Application Architecture

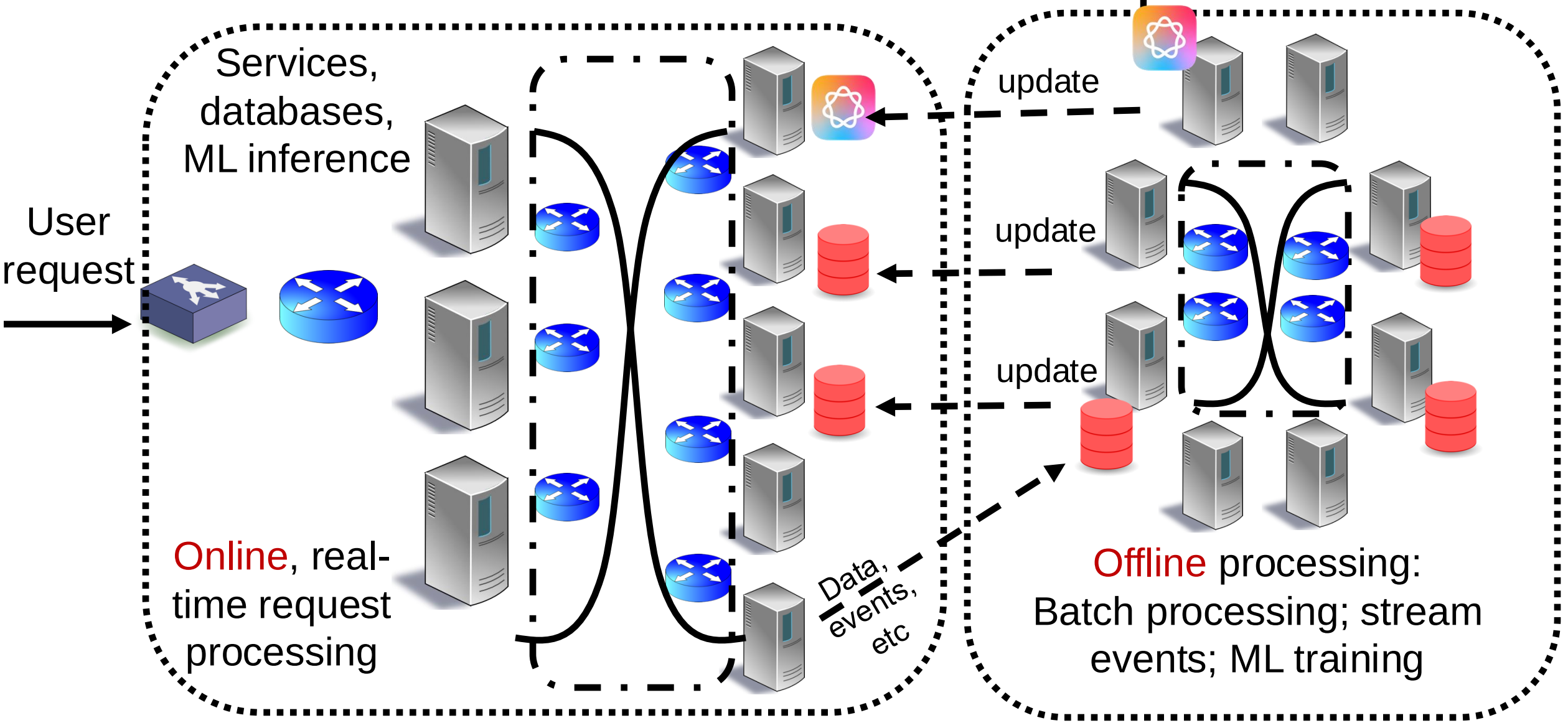
Lecture 7

Srinivas Narayana

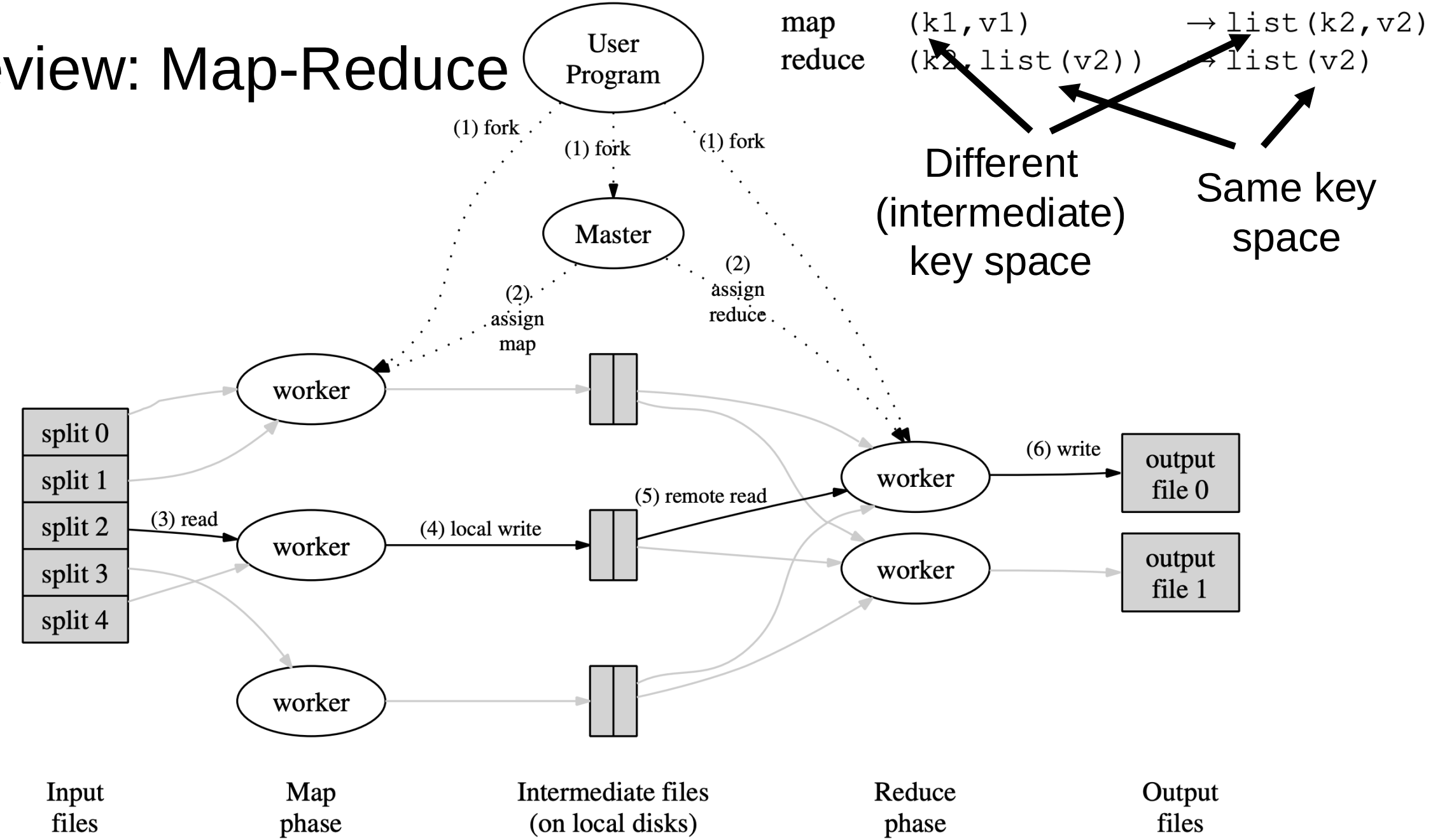
http://www.cs.rutgers.edu/~sn624/553-S25

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Review: Offline and Online components



Review: Map-Reduce

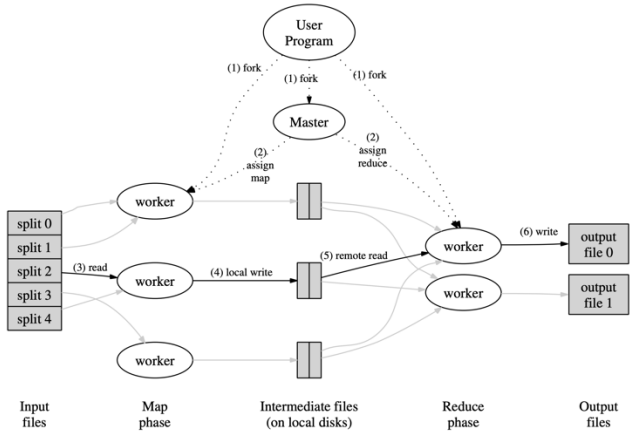


map (k1,v1) → list(k2,v2)
reduce (k2,list(v2)) → list(v2)

