Why MongoDB

Architecture and Technology Overview

Safe Harbor Statement

This presentation contains “forward-looking statements” within the meaning of Section 27A of the Securities Act of 1933, as amended, and Section 21E of the Securities Exchange Act of 1934, as amended. Such forward-looking statements are subject to a number of risks, uncertainties, assumptions and other factors that could cause actual results and the timing of certain events to differ materially from future results expressed or implied by the forward-looking statements. Factors that could cause or contribute to such differences include, but are not limited to, those identified our filings with the Securities and Exchange Commission. You should not rely upon forward-looking statements as predictions of future events. Furthermore, such forward-looking statements speak only as of the date of this presentation.

In particular, the development, release, and timing of any features or functionality described for MongoDB products remains at MongoDB’s sole discretion. This information is merely intended to outline our general product direction and it should not be relied on in making a purchasing decision nor is this a commitment, promise or legal obligation to deliver any material, code, or functionality. Except as required by law, we undertake no obligation to update any forward-looking statements to reflect events or circumstances after the date of such statements.

MongoDB's

Intelligent Data Platform

is the best way of making your data simple

to **Organize, Use, & Enrich Anywhere**

Data Is Everywhere

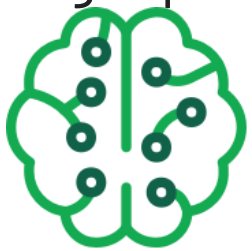
$4.88tn in eCommerce sales

by 2021

75bn connected IoT devices

by 2025

83% see AI as strategic priority



in 2019

20x faster, 120x lower latency

by 2021

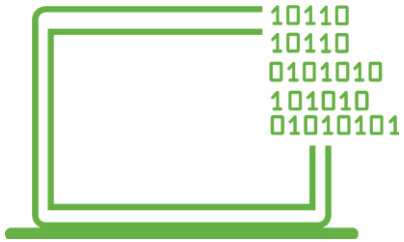
3.8bn smartphone users

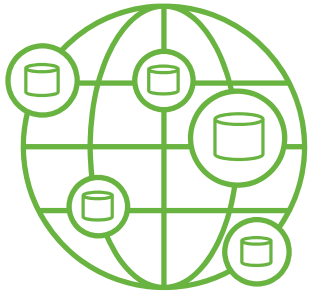
by 2021

$6tn+ in cyber-crime damage

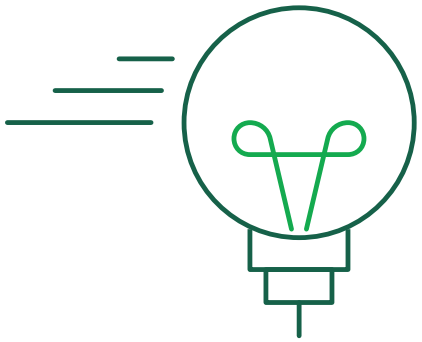
by 2021

Every company must become a **data & software company**





Demands from the Business



Innovate Faster

Data Driven

Lower Cost

Developers are key….

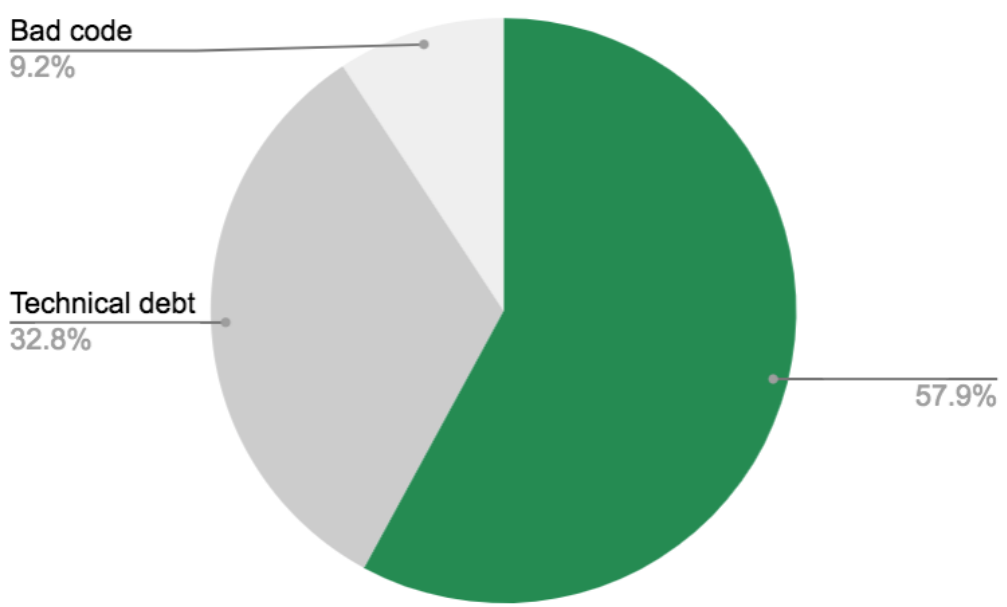
Developers are key….but

Access to developers is a bigger constraint on growth than access to capital, according to survey of thousands of C-level executives.

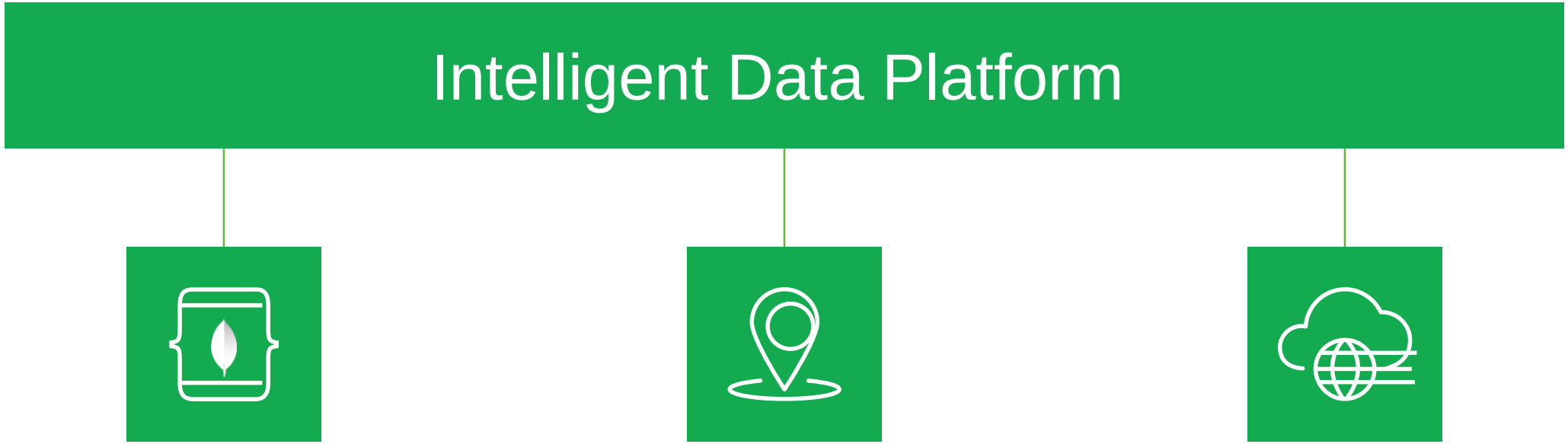
The Developer Coefficient, Stripe

stripe

Developers spend **42% of their work week** on maintenance issues and fixing bad code.



MongoDB: Built for Developer Productivity



The evolution of MongoDB

3.2

Document Validation
$lookup
Fast Failover
Simpler Scalability
Aggregation ++
Encryption At Rest
In-Memory Storage Engine
BI Connector
MongoDB Compass
APM Integration
Auto Index Builds
Backups to File System

3.4

Linearizable reads
Intra-cluster compression
Read only views
Log Redaction
Graph Processing
Decimal
Collations
Faceted Navigation
Aggregation ++
Auto-balancing ++
ARM, Power, zSeries
BI & Spark Connectors ++
Compass ++
LDAP Authorization
Encrypted Backups
Cloud Foundry Integration

3.6

Change Streams
Retryable Writes
Expressive Array Updates
Query Expressivity
Causal Consistency
Consistent Sharded Sec. Reads
Ops Manager ++
Query Advisor
Schema Validation
End to End Compression
IP Whitelisting
Default Bind to Localhost
Sessions
WiredTiger 1m+ Collections
Expressive $lookUp
R Driver
Atlas Cross Region Replication
Atlas Auto Storage Scaling

4.0

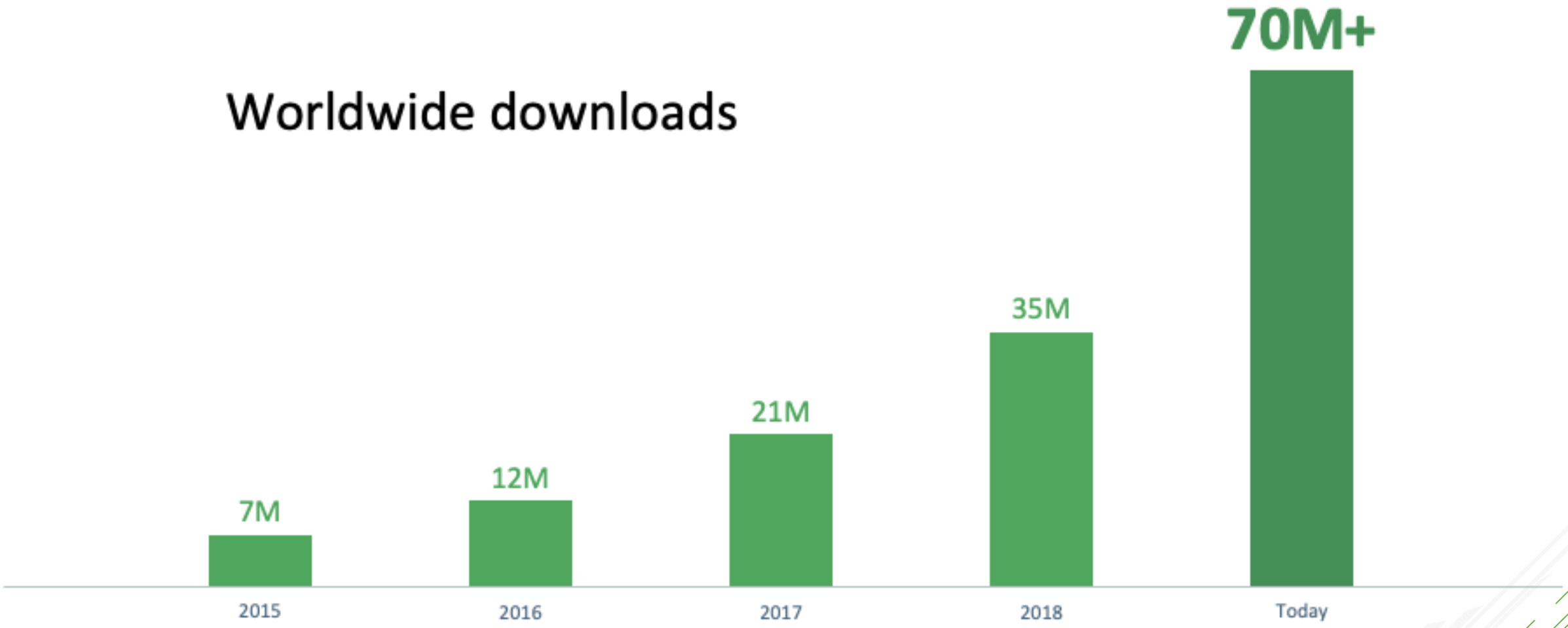
Replica Set Transactions
Atlas Global Clusters
Atlas HIPAA
Atlas LDAP
Atlas Audit
Atlas Enc. Storage Engine
Atlas Backup Snapshots
Type Conversions
40% Faster Shard Migrations
Snapshot Reads
Non-Blocking Sec. Reads
SHA-2
TLS 1.1+
Compass Agg Pipeline Builder
Compass Export to Code
Charts Beta
Free Monitoring Cloud Service
Ops Manager K8s Beta
MongoDB Stitch GA
MongoDB Mobile Beta

4.2

Distributed Transactions
Global Point in Time Reads
Large Transactions
Mutable Shard Key Values
Atlas Data Lake (Beta)
Atlas Auto Scaling (Beta)
Atlas Full-Text Search (Beta)
Atlas ISO Compliance
Atlas Service Broker
Field Level Encryption (Beta)
Multi-CAs & Online Rotation
On-Demand Materialized Views
Wildcard Indexes
Agg Pipeline ++
Expressive Updates
Apache Kafka Connector (Beta)
MongoDB Charts GA
Retryable Reads & Writes
New Index Builds
10x Faster stepDown
Storage Node Watchdog
Zstandard Compression
Ops Manager Headless Backup
Ops Manager K8s GA
Ops Manager Single Agent

mongoDB

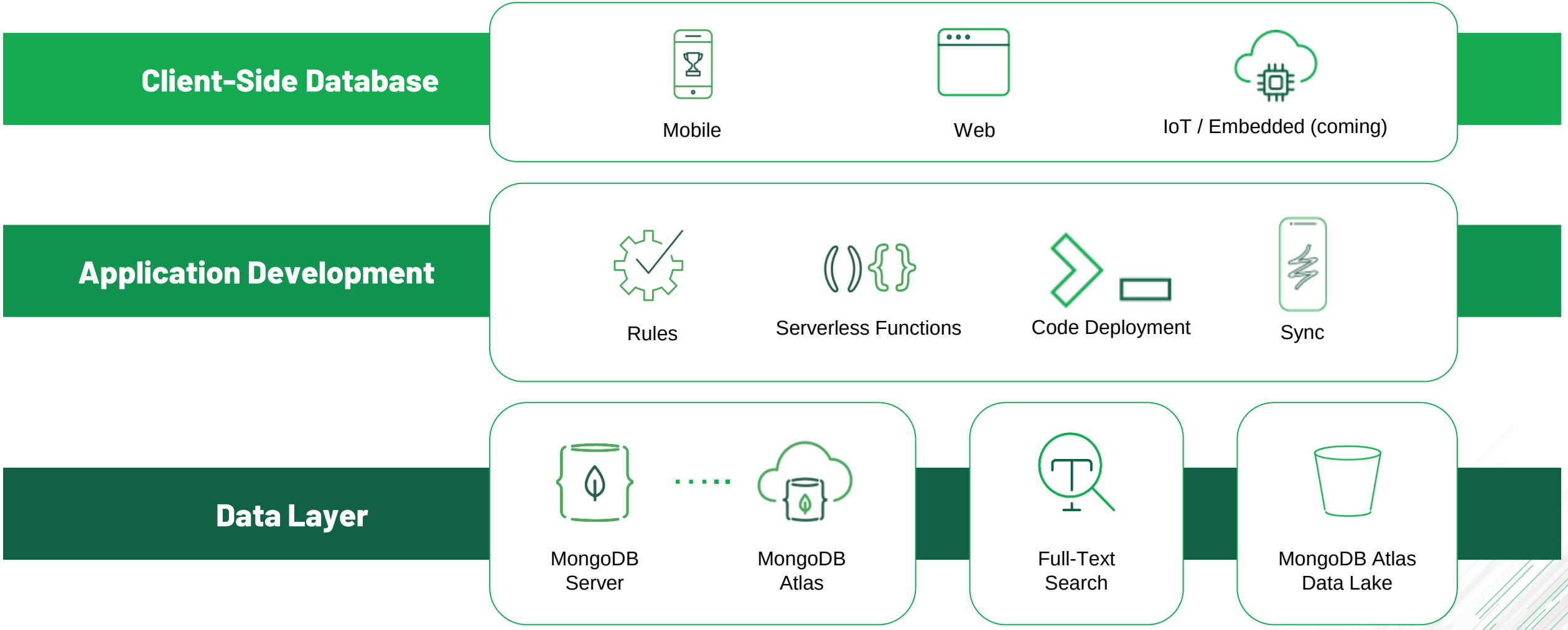
Increasingly the Database of Choice



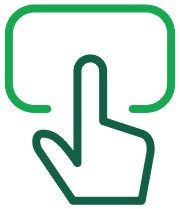
mongoDB

Stack Overflow Survey: Most Wanted Database

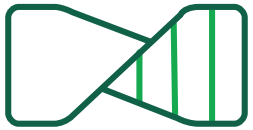
MongoDB Data Platform



Best way to work with data

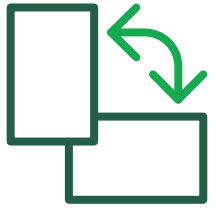


Easy:
Work with data in a natural, intuitive way, fully transactional

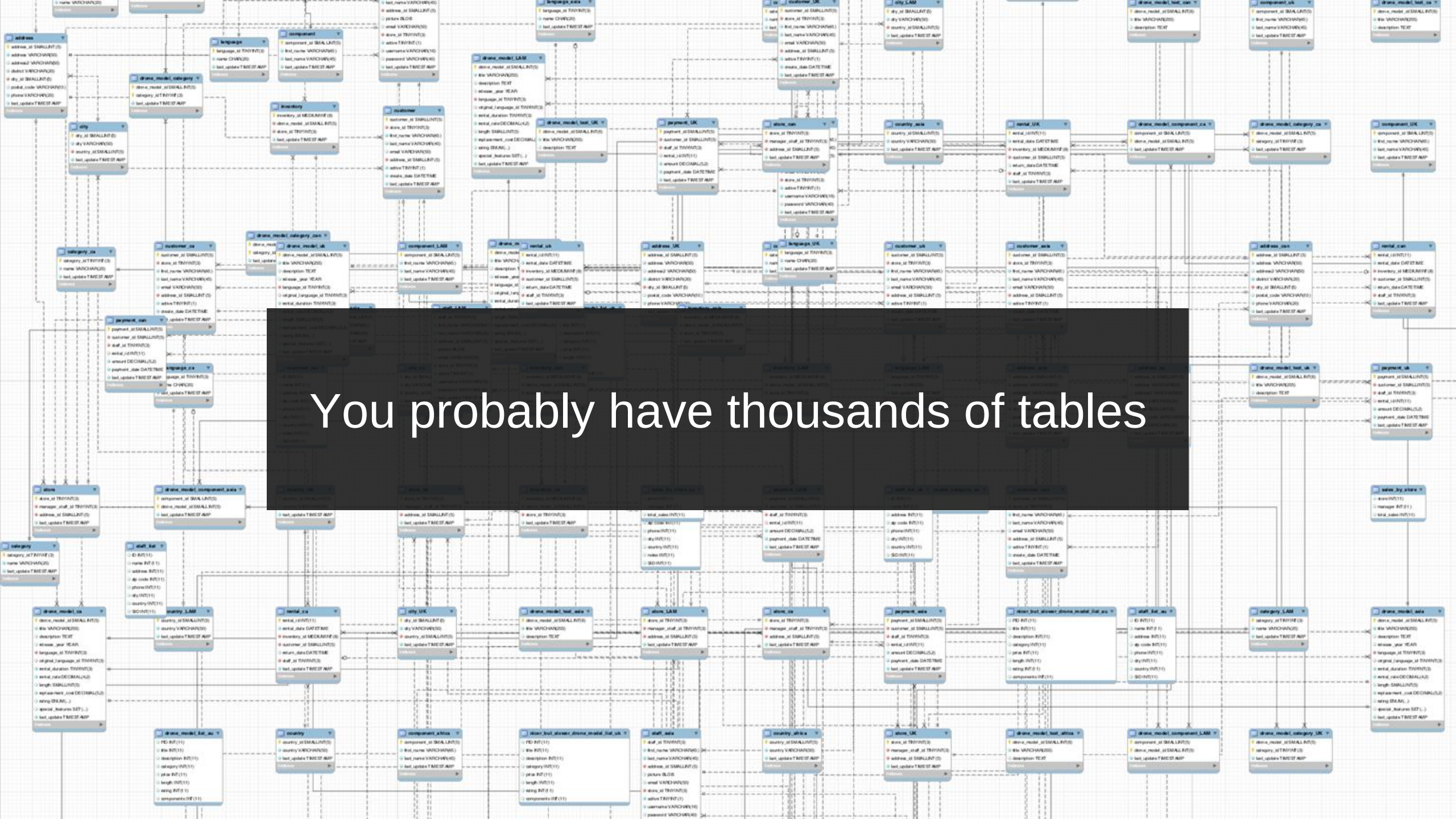


Flexible:
Adapt and make changes quickly

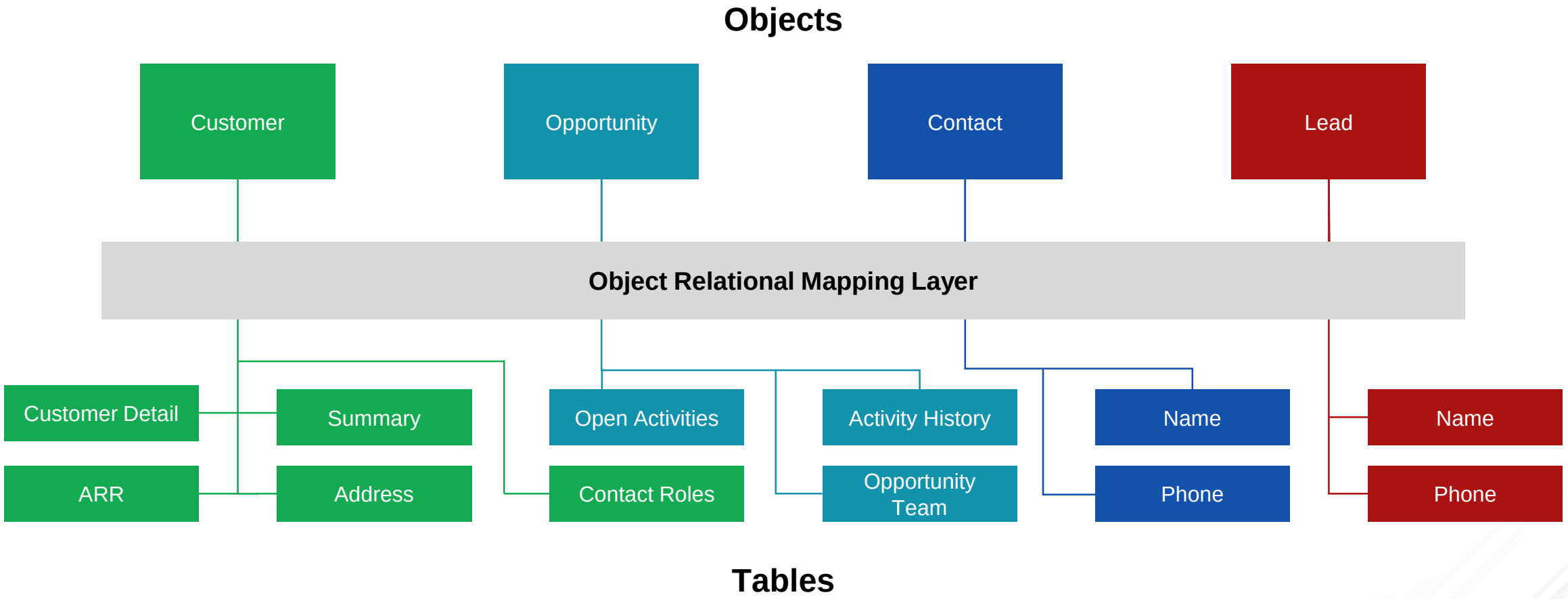
Fast:
Get great performance with less code



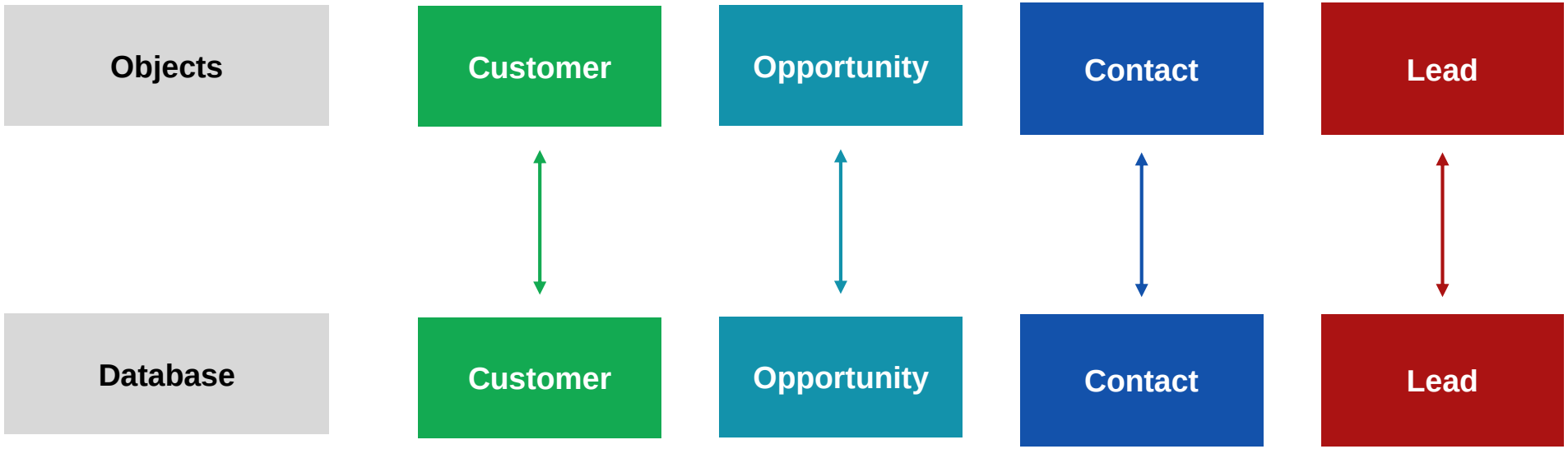
Versatile:
Supports a wide variety of data models and queries


You probably have thousands of tables

Go from this….

