Serverless Data Processing & Cloud-Native Query Engines

Part 1

Introduction to Serverless Computing

- Cloud provider dynamically manages the allocation and provisioning of servers
- Serverless applications run in stateless compute containers
  - event-triggered, ephemeral, fully managed by the cloud provider

Historical Context

- **2014**: AWS introduces Lambda, the first mainstream FaaS offering
- **2016**: Google Cloud Functions and Azure Functions launch
- **2017**: Kubernetes-based serverless platforms emerge (Knative)
- **2018**: Standardization efforts begin (CloudEvents)
- **2019-Present**: Enterprise adoption accelerates, specialized serverless data services emerge

Evolution of Computing Models

Traditional On-Premises	Infrastructure as a Service (IaaS)	Platform as a Service (PaaS)	Function as a Service (FaaS)
• Physical servers in data centers • Capital expenditure model • Manual scaling and maintenance • Fixed capacity • Long procurement cycles • High operational overhead	• Virtual machines in the cloud • Pay for allocated resources • Self-managed operating systems • Manual or auto-scaling • Reduced procurement time • Moderate operational overhead	• Managed application platforms • Simplified deployment • Limited configuration options • Platform-level scaling • Focus on application code • Reduced operational overhead	• Code-level abstraction • Event-driven execution • Automatic scaling to zero • Pay-per-execution • No infrastructure management • Minimal operational overhead

Comparing Cloud Computing Models

Aspect	On-Premises	IaaS	PaaS	FaaS/Serverless
Infrastructure Management	Full	Partial	Minimal	None
Scaling Responsibility	User	User	Shared	Provider
Pricing Model	CapEx	Reserved/On-demand	Reserved/On-demand	Pay-per-use
Utilization Efficiency	Low	Medium	High	Very High
Operational Overhead	High	Medium	Low	Very Low
Development Focus	Infrastructure + App	App + Config	App	Business Logic
Cold Start	N/A	N/A	Minimal	Yes
Vendor Lock-in	Low	Medium	High	Very High

Key Characteristics of Serverless Computing

No Server Management

- Infrastructure is completely abstracted
- No provisioning, patching, or maintenance
- No operating system management
- No capacity planning

Auto-scaling

- Automatic scaling from zero to peak demand
- Instantaneous response to load changes
- No pre-provisioning required
- No scaling configuration needed

Key Characteristics: Cont.

Pay-per-use

- Billing based on actual resource consumption
- No charges when code is not running
- Typically measured in milliseconds
- Resource-based pricing (memory, CPU)

Event-driven

- Functions triggered by events
- Common event sources:
  - HTTP requests,
  - Database changes,
  - File uploads,
  - Message queues,
  - Scheduled triggers

Key Characteristics: Execution Model

Stateless Execution

- Functions typically don't maintain state between invocations
- State must be externalized (databases, object storage)
- Each invocation is independent
- Enables horizontal scaling