Different (intermediate) key space

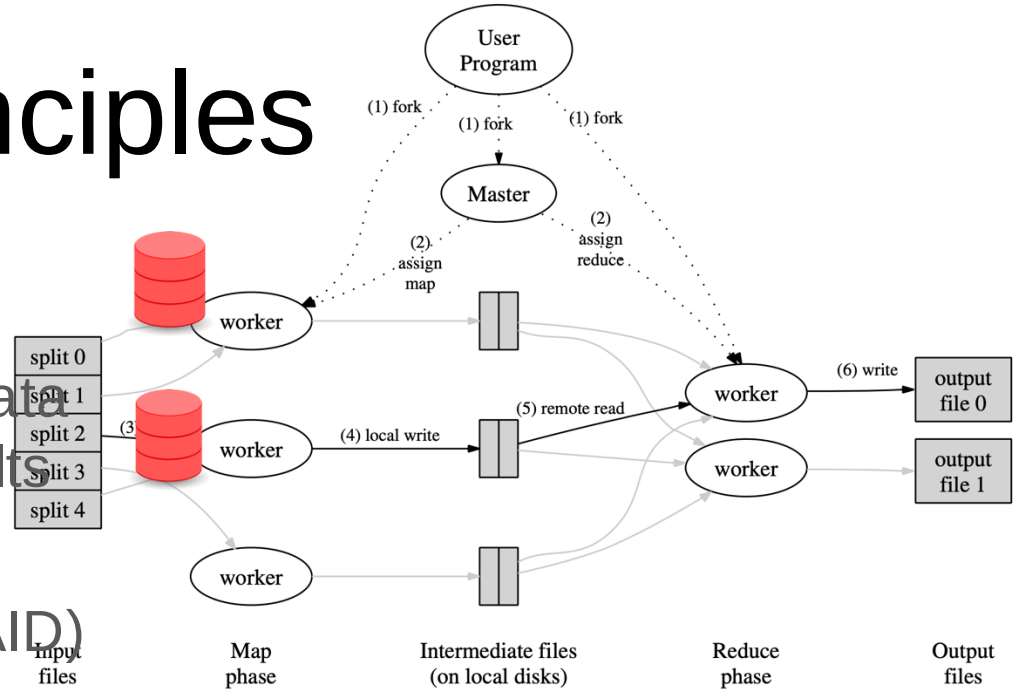
Same key space

Processing steps in MapReduce



- Input data consumed from a distributed filesystem
- Master ships code to the worker node closest to data, if possible (CPU, memory constraints permitting)
- Each mapper partitions its input data by the reducer key
  - Typically, through a hash function, e.g., hash (key) mod R == r
- Sort output data (per partition) by the key; run map function
- Reducers are informed of partial result at each mapper
- Reducer pulls files from mappers through RPC
- Output persisted to distributed filesystem (typically involves replication)
- Result: R output files in the DFS (one per reducer partition)

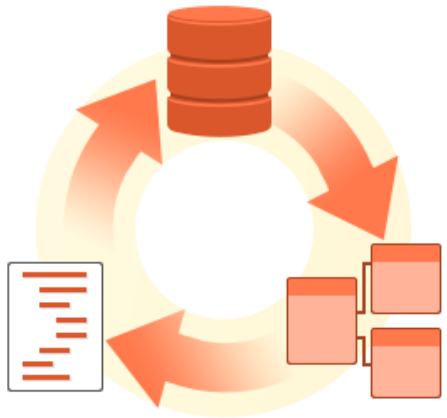
Implementation Key Principles



- Data locality
  - Reduce network bandwidth: ship code to data
  - Locally persist (not DFS) intermediate results
- Handle failures by re-doing compute
  - No fancy hardware fault tolerance (e.g., RAID)
  - Mapper failure: restart map job
  - Assume deterministic operations
  - Reducer failure (after completion): no problem (DFS)
- Identify and skip shards with deterministic faults
- Mitigate stragglers through eager replication of compute close to job completion
- Combiners at mapper: preliminary reduce for associative and commutative functions

More examples of using map-reduce

- Database Joins
  - Example: user activity (e.g. URLs) with user information (e.g. age)
- Grouping (GROUPBY) aggregations:
  - Count, sum, etc
  - Creating the sequence of events in a user session, determining whether e.g. a new version of a web page resulted in better sales
- Large distributed sorting
- Output sorting after mapper: important!



