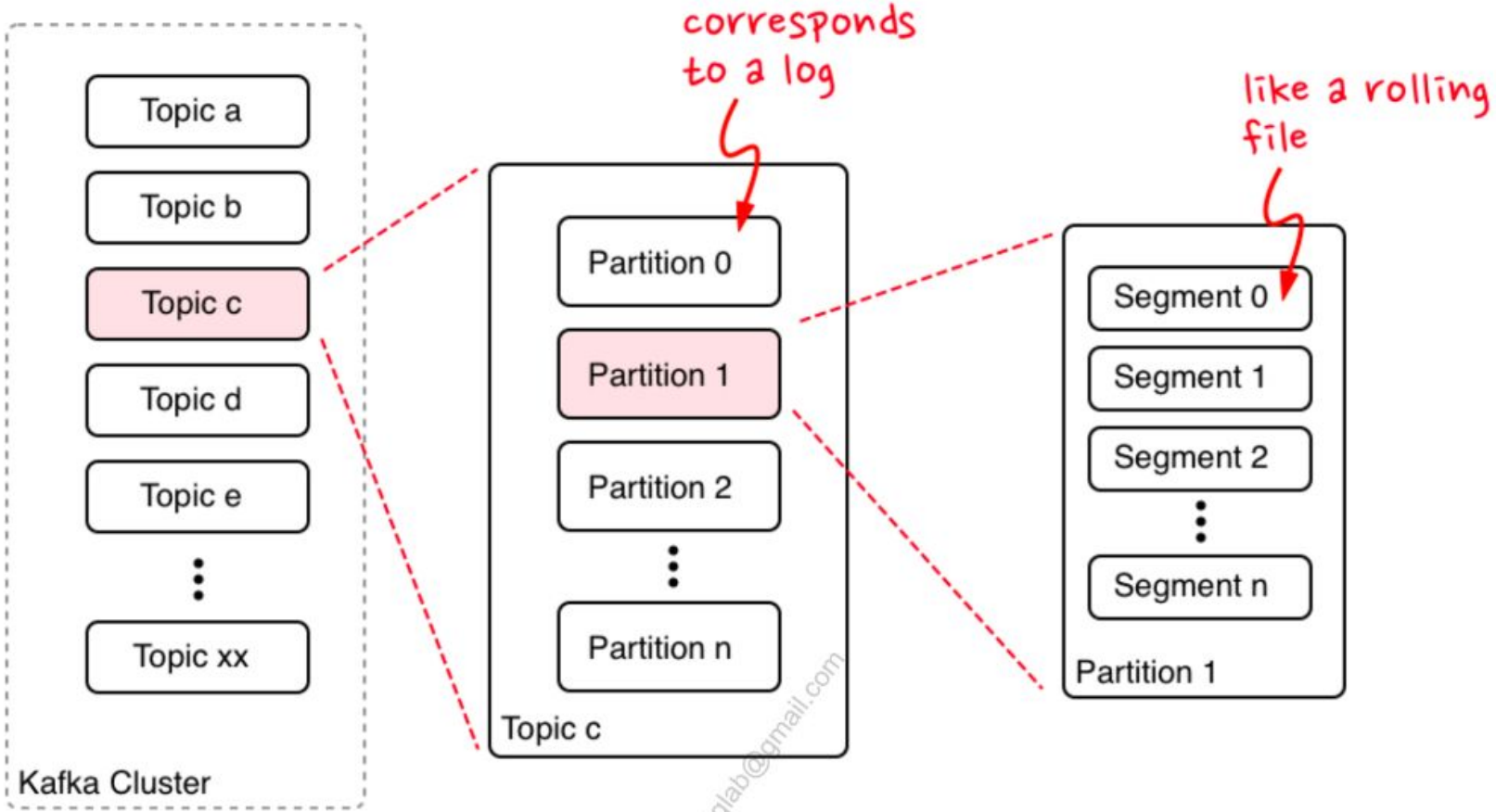
Role of Zookeeper in Kafka

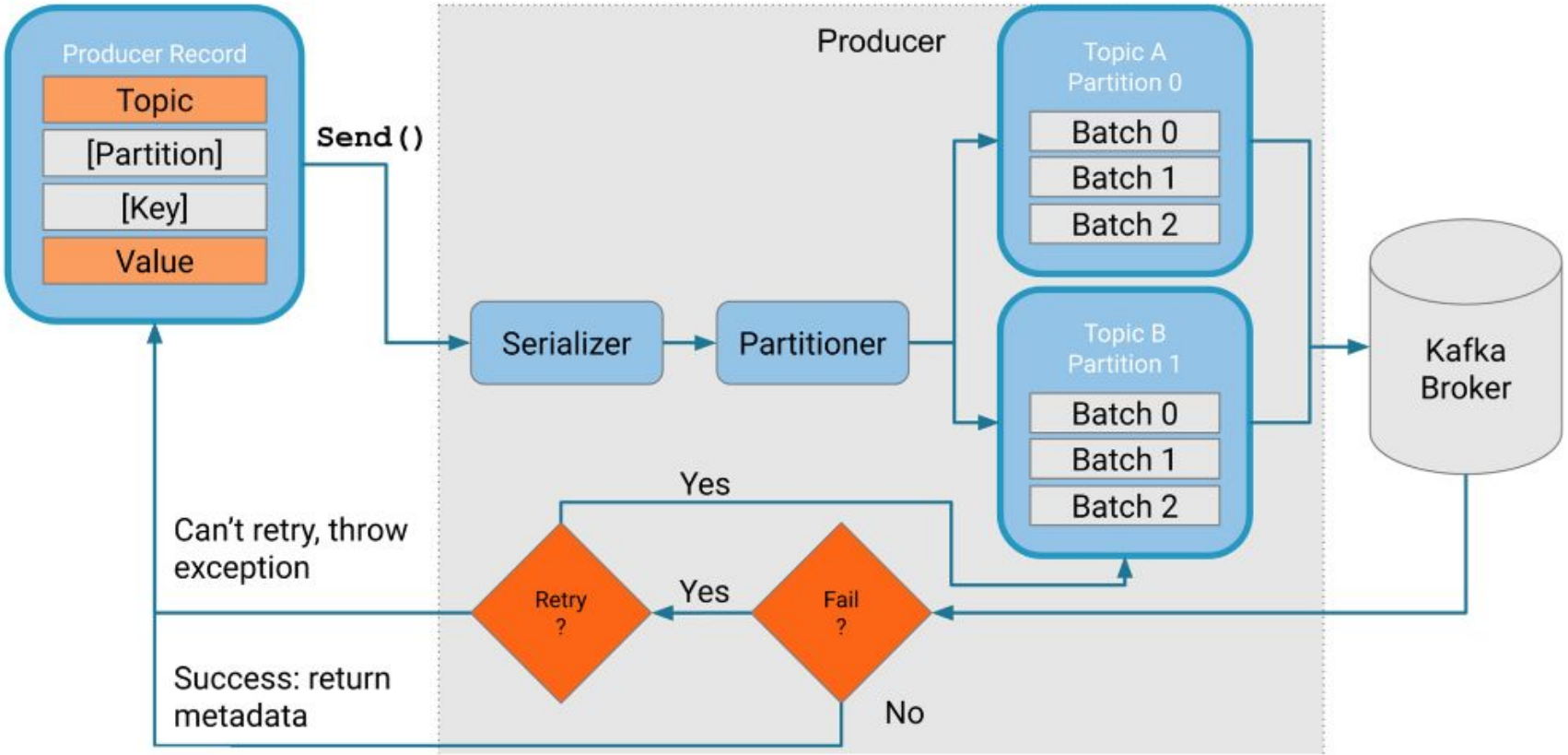
1. Brokers registration, with heartbeat mechanism to keep the broker list updated
2. Maintaining a list of topics
   - ➔ Topic configuration (Partitions, Replication factor, additional configs etc.)
   - ➔ The list of In-sync replicas for Partitions
3. Performs leader election in case any broker is down
4. Store Access Control Lists, if security is enabled
   - ➔ Topics
   - ➔ Consumer groups
   - ➔ Users

Apache Kafka Broker

- An computer, instance, or container running the Kafka process
- Manage partitions
- Handle write and read requests
- Manage replication of partitions
- Intentionally very simple

Producer


corresponds to a log
like a rolling file
Topic a
Topic b
Topic c
Topic d
Topic e
Topic xx
Kafka Cluster
Partition 0
Partition 1
Partition 2
Partition n
Topic c
Segment 0
Segment 1
Segment 2
Segment n
Partition 1


Producer Record
Topic
[Partition]
[Key]
Value
Send()
Producer
Topic A
Partition 0
Batch 0
Batch 1
Batch 2
Topic B
Partition 1
Batch 0
Batch 1
Batch 2
Kafka
Broker
Serializer
Partitioner
Yes
Can't retry, throw
exception
Retry
?
Yes
Fail
?
Success: return
metadata
No

Producer is to decide which partition to send the messages to.