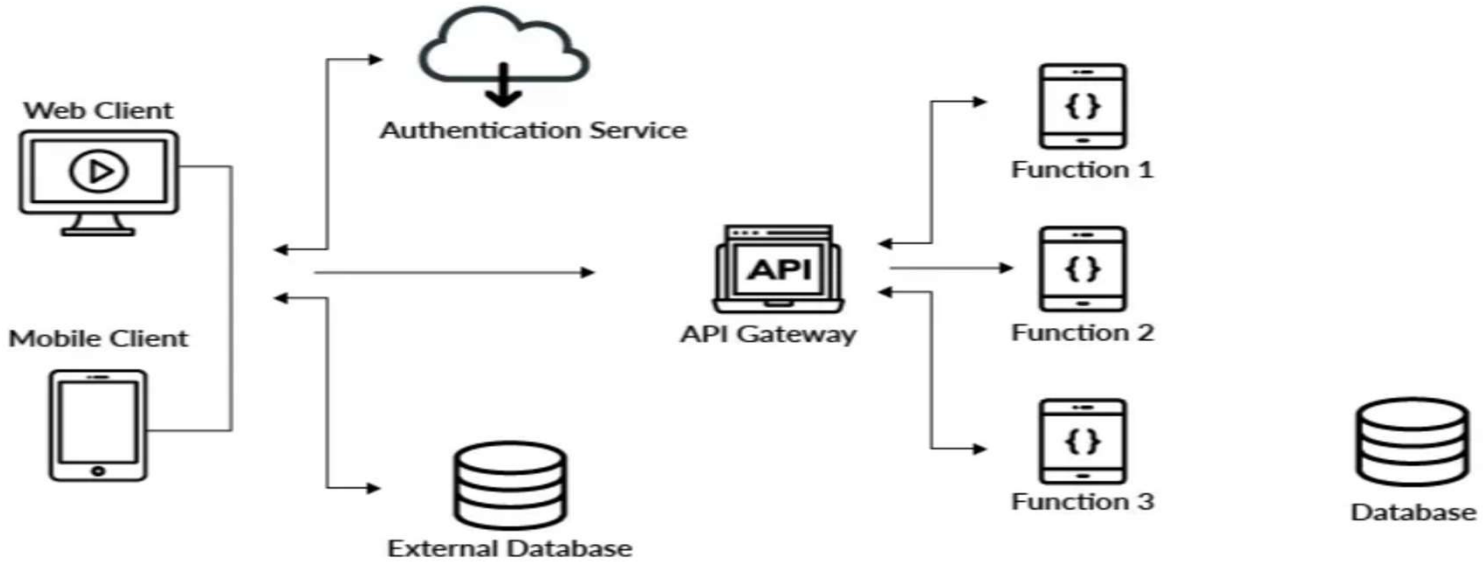
Short-lived Execution

- Functions designed for brief runtime
- Execution time limits (seconds to minutes)
- Not suitable for long-running processes
- Encourages microservice design patterns

Cold Starts

- Latency when initializing new function instances
- Varies by platform, language, and configuration
- Mitigation strategies available
- Trade-off for auto-scaling to zero

Serverless Computing Architecture



Source: https://appinventiv.com/blog/serverless-architecture/

Limitations and Challenges

Cold Start Latency

- Initial invocation delay
- Varies by language and memory allocation
- Impact on interactive applications
- Mitigation strategies required for latency-sensitive workloads

Execution Duration Limits

- Maximum runtime constraints (minutes)
- Not suitable for long-running processes
- Requires workflow orchestration for complex processes
- Different limits across providers

Limitations and Challenges, Cont.

Debugging and Monitoring Complexity

- Distributed execution challenges
- Limited local testing options
- Complex logging and tracing
- Specialized monitoring tools required

Vendor Lock-in

- Provider-specific APIs and services
- Migration challenges
- Different feature sets across providers
- Limited standardization