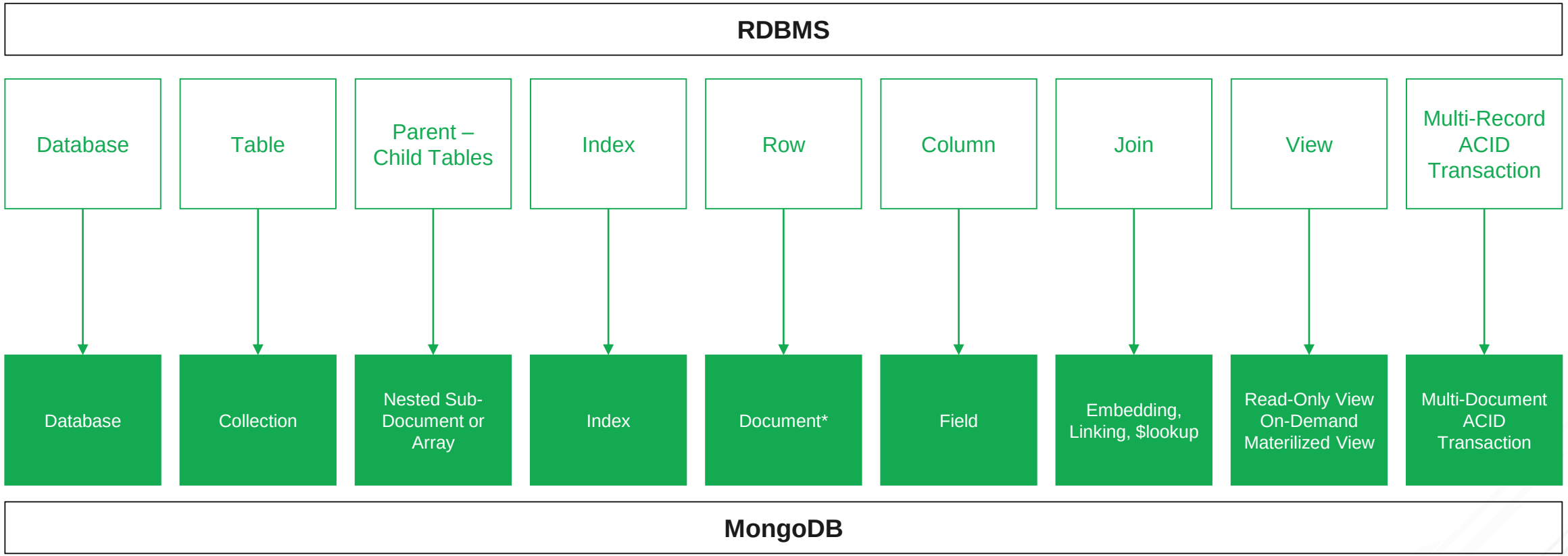

Objects
Customer
Opportunity
Contact
Lead
Object Relational Mapping Layer
Customer Detail
Summary
Open Activities
Activity History
Name
Name
ARR
Address
Contact Roles
Opportunity Team
Phone
Phone
Tables

To this: store objects directly…



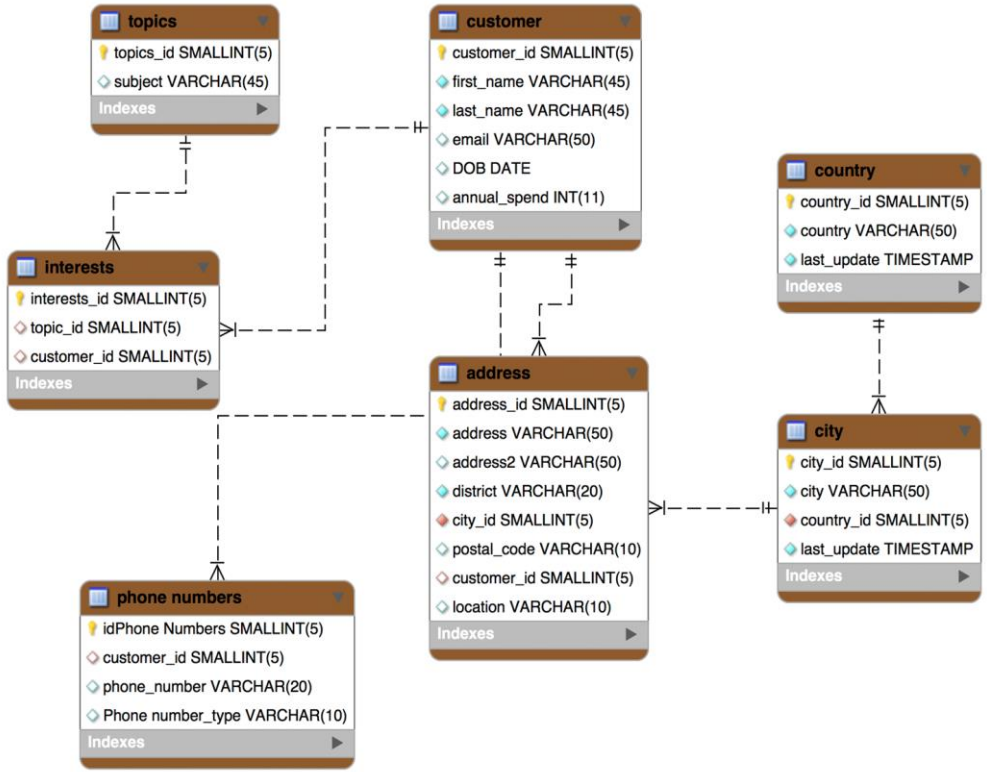
Some Terminology

A comparison



* Proper document schema design yields more entity data per document than found in a relational database row

Easy: Contrasting data models



Tabular (Relational) Data Model

Related data split across multiple records and tables

```
{
  "_id" : ObjectId("5ad88534e3632e1a35a58d00"),
  "name" : {
    "first" : "John",
    "last" : "Doe" },
  "address" : [
    { "location" : "work",
      "address" : {
        "street" : "16 Hatfields",
        "city" : "London",
        "postal_code" : "SE1 8DJ"},
      "geo" : { "type" : "Point", "coord" : [
          51.5065752,-0.109081]}},
+   {...}
  ],
  "phone" : [
    { "location" : "work",
      "number" : "+44-1234567890"},
+   {...}
  ],
  "dob" : ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund" : NumberDecimal("1292815.75")
}
```

Document Data Model

Related data contained in a single, rich document

Easy: Document data model

- Naturally maps to objects in code
- Represent data of any structure
- Strongly typed for ease of processing
  - Over 20 binary encoded JSON data types
- Access by idiomatic drivers in all major programming language

```
{
  "_id" : ObjectId("5ad88534e3632e1a35a58d00"),
  "name" : {
    "first" : "John",
    "last" : "Doe" },
  "address" : [
    { "location" : "work",
      "address" : {
        "street" : "16 Hatfields",
        "city" : "London",
        "postal_code" : "SE1 8DJ"},
      "geo" : { "type" : "Point", "coord" : [
          51.5065752,-0.109081]}},
+   {...}
  ],
  "phone" : [
    { "location" : "work",
      "number" : "+44-1234567890"},
+   {...}
  ],
  "dob" : ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund" : NumberDecimal("1292815.75")
}
```

Easy: Drivers and Frameworks

Drivers

Frameworks

Easy: Transactional Data Guarantees

Single & Multi-Document ACID Transactions

For many apps, single document transactions meet the majority of needs

```
  _id: 12345678
> name: Object
> address: Array
> phone: Array
  email: "john.doe.@mongodb.com"
  dob: 1966-07-30 01:00:00:000
v interests:Array
        0: "Cycling"
        1: "IoT"
```

Related data modeled in a single, rich document against which ACID guarantees are applied

Easy: MongoDB Multi-Document ACID Transactions

Just like relational transactions

- Multi-statement, familiar relational syntax
- Easy to add to any application
- Multiple documents in 1 or many collections and databases, across replica sets and sharded clusters

ACID guarantees

- Snapshot isolation, all or nothing execution
- No performance impact for non-transactional operations

Syntax

```
with client.start_session() as s:
    s.start_transaction()
    collection_one.insert_one(doc_one, session=s)
    collection_two.insert_one(doc_two, session=s)
    s.commit_transaction()
```

Natural for developers

- Idiomatic to the programming language
- Familiar to relational developers
- Simple

Syntax

```
try (ClientSession clientSession = client.startSession()) {
            clientSession.startTransaction();
            collection.insertOne(clientSession, docOne);
            collection.insertOne(clientSession, docTwo);
            clientSession.commitTransaction();
        }
```

Comparing Syntax with Relational

MySQL

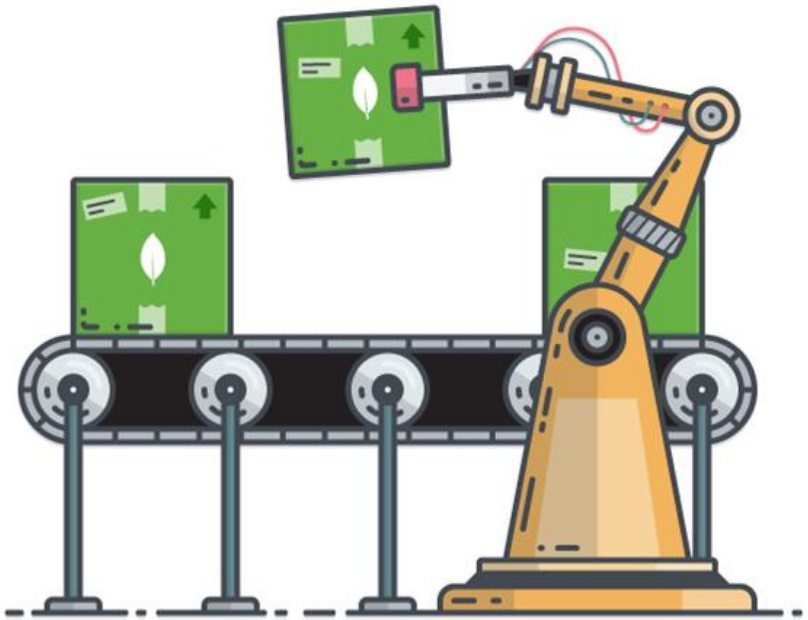
```
db.start_transaction()
cursor.execute(orderInsert, orderData)
cursor.execute(stockUpdate, stockData)
db.commit()
```

mongoDB

```
s.start_transaction()
orders.insert_one(order, session=s)
stock.update_one(item, stockUpdate, session=s)
s.commit_transaction()
```

Transactions Best Practises

- Use MongoDB 4.2 drivers
- Transactions targeting a single shard will be faster than those spanning multiple shards
- Transactions automatically abort after 60 seconds – TUNABLE. Aborted transactions are fully rolled back by the database
- Best practice: No more than 1,000 documents modified in a single transaction
- No arbiters when running Distributed Transactions
- Chunk migrations will block behind running transactions



The Journey to Transactions

Major engineering investment over 3+ years touching every part of the server and drivers

- Storage layer
- Replication consensus protocol
- Sharding architecture
- Consistency and durability guarantees
- Global logical clock
- Cluster metadata management
- Exposed to drivers through API enhancements

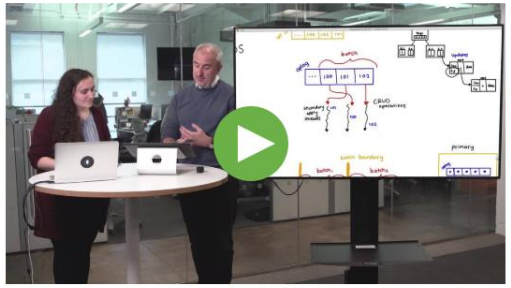
Major Engineering Projects

MongoDB 3.0	MongoDB 3.2	MongoDB 3.4	MongoDB 3.6	MongoDB 4.0	MongoDB 4.2
New Storage engine (WiredTiger)	Enhanced replication protocol: stricter consistency & durability	Shard membership awareness	Consistent secondary reads in sharded clusters	Replica Set Transactions	Global Transactions
	WiredTiger default storage engine		Logical sessions	Make catalog timestamp-aware	Oplog applier prepare support
	Config server manageability improvements		Retryable writes	Snapshot reads	Distributed commit protocol
	Read concern “majority”		Causal Consistency	Recoverable rollback via WT checkpoints	Global point-in-time reads
			Cluster-wide logical clock	Recover to a timestamp	More extensive WiredTiger repair
			Storage API to changes to use timestamps	Sharded catalog improvements	Transaction manager
			Read concern majority feature always available		
			Collection catalog versioning		
			UUIDs in sharding		
			Fast in-place updates to large documents in WT		

Learn More

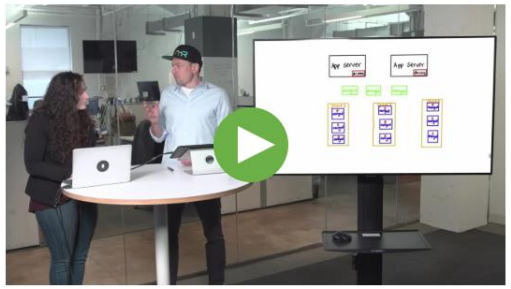
Engineering Chalk and Talks

Hear from the engineers who implemented transactions in MongoDB



WiredTiger timestamps: enforcing correctness in operation ordering across the distributed storage layer. Hear from Dr. Michael Cahill, Director of Engineering for storage at MongoDB

Logical sessions: coordinating operations across a distributed cluster, presented by Jason Carey, Lead Engineer for platforms at MongoDB



Global logical clock: establishing a global snapshot of

Safe secondary reads: providing consistent reads against

Transactions Page

- Chalk and talks
- Code snippets & documentation
- Blogs

mongoDB

Flexible: Adapt to change

```
{
  "_id" : ObjectId("5ad88534e3632e1a35a58d00"),
  "name" : {
    "first" : "John",
    "last" : "Doe" },
  "address" : [
    { "location" : "work",
      "address" : {
        "street" : "16 Hatfields",
        "city" : "London",
        "postal_code" : "SE1 8DJ"},
      "geo" : { "type" : "Point", "coord" : [
        51.5065752,-0.109081]}},
+   {...}
  ],
"dob" : ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund" : NumberDecimal("1292815.75")

}
```

```
{
  "_id" : ObjectId("5ad88534e3632e1a35a58d00"),
  "name" : {
    "first" : "John",
    "last" : "Doe" },
  "address" : [
    { "location" : "work",
      "address" : {
        "street" : "16 Hatfields",
        "city" : "London",
        "postal_code" : "SE1 8DJ"},
      "geo" : { "type" : "Point", "coord" : [
        51.5065752,-0.109081]}},
+   {...}
  ],
  "phone" : [
    { "location" : "work",
      "number" : "+44-1234567890"},
+   {...}
  ],
  "dob" : ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund" : NumberDecimal("1292815.75")
}
```

Add new fields dynamically at runtime

Stepping through the code: Version 1

For each customer store:

- ID
- Name

Version 1: Initial efforts for both technologies

SQL

```
DDL: create table contact ( … )

init()
{
  contactInsertStmt = connection.prepareStatement
  (“insert into contact ( id, name ) values ( ?,? )”);
  fetchStmt = connection.prepareStatement
  (“select id, name from contact where id = ?”);
}

save(Map m)
{
    contactInsertStmt.setStr
    contactInsertStmt.setStr
    contactInsertStmt.execut
}

Map fetch(String id)
{
    Map m = null;
    fetchStmt.setString(1, id);
    rs = fetchStmt.execute();
    if(rs.next()) {
              m = new HashMap();
              m.put(“id”, rs.getString(1));
              m.put(“name”, rs.getString(2));
    }
    return m;
}
```

MongoDB

```
DDL:  none

save(Map m)

Map fetch(String id)
{
    Map m = null;
    DBObject dbo = new BasicDBObject();
    dbo.put(“id”, id);
    c = collection.find(dbo);
    if(c.hasNext()) }
              m = (Map) c.next();
    }
    return m;
}
```

Let’s assume for argument’s sake that both approaches take the same amount of time

Version 1: Initial efforts for both technologies

SQL

```
DDL: create table contact ( … )

init()
{
  contactInsertStmt = connection.prepareStatement
  (“insert into contact ( id, name ) values ( ?,? )”);
  fetchStmt = connection.prepareStatement
  (“select id, name from contact where id = ?”);
}

save(Map m)
{
    contactInsertStmt.setStr
    contactInsertStmt.setStr
    contactInsertStmt.execut
}

Map fetch(String id)
{
    Map m = null;
    fetchStmt.setString(1, id);
    rs = fetchStmt.execute();
    if(rs.next()) {
              m = new HashMap();
              m.put(“id”, rs.getString(1));
              m.put(“name”, rs.getString(2));
    }
    return m;
}
```

MongoDB

```
DDL:  none

save(Map m)

Map fetch(String id)
{
    Map m = null;
    DBObject dbo = new BasicDBObject();
    dbo.put(“id”, id);
    c = collection.find(dbo);
    if(c.hasNext()) }
              m = (Map) c.next();
    }
    return m;
}
```

Let’s assume for argument’s sake that both approaches take the same amount of time

Version 2

New feature!

For each customer store:

- ID
- Name
- Title
- ContactDate

} ***New Requirements***

Version 2: Add simple fields

```
m.put("name", "John");
m.put("id", "K1");
m.put("title", "Mr.");
m.put("contactDate", new
Date(2018, 05, 1));
```

- In the data access layer, the map is easy to change
- …but now we have to change our persistence code

Brace yourself…

SQL Version 2 (changes in bold)

```
DDL:  alter table contact add title varchar(8);
      alter table contact add contactDate date;

init()
{
  contactInsertStmt = connection.prepareStatement
  ("insert into contact ( id, name, title, contactdate ) values ( ?,?,?,? )");
  fetchStmt = connection.prepareStatement
  ("select id, name, title, contactdate from contact where id = ?");
}

save(Map m)
{
    contactInsertStmt.setString(1, m.get("id"));
    contactInsertStmt.setString(2, m.get("name"));
    contactInsertStmt.setString(3, m.get("title"));
    contactInsertStmt.setDate(4, m.get("contactDate"));
    contactInsertStmt.execute();
}

Map fetch(String id)
{
    Map m = null;
    fetchStmt.setString(1, id);
    rs = fetchStmt.execute();
    if(rs.next()) {
            m = new HashMap();
            m.put("id", rs.getString(1));
            m.put("name", rs.getString(2));
            m.put("title", rs.getString(3));
            m.put("contactDate", rs.getDate(4));
    }
    return m;
}
```

Consequences:

1. Code release schedule linked to database upgrade (new code cannot run on old schema)
2. DAL closely tied to RDBMS columns
3. Issues with case sensitivity starting to creep in (many RDBMS are case insensitive for column names, but code is case sensitive)
4. Changes require careful modifications in 4 places

1. Beginning of **technical debt**

SQL Version 2 (changes in bold)

```
DDL:  alter table contact add title varchar(8);
      alter table contact add contactDate date;

init()
{
  contactInsertStmt = connection.prepareStatement
  ("insert into contact ( id, name, title, contactdate ) values (
?,?,?,? )");
  fetchStmt = connection.prepareStatement
  ("select id, name, title, contactdate from contact where id = ?");
}

save(Map m)
{
    contactInsertStmt.setString(1, m.get("id"));
    contactInsertStmt.setString(2, m.get("name"));
    contactInsertStmt.setString(3, m.get("title"));
    contactInsertStmt.setDate(4, m.get("contactDate"));
    contactInsertStmt.execute();
}

Map fetch(String id)
{
    Map m = null;
    fetchStmt.setString(1, id);
    rs = fetchStmt.execute();
    if(rs.next()) {
            m = new HashMap();
            m.put("id", rs.getString(1));
            m.put("name", rs.getString(2));
            m.put("title", rs.getString(3));
            m.put("contactDate", rs.getDate(4));
    }
    return m;
```

Consequences:

1. Code release schedule linked to database upgrade (new code cannot run on old schema)
2. DAL closely tied to RDBMS columns
3. Issues with case sensitivity starting to creep in (many RDBMS are case insensitive for column names, but code is case sensitive)
4. Changes require careful modifications in 4 places

1. Beginning of **technical debt**

MongoDB Version 2

```
save(Map m)
{
    collection.insert(m);
}

Map fetch(String id)
{
    Map m = null;
    DBObject dbo = new BasicDBObject();
    dbo.put("id", id);
    c = collection.find(dbo);
    if(c.hasNext()) }
          m = (Map) c.next();
    }
    return m;
}
```

NO CHANGE

Advantages:

1. **Zero time and money spent on overhead code**
2. Code and database not physically linked
3. New material with more fields can be added into existing collections; backfill is optional
4. Names of fields in database precisely match key names in code layer and directly match on name, not indirectly via positional offset
1. No technical debt is created

Version 3

New feature!

For each customer store:

- ID
- Name
- Title
- HireDate
- Zero to n phone numbers

} ***New Requirement***

Version 3: Add list of phone numbers

```
m.put("name", "John");
m.put("id", "K1");
m.put("title", "Mr.");
m.put("contactDate", new Date(2018, 05, 1));

n1.put("type", "work");
n1.put("number", "1-800-555-1212"));
list.add(n1);
n2.put("type", "home"));
n2.put("number", "1-866-444-3131"));
list.add(n2);
m.put("phones", list);
```

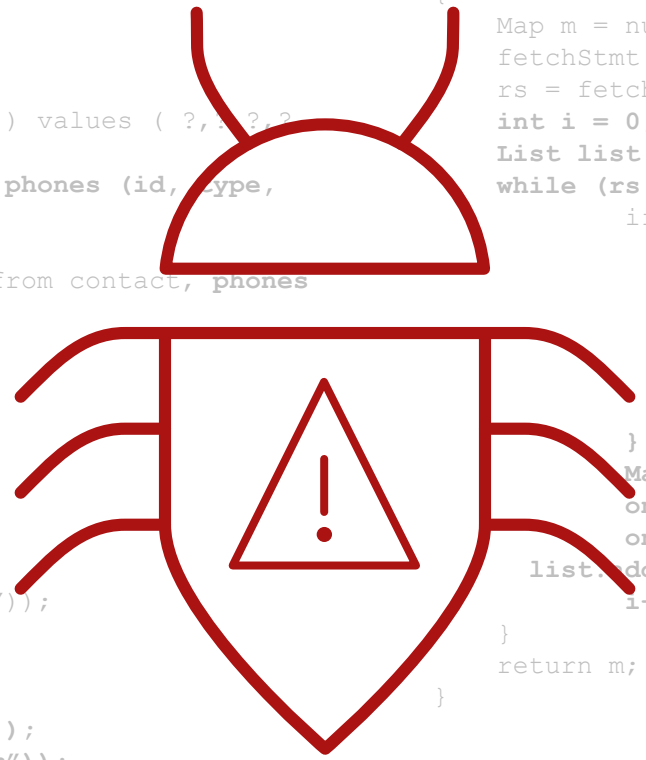
- It was still pretty easy to add this data to the map
- ... but meanwhile, in the persistence code …

REALLY brace yourself…

SQL Version 3

```
DDL:  create table phones ( … )

init()
{
  contactInsertStmt = connection.prepareStatement
  (“insert into contact ( id, name, title, hiredate ) values ( ?, ?, ?
)”);
  c2stmt = connection.prepareStatement(“insert into phones (id, type,
number) values (?, ?, ?)”;
  fetchStmt = connection.prepareStatement
  (“select id, name, title, hiredate, type, number from contact, phones
where phones.id = contact.id and   contact.id = ?”);
}

save(Map m)
{
    startTrans();
    contactInsertStmt.setString(1, m.get(“id”));
    contactInsertStmt.setString(2, m.get(“name”));
    contactInsertStmt.setString(3, m.get(“title”));
    contactInsertStmt.setDate(4, m.get(“contactDate”));

    for(Map onePhone : m.get(“phones”)) {
            c2stmt.setString(1, m.get(“id”));
            c2stmt.setString(2, onePhone.get(“type”));
            c2stmt.setString(3, onePhone.get(“number”));
            c2stmt.execute();
    }
    contactInsertStmt.execute();
    endTrans();
}
```

```
Map fetch(String id)
{
  Map m = null;
  fetchStmt.setString(1, id);
  rs = fetchStmt.execute();
  int i = 0;
  List list = new ArrayList();
  while (rs.next()) {
        if(i == 0) {
              m = new HashMap();
              m.put(“id”, rs.getString(1));
              m.put(“name”, rs.getString(2));
              m.put(“title”, rs.getString(3));
              m.put(“contactDate”, rs.getDate(4));
              m.put(“phones”, list);
        }
        Map onePhone = new HashMap();
        onePhone.put(“type”, rs.getString(5));
        onePhone.put(“number”, rs.getString(6));
      list.add(onePhone);
        i++;
  }
  return m;
}
```



This took time and money

SQL Version 3: With Bugs Removed

```
init()
{
  contactInsertStmt = connection.prepareStatement
  (“insert into contact ( id, name, title, contactdate ) values (
?,?,?,? )”);
  c2stmt = connection.prepareStatement(“insert into phones (id,
type, number) values (?, ?, ?)”;
  fetchStmt = connection.prepareStatement
  (“select A.id, A.name, A.title, A.hiredate, B.type, B.number from
contact A left outer join phones B on (A.id = B. id) where A.id =
?”);
}
```

```
  while (rs.next()) {
            if(i == 0) {
                // …
            }
            String s = rs.getString(5);
            if(s != null) {
                Map onePhone = new HashMap();
                onePhone.put(“type”, s);
                onePhone.put(“number”, rs.getString(6));
           list.add(onePhone);
        }
     }
```

Some contacts that have no phone number → outer join.

But this ALSO means we have to change the unwind logic

This took more time and money!