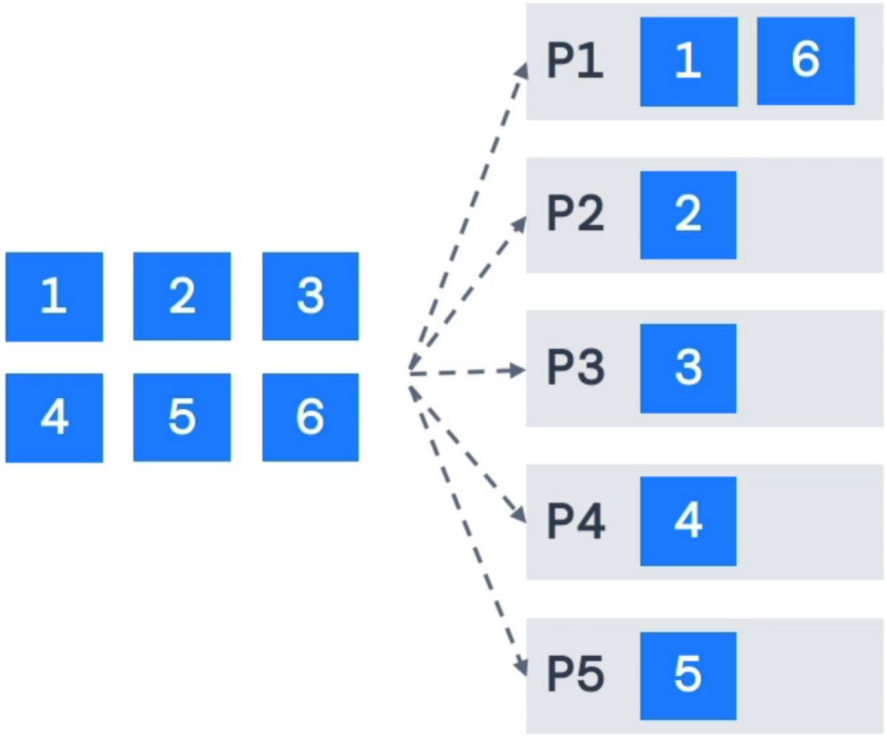
1. **No key specified** :- producer will randomly decide partition and would try to balance the total number of messages on all partitions
2. **Key Specified :-** the producer uses Consistent Hashing to map the key to a partition.
3. **Partition Specified :-** You can hardcode the destination partition as well.
4. **Custom Partitioning logic:-** We can write some rules depending on which the partition can be decided.

Producer Default Partitioner

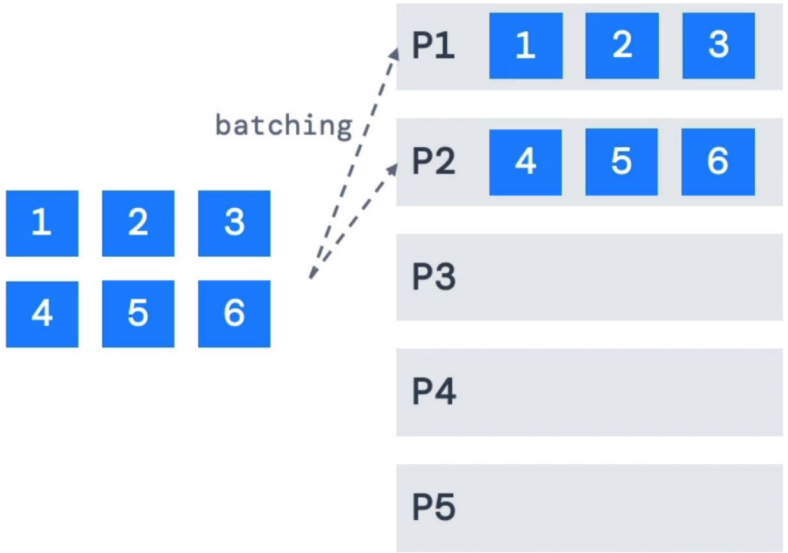
Producer Default Partitioner

- By default, your keys are hashed using the "murmur2" algorithm.
- It is most likely preferred to not override the behavior of the partitioner, but it is possible to do so (partitioner.class).
- The formula is:
  targetPartition = Utils.abs(Utils.murmur2(record.key())) % numPartitions;
- This means that same key will go to the same partition (we already know this), and adding partitions to a topic will completely alter the formula

Kafka <= 2.3



Kafka >= 2.4

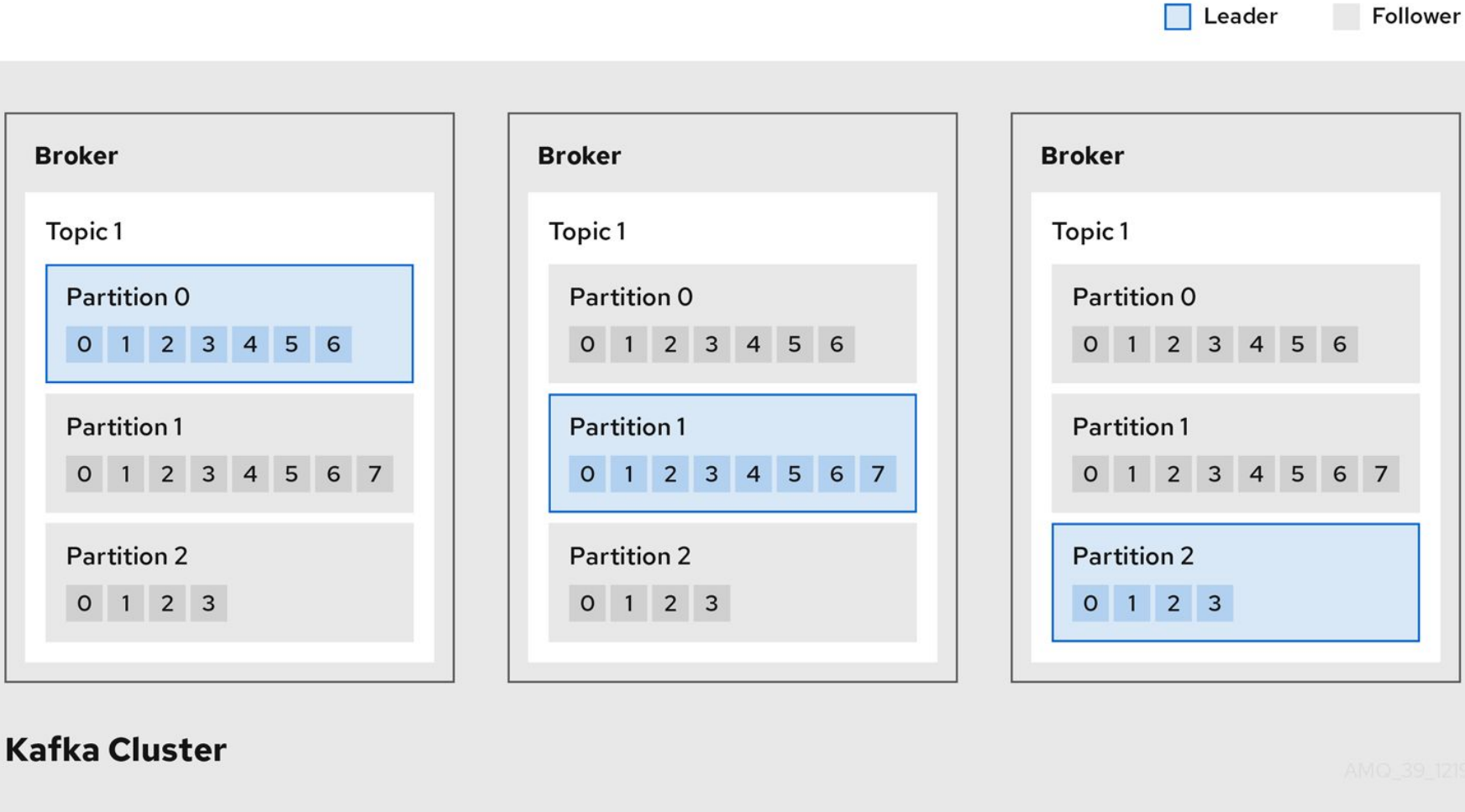


Sticky Partitioner
(performance improvement)