Building on Map-Reduce: (1) Workflows

- One Map-Reduce job isn't usually enough
- Google web search index: pipeline of 10 jobs; recommendation systems: 50—100
- Workflows: Chains of map-reduce jobs
  - E.g., one MR for counting requests by URL; another to sort count
- Explicit output files from each?
  - Like writing to file at the end of each tool in Unix pipeline
  - Materialization of the intermediate results needed?
- Stragglers make workflows slower
- Separate systems needed just to orchestrate the workflows correctly

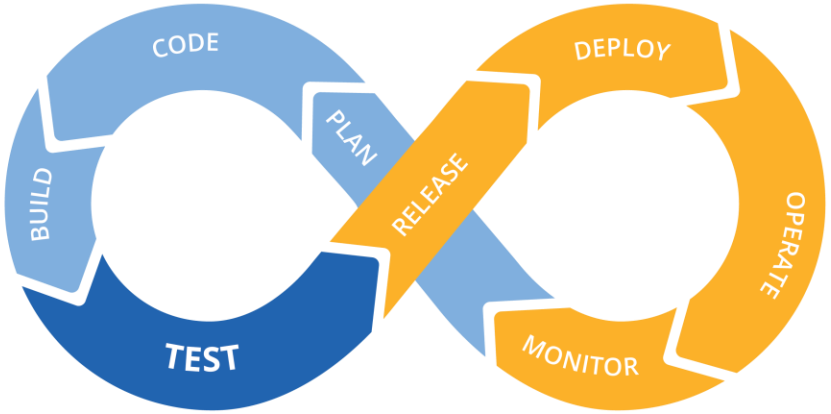
Building on Map-Reduce: (2) Dataflow

- Dataflow engines: handle the entire workflow
  - “Operators”: chain map-reduce functions
  - Only persist intermediate outputs to DFS when necessary
  - Chain reducers (no explicit mappers) when the key is the same
  - Don’t wait for stragglers of the previous job
- Stream Processing
  - Incremental execution of batch jobs when new data arrives
- Selectively materialize or recompute intermediate results
  - Lineages (RDD/Spark) or checkpoint