MongoDB Version 3

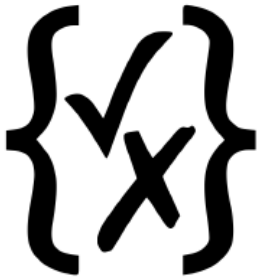
```
save(Map m)
{
    collection.insert(m);
}

Map fetch(String id)
{
    Map m = null;
    DBObject dbo = new BasicDBObject();
    dbo.put("id", id);
    c = collection.find(dbo);
    if(c.hasNext()) }
          m = (Map) c.next();
    }
    return m;
}
```

NO CHANGE

Advantages:

1. **Zero time and money spent on overhead code**
2. Code and database not physically linked
3. New material with more fields can be added into existing collections; backfill is optional
4. Names of fields in database precisely match key names in code layer and directly match on name, not indirectly via positional offset
1. No technical debt is created

Flexible: Govern



JSON Schema

Enforces strict schema structure over a complete collection for data governance & quality

- Builds on document validation introduced by restricting new content that can be added to a document
- Enforces presence, type, and values for document content, including nested array
- Simplifies application logic

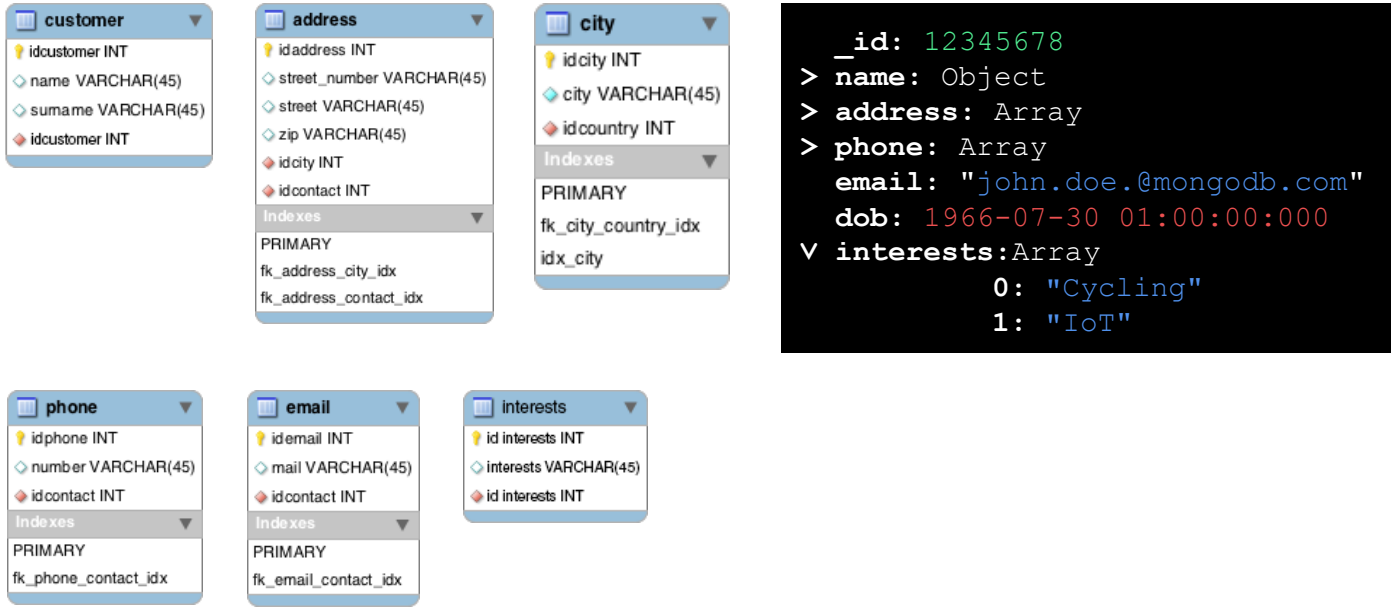
Tunable: enforce document structure, log warnings, or allow complete schema flexibility

Queryable: identify all existing documents that do not comply

Fast: To work with data

Compared to storing data across multiple tables, a single document data structure:

- Presents a single place for the database to read and write data
- Denormalized data eliminates JOINs for most operational queries
- Simplifies query development and optimization



SQL needed to insert a customer

```
import mysql.connector
from mysql.connector import errorcode

def addUser(connection, user):
    cursor = connection.cursor()

    customerInsert = (
        "INSERT INTO customer (first_name, last_name,
            email, "
        "DOB, annual_spend) VALUES "
        "(%(first)s, %(last)s, %(email)s, %(dob)s, %(spend)
            s)")

    customerData = {
        'first': user['name']['first'],
        'last': user['name']['second'],
        'email': user['email'],
        'dob': user['dob'],
        'spend': user['annualSpend']
    }

    cursor.execute(customerInsert, customerData)
    customerId = cursor.lastrowid

    cityQuery = ("SELECT city_id FROM city WHERE city = %(
        city)s")
    for address in user['address']:
        cursor.execute(cityQuery, {'city': address['city']}
            )
        city_id = cursor.fetchone()[0]

        addressInsert = (
            "INSERT INTO address (address, address2,
                district, "
            "city_id, postal_code, customer_id, location) "
            "VALUES (%(add)s, %(add2)s, %(dist)s, %(city)s
                , %(post)s, %(cust)s, %(loc)s)")

        addressData = {
            'add': address['number'],
            'add2': address['street'],
            'dist': address['state'],
            'city': city_id,
            'post': address['postalCode'],
            'cust': customerId,
            'loc':  address['location']
        }

        cursor.execute(addressInsert, addressData)

    topicQuery = ("SELECT topics_id FROM topics WHERE
        subject = %(subj)s")
    interestInsert = (
        "INSERT into interests (topic_id, customer_id) "
        "VALUES (%(topic)s, %(cust)s)")

    for interest in user['interests']:
        topicId = 0
        topicData = {
            'subj': interest['interest']
        }

        cursor.execute(topicQuery, topicData)
        row = cursor.fetchone()
        if row is None:
            topicInsert = ("INSERT INTO topics (subject)
                VALUES (%(subj)s)")
            cursor.execute(topicInsert, topicData)
            topicId = cursor.lastrowid
        else:
            topicId = row[0]

        interestData = {
            'topic': topicId,
            'cust': customerId
        }
        cursor.execute(interestInsert, interestData)

    phoneInsert = (
        "INSERT INTO `phone numbers` (customer_id,
            phone_number, `Phone number_type`) "
        "VALUES (%(cust)s, %(num)s, %(type)s)")
    for phoneNumber in user['phone']:
        phoneData = {
            'cust': customerId,
            'num': phoneNumber['number'],
            'type': phoneNumber['location']
        }
        cursor.execute(phoneInsert, phoneData)

    connection.commit()
    cursor.close()
    return customerId
```

Code snippet:
https://git.io/vpnxX

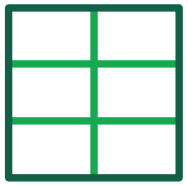
Fast: MongoDB requires just 2 lines of code

```
def addUser(database, user):
  return database.customers.insert_one(user).inserted_id
```

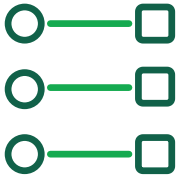
Code snippet: https://git.io/vpnpG

Versatile: Multiple data models, rich query functionality

JSON Documents



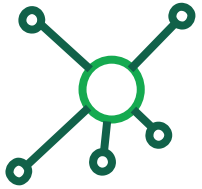
Tabular



Key-Value

Text

Geospatial



Graph

Rich Queries

Point | Range | Geospatial | Faceted Search | Aggregations | JOINs | Graph Traversals

Versatile: Rich query functionality

Expressive Queries	• Find anyone with phone # “1-212…” • Check if the person with number “555…” is on the “do not call” list
Geospatial	• Find the best offer for the customer at geo coordinates of 42nd St. and 6th Ave
Text Search	• Find all tweets that mention the firm within the last 2 days
Aggregation	• Count and sort number of customers by city, compute min, max, and average spend
Native Binary JSON Support	• Add an additional phone number to Mark Smith’s record without rewriting the document • Update just 2 phone numbers out of 10 • Sort on the modified date
JOIN ($lookup)	• Query for all San Francisco residences, lookup their transactions, and sum the amount by person
Graph Queries ($graphLookup)	• Query for all people within 3 degrees of separation from Mark

MongoDB

```
{       customer_id : 1,
        first_name : "Mark",
        last_name : "Smith",
        city : "San Francisco",
        phones: [       {
                        number : "1-212-777-1212",
                        type : "work"
        },
        {
                        number : "1-212-777-1213",
                        type : "cell"
        }]
……...
```

Fully Indexable

Fully featured secondary indexes

Index Types

- Primary Index
  - Every Collection has a primary key index
- Compound Index
  - Index against multiple keys in the document
- MultiKey Index
  - Index into arrays
- Wildcard Index
  - Auto-index all matching fields, sub-documents & arrays
- Text Indexes
  - Support for text searches
- GeoSpatial Indexes
  - 2d & 2dSphere indexes for spatial geometries
- Hashed Indexes
  - Hashed based values for sharding

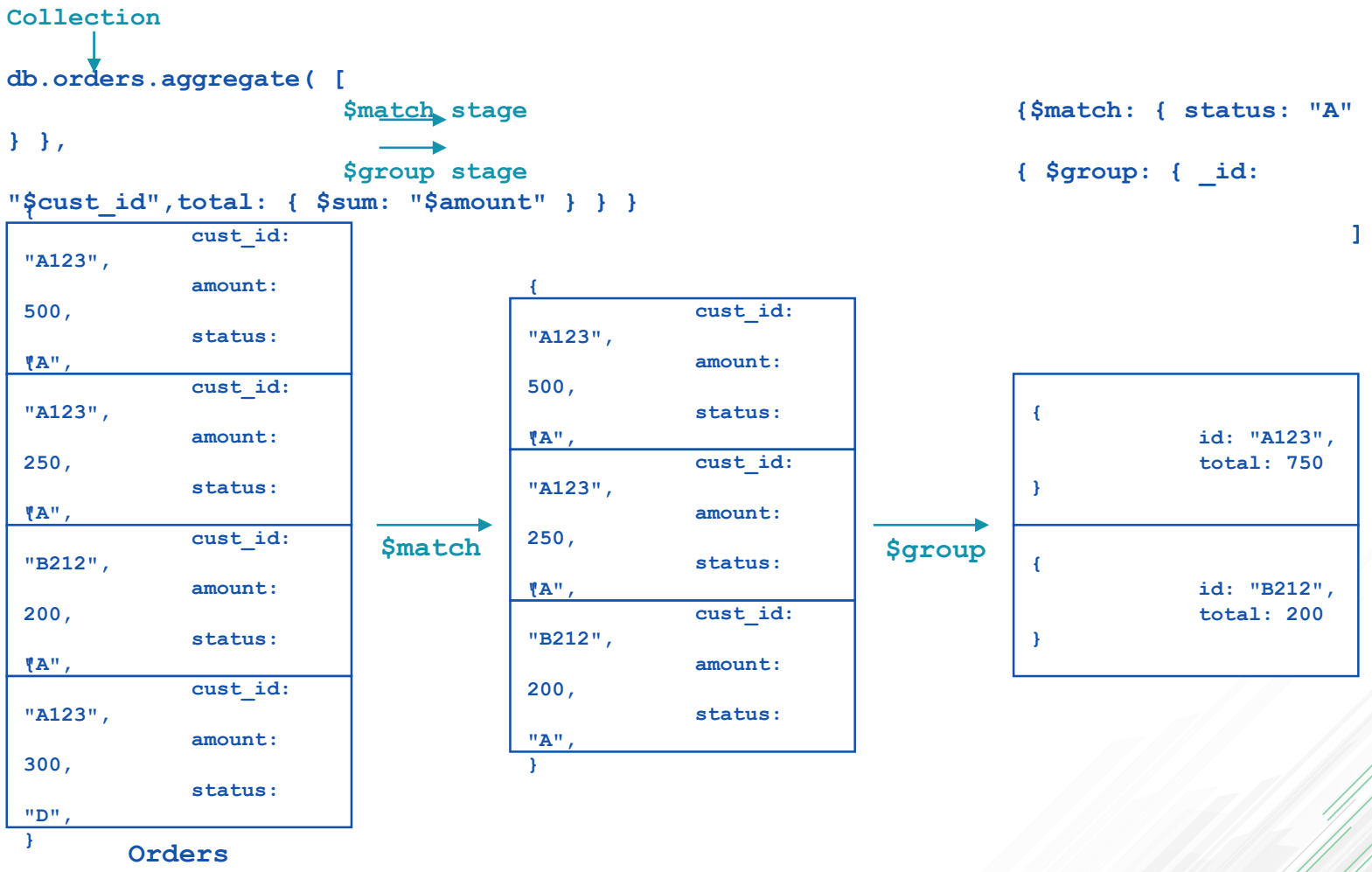
Index Features

- TTL Indexes
  - Single Field indexes, when expired delete the document
- Unique Indexes
  - Ensures value is not duplicated
- Partial Indexes
  - Expression based indexes, allowing indexes on subsets of data
- Case Insensitive Indexes
- supports text search using case insensitive search
- Sparse Indexes
  - Only index documents which have the given field

Aggregations

Advanced data processing pipeline for transformations and analytics

- Multiple stages
- Similar to a unix pipe
  - Build complex pipeline by chaining commands together
- Rich Expressions



Aggregation Features

A feature rich analytical framework

Pipeline Stages

- $match
- $group
- $facet
- $geoNear
- $graphLookup
- $lookup
- $merge
- $project
- $sort
- $unwind
- ...and more

Operators

- Mathematical
  - $add, $abs, $substract, $multiply, $divide, $log, $log10, $stdDevPop, $stdDevSam, $avg, $sqrt, $pow, $sum, $zip, $convert, $round, etc
- Array
  - $push, $reduce, $reverseArray, $addToSet, $arrayElemAt, $slice, etc.
- Conditionals
  - $and, $or, $eq, $lt, $lte, $gt, $gte, $cmp, $cond, $switch, $in, etc
- Date
  - $dateFromParts, $dateToParts, $dateFromString, $dateToString, $dayOfMonth, $isoWeek, $minute, $month, $year, etc.
- String
  - $toUpper, $toLower, $substr, $strcasecmp, $concat, $split, etc.
- Laterals
  - $exp, $let, $literal, $map, $type, etc
- Regex
  - $regexfFnd, $regexMatch, etc
- Trigonometry
  - $sin, $cos, $degreesToRadians, etc

Compared to SQL JOINs and aggregation

```
SELECT
    city,
    SUM(annual_spend) Total_Spend,
    AVG(annual_spend) Average_Spend,
    MAX(annual_spend) Max_Spend,
    COUNT(annual_spend) customers
FROM (
    SELECT t1.city, customer.annual_spend
    FROM customer
    LEFT JOIN (
        SELECT address.address_id, city.city,
               address.customer_id, address.location
        FROM address LEFT JOIN city
        ON address.city_id = city.city_id
        ) AS t1
    ON
    (customer.customer_id = t1.customer_id AND
             t1.location = "home")
) AS t2
GROUP BY city;
```

SQL queries have a nested structure

Understanding the outer layers requires understanding the inner ones

So SQL has to be read “inside-out”

Versatile: Complex queries fast to create, optimize, & maintain

```
db.customers.aggregate([
        {
                $unwind: "$address",
        },
        {
                $match: {"address.location": "home"}
        },
        {
                $group: {
                        _id: "$address.city",
                        totalSpend: {$sum: "$annualSpend"},
                        averageSpend: {$avg: "$annualSpend"},
                        maximumSpend: {$max: "$annualSpend"},
                        customers: {$sum: 1}
                }
        }
])
```

These “phases” are distinct and easy to understand

They can be thought about in order… no nesting.

MongoDB’s aggregation framework has the flexibility you need to get value from your data, but without the complexity and fragility of SQL

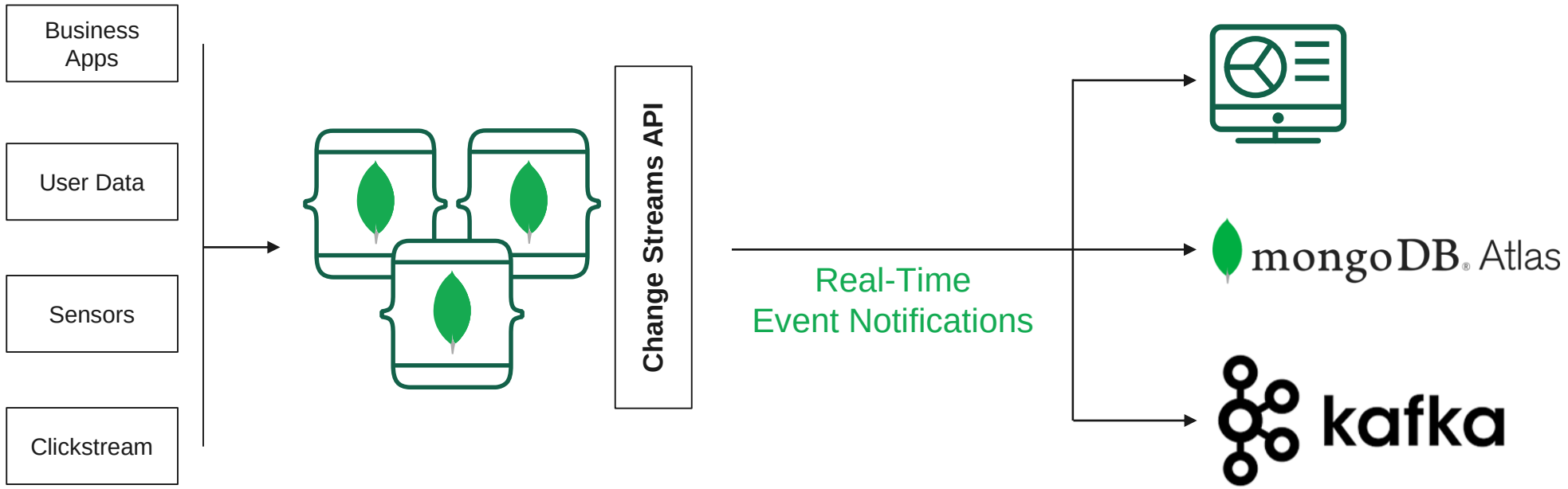
Versatile: On-Demand Materialized Views

- Faster insights on your data: pre-compute and store results of common analytics queries
- With $merge stage aggregation pipeline outputs with existing result sets to increment and enrich views
  - Updated each time the pipeline is run
  - Output to sharded and unsharded collections
  - Define indexes on each view
- With uniqueKey, control how documents are added to the view: Insert, Replace, Merge

$merge Syntax

```
{ $merge: {
    to: "<output-collection>",
    on: <field1> | [ <field1>,<field2>,... ],
    whenNotMatched: <"insert" | "discard" | "fail">,
    whenMatched: <"merge" | "replace" | "keepExisting" | "fail" | [...] >
} }
```

Versatile: MongoDB Change Streams



Enabling developers to build
reactive, real-time services

MongoDB Connector for Apache Kafka (Beta)

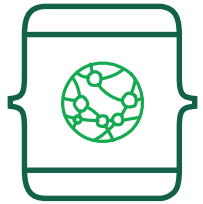
- Build robust data pipelines for microservices and Event Driven Architectures
- Developed with the community and supported by MongoDB engineers, verified by Confluent
- Supports MongoDB as a sink and a source for Kafka
- Integrate with Change Streams and Atlas triggers to create fully reactive, event driven pipelines

mongoDB

GitHub

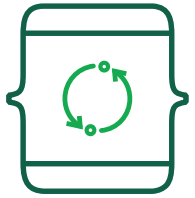
Confluent Hub

Intelligently put data where you need it



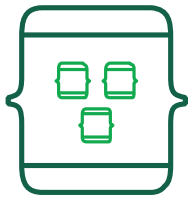
Highly Available

Built-in **multi-region** high availability, replication & automated failover



Workload Isolation

Ability to run both **operational & analytics workloads** on same cluster, for timely insight and lower cost



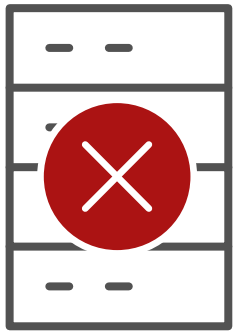
Scalability

Elastic horizontal scalability – add/remove capacity dynamically **without downtime**

Locality

Declare **data locality rules** for governance (e.g. data sovereignty), class of service & local low latency access

Challenges of scale & HA with tabular databases



Single Node, Vertically Scaled

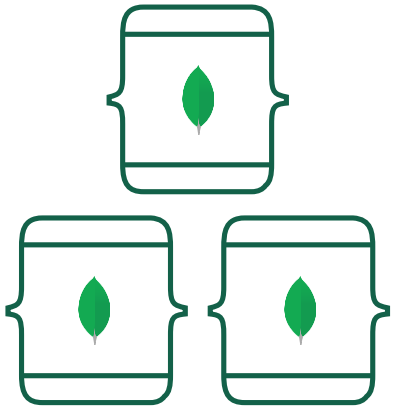
Scale-Out

- **Developer pain:** complex application-level sharding or external sharding frameworks
- **Static:** Limited elasticity
- **Trade-Offs:** Sacrifice key RDBMS functionality: cross-shard transactions, JOINs, referential integrity

Availability

- **Complex:** requires integration of external cluster managers for failure detection and recovery
- **Downtime:** extended outages with multi-minute recovery times
- **Developer pain:** write complex exception-handling code to handle multi-minute node failovers

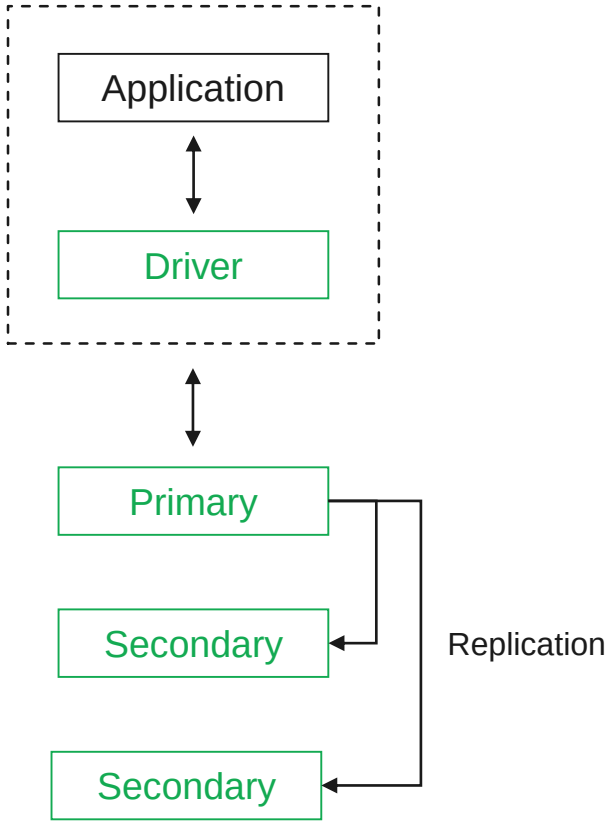
Put data where you need it: Availability



Replica Sets

- Up to 50 replicas, distributed across racks, data centers, and regions
- Self-healing using RAFT-based consensus protocol for automated failover & recovery
- Tunable durability and consistency controls
- Resilience with less code: retryable reads and writes

MongoDB Replica Sets



Replica Set – 2 to 50 copies

Self-healing

Data Center Aware

Addresses availability considerations:

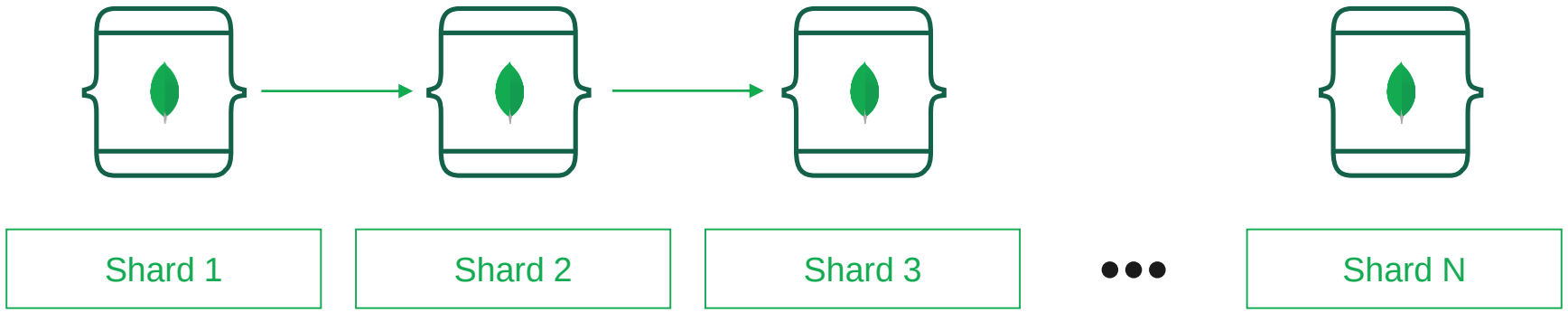
- High Availability
- Disaster Recovery
- Maintenance

Workload Isolation: operational & analytics

Serving global audiences with relational databases

MongoDB’s native replica sets puts your entire database right next to users

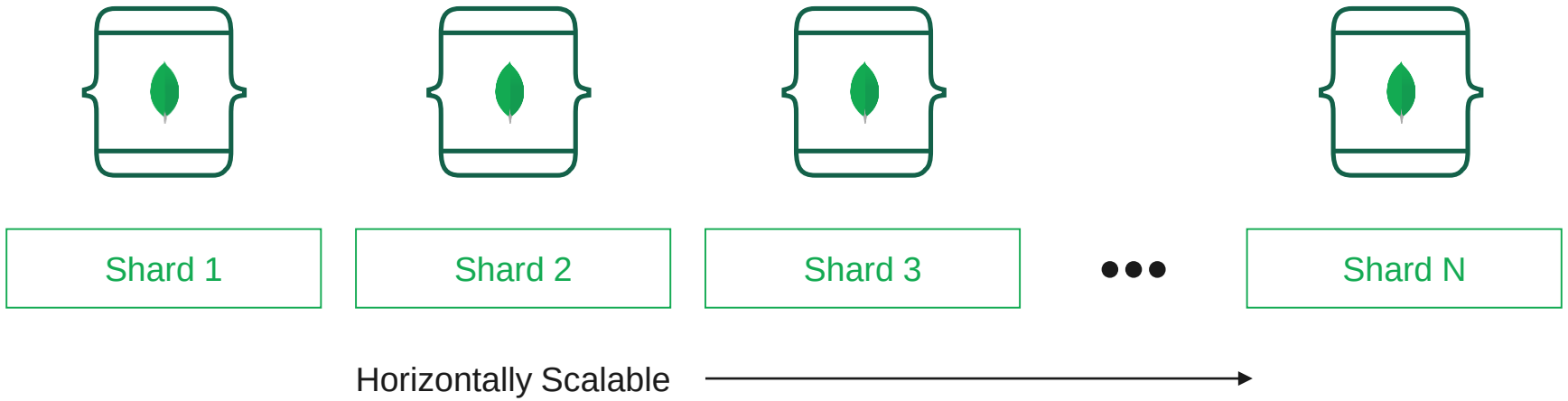
Put data where you need it: Scalability



Auto-Sharding

- Automatically scale beyond the constraints of a single node
- Application transparent
- Scale and rebalance incrementally, in real time
- Unlike NoSQL systems that randomly spray data across a cluster, MongoDB exposes multiple data distribution policies to optimize for query patterns and locality