Data Plane: Replication Protocol

Replication

- Copies of data for fault tolerance
- One lead partition and N-1 followers
- In general, writes and reads happen to the leader
- An invisible process to most developers
- Tunable in the producer


Leader
Follower
Broker
Topic 1
Partition 0
0 1 2 3 4 5 6
Partition 1
0 1 2 3 4 5 6 7
Partition 2
0 1 2 3
Broker
Topic 1
Partition 0
0 1 2 3 4 5 6
Partition 1
0 1 2 3 4 5 6 7
Partition 2
0 1 2 3
Broker
Topic 1
Partition 0
0 1 2 3 4 5 6
Partition 1
0 1 2 3 4 5 6 7
Partition 2
0 1 2 3
Kafka Cluster
AMQ_39_1219

https://docs.google.com/presentation/d/1oZU7PEcszM1C1HmR7u7EI5-DhRiQJNKi/edit#slide=id.p28

Durability,Availability & Ordering Guarantees

https://docs.google.com/presentation/d/1oZU7PEcszM1C1HmR7u7EI5-DhRiQJNKi/edit#slide=id.p86

Producer Retries

- In case of transient failures, developers are expected to handle exceptions, otherwise the data will be lost.
- Example of transient failure:
  - NotEnoughReplicasException

- There is a "retries" setting
  - defaults to 0 for Kafka <= 2.0
  - defaults to 2147483647 for Kafka >= 2.1

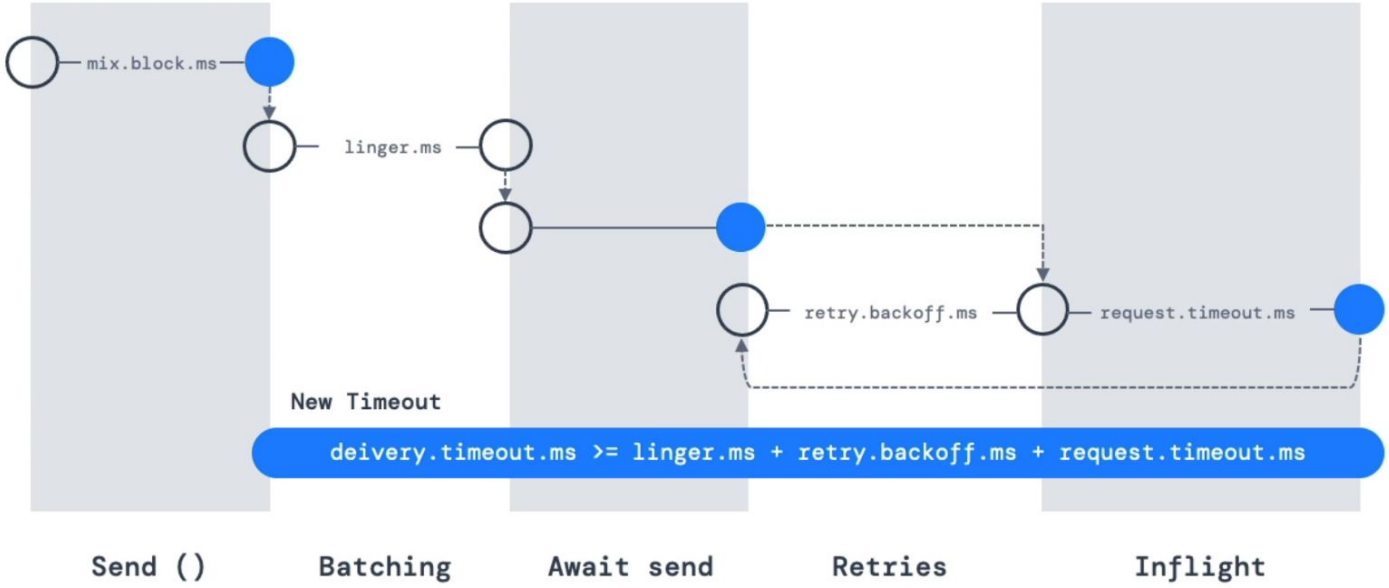
- The retry.backoff.ms setting is by default 100 ms

Producer Timeouts

- If retries > 0, for example retries = 2147483647
- the producer won't try the request for ever, it's bounded by a timeout

- For this, you can set an intuitive Producer Timeout (KIP-91 – Kafka 2.1)
- delivery.timeout.ms = 120 000 ms == 2 minutes

- Records will be failed if they can't be acknowledged in delivery.timeout.ms


mix.block.ms
linger.ms
retry.backoff.ms
request.timeout.ms
New Timeout
deivery.timeout.ms >= linger.ms + retry.backoff.ms + request.timeout.ms
Send ()
Batching
Await send
Retries
Inflight

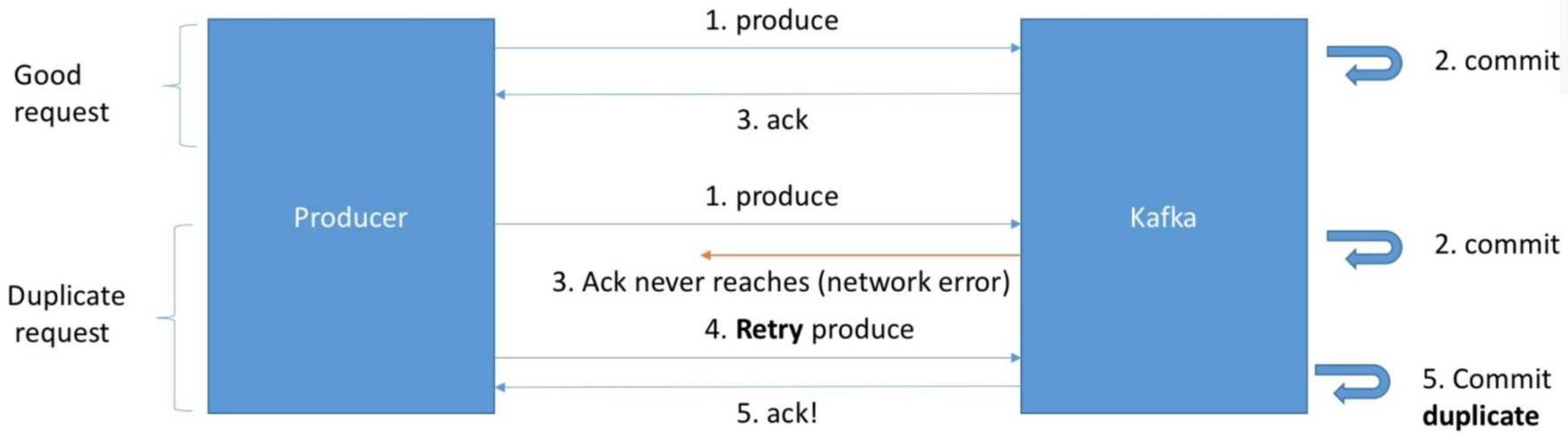
Producer Retries : Warnings

- In case of retries, there is a chance that messages will be sent out of order (if a batch has failed to be sent).
- If you rely on key-based ordering, that can be an issue.