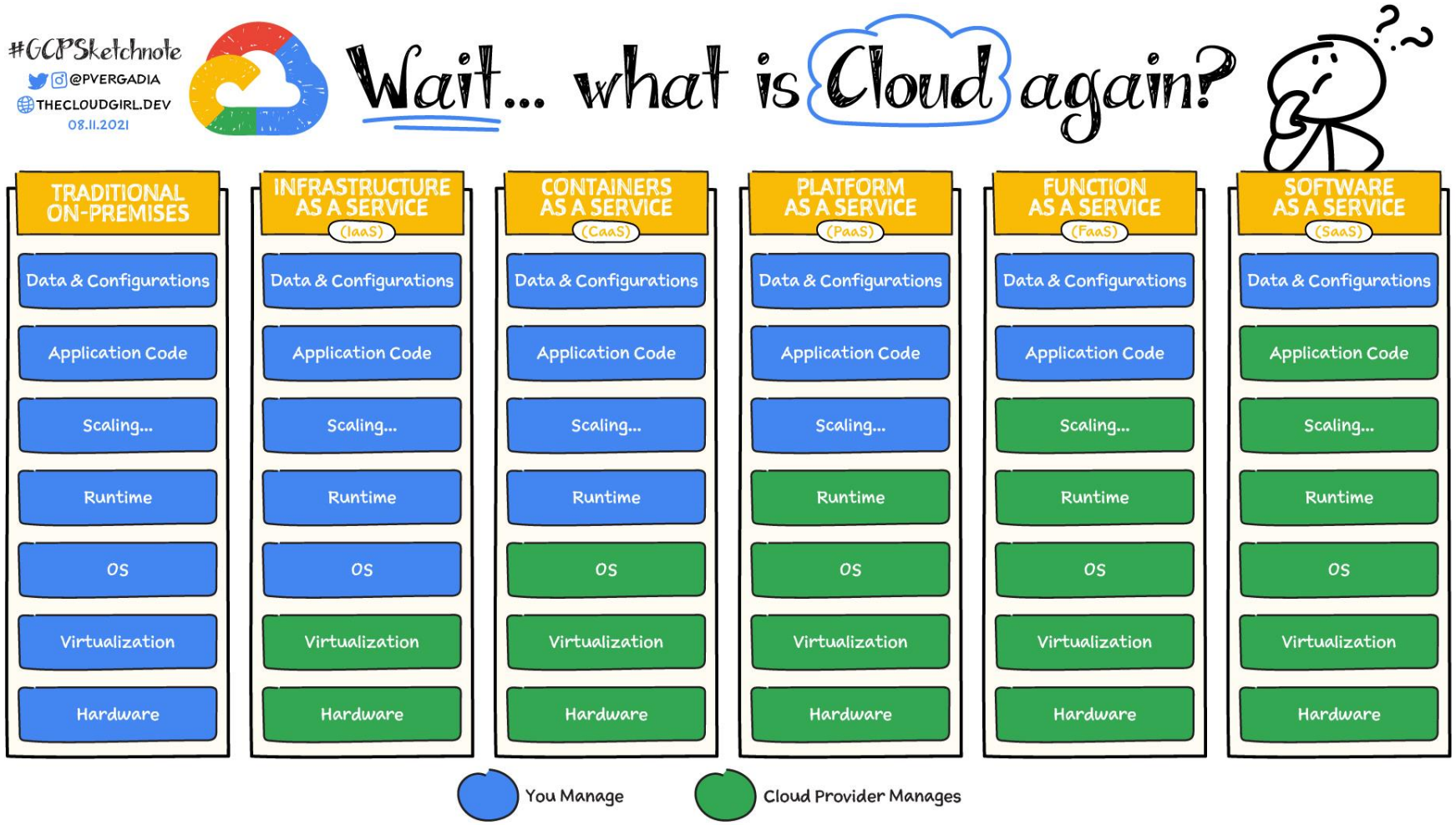
Serverless

Managing app deployments

- Consider a simple model to build and run your Internet application:
- Develop application in the language of choice
- Build & test
- Deploy over on-premise equipment (bare-metal servers) or cloud (virtual machines, containers, etc)
- Manage these resources



“_____ as a service”



https://cloud.google.com/learn/paas-vs-iaas-vs-saas

Serverless computing

- What if you could write code that deploys directly into a managed (cloud-like) environment?
- Programming model/abstraction reminiscent of functional programming
- Even those unfamiliar with such resource management can deliver (scalable, high performance, high availability, etc.) services
- Often only provide source code for the framework/runtime (e.g. python)
- Many commercial offerings available today (AWS, Azure, Google)
  - Bindings in Javascript, Python, C#, etc

Serverless Abstractions

Servers required
But someone else manages them

- Stateless functions
  - No persistent memory across invocations
- Event-triggered:
  - E.g. HTTP request
  - E.g. notification from a message router
- Computation decoupled from where/how it is run
  - Placement, scaling, and fault tolerance are managed by someone else
- Charged for execution time, not allocated resources
  - “pay as you go”
  - Scale down to zero cost when service is idle

Example (1/3)

```
import json
def hello(event, context):
    body = {
        "message": "Go Serverless v4.0! Your function executed successfully!",
    }
    response = {"statusCode": 200, "body": json.dumps(body)}
    return response
```

https://github.com/serverless/examples/blob/v4/aws-python-http-api/handler.py

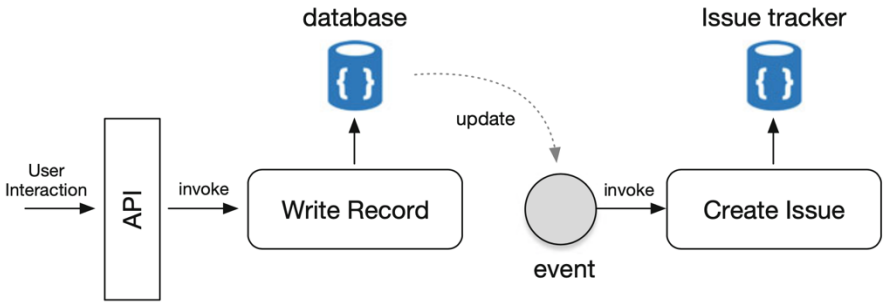
Example (2/3)