Scaling MongoDB: Automatic Sharding

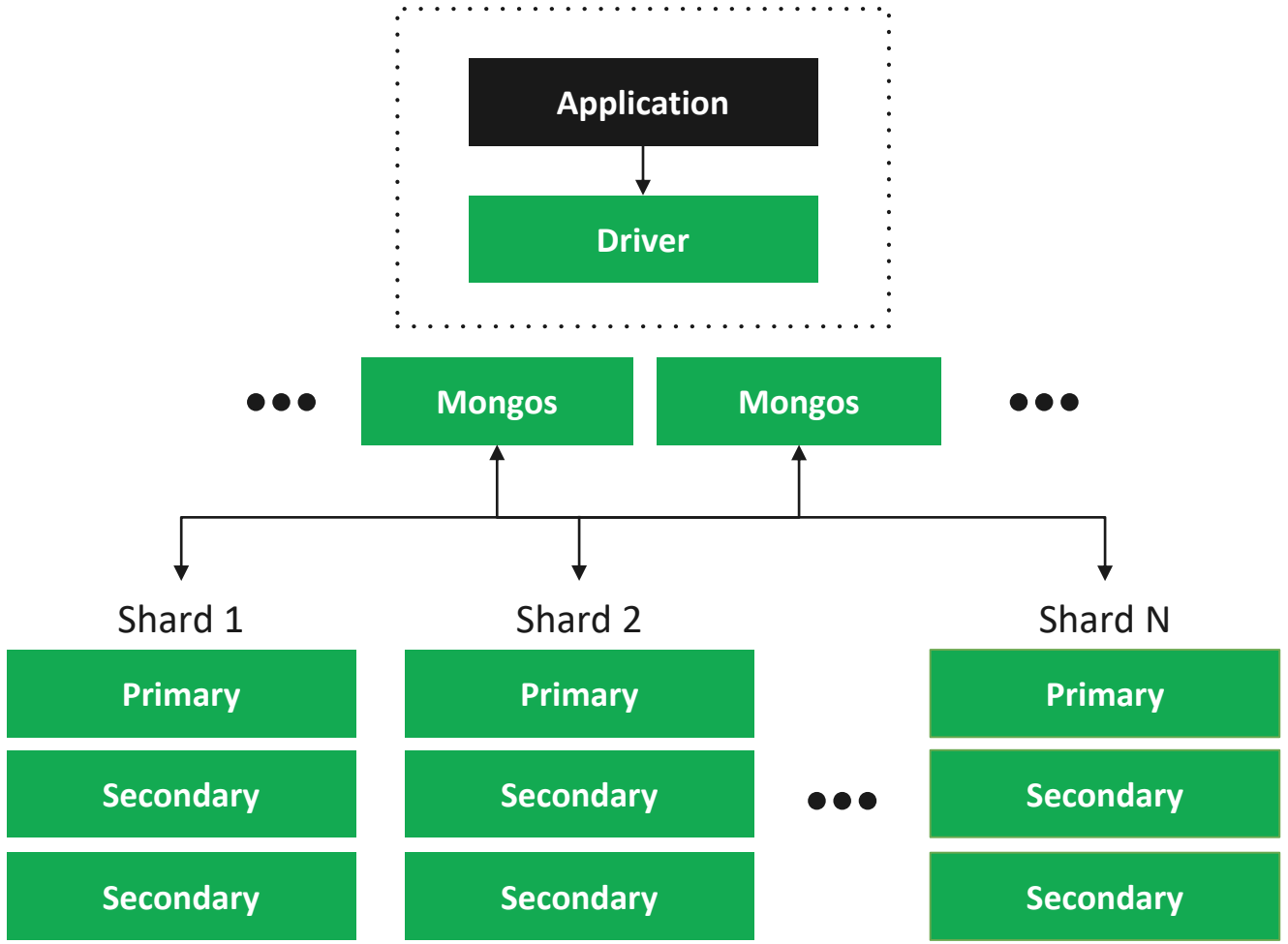


Multiple sharding policies: hashed, ranged, zoned

Increase or decrease capacity as you go

Automatic balancing for elasticity

Sharding Architecture



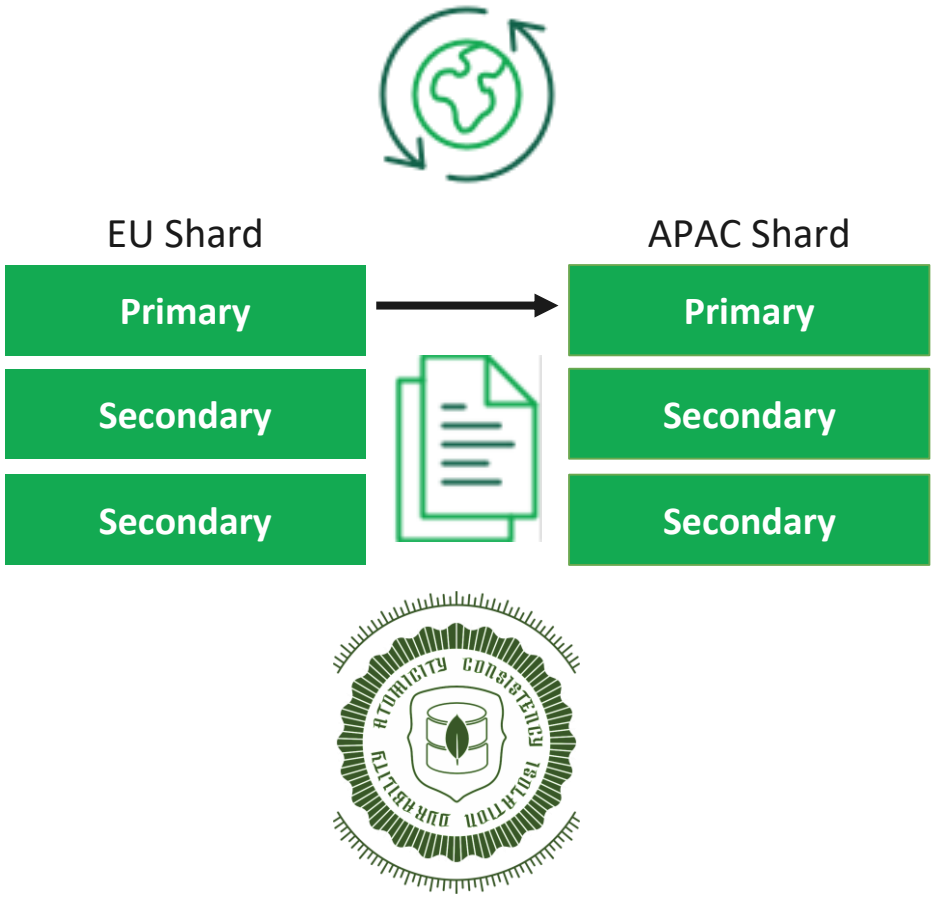
High availability

- Replica sets

Horizontal scalability

- Sharding

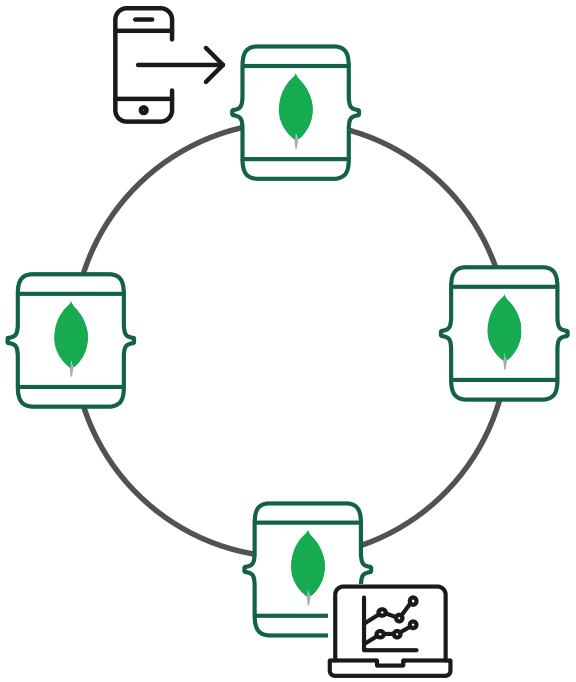
Mutable Shard Key Values with Distributed Transactions



Default 60-second run time limit

- Improved sharding flexibility
- Modifying a shard key value will migrate a document using a Distributed Transaction
  - **Global-redistribution:** Rehoming documents to a new region
  - **Tiered storage:** Aging out older documents to low cost storage shard
- No need to delete then re-insert document with the new shard key value, and control atomicity in the app

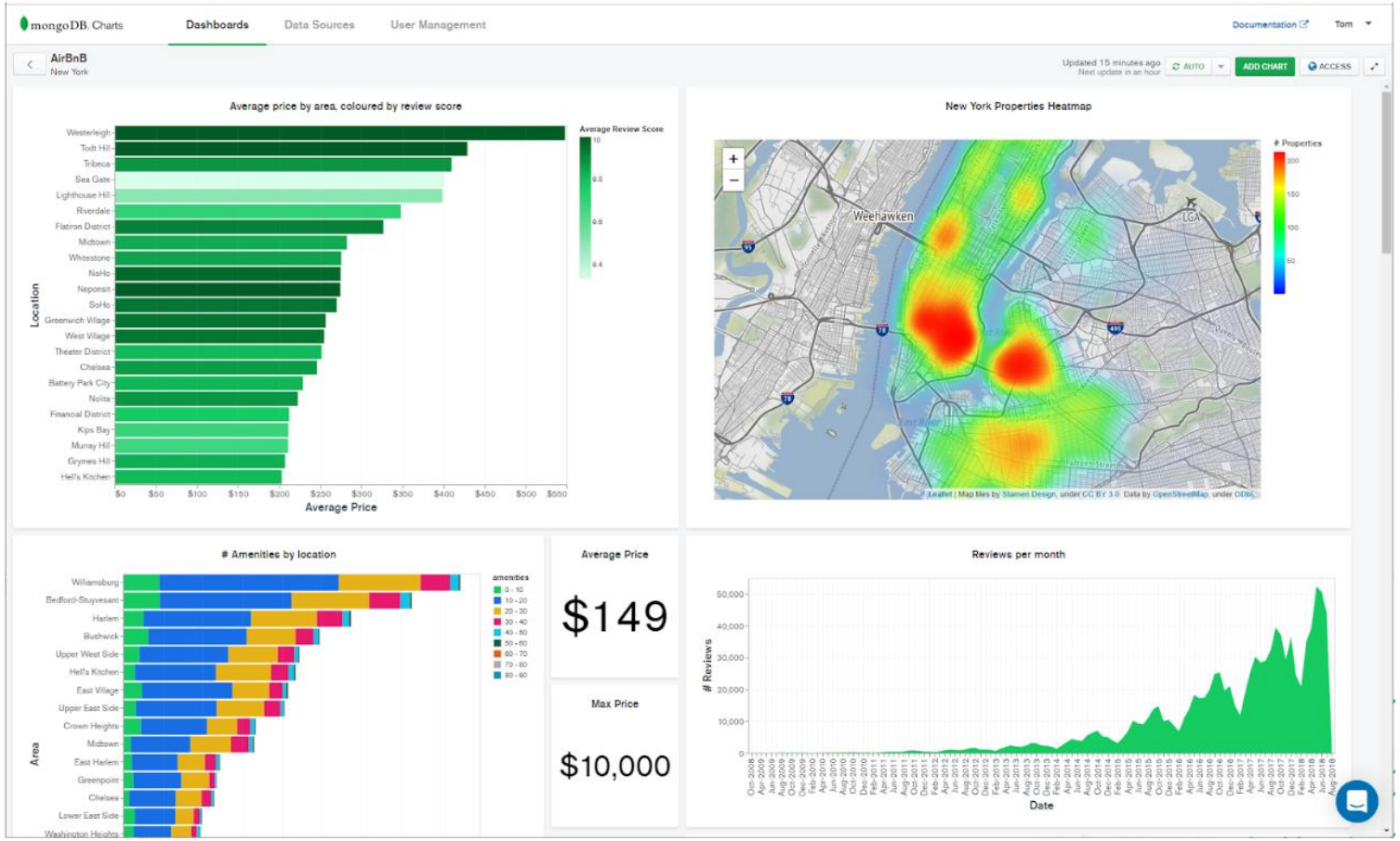
Put data where you need it: Workload Isolation



Enable different workloads on the same data

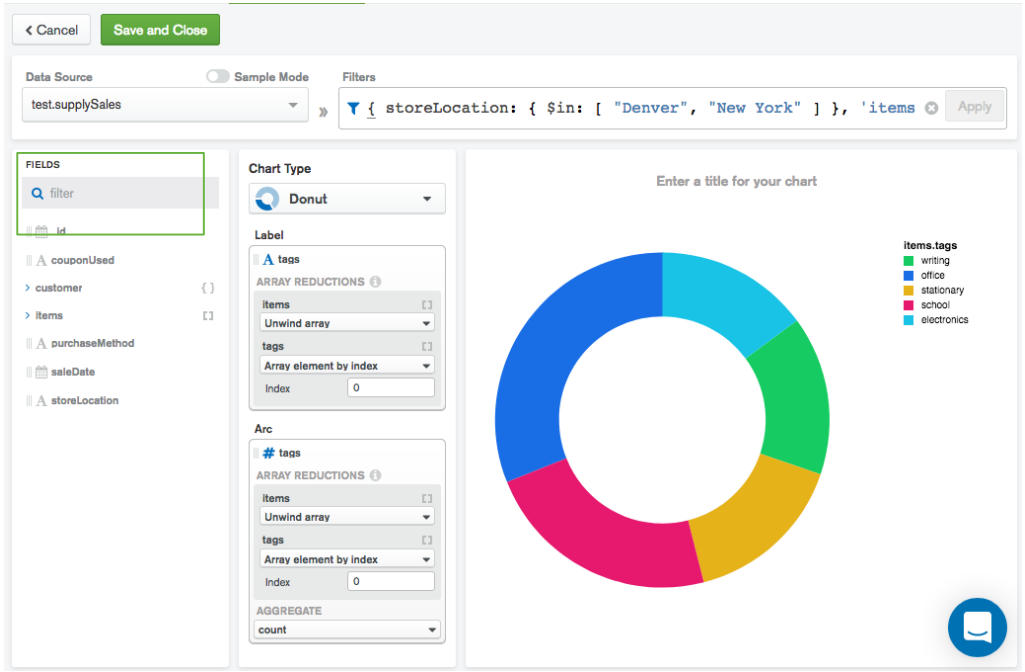
- Combine operational and analytical workloads on a single data platform
- Extract live insights from real-time data to enrich applications
- One set of nodes serving operational apps, replicating to dedicated nodes serving analytics: up to 50 nodes in a single replica set
- ETL-free

Sophisticated Analytics & Visualizations Of Data In Place



- Rich MongoDB query language & idiomatic drivers
- Charts
- Connector for BI
- Connector for Spark

MongoDB Charts: Create, Visualize, Share



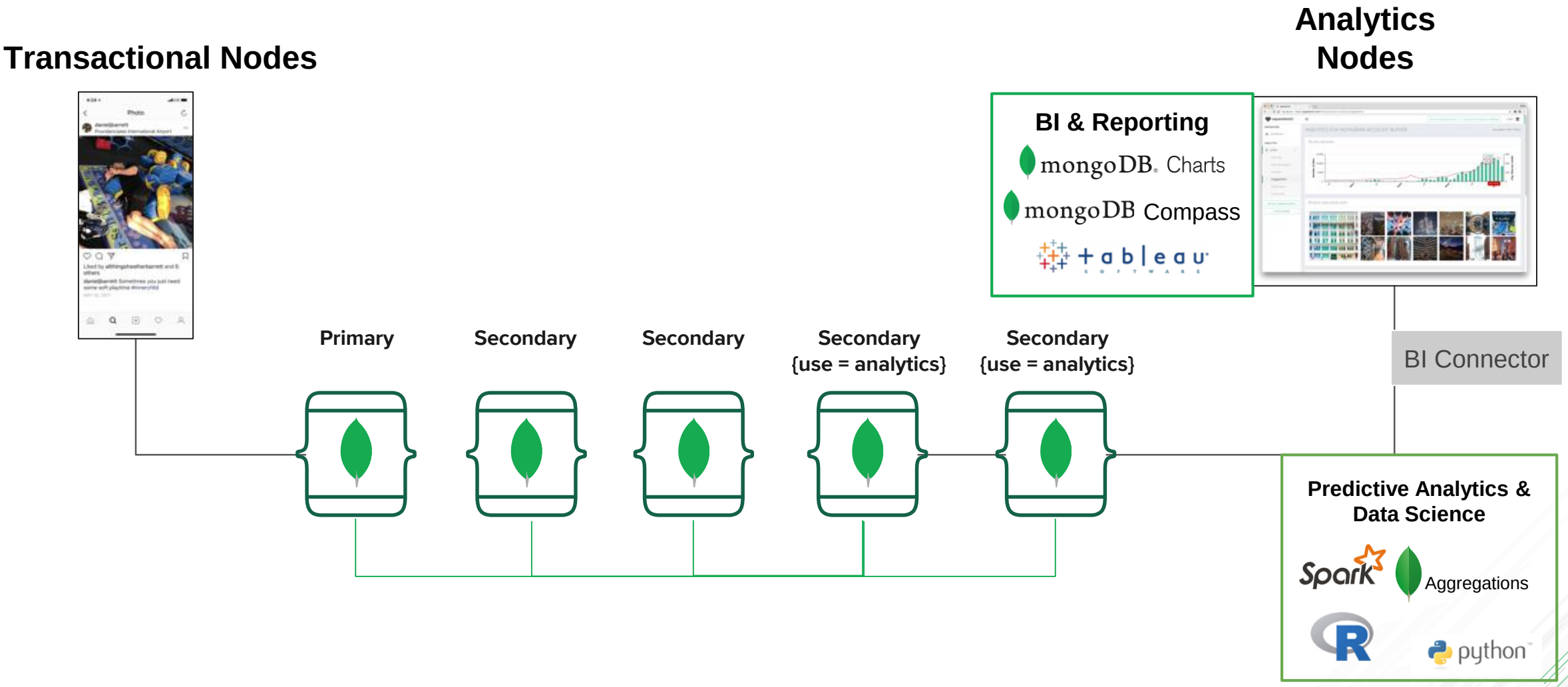
Work with complex data

Connect to data sources securely. Filter. Sample. Visualize. Share

Create standalone dashboards or embed directly into web apps

mongoDB

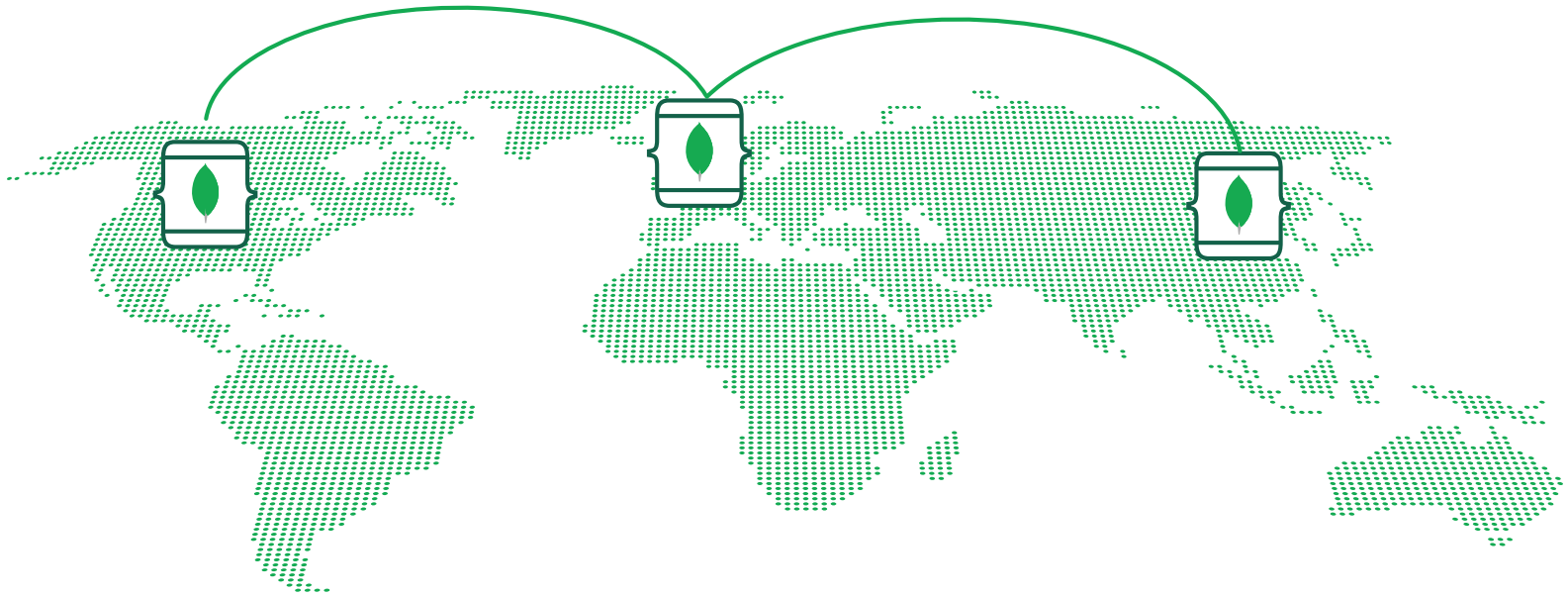
Co-locating operational and analytical workloads



Technical Implementation (automated in Atlas)

```
// Set up replica set
rs.initiate(
{
 "_id": "rs1",
 "version": 1,
 "members":
  [
    {"priority": 1, "host": "localhost:27001", "_id": 0, "tags": {"use": "op"}},
    {"priority": 1, "host": "localhost:27002", "_id": 1, "tags": {"use": "op"}},
    {"priority": 1, "host": "localhost:27003", "_id": 2, "tags": {"use": "op"}},
    {"priority": 0, "host": "localhost:27004", "_id": 3, "tags": {"use": "analytics"}},
    {"priority": 0, "host": "localhost:27005", "_id": 4, "tags": {"use": "analytics"}}]
})

// Force reads to analytics servers
read_client = MongoClient(host = 'localhost:27001', replicaset = 'rs1',
        readPreference='secondary', readPreferenceTags = ["use:analytics"])
```

Put data where you need it: Locality



Governance

Class of Service

Latency

Intelligent Distribution via Zoned Sharding

- Policies to define data placement
- Name a server by region, tag your documents by region and MongoDB does the rest
- Documents automatically migrated as shard key ranges are modified
- Global queries across all data in all zones
- Fully managed as Global Clusters in Atlas

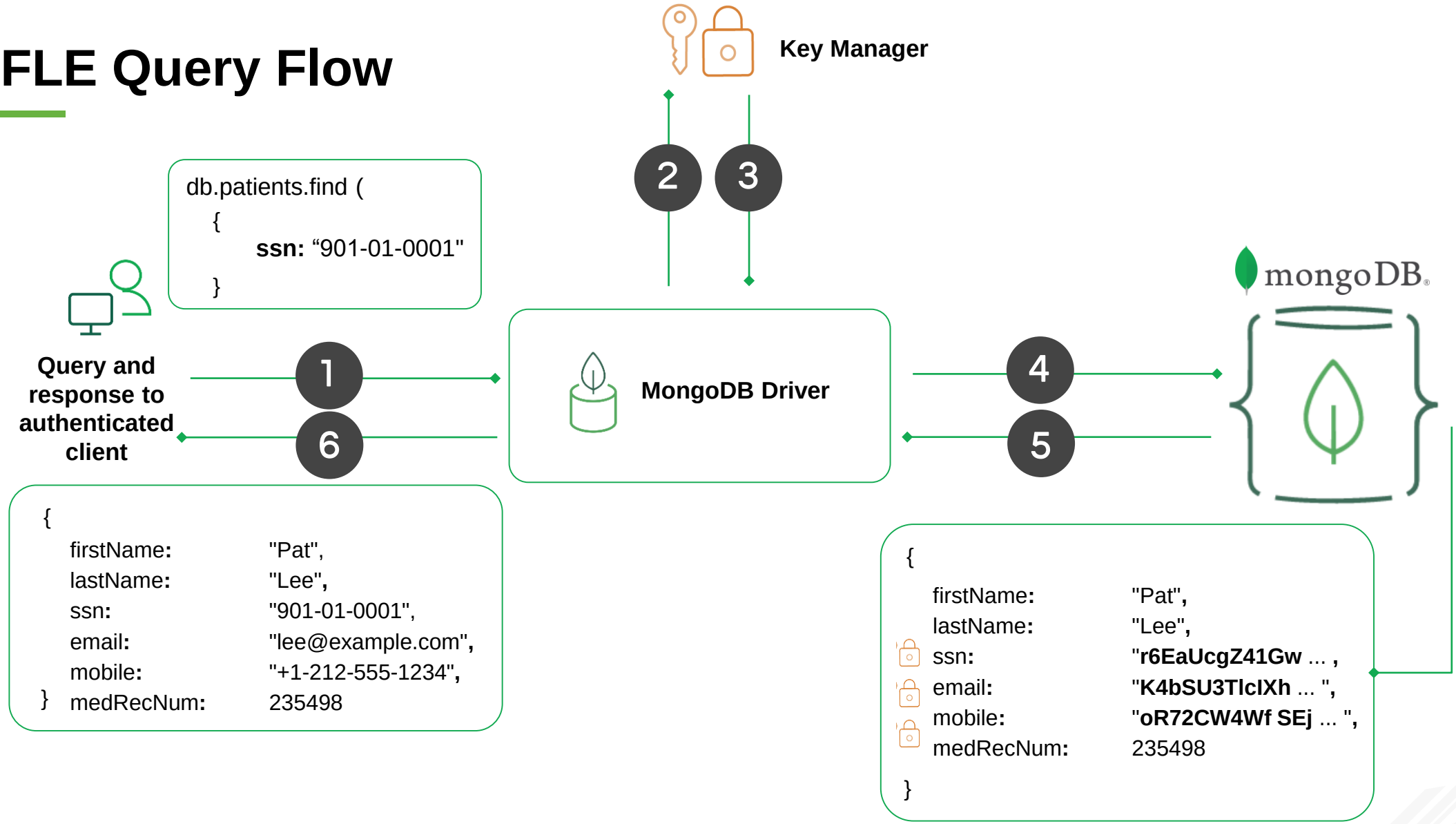
Secure: Wherever you put your data

Business Needs	Security Features
Authentication	SHA-2, LDAP, Kerberos, x.509 Certificates
Authorization	RBAC, Read-Only Views, Field-Level Redaction
Auditing	Admin, DML, DDL, Role-based
Encryption	Network: TLS (with FIPS 140-2) In-Use: Client-Side Field Level Encryption (Beta) Disk: Encrypted Storage Engine or Partner Solutions

New: Client-Side Field Level Encryption (Beta)

- Individual document fields encrypted by own key
- Database only sees ciphertext
- Many Advantages
  - Easy: Automatic and Transparent
  - Separation of Duties: (simplifies move to service-based systems as no service engineers ever see plaintext)
  - Compliant: Regulatory “right to be forgotten”
  - Fast: Minimal performance penalty

FLE Query Flow

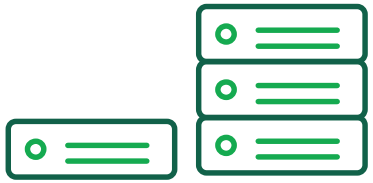


Encrypted fields always stored, transmitted, and retrieved as ciphertext

Freedom to run anywhere

Local

On-premises
Server & Mainframe



Private cloud

Hybrid cloud

Public cloud

Fully managed cloud service

- Database that runs the same everywhere
- Leverage the benefits of a multi-cloud strategy
- Global coverage
- Avoid lock-in

Convenience: same codebase, same APIs, same tools, wherever you run

Manage & Run MongoDB Yourself Or Use DBaaS

Laptop

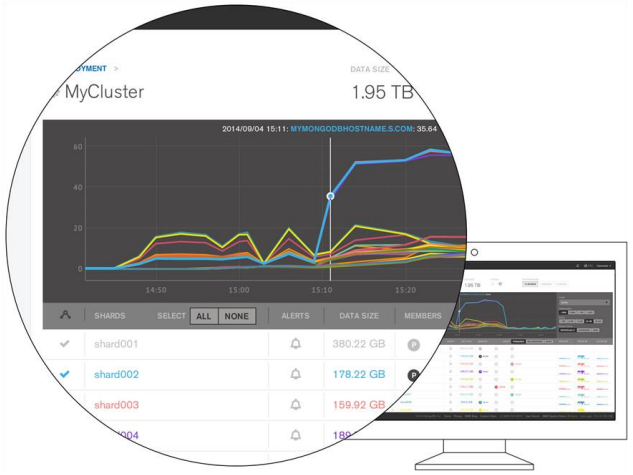
Mainframe

Server

Self-managed in the cloud

Database as a Service

mongoDB. Atlas



Self-Managed: Ops Mgr

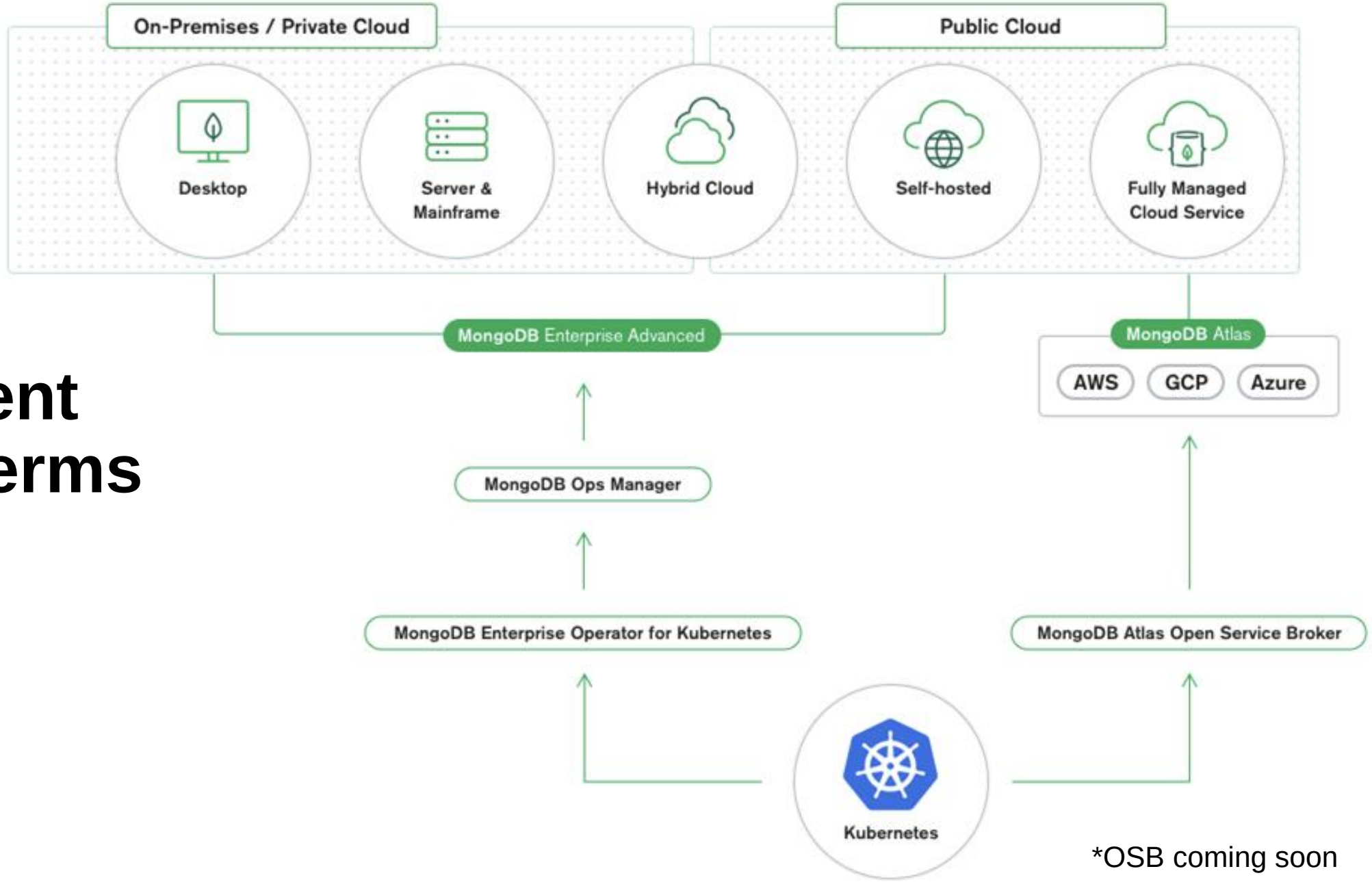
- Automation
- Monitoring Alerting
- Backup/Restore
- Patching
- Scaling
- Performance Advice
- Kubernetes Operator

Database as a Service

- Self-service & fully automated
- Global, elastic clusters
- Highly available and secure out of the box
- Operational tooling built in: monitoring, backups

- **On AWS, Azure, GCP, 60+ regions total**

Deployment on your terms



*OSB coming soon

Don’t I get this freedom with relational databases?