- For this, you can set the setting while controls how many produce requests can be made in parallel: max.in.flight.requests.per.connection
  - Default: 5
  - Set it to 1 if you need to ensure ordering (may impact throughput)

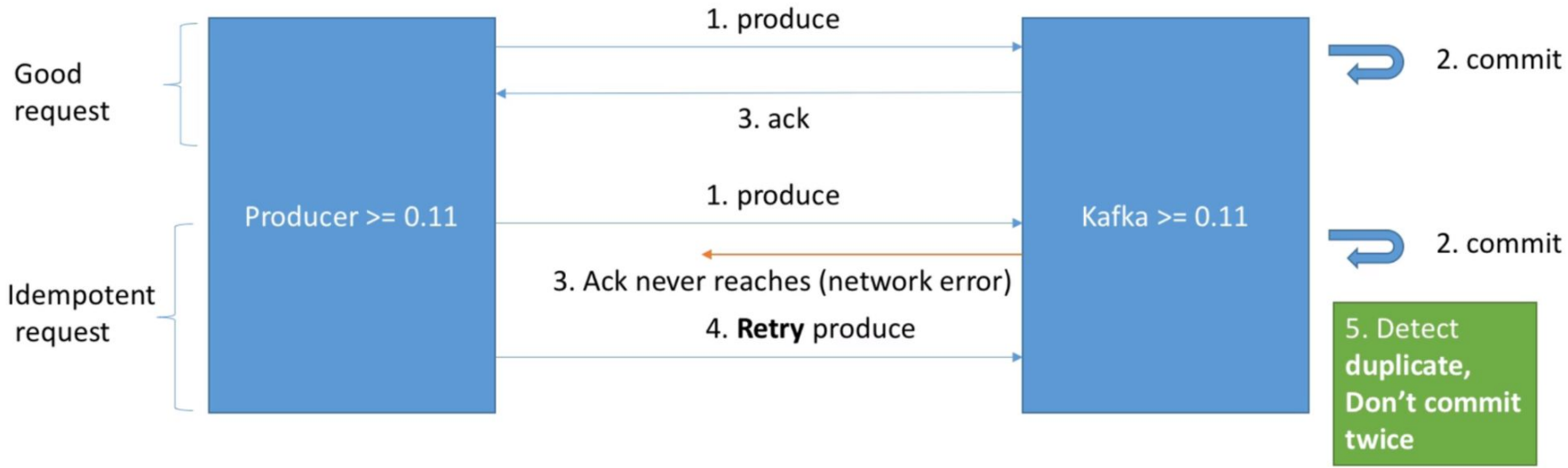
Idempotent Producer

- Here's the problem: the Producer can introduce duplicate messages in Kafka due to network errors



Idempotent Producer

- In Kafka >= 0.11, you can define a "idempotent producer" which won't introduce duplicates on network error



Idempotent Producer

- Idempotent producers are great to guarantee a stable and safe pipeline!
- They come with:
  - retries = Integer.MAX_VALUE (2^31−1 = 2147483647)
  - max.in.flight.requests=1 (Kafka == 0.11) or
  - max.in.flight.requests=5 (Kafka >= 1.0 – higher performance & keep ordering)
    See https://issues.apache.org/jira/browse/KAFKA-5494
  - acks=all
- These settings are applied automatically after your producer has started if you don't set them manually
- Just set:
  - **producerProps.put("enable.idempotence", true);**

Safe Producer

Kafka < 0.11

- acks=all (producer level)
  - Ensures data is properly replicated before an ack is received
- min.insync.replicas=2 (broker/topic level)
  - Ensures two brokers in ISR at least have the data after an ack
- retries=MAX_INT (producer level)
  - Ensures transient errors are retried indefinitely
- max.in.flight.requests.per.connection=1 (producer level)
  - Ensures only one request is tried at any time, preventing message re-ordering in case of retries

Kafka >= 0.11

- enable.idempotence=true (producer level) + min.insync.replicas=2 (broker/topic level)
  - Implies acks=all, retries=MAX_INT, max.in.flight.requests.per.connection=1 if Kafka 0.11 or 5 if Kafka >= 1.0
  - while keeping ordering guarantees and improving performance!

- Running a "safe producer" might impact throughput and latency, always test for your use case

High-Throughput producer

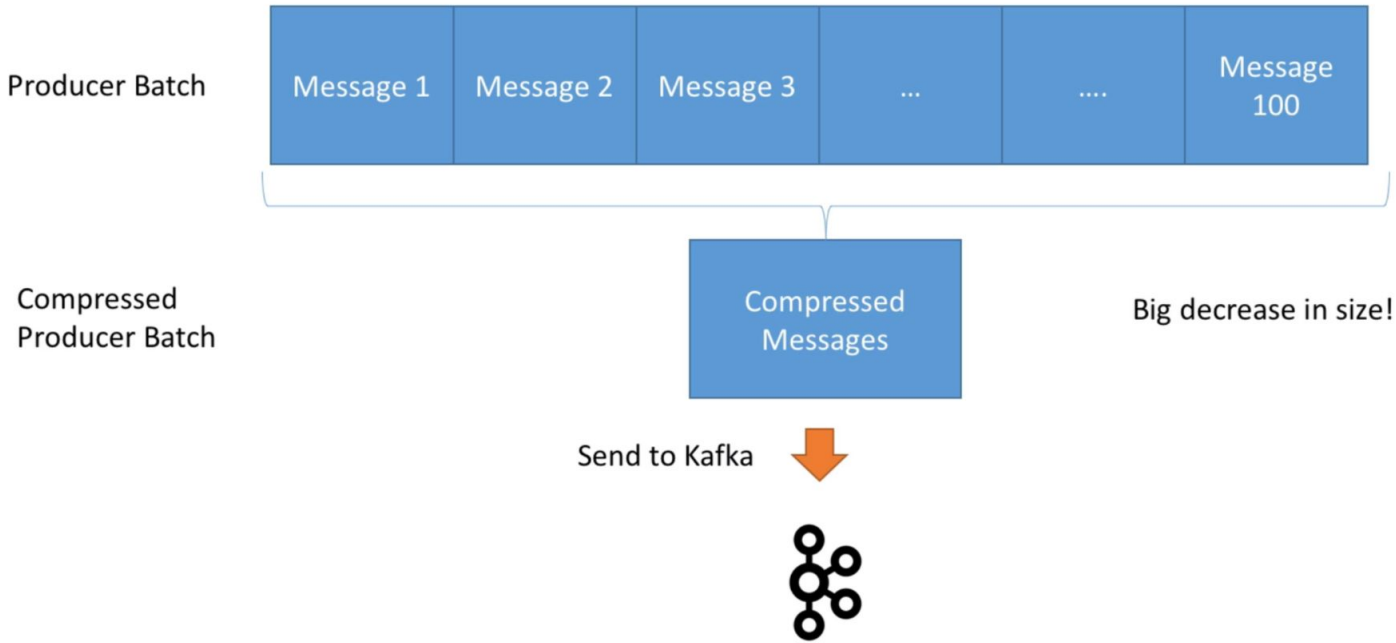
Producer : Message Compression

- Producer usually send data that is text-based, for example with JSON data
- In this case, it is important to apply compression to the producer.

- Compression is enabled at the Producer level and doesn't require any configuration change in the Brokers or in the Consumers
- "compression.type" can be 'none' (default), 'gzip', 'lz4', 'snappy'

- Compression is more effective the bigger the batch of message being sent to Kafka!
- Benchmarks here: https://blog.cloudflare.com/squeezing-the-firehose/

Message Compression



Message Compression

- The compressed batch has the following advantage:
  - Much smaller producer request size (compression ratio up to 4x!)
  - Faster to transfer data over the network => less latency
  - Better throughput
  - Better disk utilisation in Kafka (stored messages on disk are smaller)
- Disadvantages (very minor):
  - Producers must commit some CPU cycles to compression
  - Consumers must commit some CPU cycles to decompression
- Overall:
  - Consider testing snappy or lz4 for optimal speed / compression ratio

Message Compression Recommendations

- Find a compression algorithm that gives you the best performance for your specific data. Test all of them!
- Always use compression in production and especially if you have high throughput
- Consider tweaking linger.ms and batch.size to have bigger batches, and therefore more compression and higher throughput

- By default, Kafka tries to send records as soon as possible
  - It will have up to 5 requests in flight, meaning up to 5 messages individually sent at the same time.
  - After this, if more messages have to be sent while others are in flight, Kafka is smart and will start batching them while they wait to send them all at once.

