

Serverless Computing



NORDICAPIS.COM

Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu
Télécom SudParis, IMT, IP Paris, Inria

Do we really want Containers?

- An **environment** is required
 - Overhead of building containers and pods
- A management layer is required
 - Overhead of configuring service availability (orchestrator)
- **Backends** are required
 - Overhead of management of support features
 - Database servers, monitoring...
 - Always-on servers
 - May scale down to 0, but then latency overhead on next request

Introducing: Serverless

- Real cloud-native applications: only provide **code** to implement business core features
 - A continuation from micro-services
- Management and execution service provided by the cloud platform
 - From execution environment to service availability
- Serverless = **Function-as-a-Service** + **Backend-as-a-Service**

Backend-as-a-Service (BaaS)

- **Common backend components** in applications architectures
 - Database servers, messages queues, (object) storage, Pub/Sub services...
- A good thing: better served by the cloud provider
 - Mutualized, no overhead for the user, always available
 - Provides an ecosystem of ready-to-use components
 - But beware of vendor lock-in!
- **Elasticity** requirements: scale in synchronization with the FaaS workload (*see next*)
 - dynamically
 - quickly
 - up and also down to zero

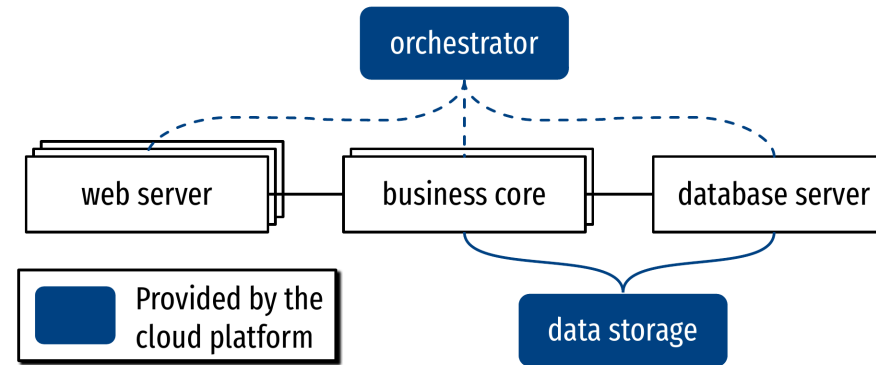
Function-as-a-Service (FaaS)

- Run application code without fixed long-lived servers
 - Execution runtimes are spawned on-demand
 - Fully managed by the cloud FaaS platform
- Unit of execution, and unit of application architecture: **a function**
 - I.e., a singular feature of the application
- Applications are (mostly) **event-driven**
 - Triggered by requests or events from BaaS sources
 - Mainly **asynchronous** processing
- Parallelism at the level of the cloud function, managed by the platform
 - *Technically, concurrency is possible inside the cloud function*
- Core feature of serverless