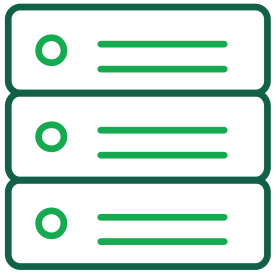
Google Cloud Platform

Fragmentation across different cloud platforms

- Different versions
- Different database features and options
- Different scaling characteristics
- Different cross-region options
- Different monitoring, alerting, backup, and recovery
- Different security controls

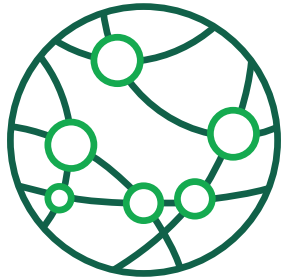
Architectural mis-alignment

Choosing the right data platform for the cloud



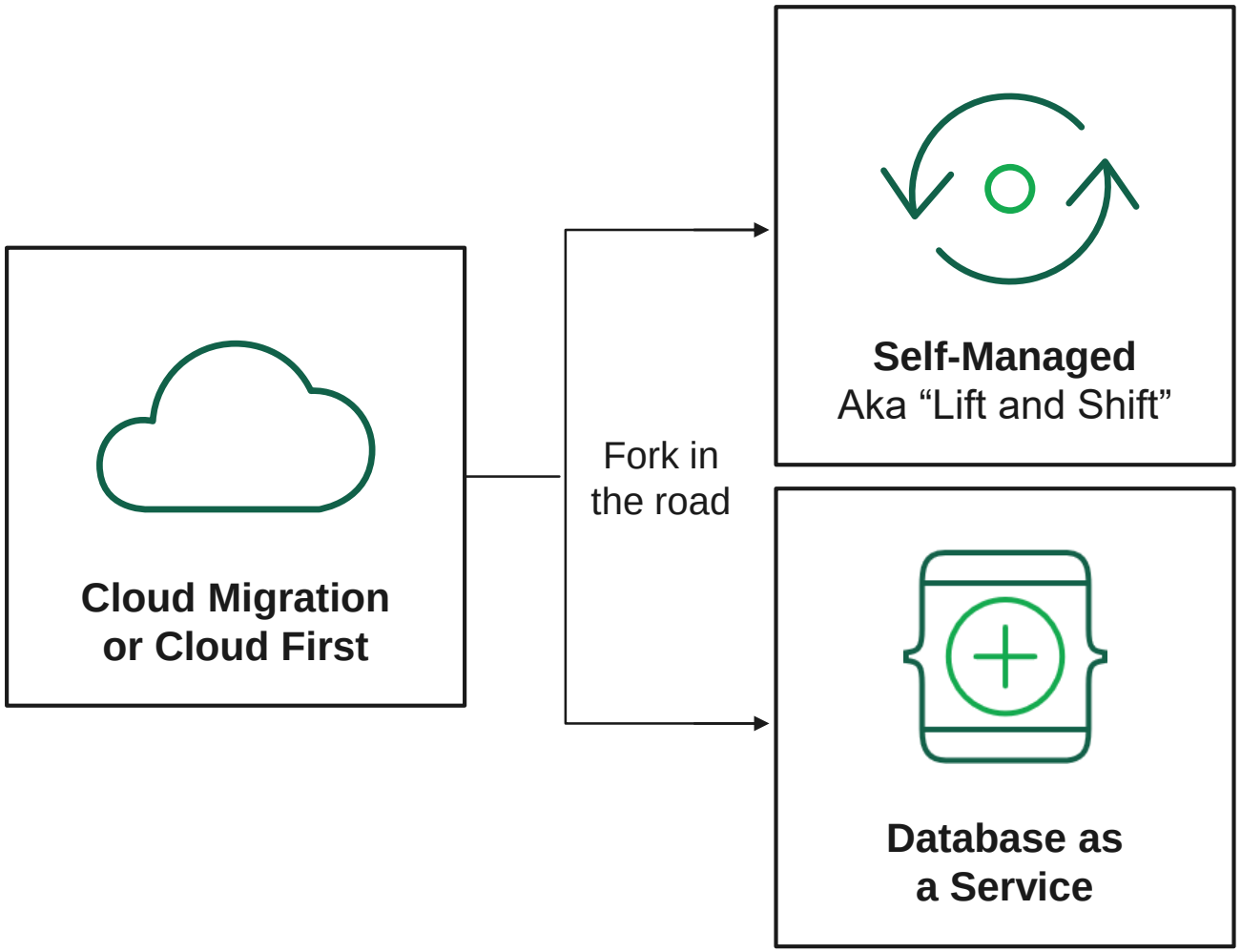
Exploit scale-out, commodity platforms

Support velocity of modern app development

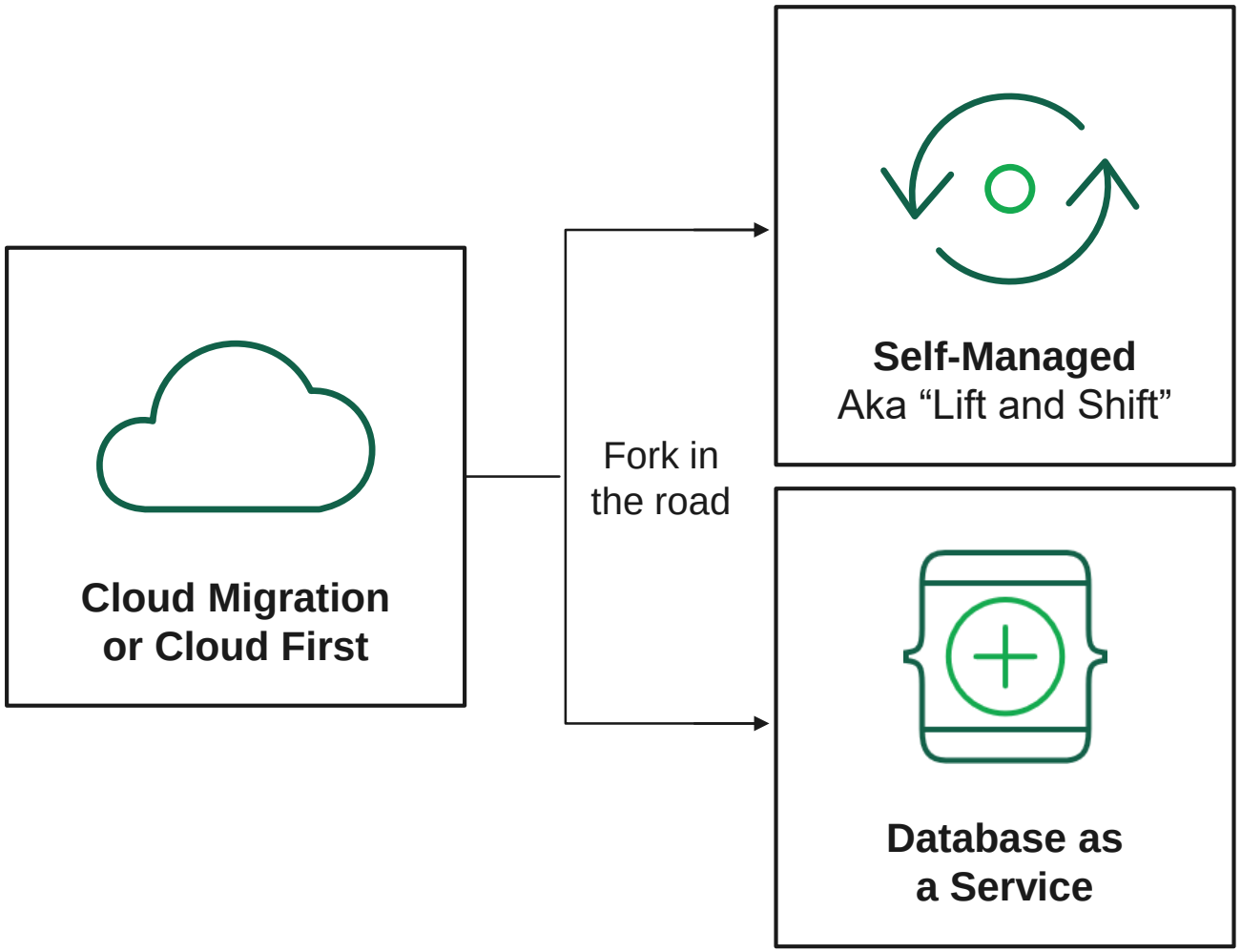


Deploy globally, on-demand

2 choices as you move to the cloud: Self-Managed or DBaaS

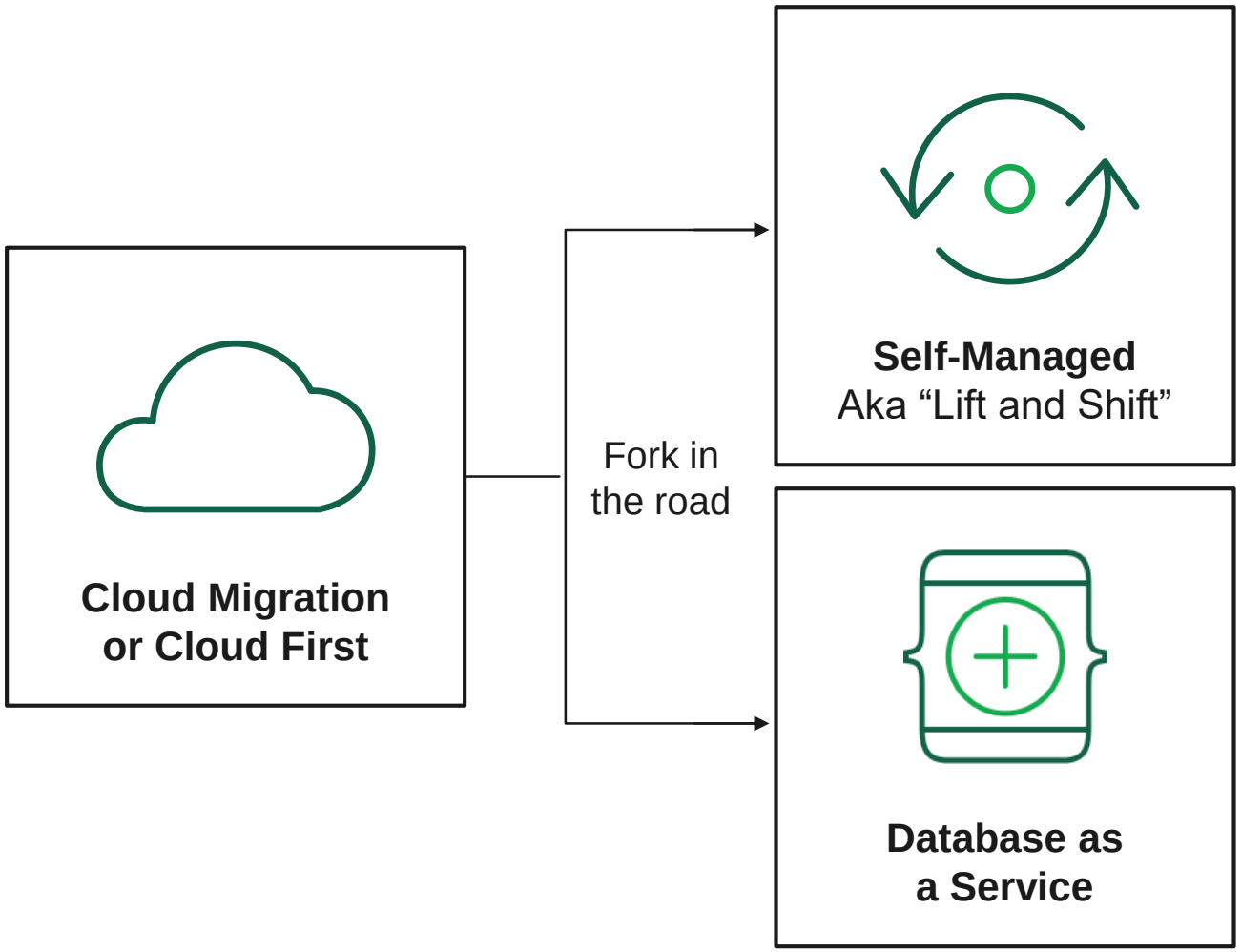


2 choices as you move to the cloud: Self-Managed or DBaaS



1. Provision instances and storage
2. Configure HA
3. Configure security
4. Configure backup/restore
5. Monitoring & alerting
6. Ongoing upgrades & maintenance

2 choices as you move to the cloud: Self-Managed or DBaaS



1. Provision instances and storage
2. Configure HA
3. Configure security
4. Configure backup/restore
5. Monitoring & alerting
6. Ongoing upgrades & maintenance

Choose instance, hit deploy, available in a couple of minutes

Atlas

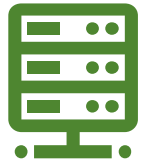
unlocks **agility** and **reduces cost**

Self-service and elastic

Global and highly available

Secure by default

Comprehensive monitoring



Managed backup

Sync & Serverless

mongoDB

MongoDB Atlas — Global Cloud Database

Self-service & elastic

Deploy, modify, and upgrade on demand with best-in-class operational automation

Automated database maintenance

Database and infrastructure resources as code

Scale up, out, or down in a few clicks or API calls

Global & cloud-agnostic

Available in 60+ regions across AWS, Azure, GCP

Global clusters for read/write anywhere deployments and multi-region fault tolerance

Easy migrations with a consistent experience across cloud providers

Enterprise-grade security & SLAs

Network isolation, VPC peering, end-to-end encryption, and role-based access controls

Encryption key management, LDAP integration, granular database auditing

SOC 2 / Privacy Shield / HIPAA

Guaranteed reliability with SLAs

Comprehensive monitoring

Deep visibility into 100+ KPIs with proactive alerting

Real-time performance tracking and Performance Advisor

APIs to integrate with monitoring dashboards

Managed backup

Point-in-time data recovery

Queryable backup snapshots

Consistent snapshots of sharded deployments

Cloud data mobility

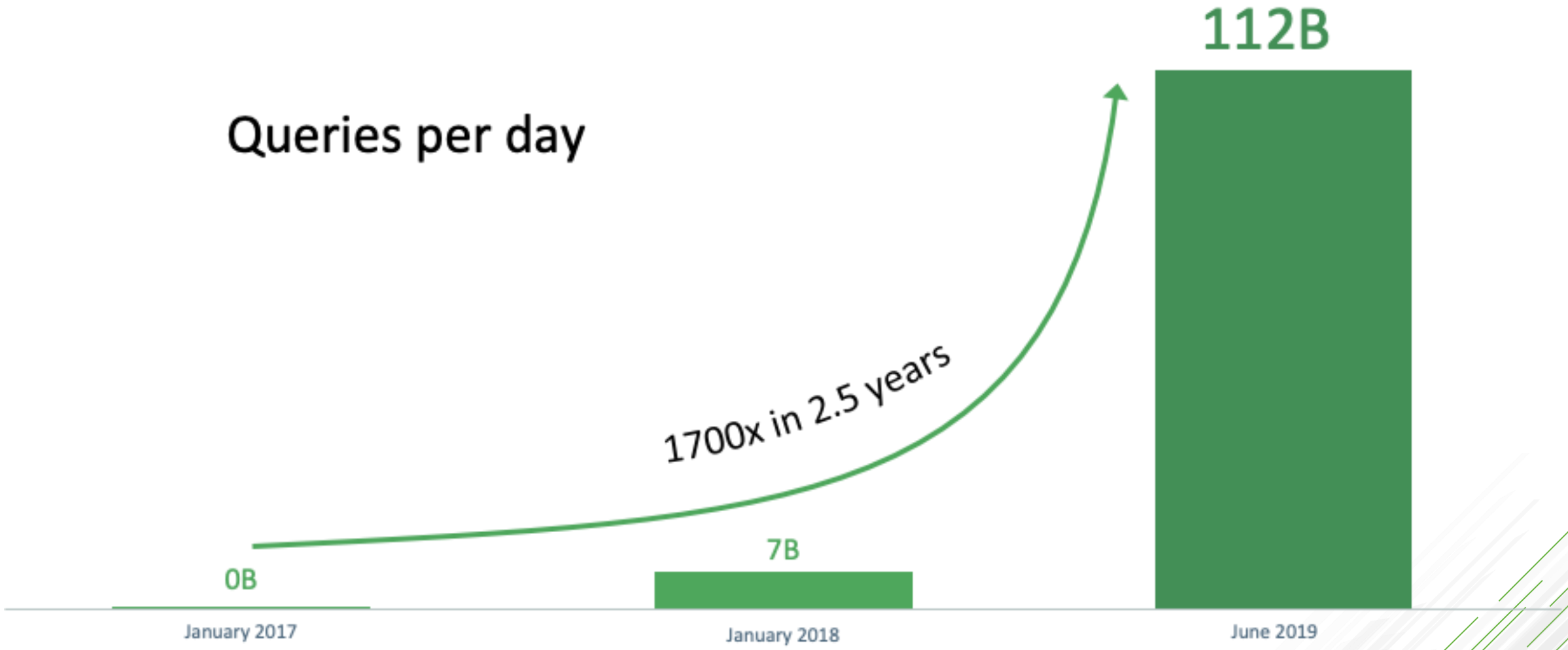
Sync and Serverless

Simple, serverless functions for backend logic, service integrations, and APIs

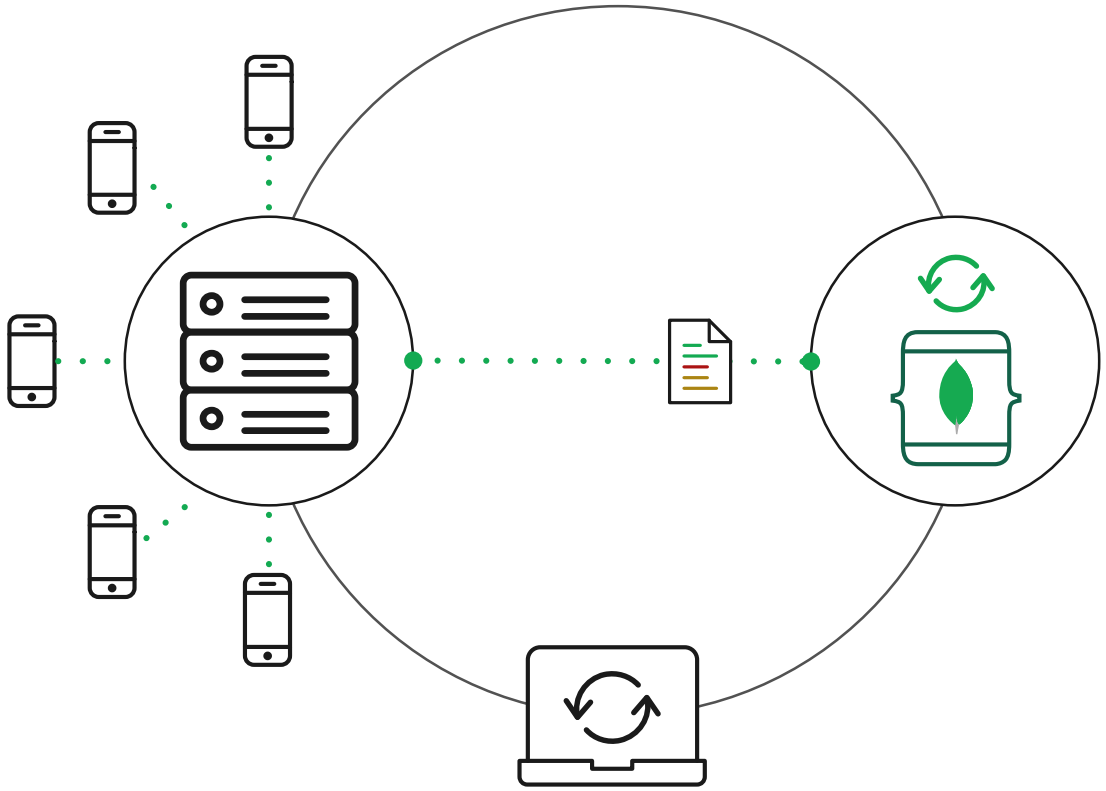
Database access from your frontend secured by straightforward, field-level access rules

Database and authentication triggers to react to changes in real time

Powering Mission-Critical Workloads



Freedom to run Anywhere: Cloud Migration



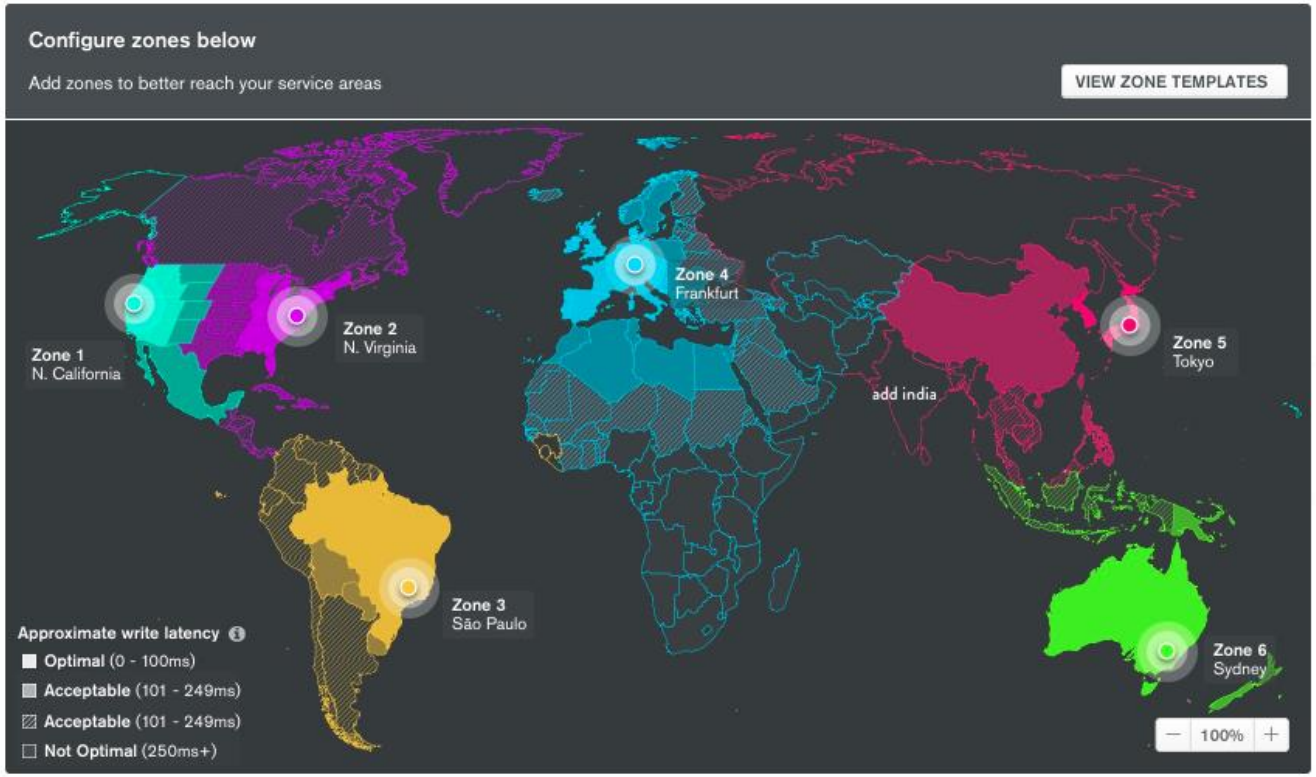
Migrate existing deployments running anywhere into MongoDB Atlas with minimal impact to your application.
Live migration works by:

- Performing a sync between your source database and a target database hosted in MongoDB Atlas
- Syncing live data between your source database and the target database by tailing the oplog
- Notifying you when it's time to cut over to the MongoDB Atlas cluster

Geo-Distributed apps: MongoDB Atlas Global Clusters

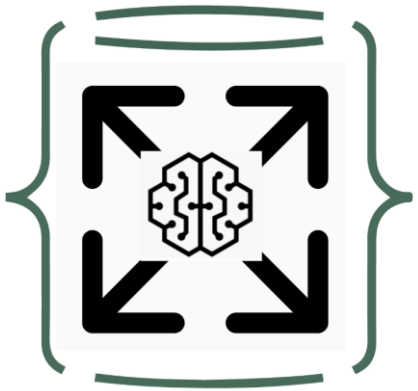
Distribute your fully automated database across multiple geographically distributed *zones* made up of one or more cloud regions

- Read and write locally to provide single-digit millisecond latency for your distributed applications
- Ensure that data from certain geographies lives in predefined zones
- Easily deploy using prebuilt zone templates or build your own zones by choosing cloud regions in an easy-to-use, visual interface



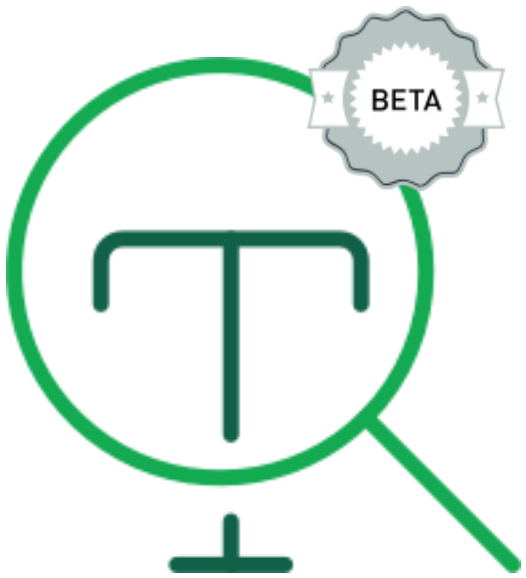
mongoDB

New: MongoDB Atlas Auto Scaling (late-Summer)