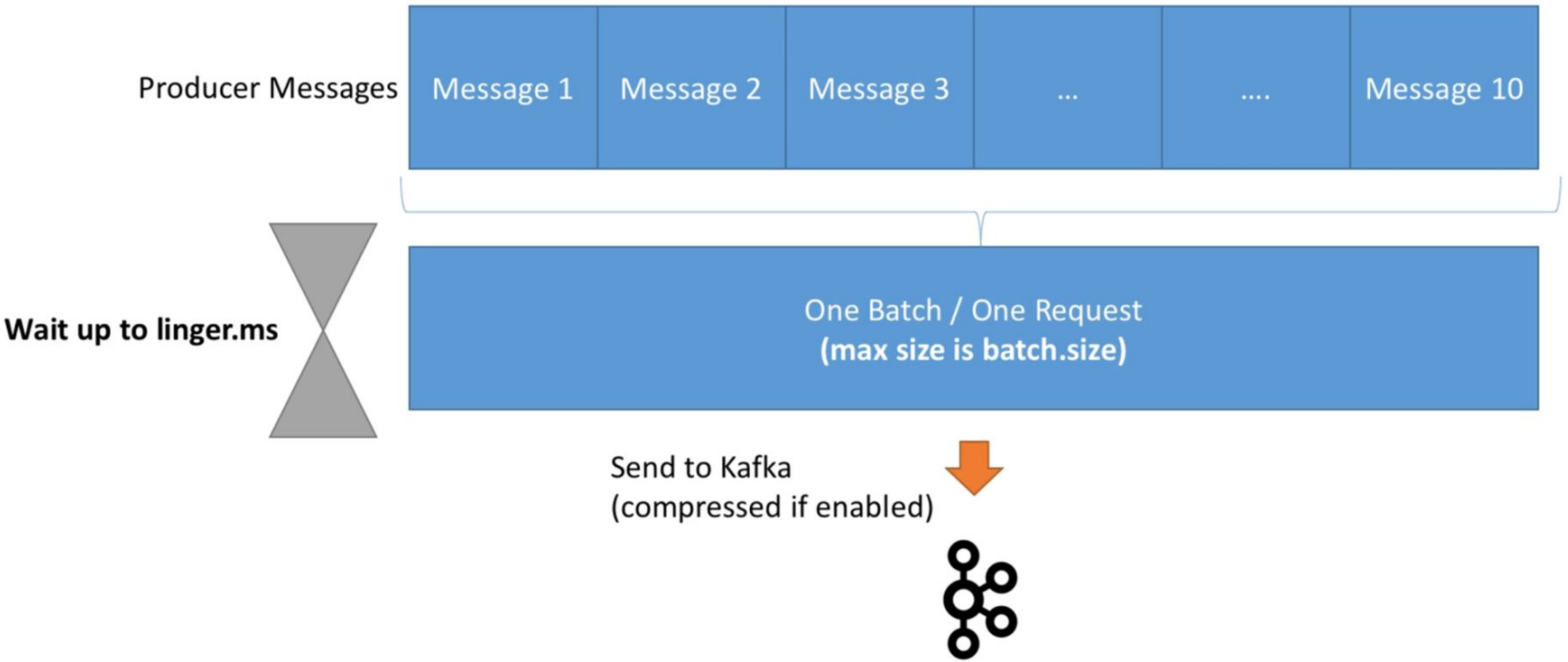
- This smart batching allows Kafka to increase throughput while maintaining very low latency.
- Batches have higher compression ratio so better efficiency

- So how can we control the batching mechanism?

Linger.ms & batch.size

- **Linger.ms:** Number of milliseconds a producer is willing to wait before sending a batch out. (default 0)
- By introducing some lag (for example linger.ms=5), we increase the chances of messages being sent together in a batch
- So at the expense of introducing a small delay, we can increase throughput, compression and efficiency of our producer.
- If a batch is full (see batch.size) before the end of the linger.ms period, it will be sent to Kafka right away!

linger.ms


Producer Messages
Message 1
Message 2
Message 3
...
....
Message 10
Wait up to linger.ms
One Batch / One Request
(max size is batch.size)
Send to Kafka
(compressed if enabled)

Linger.ms & batch.size

- **batch.size:** Maximum number of bytes that will be included in a batch. The default is 16KB.

- Increasing a batch size to something like 32KB or 64KB can help increasing the compression, throughput, and efficiency of requests
- Any message that is bigger than the batch size will not be batched
- A batch is allocated per partition, so make sure that you don't set it to a number that's too high, otherwise you'll run waste memory!

High throughput producer

- We'll add snappy message compression in our producer
- snappy is very helpful if your messages are text based, for example log lines or JSON documents
- snappy has a good balance of CPU / compression ratio

- We'll also increase the batch.size to 32KB and introduce a small delay through linger.ms (20 ms)

Max.block.ms & buffer.memory

- If the producer produces faster than the broker can take, the records will be buffered in memory
- **buffer.memory=33554432 (32MB):** the size of the send buffer

- That buffer will fill up over time and fill back down when the throughput to the broker increases