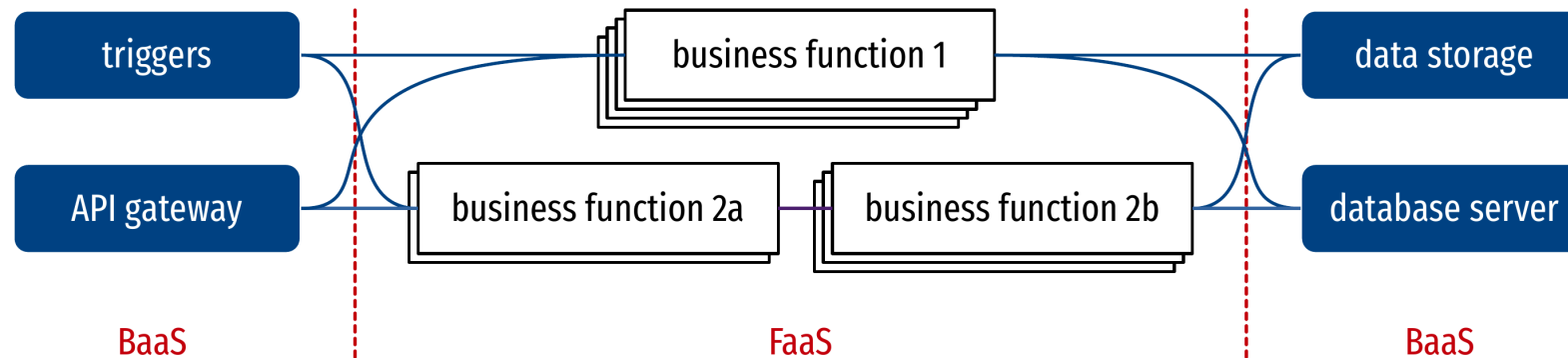
Comparison with micro-services

Application architecture

- API gateway: managed routing of HTTP REST requests to functions
- Triggers: HTTP request, message queue, event stream, complex orchestration, timer, storage, etc.



Micro-services architecture.



Serverless architecture.

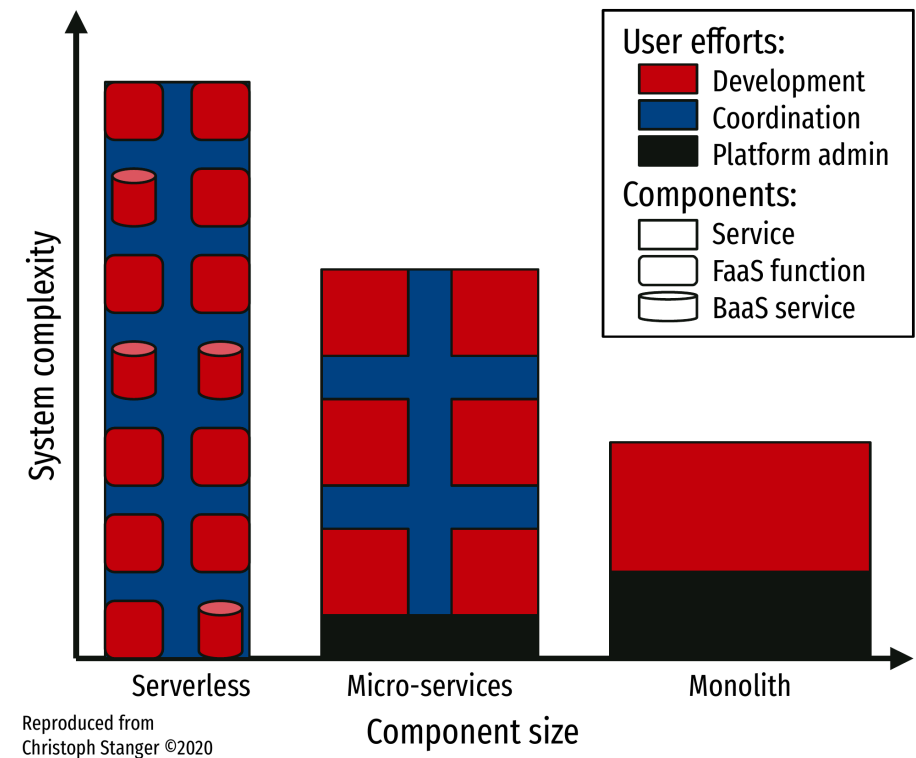
Consequences of Function-as-a-Service

Benefits

- **Elasticity:** granularity of the request handler allows more precise scaling
 - Quick scaling, each function can scale down to 0
- **Deployment:** just write and reuse code, and upload it
 - Quick experimentation, quick update
 - Easy to assemble (pluggable BaaS components)
- **Cost:** pay only the compute time you need
 - No request = no running function = no used resource = no cost
 - $\text{Cost} = \text{Compute time} \times \text{Reserved Memory}$ (approximately)

Drawbacks

- **Complexity:** many “moving pieces”
- **Spread code base:** requires good DevOps
- **Performance:** contention, *cold start*, access to state



Compared system complexity of architectures: monolithic, micro-services, serverless.