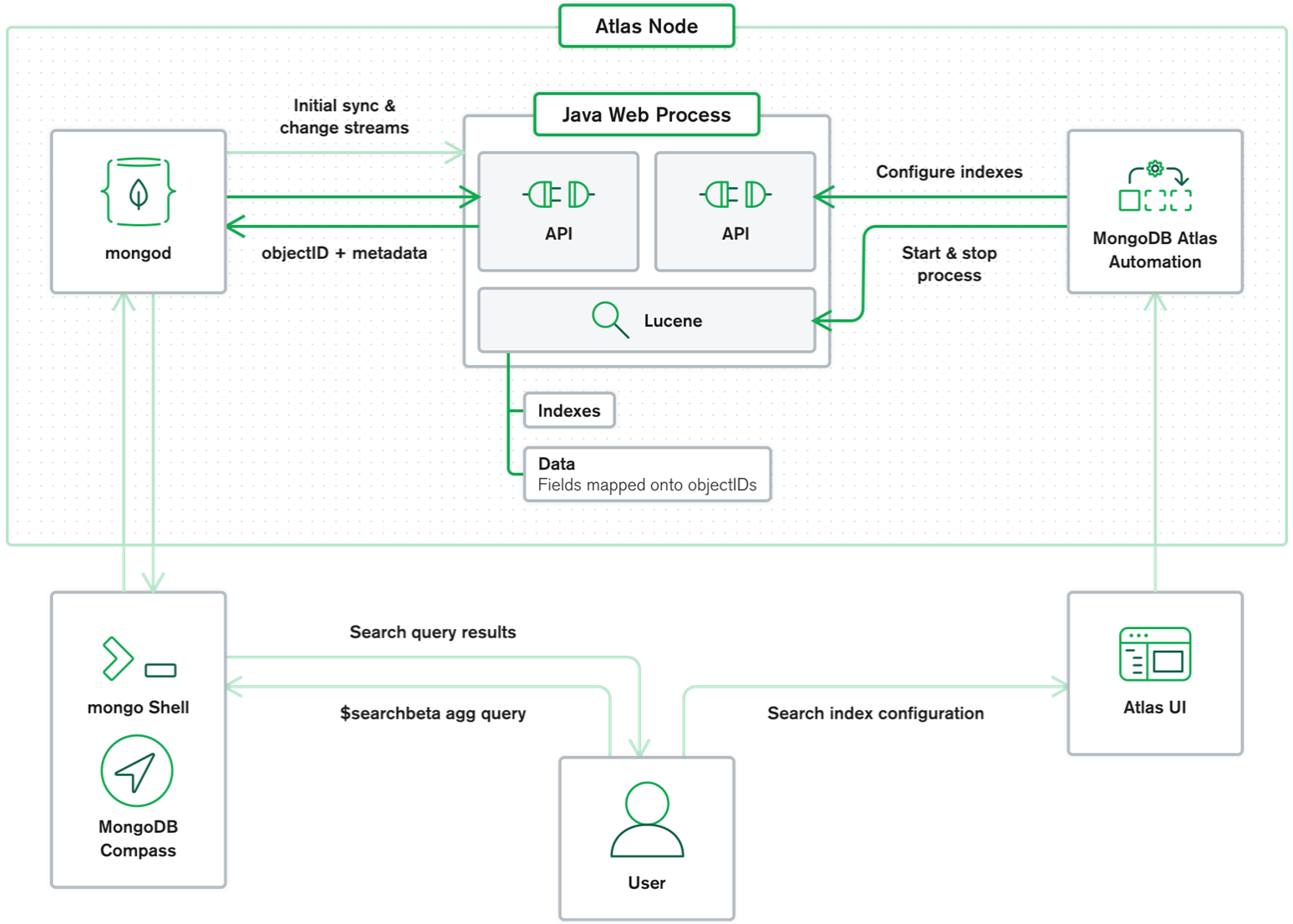
- Elastic scaling of instance sizes so your provisioned capacity responds to demand
  - Monitors key resource utilization metrics
  - Toggle on and off via UI or API
  - Cap peak instance sizes to control costs
- Rolling restarts across replica sets to maintain app availability
- Auto-storage scaling available since 2018

New: MongoDB Atlas Full Text Search (Beta)



- Adds Full Text Search as a fully managed service to your Atlas cluster
  - Power of Lucene 8, without provisioning and running a separate search platform
  - Integrated with MongoDB Query Language, so no separate APIs to learn
  - Dynamic and static indexing supporting fuzzy & wildcard search, Boolean & compound queries, language analyzers, scoring and snippets
  - Configured via Atlas Data Explorer or API

MongoDB Atlas Full Text Search Architecture

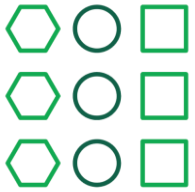


mongoDB

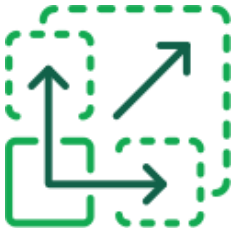
New: MongoDB Atlas Data Lake (Beta)

Analyze data in any format on S3 using the MongoDB Query Language

Multiple Formats, No Schema



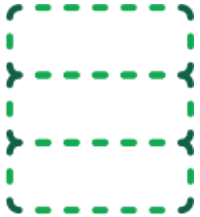
Auto-Scale, At Any Scale



Best Tools, High Productivity

Integrated with Atlas, Single UI, Billing, Permissioning

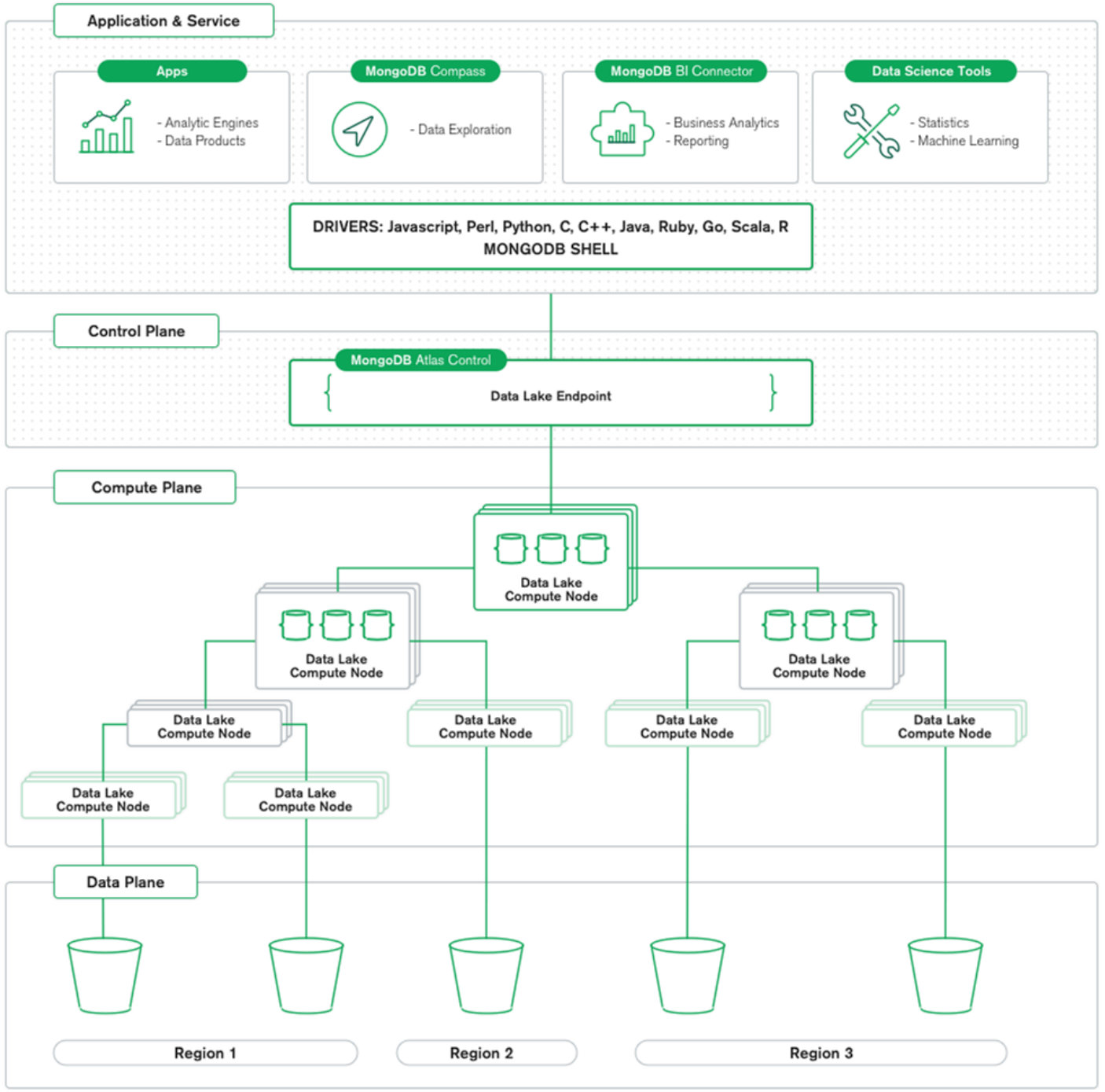
Serverless, No Infrastructure Management



On-Demand, Usage-Based Pricing

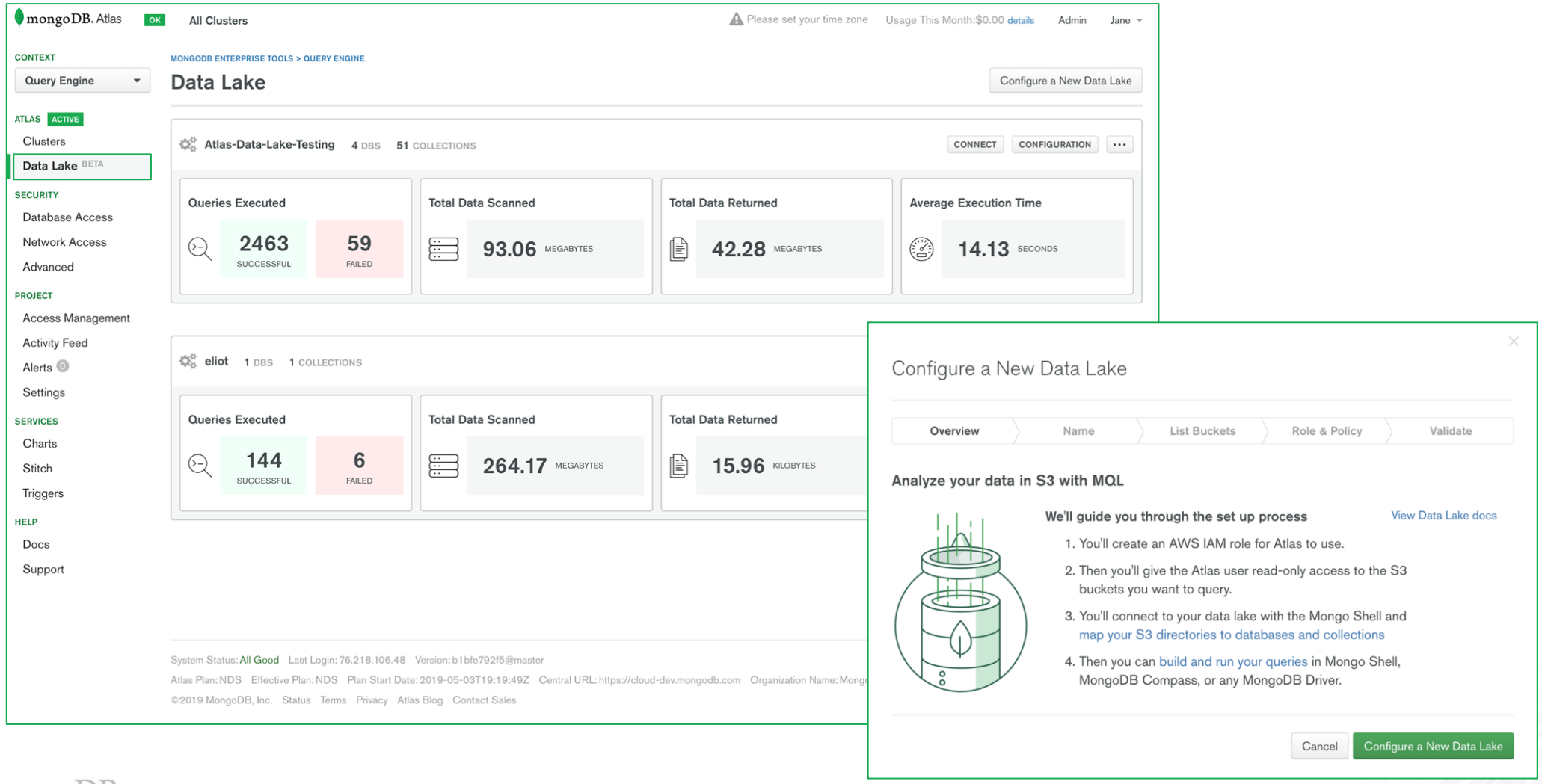
mongoDB

MongoDB Atlas Data Lake Architecture



mongoDB

MongoDB Atlas Data Lake UI



mongoDB

Use Cases: Atlas Data Lake

Data Lake Analytics

- ➔ explore all of your rich data naturally
- ➔ get to data as it lands via streams or microservices
- ➔ democratize access across diverse user groups

Data Products and Services

- ➔ monetize data
- ➔ market research, data- and insight-as-a-service
- ➔ snapshots, time series analysis, predictive analytics to innovate faster

Active Archives

- ➔ historical analysis against data assets retained in long term cold storage
- ➔ cost-effective data strategy

mongoDB. Atlas: The only *true* multi-cloud database as a service

Database that runs the same everywhere

Consistent experience across AWS, Azure, and GCP

Coverage in any geography

Deploy in 59 regions worldwide

Create globally distributed databases with a few clicks

Leverage the benefits of a multi-cloud strategy

Exploit the benefits of AWS, Azure, or GCP services on your data

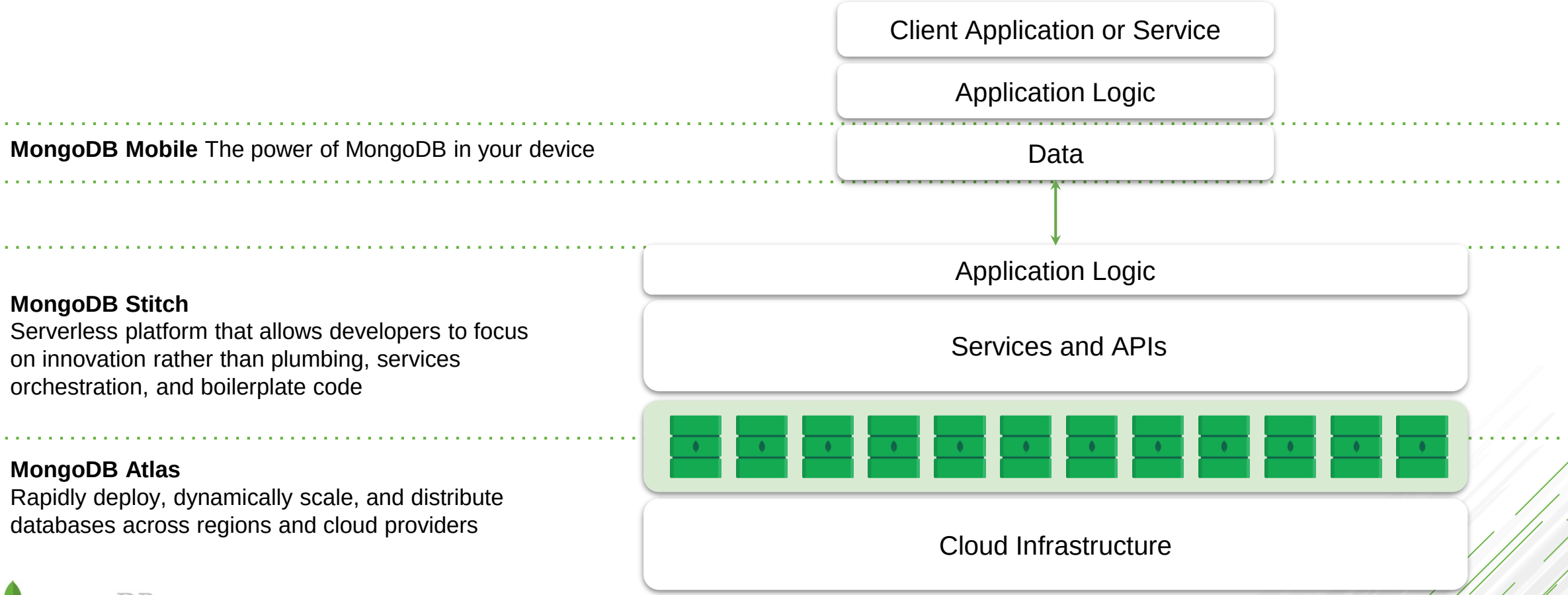
Avoid lock-in

Easily migrate data between cloud providers

Beyond the Server

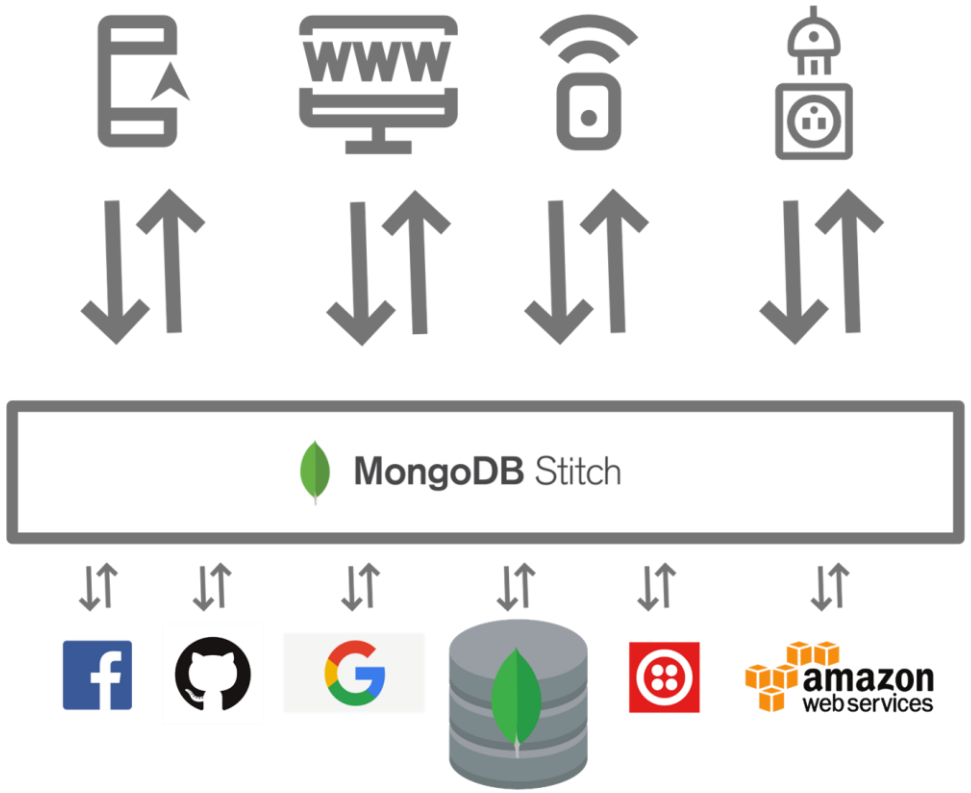
The Broader IODP Accelerates Everything

3-5x increase in productivity by leveraging the Intelligent Operational Data Platform



mongoDB

MongoDB Stitch Serverless Platform



Streamlines app development with simple, secure access to data and services from the client with thousands of lines less code to write and no infrastructure to manage.

Getting your apps to market faster while reducing operational costs.

MongoDB Stitch Serverless Platform – Services

Stitch QueryAnywhere

Brings MongoDB's rich query language safely to the edge

iOS, Android, Web, IoT

Stitch Functions

Integrate microservices + server-side logic + cloud services

Build full apps, or Data as a Service through custom APIs

Stitch Triggers

Real-time notifications let your application functions react in response to database changes, as they happen

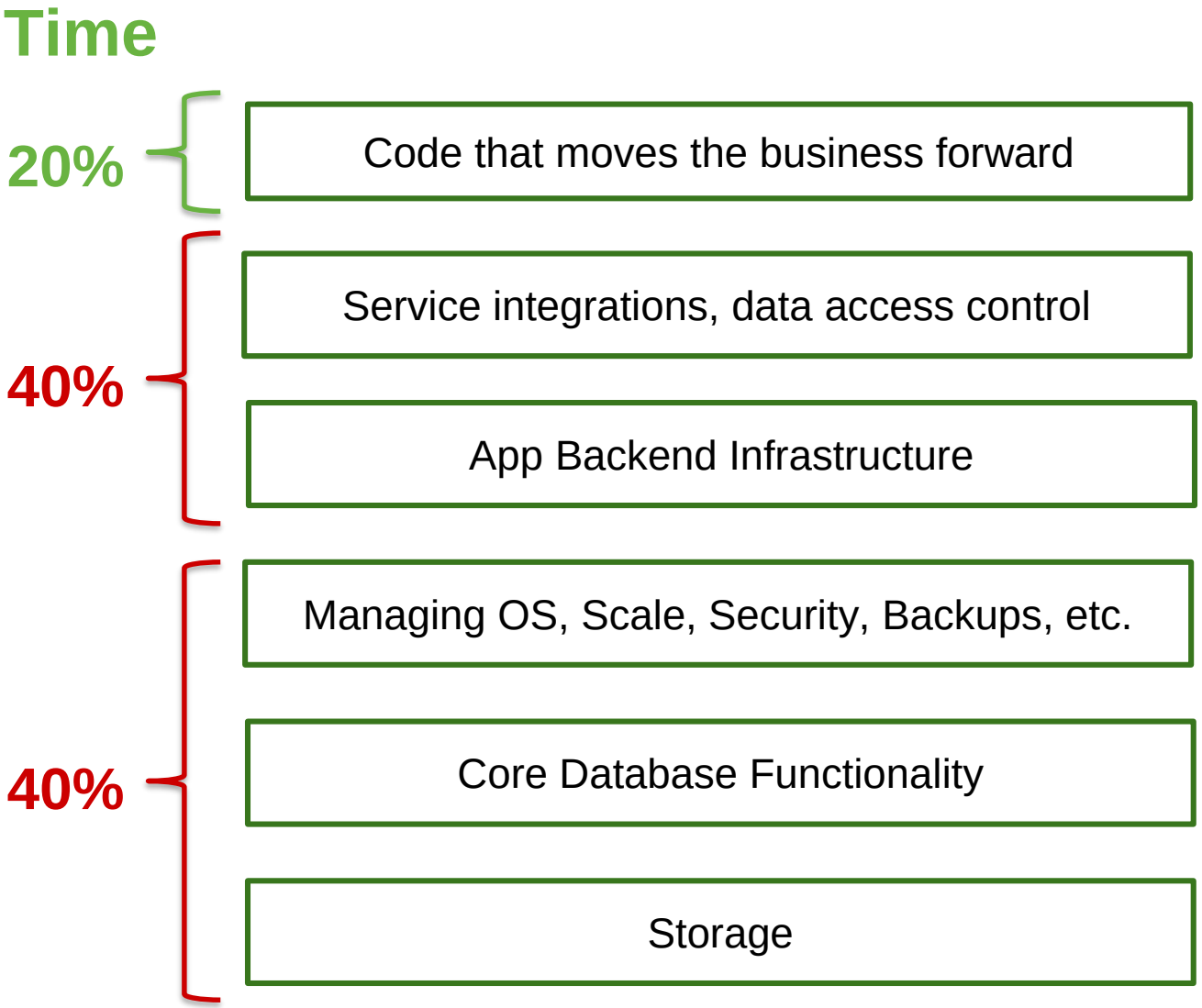
Stitch Mobile Sync

Automatically synchronizes data between documents held locally in MongoDB Mobile and your backend database

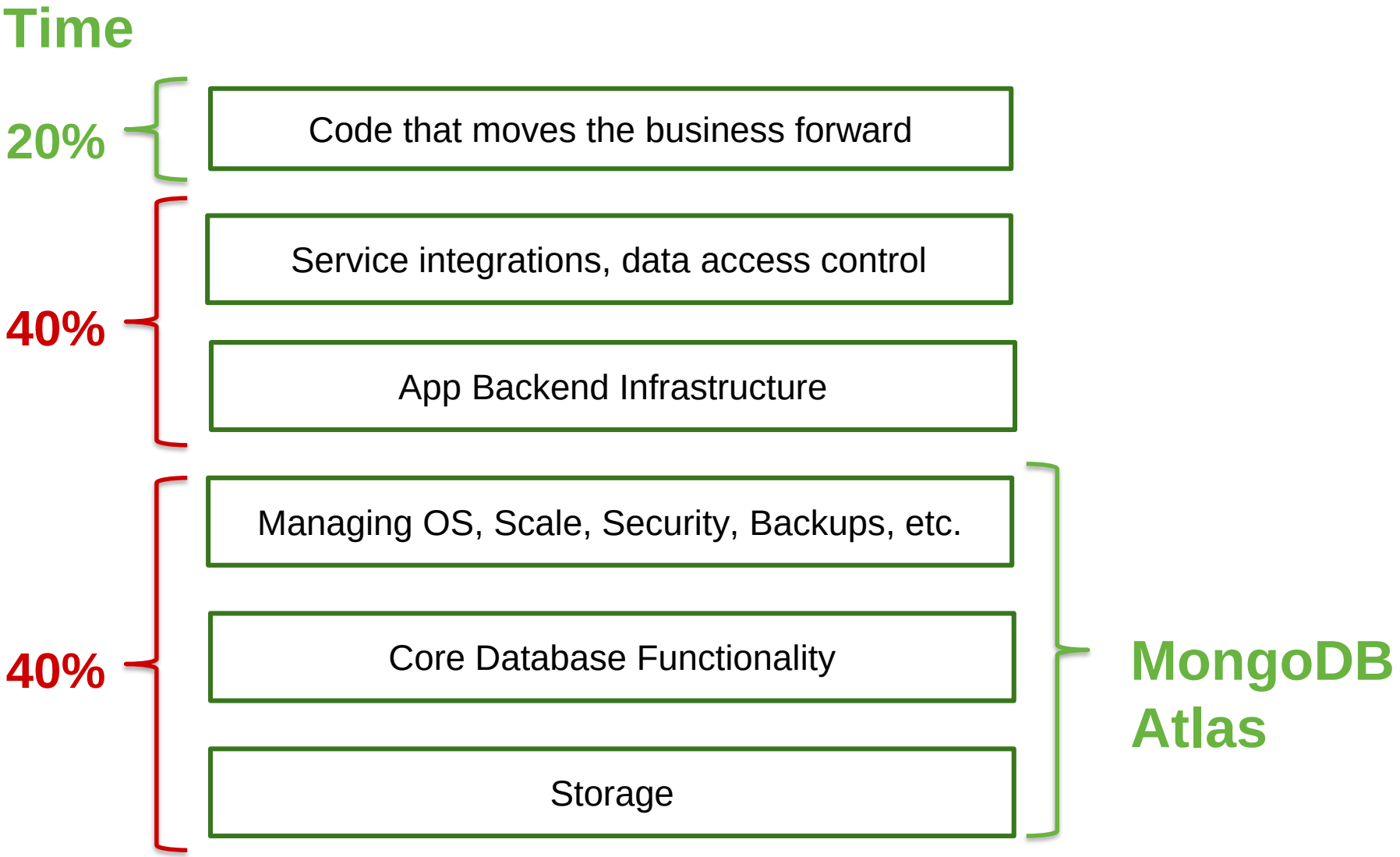
(beta)

Streamlines app development with simple, secure access to data and services from the client with thousands of lines less code to write and no infrastructure to manage – getting your apps to market faster while reducing operational costs.