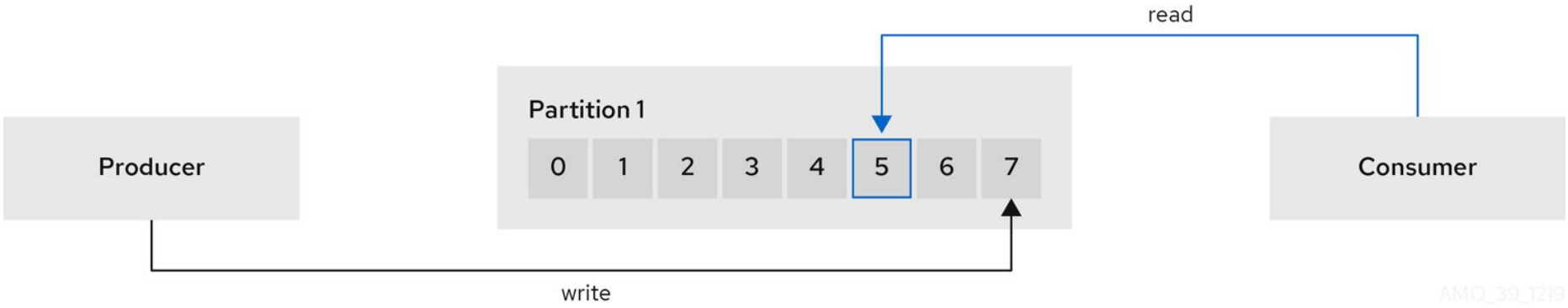
Max.block.ms & buffer.memory

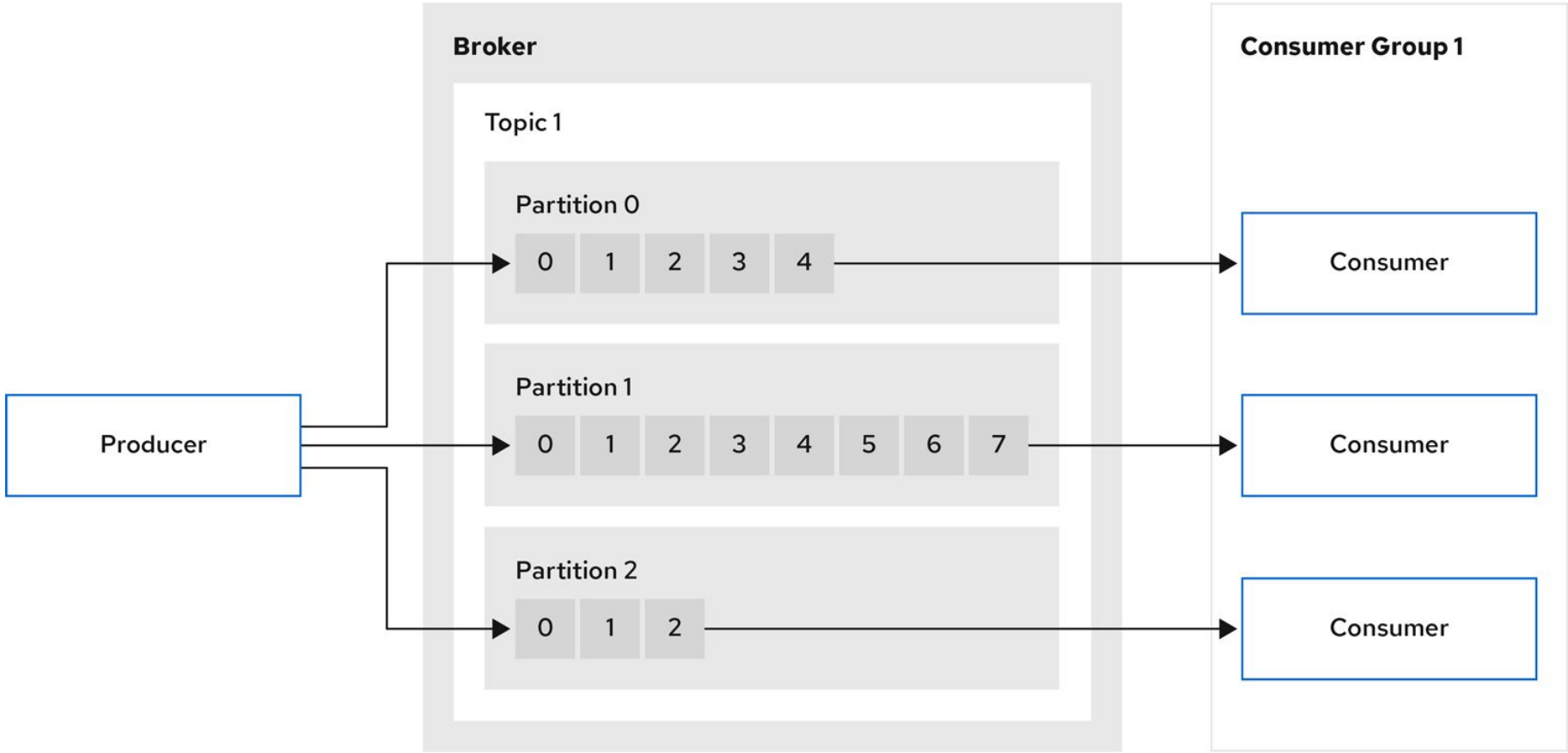
- If that buffer is full (all 32MB), then the .send() method will start to block (won't return right away)

- **max.block.ms=60000**: the time the .send() will block until throwing an exception. Exceptions are basically thrown when
  - The producer has filled up its buffer
  - The broker is not accepting any new data
  - 60 seconds has elapsed.

- If you hit an exception hit that usually means your brokers are down or overloaded as they can't respond to requests

Consumer Group Protocol

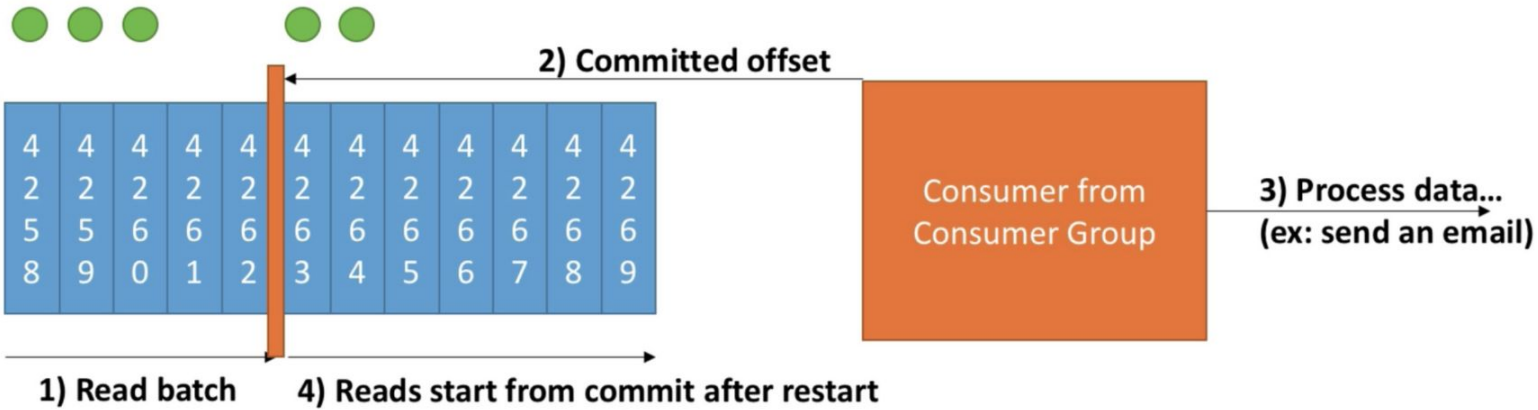

read
Partition 1
0
1
2
3
4
5
6
7
Producer
Consumer
write
AMQ_39_1219


Broker
Topic 1
Partition 0
0
1
2
3
4
Partition 1
0
1
2
3
4
5
6
7
Partition 2
0
1
2
Producer
Consumer Group 1
Consumer
Consumer
Consumer

https://docs.google.com/presentation/d/1oZU7PEcszM1C1HmR7u7EI5-DhRiQJNKi/edit#slide=id.p64

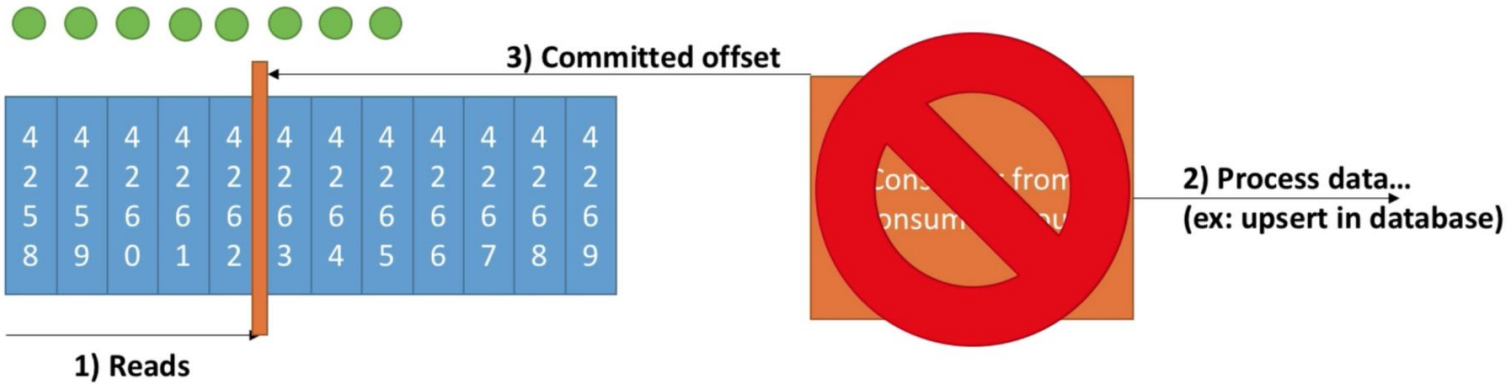
Delivery Semantics : **At Most Once**

- At most once: offsets are committed as soon as the message batch is received. If the processing goes wrong, the message will be lost (it won't be read again).