Demo: Apache OpenWhisk

1. Create a new function
2. Manually invoke a function
3. Use the API gateway to invoke a function as the backend to REST requests
4. Use triggers to integrate function invocation
5. Present warm and cold starts



APACHE
OpenWhisk

Apache OpenWhisk logo.

Example of FaaS Function

```
from platform_api import KeyValueStore

def main(args, env):
    """Entrypoint to the cloud function

    :param args: dictionary (JSON) of arguments passed in the request
    :param env: dictionary (JSON) of environment variables
    """
    name, place = args['name'], args['place']

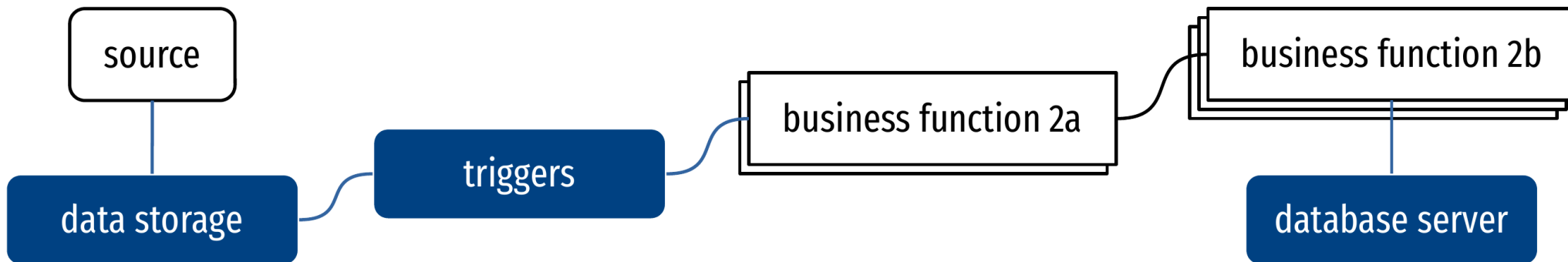
    # Totally fictitious example of using a key-value store from BaaS.
    with KeyValueStore(env['TOKEN']) as cnx:
        age = cnx.get('age', {'id': name})

    msg = f'Hello {name} from {place}!'

    # Return a result (a dictionary, translated to JSON).
    return {
        'greetings': msg,
    }
```

Application architecture with Function-as-a-Service (1/2)

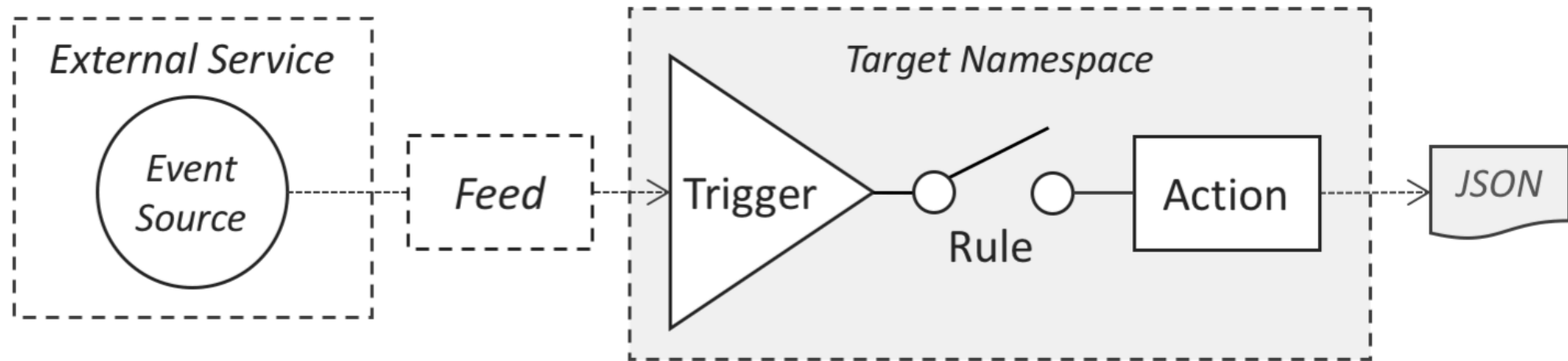
- Most often following the paradigm of **Extract – Transform – Load** (ETL)
 1. Get data
 2. Process data
 3. Output data
- Fit for event-driven processing
 - Execute when a request arrives or a trigger is fired
 - Asynchronicity is central to avoid **double-billing**
 - Double-billing happens when Fn. A queries Fn. B then waits for its result: both are billed.
- **Stateless** functions: no single instance is tied to a request
 - Consecutive invocations may be served by any instance (including new ones)
 - FaaS uses BaaS to store business data
 - Also hybrid architecture: VMs + containers + FaaS
 - FaaS relies on the API gateway or on the client to keep transient state