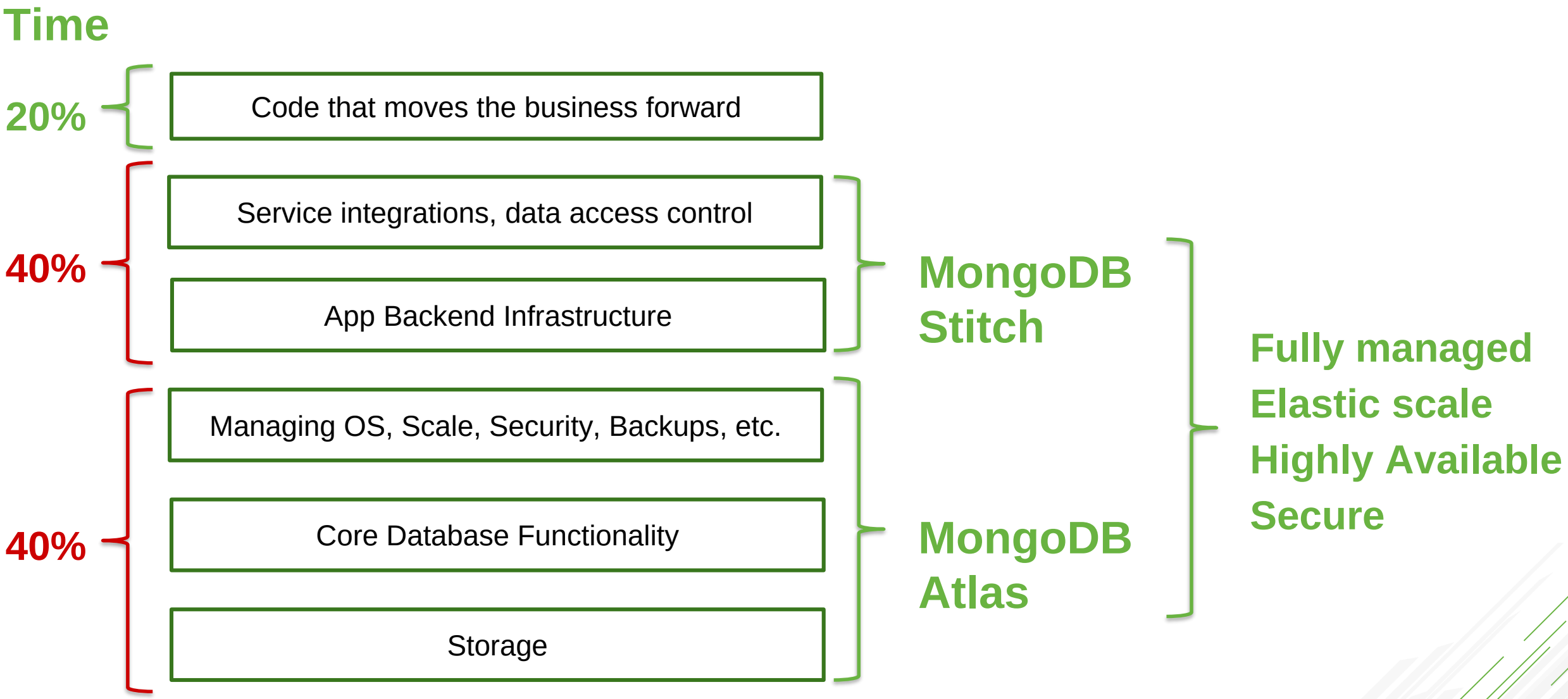
Focus Your Energy Where You Can Make a Difference



Focus Your Energy Where You Can Make a Difference

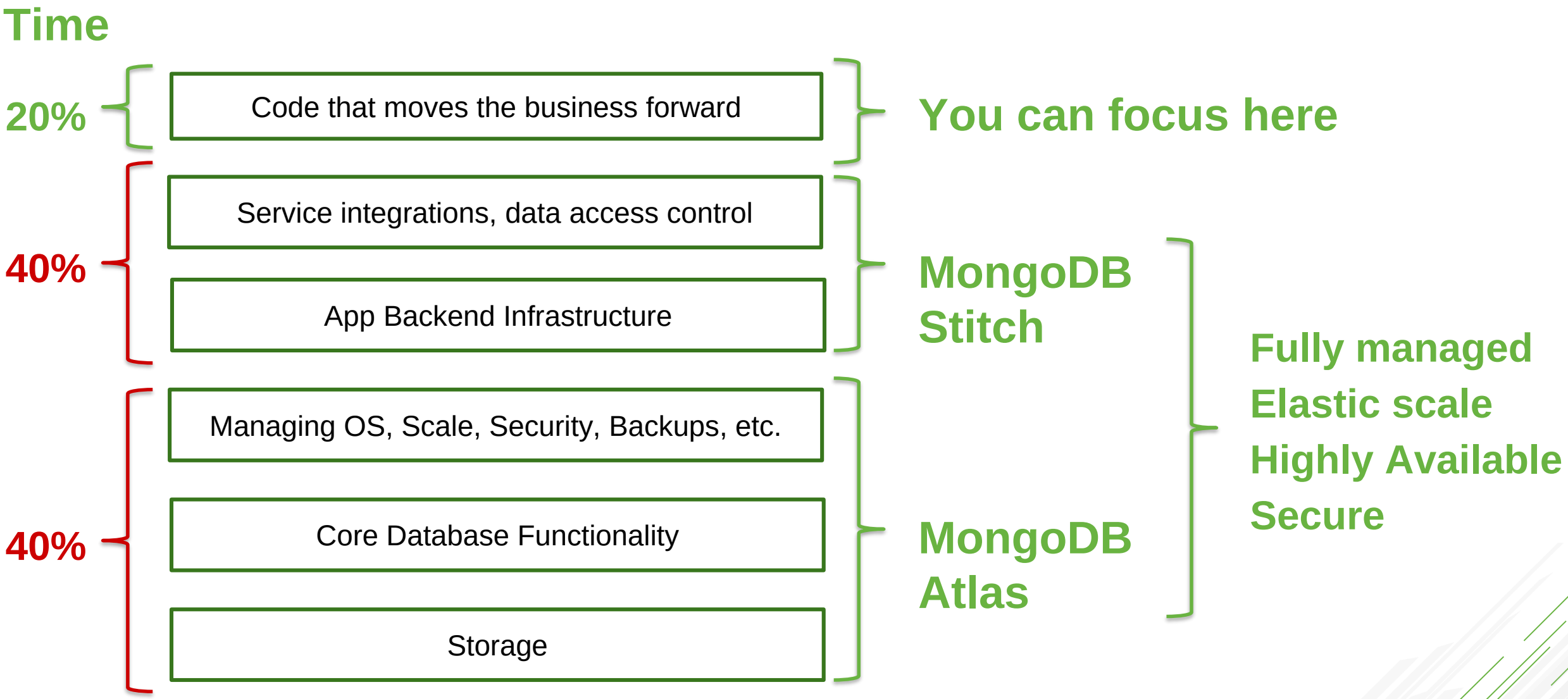


Focus Your Energy Where You Can Make a Difference



mongoDB

Focus Your Energy Where You Can Make a Difference



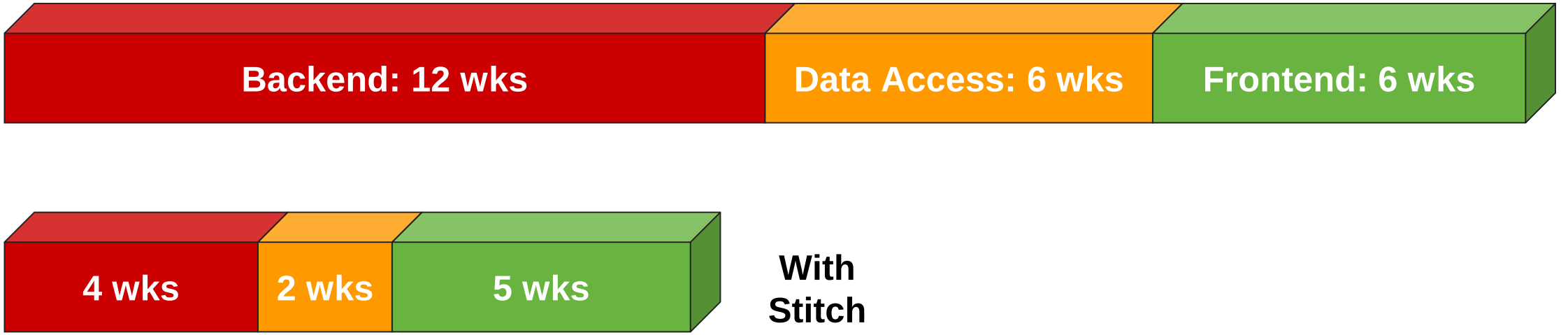
mongoDB

Save weeks of development and thousands of lines of code

	Without Stitch	With Stitch
Backend	Provision backend server Install runtime environment Add code to make backend HA Add code to scale backend Monitor & manage backend infrastructure Code REST API for frontend to use backend Code backend application logic	**Handled automatically by Stitch and Atlas** **Provide JS code for Stitch Functions**
Data Access	Code user authentication Code data access controls	**Simple JSON Config**
Frontend	Code application frontend Code against each external service API Continuously poll database for changes	**Code frontend using single SDK/API to access backend services**

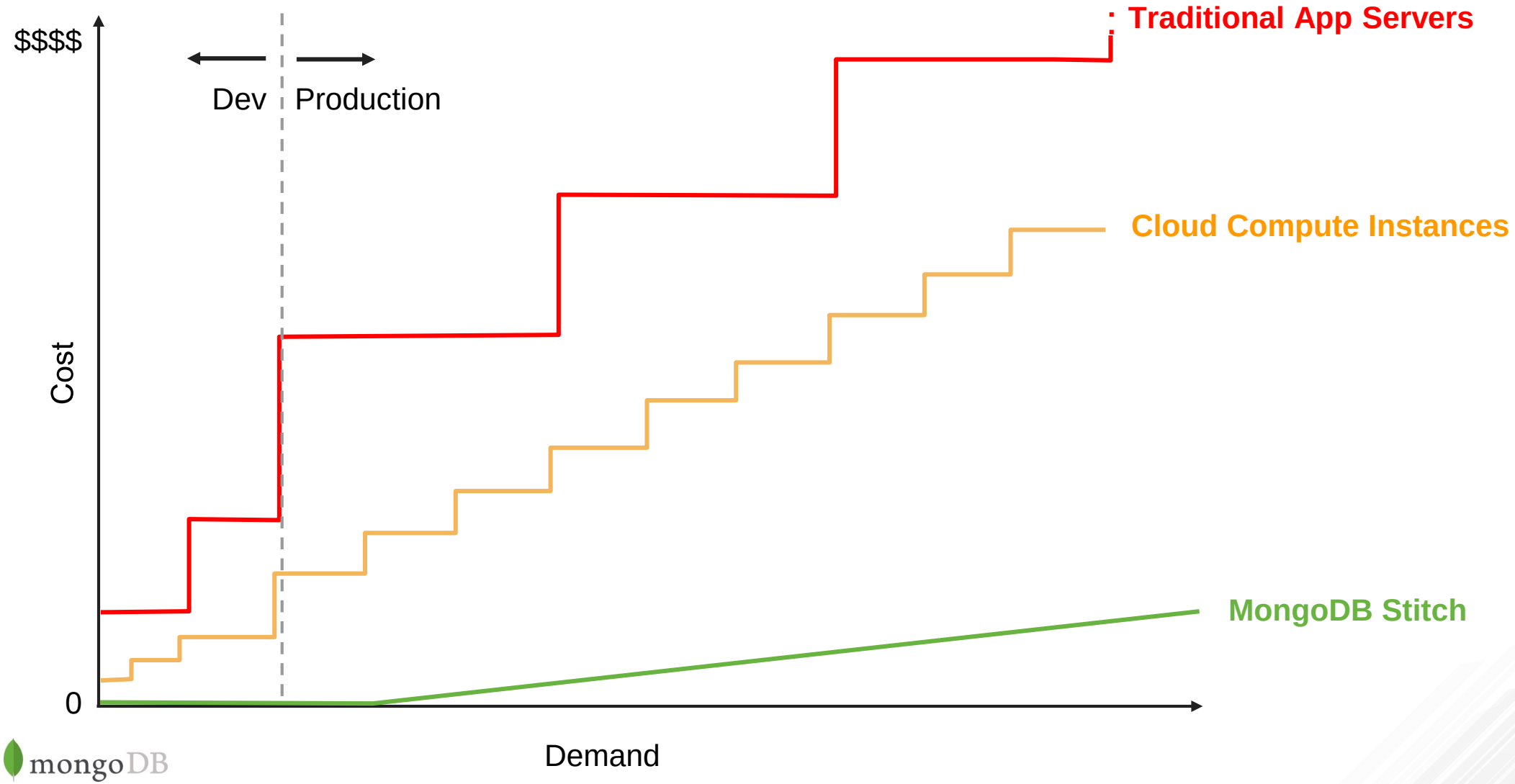
mongoDB

Development Time With/Without Stitch



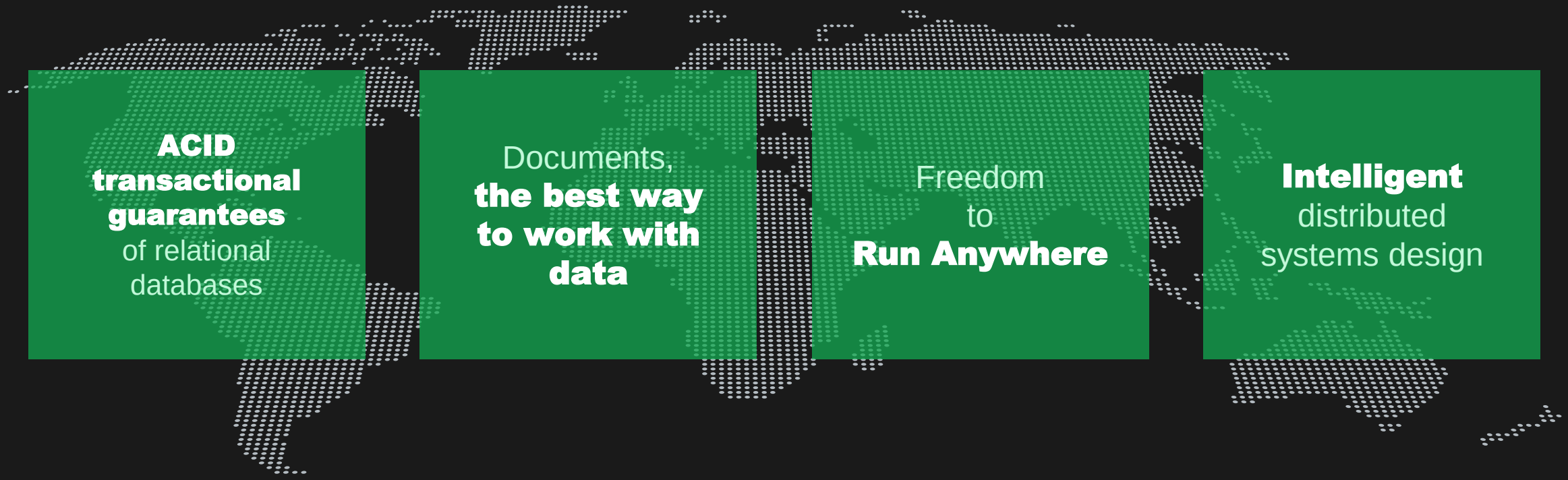
Cut development time in half

MongoDB Stitch Reduces Infrastructure Costs



mongoDB

Summary: MongoDB Uniquely Delivers…



APPENDIX

Hidden Operational Costs

Complicated Operational Process

1. **Provision servers**
2. **Configure high availability**
3. **Configure security**
4. **Configure and test backup**
5. **Install and configure monitoring and alerting**
6. **Ongoing upgrades and maintenance**

The HIDDEN COST of resources

1. Provision servers
2. Configure high availability
3. Configure security
4. Configure and test backup
5. Install and configure monitoring and alerting
6. Ongoing upgrades and maintenance

PEOPLE COSTS

DBAs
~ $150,000

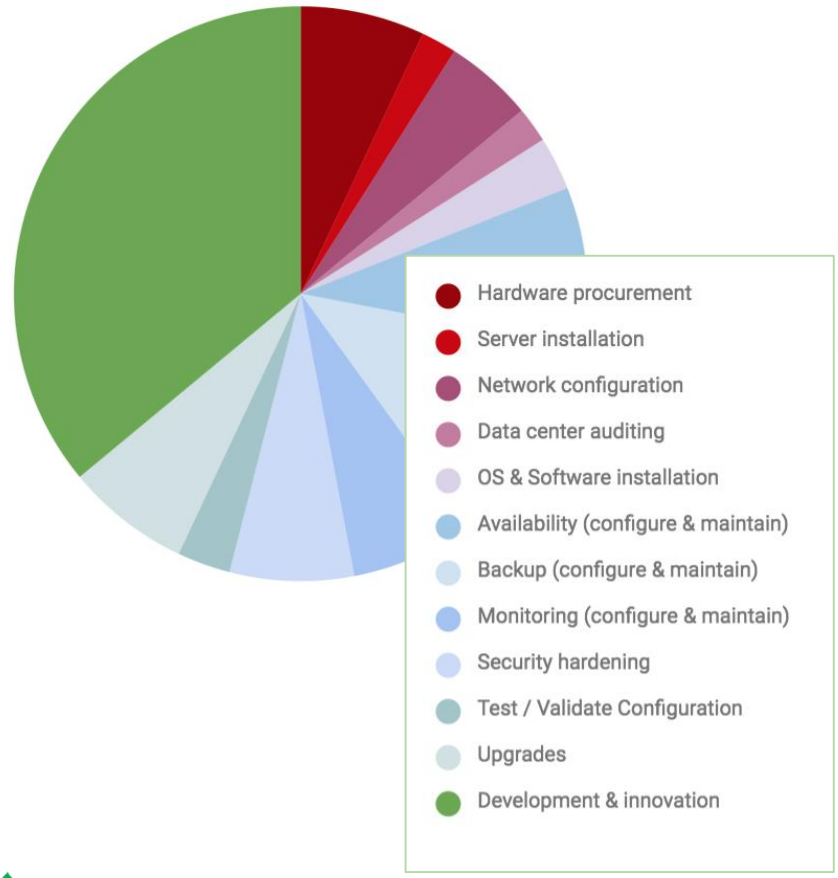
SysAdmin
~ $100,000

IT Security
~ $130,000

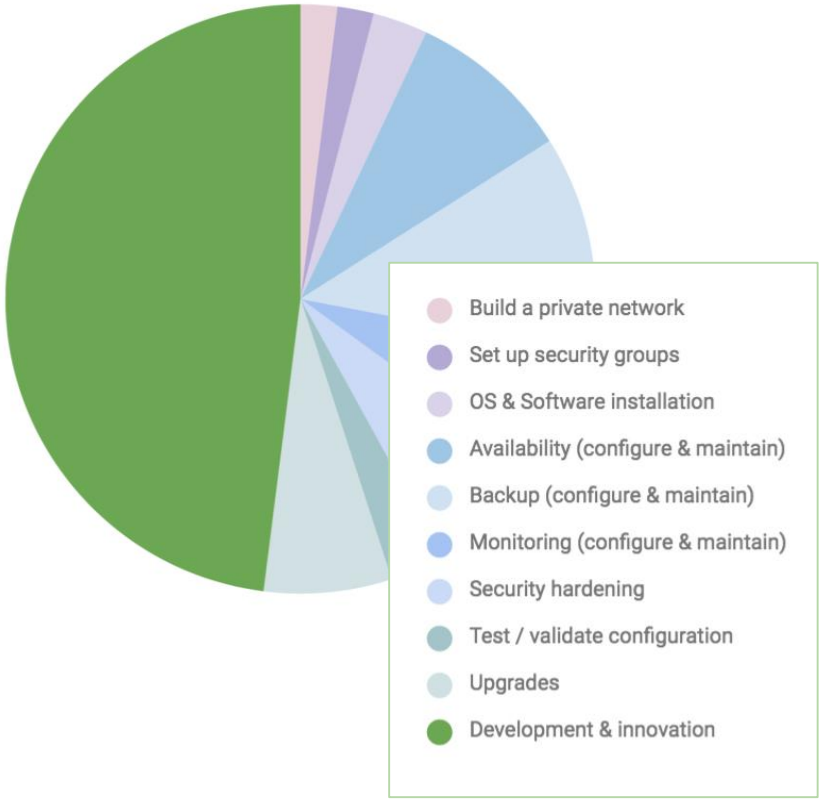
Developers
~ $250,000

Time spent on operations with different deployment models

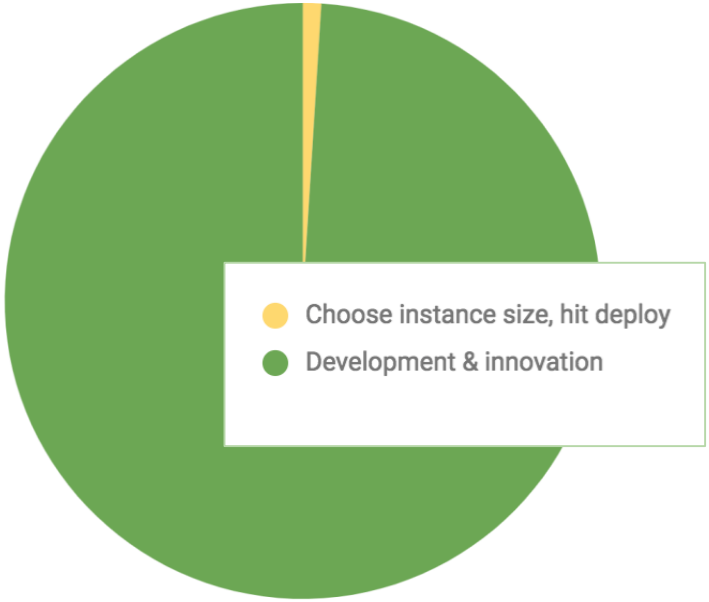
On-premises



Self-managed in the cloud

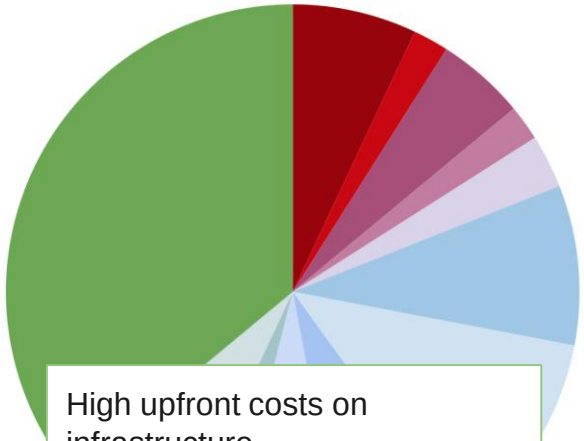


Database as a service



Operational & resource costs

On-premises

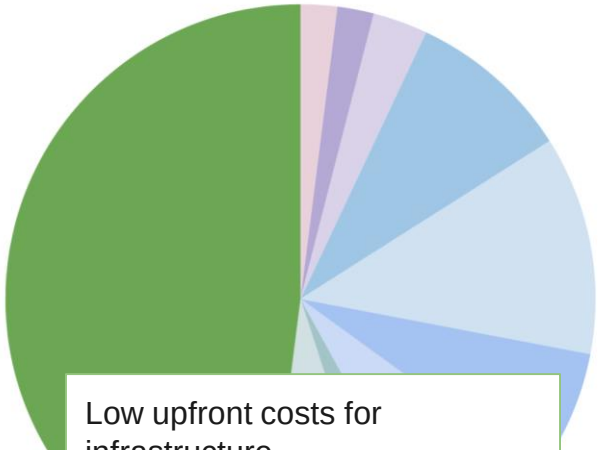


High upfront costs on infrastructure

High resource costs for installation and ongoing maintenance

Millions in operational and resource costs

Self-managed in the cloud

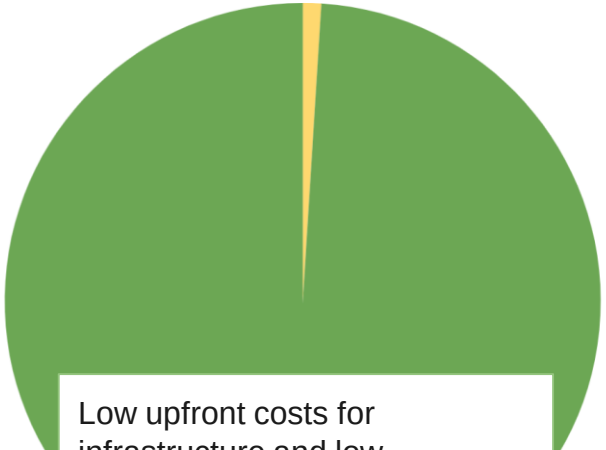


Low upfront costs for infrastructure

High resource costs for installation and ongoing maintenance

Significant resource costs as operational overhead is simply transferred to the cloud

Database as a service



Low upfront costs for infrastructure and low operational overhead

Self-service, elastic, and consumption-based service reduces TCO and accelerates time to value.