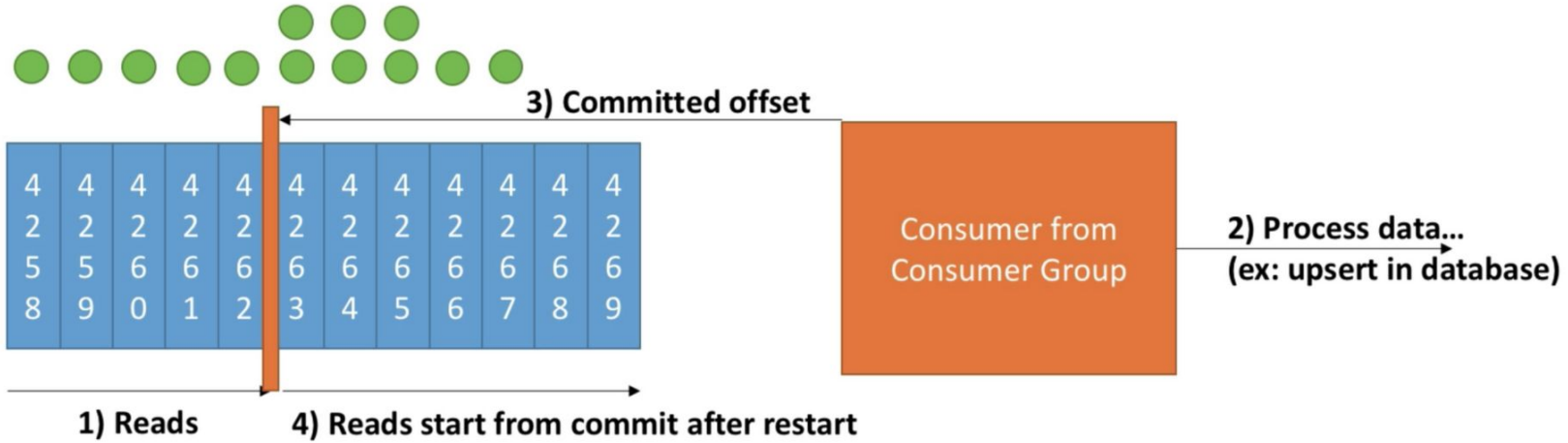
Delivery Semantics : **At Least Once**

- At least once: offsets are committed after the message is processed. If the processing goes wrong, the message will be read again. This can result in duplicate processing of messages. Make sure your processing is **idempotent** (i.e. processing again the messages won't impact your systems)



Delivery Semantics : **At Least Once**

- At least once: offsets are committed after the message is processed. If the processing goes wrong, the message will be read again. This can result in duplicate processing of messages. Make sure your processing is **idempotent** (i.e. processing again the messages won't impact your systems)



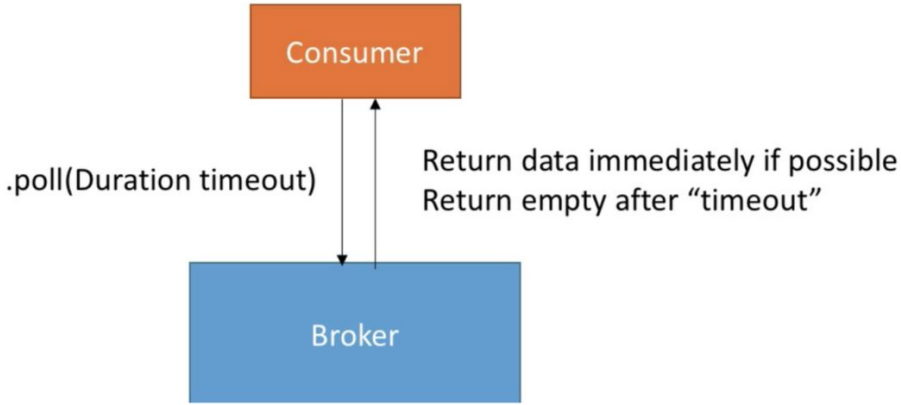
Delivery semantics

- At most once: offsets are committed as soon as the message is received. If the processing goes wrong, the message will be lost (it won't be read again).
- At least once: offsets are committed after the message is processed. If the processing goes wrong, the message will be read again. This can result in duplicate processing of messages. Make sure your processing is **idempotent** (i.e. processing again the messages won't impact your systems)
- Exactly once: Can be achieved for Kafka => Kafka workflows using Kafka Streams API. For Kafka => Sink workflows, use an idempotent consumer.

Bottom line: for most applications you should use at least once processing (we'll see in practice how to do it) and ensure your transformations / processing are idempotent

Consumer Poll Behavior

- Kafka Consumers have a "poll" model, while many other messaging bus in enterprises have a "push" model.
- This allows consumers to control where in the log they want to consume, how fast, and gives them the ability to replay events



Consumer Poll Behavior

- **Fetch.min.bytes (default 1):**
  - Controls how much data you want to pull at least on each request
  - Helps improving throughput and decreasing request number
  - At the cost of latency

- **Max.poll.records (default 500):**
  - Controls how many records to receive per poll request
  - Increase if your messages are very small and have a lot of available RAM
  - Good to monitor how many records are polled per request

Consumer Poll Behavior

- **Max.partitions.fetch.bytes (default 1MB):**
  - Maximum data returned by the broker per partition
  - If you read from 100 partitions, you'll need a lot of memory (RAM)
- **Fetch.max.bytes (default 50MB):**
  - Maximum data returned for each fetch request (covers multiple partitions)
  - The consumer performs multiple fetches in parallel
- Change these settings only if your consumer maxes out on throughput already

Consumer Offset Commits Strategies

- There are two most common patterns for committing offsets in a consumer application.

- **2 strategies:**
  - (easy) enable.auto.commit = true & synchronous processing of batches
  - (medium) enable.auto.commit = false & manual commit of offsets

Consumer Offset Commits Strategies

- enable.auto.commit = true & synchronous processing of batches

```
while(true){
    List<Records> batch = consumer.poll(Duration.ofMillis(100))
    doSomethingSynchronous(batch)
}
```

- With auto-commit, offsets will be committed automatically for you at regular interval **(auto.commit.interval.ms=5000 by default)** every-time you call .poll()
- If you don't use synchronous processing, you will be in "at-most-once" behavior because offsets will be committed before your data is processed

Consumer Offset Commits Strategies

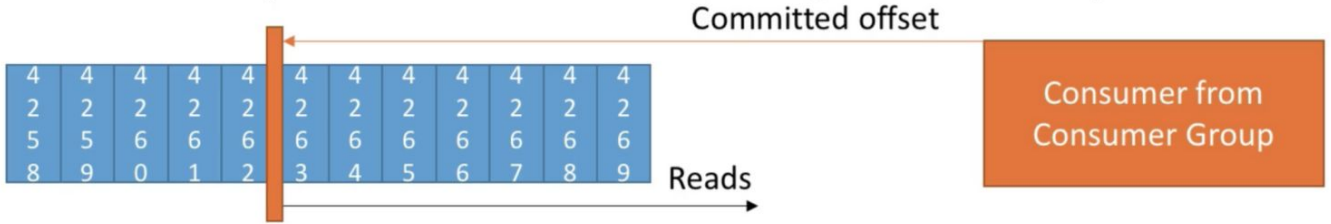
- enable.auto.commit = false & synchronous processing of batches

```
while(true){
    batch += consumer.poll(Duration.ofMillis(100))
    if isReady(batch) {
        doSomethingSynchronous(batch)
        consumer.commitSync();
    }
}
```

- You control when you commit offsets and what's the condition for committing them.
- *Example: accumulating records into a buffer and then flushing the buffer to a database + committing offsets then.*

Consumer Offset Reset Behavior

- A consumer is expected to read from a log continuously.



- But if your application has a bug, your consumer can be down
- If Kafka has a retention of 7 days, and your consumer is down for more than 7 days, the offsets are "invalid"

Consumer Offset Reset Behavior

- The behavior for the consumer is to then use:
  - auto.offset.reset=latest: will read from the end of the log
  - auto.offset.reset=earliest: will read from the start of the log
  - auto.offset.reset=none: will throw exception if no offset is found

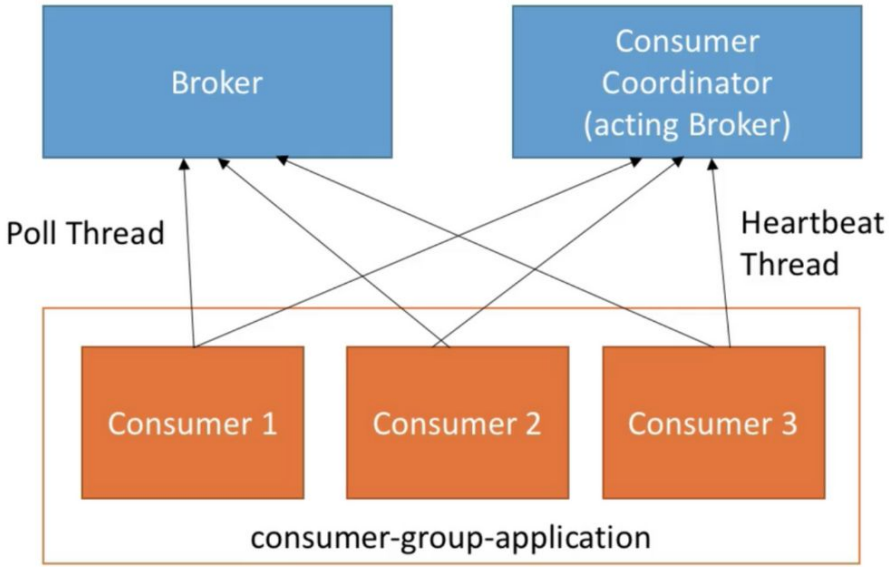
- Additionally, consumer offsets can be lost:
  - If a consumer hasn't read new data in 1 day (Kafka < 2.0)
  - If a consumer hasn't read new data in 7 days (Kafka >= 2.0)
- This can be controlled by the broker setting offset.retention.minutes