Example of the Extract – Transform – Load paradigm in Serverless computing.

Application architecture with Function-as-a-Service (2/2)

- Most often following the paradigm of **Extract – Transform – Load** (ETL)
 1. Get data
 2. Process data
 3. Output data
- Fit for event-driven processing
 - Execute when a request arrives or a trigger is fired
 - Asynchronicity is central to avoid **double-billing**
 - Double-billing happens when Fn. A queries Fn. B then waits for its result: both are billed.
- **Stateless** functions: no single instance is tied to a request
 - Consecutive invocations may be served by any instance (including new ones)
 - FaaS uses BaaS to store business data
 - Also hybrid architecture: VMs + containers + FaaS
 - FaaS relies on the API gateway or on the client to keep transient state

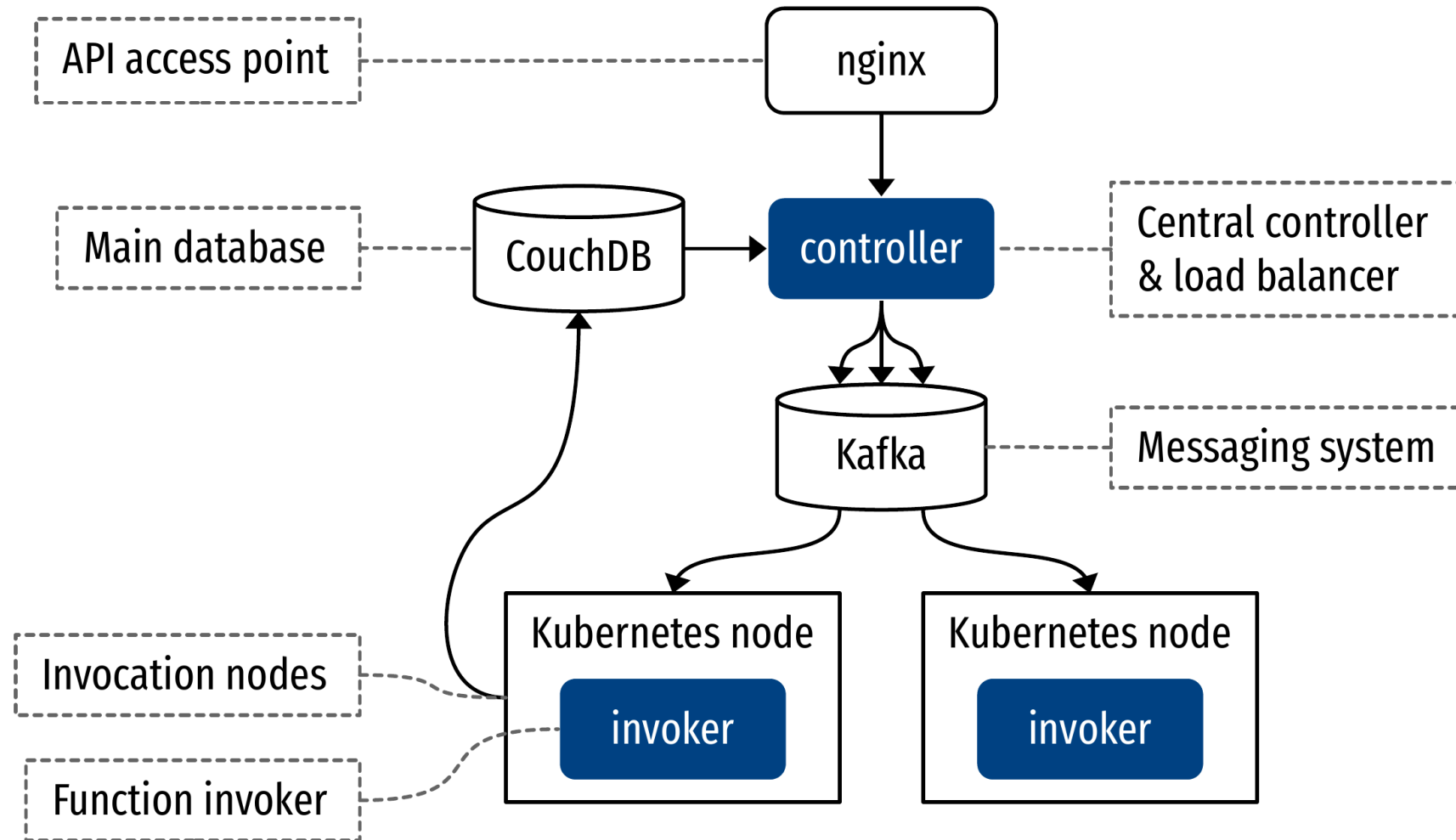


Programming model of Apache OpenWhisk.

What makes a FaaS application

- Naturally multi-process thanks to platform-level parallelism
- **Chains** and **graphs** of processing dependency between functions
 - Chain function calls to implement complex features with fine control granularity
 - Next level: handle a dependency graph (MapReduce pattern, etc.)
 - Some vendors integrate it, often must be done manually
- In practice: platform-level configuration + code pieces
- To execute functions:
 - Use provided environments: NodeJS, Python, Java...
 - Or provide a custom environment: Docker images, opaque binary executables

Internals of Apache OpenWhisk



Architecture of Apache OpenWhisk.

Invocation of a function

1. Control authentication, rights to invoke, housekeeping...
2. Spawn a new container with the function's runtime
3. Inject the function's code