Aggregations

Advanced data processing pipeline for transformations and analytics using an Intelligent Operational Data Platform

```
db.orders.aggregate([
 {$match: {status: 'A'}},
 {
  $group: {
   _id: '$cust_id',
   total: {$sum: '$amount'}
  }
 },
  {
   $project: {
    customerId: '$_id',
    total: 1,
    _id: 0
    }
  }
])
```

Framework for Data Aggregation

- multiple stages

Similar to a unix pipe

- Build complex pipeline by chaining commands together

Rich Expressions

Examples using JavaScript shell

Fast at Scale

Cluster Scale		Performance Scale		Data Scale	
Entertainment Company	1,400 servers	Man AHL	250 Million Ticks / Sec	Adobe	**Petabytes**
Asian Internet Company	1,000+ servers	FOURSQUARE	300k Ops / Sec	Yandex	**10s of billions of objects**
EA SPORTS FIFA	250+ servers	Federal Agency	500k Ops / Sec	Baidu 百度	**200 billion documents / 1+ PB of data**

Slide Executions with Different Colored Backgrounds
Select best option that matches your projector

Flexible: Adapt to change

- **Polymorphic:** documents in a collection can contain different fields
- **Dynamic:** Modify schema without downtime: new app features or data sources
- **Govern:** JSON Schema to enforce structure

```
{
  "_id" : ObjectId("5ad88534e3632e1a35a58d00"),
  "name" : {
    "first" : "John",
    "last" : "Doe" },
  "address" : [
    { "location" : "work",
      "address" : {
        "street" : "16 Hatfields",
        "city" : "London",
        "postal_code" : "SE1 8DJ"},
      "geo" : { "type" : "Point", "coord" : [
         51.5065752,-0.109081]}},
+   {...}
  ],
  "phone" : [
    { "location" : "work",
      "number" : "+44-1234567890"},
+   {...}
  ],
  "dob" : ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund" : NumberDecimal("1292815.75")
}
```

MongoDB Version 2

```
save(Map m)
{
    collection.insert(m);
}

Map fetch(String id)
{
    Map m = null;
    DBObject dbo = new BasicDBObject();
    dbo.put("id", id);
    c = collection.find(dbo);
    if(c.hasNext()) }
          m = (Map) c.next();
    }
    return m;
}
```

NO CHANGE

Advantages:

1. **Zero time and money spent on overhead code**
2. Code and database not physically linked
3. New material with more fields can be added into existing collections; backfill is optional
4. Names of fields in database precisely match key names in code layer and directly match on name, not indirectly via positional offset
1. No technical debt is created

MongoDB Version 3

```
save(Map m)
{
    collection.insert(m);
}

Map fetch(String id)
{
    Map m = null;
    DBObject dbo = new BasicDBObject();
    dbo.put("id", id);
    c = collection.find(dbo);
    if(c.hasNext()) }
          m = (Map) c.next();
    }
    return m;
}
```

NO CHANGE

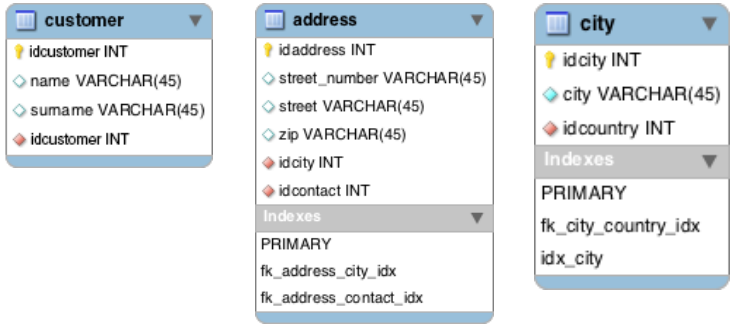
Advantages:

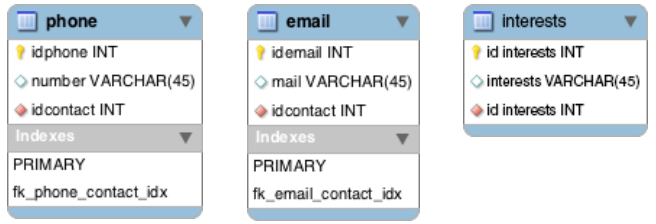
1. **Zero time and money spent on overhead code**
2. Code and database not physically linked
3. New material with more fields can be added into existing collections; backfill is optional
4. Names of fields in database precisely match key names in code layer and directly match on name, not indirectly via positional offset
1. No technical debt is created

Fast: To work with data

Compared to storing data across multiple tables, a single document data structure:

- Presents a single place for the database to read and write data
- Denormalized data eliminates JOINs for most operational queries
- Simplifies query development and optimization






```
  _id: 12345678
> name: Object
> address: Array
> phone: Array
  email: "john.doe.@mongodb.com"
  dob: 1966-07-30 01:00:00:000
v interests:Array
            0: "Cycling"
            1: "IoT"
```

Versatile: Rich query functionality

Expressive Queries	• Find anyone with phone # “1-212…” • Check if the person with number “555…” is on the “do not call” list
Geospatial	• Find the best offer for the customer at geo coordinates of 42nd St. and 6th Ave
Text Search	• Find all tweets that mention the firm within the last 2 days
Aggregation	• Count and sort number of customers by city, compute min, max, and average spend
Native Binary JSON Support	• Add an additional phone number to Mark Smith’s record without rewriting the document • Update just 2 phone numbers out of 10 • Sort on the modified date
JOIN ($lookup)	• Query for all San Francisco residences, lookup their transactions, and sum the amount by person
Graph Queries ($graphLookup)	• Query for all people within 3 degrees of separation from Mark

MongoDB

```
{       customer_id : 1,
        first_name : "Mark",
        last_name : "Smith",
        city : "San Francisco",
        phones: [       {
                        number : "1-212-777-1212",
                        type : "work"
        },
        {
                        number : "1-212-777-1213",
                        type : "cell"
        }]
.....
```

Single vs Multi-Document ACID Transactions

For many apps, single document transactions meet the majority of needs

```
  _id: 12345678
> name: Object
> address: Array
> phone: Array
  email: "john.doe.@mongodb.com"
  dob: 1966-07-30 01:00:00:000
v interests:Array
        0: "Cycling"
        1: "IoT"
```

Related data modeled in a single, rich document against which ACID guarantees are applied

Syntax*

```
with client.start_session() as s:
    s.start_transaction()
    try:
        collection.insert_one(doc1, session=s)
        collection.insert_one(doc2, session=s)
        s.commit_transaction()
    except Exception:
        s.abort_transaction()
```

*Python. Syntax is subject to change

Natural for developers

- Idiomatic to the programming language
- Familiar to relational developers
- Simple

Comparing Syntax with Relational

MySQL

```
db.start_transaction()
cursor.execute(orderInsert, orderData)
cursor.execute(stockUpdate, stockData)
db.commit()
```

mongoDB.

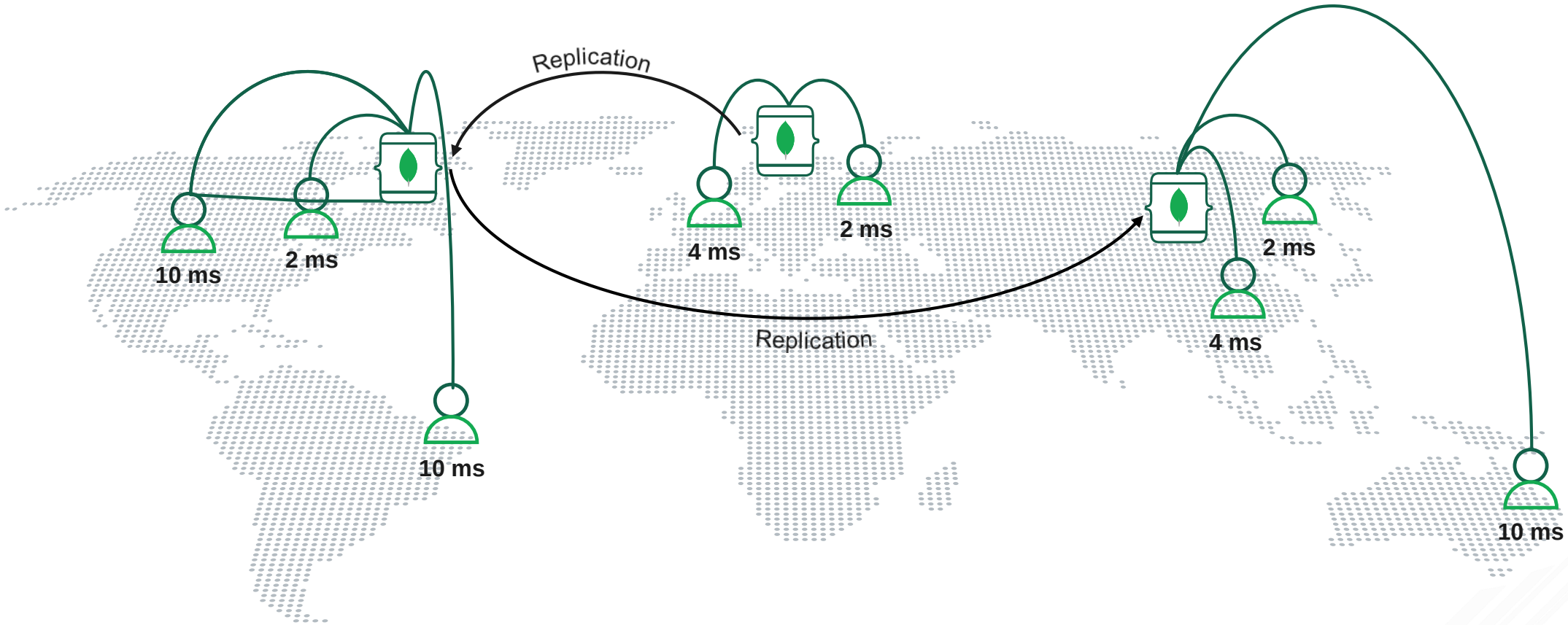
```
s.start_transaction()
orders.insert_one(order, session=s)
stock.update_one(item, stockUpdate, session=s)
s.commit_transaction()
```


*Python. Syntax is subject to change

Write Everywhere:
zoned sharding routes data to the regions where it’s needed

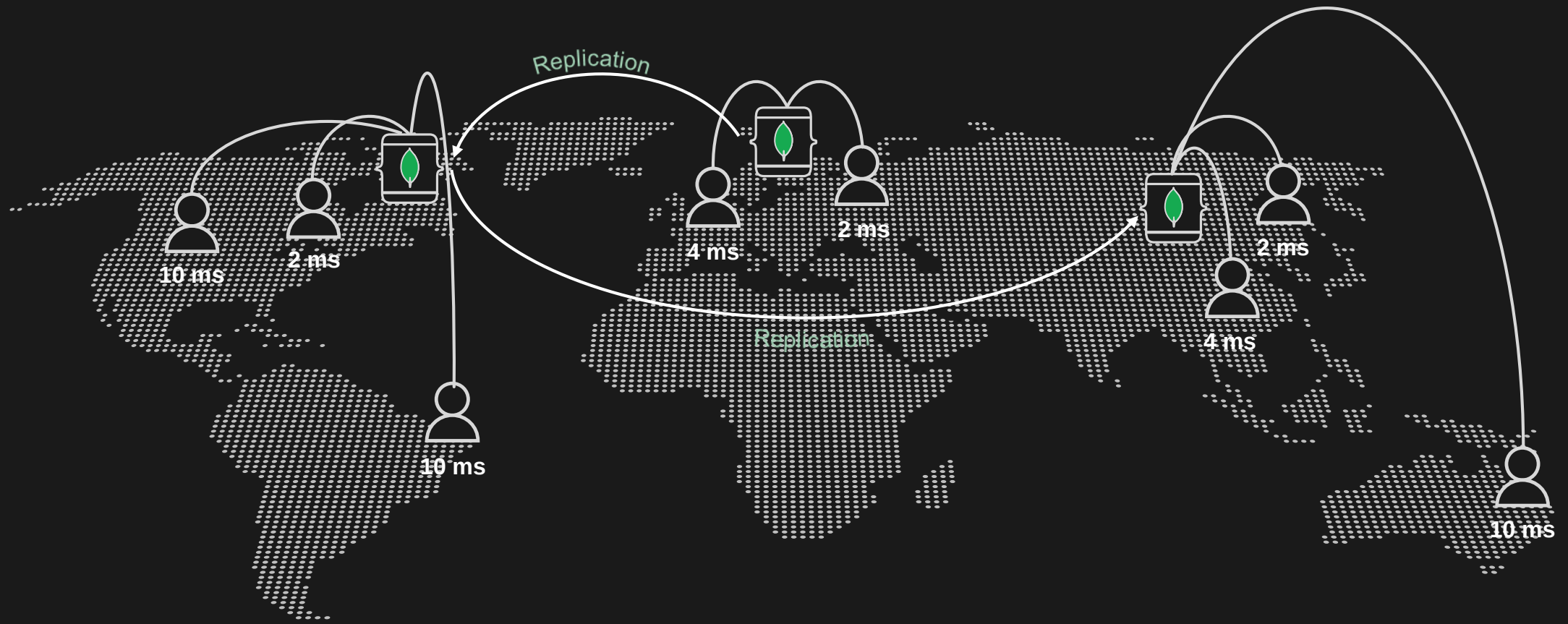
Summary: MongoDB Uniquely Delivers…

MongoDB's native replica sets puts your entire database right next to users

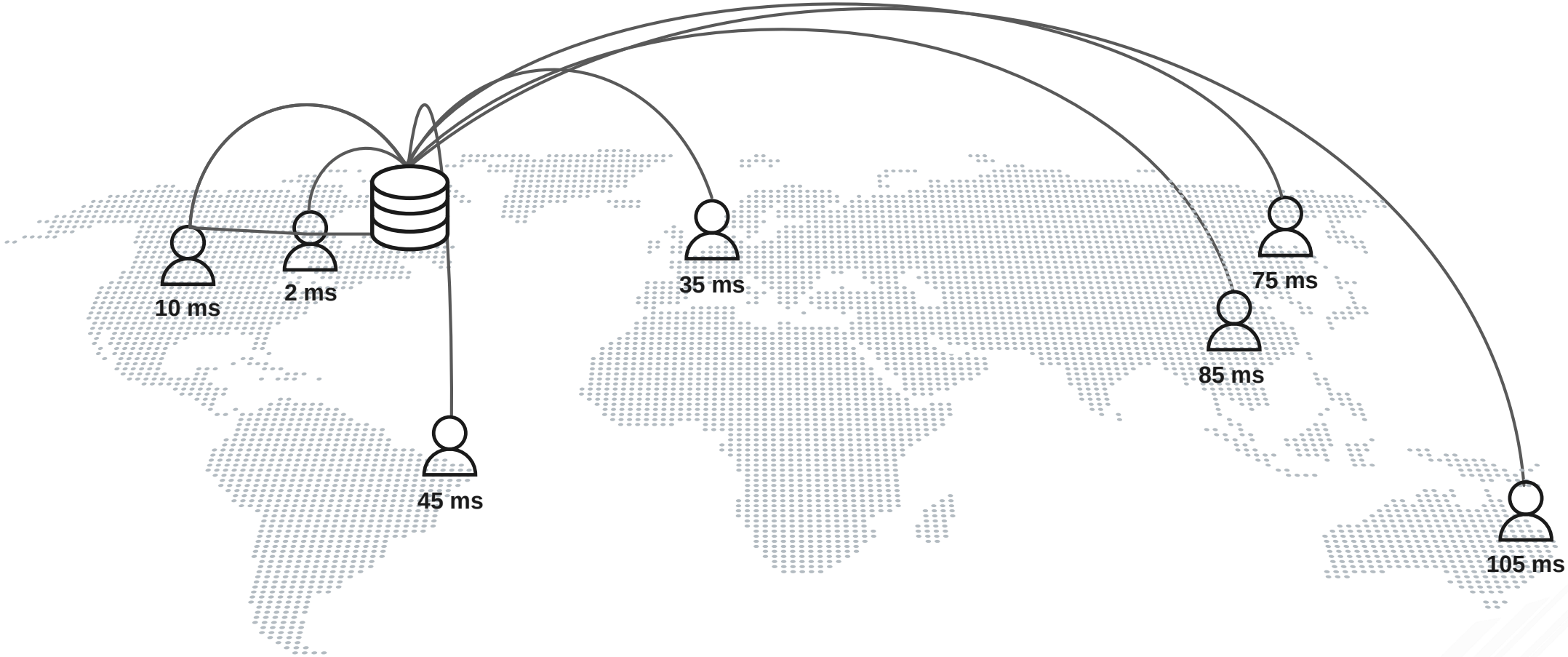


With higher map contrast

MongoDB’s native replica sets puts your entire database right next to users



Serving global audiences with relational databases



With higher map contrast

Serving global audiences with relational databases

Write Everywhere:
zoned sharding routes data to the regions where it's needed

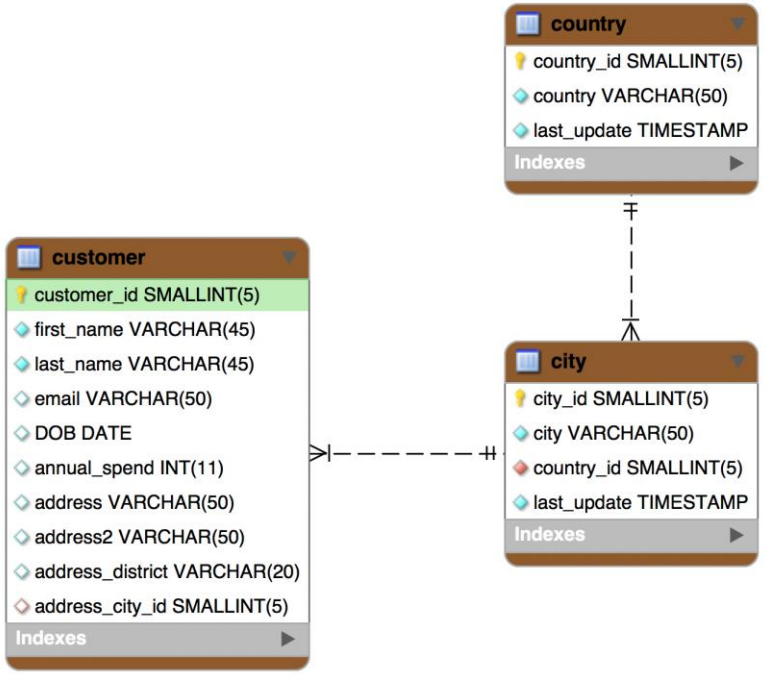
With higher map contrast

Write Everywhere:
zoned sharding routes data to the regions where it’s needed

Modeling a customer in a relational database

Customer ID
Name
Address (multiple lines)
Phone
eMail Address
DoB

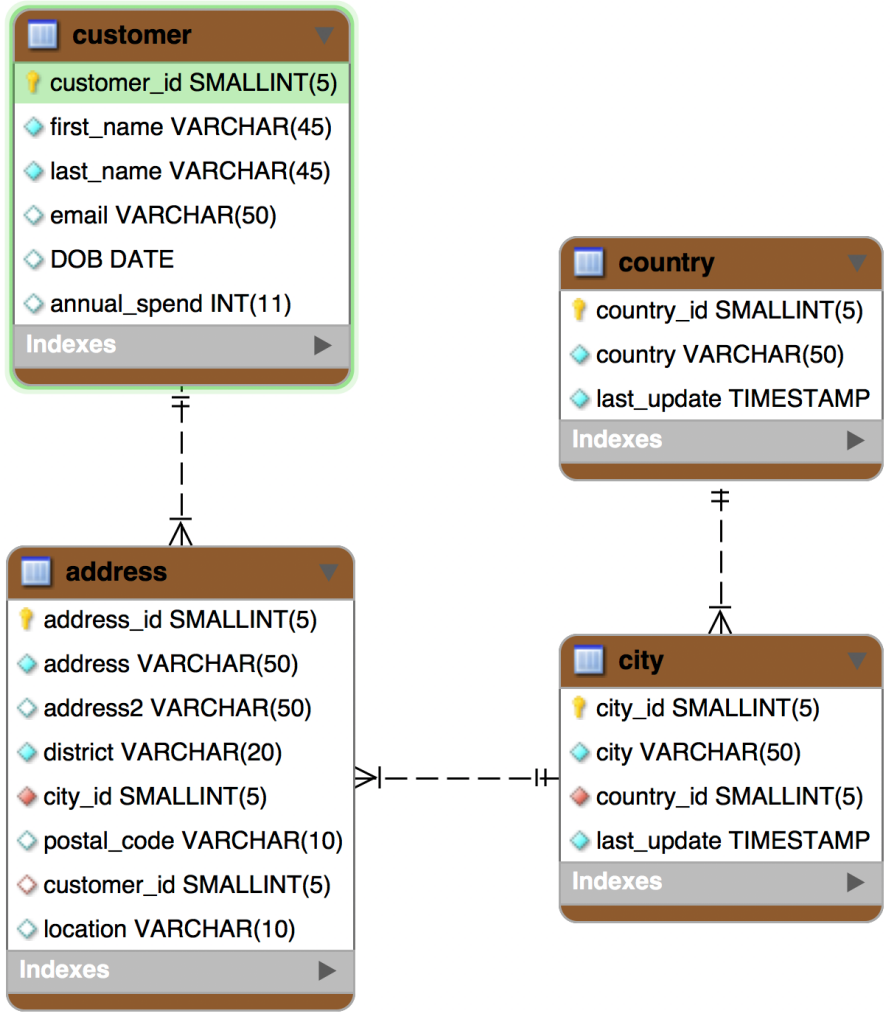
Simple Customer Data



Relational Data Model

Adding basic customer data requires a new table

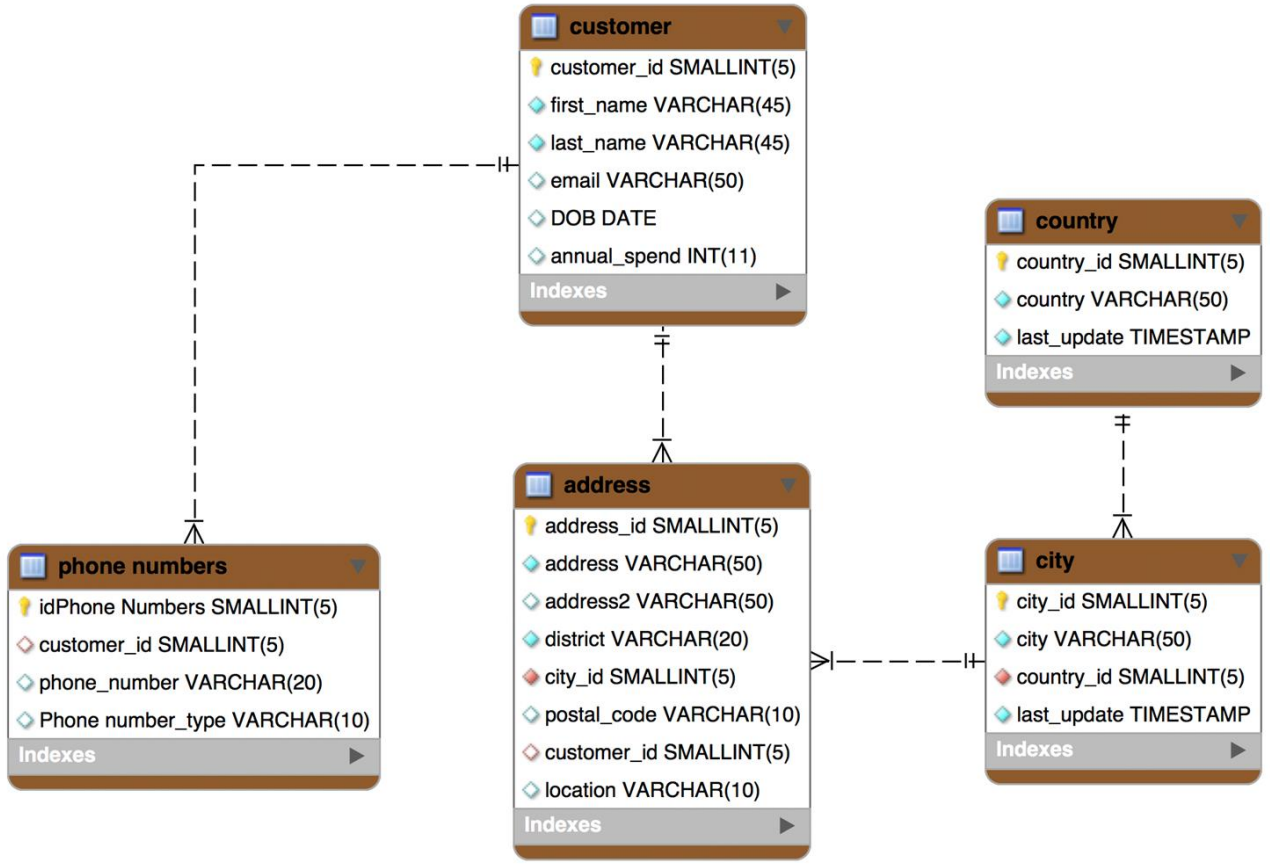
Customer ID
Name
Work Address (multiple lines)
Home Address (multiple lines)
Phone
eMail Address
DoB



Enriched Customer Data

Enriching data further complicates the data model

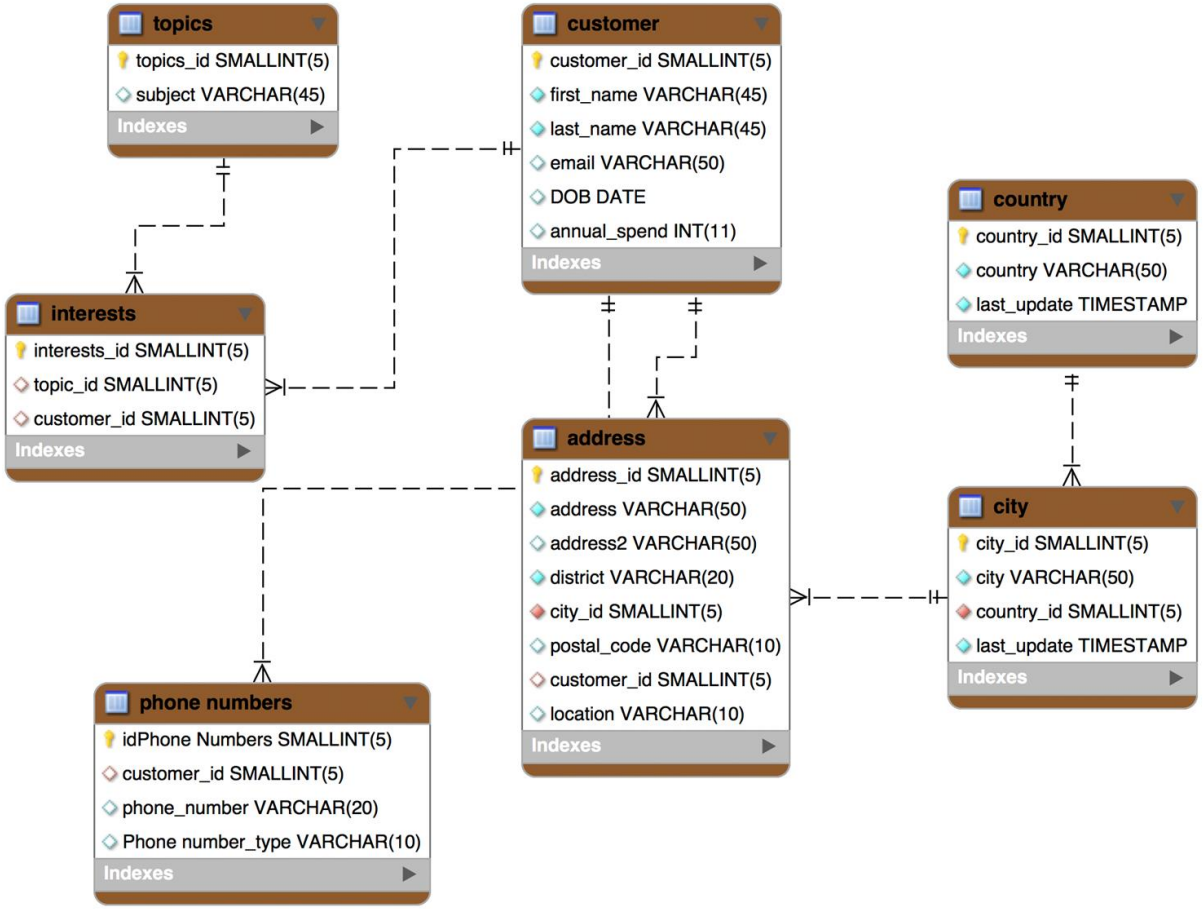
Customer ID
Name
Work Address (multiple lines)
Home Address (multiple lines)
Daytime Phone
Evening Phone
Mobile Phone
eMail Address
DoB



Enriched Customer Data

Quickly, the relational model becomes unwieldy

Customer ID
Name
Work Address (multiple lines)
Home Address (multiple lines)
Daytime Phone
Evening Phone
Mobile Phone
eMail Address
DoB
Interests



Enriched Customer Data