Replaying data for Consumers

- To replay data for a consumer group:
  - Take all the consumers from a specific group down
  - Use `kafka-consumer-groups` command to set offset to what you want
  - Restart consumers

- Bottom line:
  - Set proper data retention period & offset retention period
  - Ensure the auto offset reset behavior is the one you expect / want
  - Use replay capability in case of unexpected behaviour

Controlling Consumer Liveliness



Note: heartbeats and poll() are decoupled since Kafka 0.10.1

- Consumers in a Group talk to a Consumer Groups Coordinator
- To detect consumers that are "down", there is a "heartbeat" mechanism and a "poll" mechanism
- To avoid issues, consumers are encouraged to process data fast and poll often

Consumer Heartbeat Thread

45s in v3

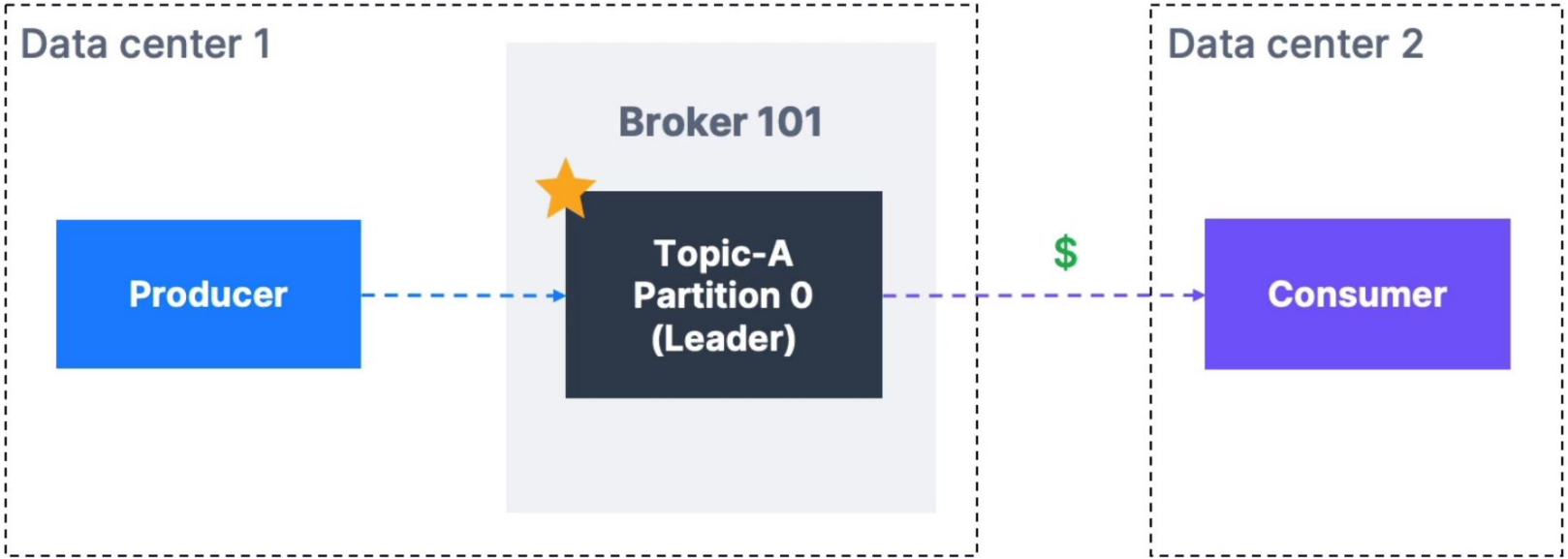
- **Session.timeout.ms (default 10 seconds):**
  - Heartbeats are sent periodically to the broker
  - If no heartbeat is sent during that period, the consumer is considered dead
  - Set even lower to faster consumer rebalances
- **Heartbeat.interval.ms (default 3 seconds):**
  - How often to send heartbeats
  - Usually set to 1/3rd of session.timeout.ms
- Take-away: This mechanism is used to detect a consumer application being down

Consumer Poll Thread

- **max.poll.interval.ms (default 5 minutes):**
  - Maximum amount of time between two .poll() calls before declaring the consumer dead
  - This is particularly relevant for Big Data frameworks like Spark in case the processing takes time

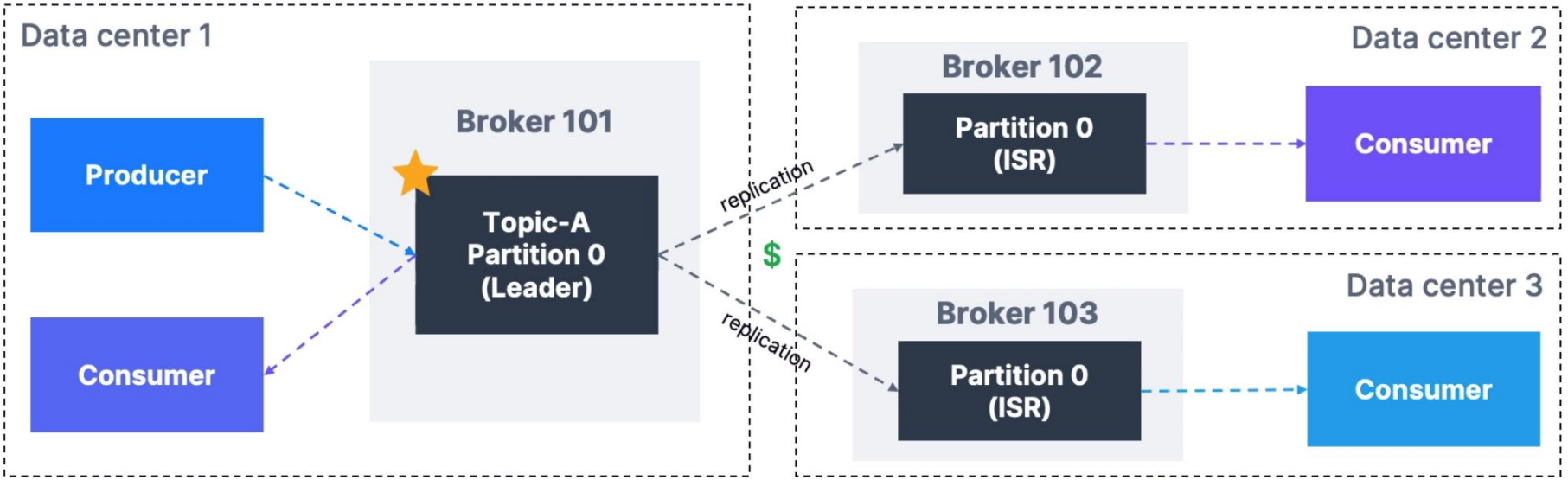
- Take-away: This mechanism is used to detect a data processing issue with the consumer

Default Consumer behavior with partition leader



Consumers Replica Fetching (v2.4+)

- Since Kafka 2.4, it is possible to configure consumers to read from **the closest replica**
- This may help improve latency, and also decrease network costs if using the cloud



Consumer Rack Awareness - How to set up

- Broker setting:
  - Must be version Kafka v2.4+
  - **`rack.id`** config must be set to the data centre ID (ex: AZ ID in AWS)
  - Example for AWS: AZ ID **`rack.id=usw2-az1`**
  - **`replica.selector.class`** must be set to `org.apache.kafka.common.replica.RackAwareReplicaSelector`
- Consumer client setting:
  - Set **`client.rack`** to the data centre ID the consumer is launched on