4. Execute the function with the request's parameters
5. Retrieve the function's result
6. Destroy the container
 - **Optimizations** for steps 2 and 6

Management of function runtime containers

- Creating a new runtime container is slow (compared to serving a request)
 - Hundreds of milliseconds vs. execution time of 100ms
- Solution: **reuse existing containers!**
 - Functions are stateless: all containers with a function's runtime are equivalent
- **Cold starts** and **warm starts**
 - No runtime container available: cold start
 - Available runtime (ready and idle): warm start
- Cold starts are around **40 times slower!**
 - Depends on function runtime, language...
- Importance of managing the runtime container pool
- Concerns:
 - Maintain a pool of pre-warmed containers to speed up first requests
 - But balance between occupied resources and execution latency
 - Pre-warmed containers use resources that are not billed to the user!

Limits of serverless computing

- **Latency:**
 - cold starts
 - overhead of FaaS platform
- **Compatibility** with serverful applications
 - What about stateful applications?
 - No per-request local state
 - What about massively parallel applications?
 - Strong isolation between functions makes MPI hard
 - FaaS is not fit for long-running processing
 - Costs more, is less efficient
- Still a recent cloud paradigm!
 - Promising for easy cloud access and cloud-native applications

Serverless computing

- Function-as-a-Service for core business code and features
- Backend-as-a-Service to provide architectural services
- **Cloud-native paradigm**
 - Fine-grained, elastic, pay-as-you-go, no-management-overhead
- But not suited to all applications
 - Yet?