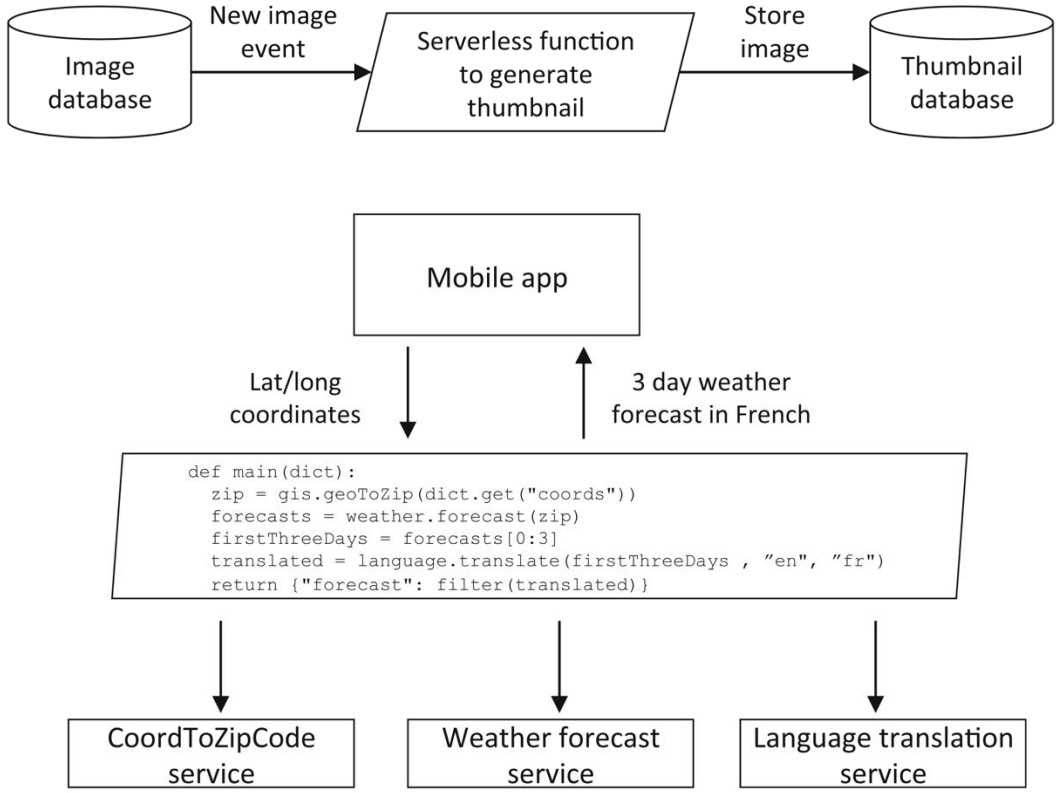
```
def lucky_number(event, context):
    upperLimit = 100
    number = random.randint(0, upperLimit)
    response = {
        'version': '1.0',
        'response': {
            'outputSpeech': {
                'type': 'PlainText',
                'text': 'Your lucky number is ' + str(number),
            }
        }
    }
    return response
```

https://github.com/serverless/examples/blob/v4/aws-python-alexa-skill/handler.py
https://github.com/serverless/examples

More examples (3/3)

- Event processing
- Offloading computation from mobile apps
- Coordination service





- Edge computation (CDNs, IoTs)
- Chatbots

Bursty, compute-intensive, short lived

Consequences of serverless abstraction

- Stateless: Database/storage integration becomes necessary
  - Storage options: ephemeral? Long-term? High IOPS? Low latency?
- Ecosystem of other tools:
  - Manage state, record logs, send alerts, perform authentication and authorization, messaging queues, cloud-based storage
  - Prone to vendor lock-in!
- Relinquish control of resource provisioning, monitoring, maintenance, scalability, fault tolerance, placement, logging, …
- Cost-driven decision making
  - Your billing tier depends on memory provisioned
  - Packages and software libraries that inflate memory size
  - Consequences of long polling, websockets?

Consequences of serverless abstraction

- Supporting stateful interactions (HTTP cookies, sessions)
  - Retrieve session state from storage across several connections/requests
  - Dedicated proxy to supply the session state
- How to compose multiple stateless functions?
  - Coordination server (always on, itself not serverless)
  - Use a messaging queue (message router/broker)
- May need frameworks and tools to turn existing high-level language code (e.g. legacy applications) into serverless functions
  - e.g. python decorator app.route(…) web-hook
- Unpredictable performance
  - Functions may run on different kinds of machines

What should the hosting platform do?

- Take an event sent over HTTP or from an event source
  - Determine which function(s) to which to dispatch the event
  - Send event to a function instance
  - Wait for a response, make it available to the user
  - Gather execution logs
  - Stop function when idle
- Isolate functions
- Orchestrate, scale
- Fault tolerance
- Locality: code, data, package