Serverless Data Processing Architecture

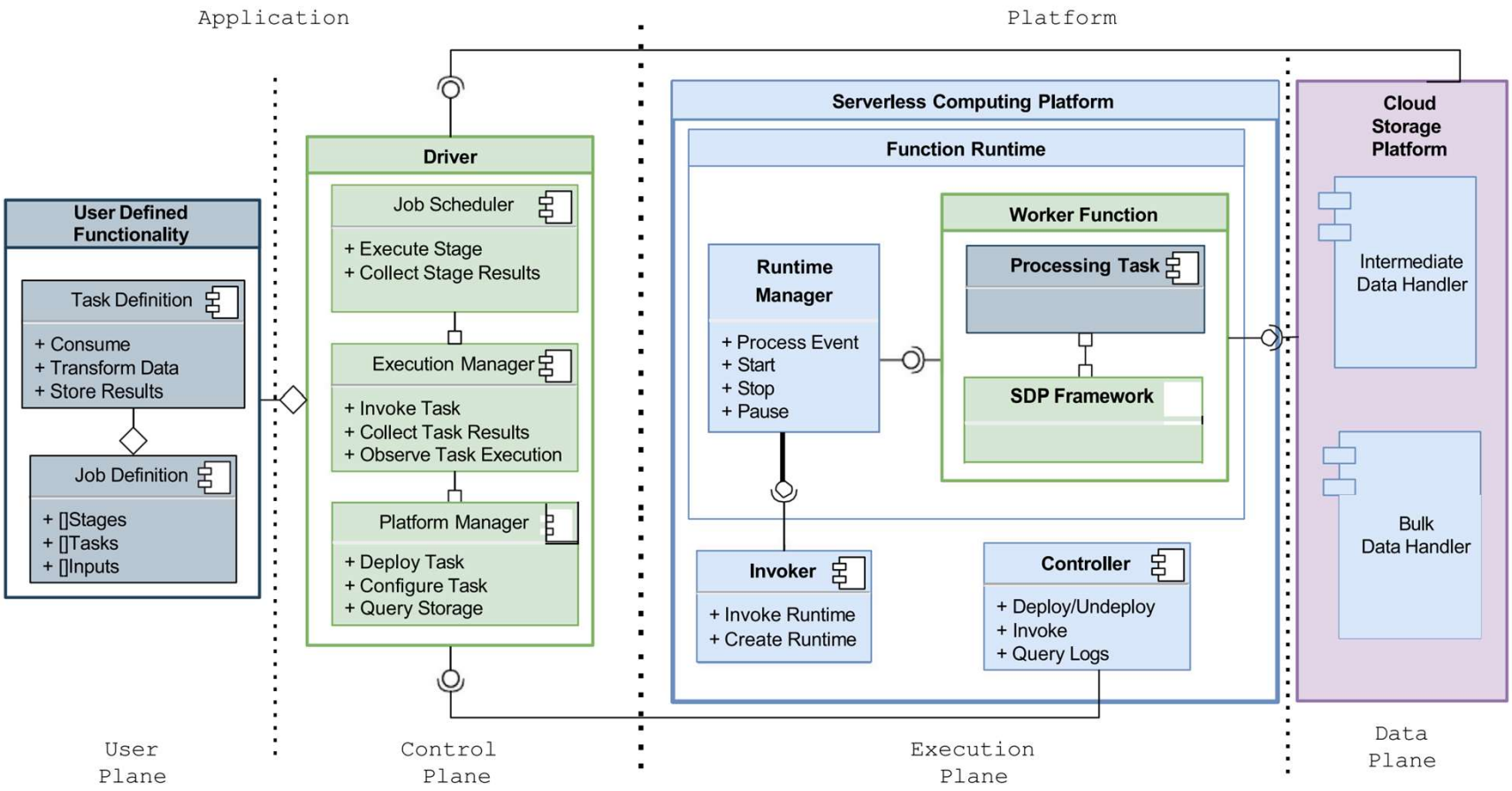


Figure Source: https://www.sciencedirect.com/science/article/pii/S0167739X24000360

Serverless Data Processing Patterns

Event-Driven Processing

Process data in response to events such as:

- File uploads to storage
- Database changes
- Message queue events
- API calls
- IoT device signals

Example Use Cases

- Image and video processing
- Real-time data transformation
- Notification systems
- Webhook handling
- Audit logging

Scheduled Processing

Regular data processing jobs triggered by time-based schedules:

- Fixed intervals (hourly, daily, weekly)
- Cron expressions for complex schedules
- One-time scheduled executions
- Calendar-based scheduling

Example Use Cases

- Daily report generation
- Data aggregation and summarization
- Cleanup operations
- Periodic data exports

Stream Processing

Process data as it arrives in continuous streams:

- Real-time data analysis
- Continuous data transformation
- Stream enrichment
- Pattern detection
- Aggregation windows

Example Use Cases

- Real-time analytics
- Fraud detection
- IoT sensor data processing
- Log analysis
- Social media monitoring

Major Serverless Computing Services: AWS Lambda

Overview

- Pioneered FaaS in 2014
- Supports multiple languages (Node.js, Python, Java, Go, Ruby, .NET)
- Integrated with AWS ecosystem
- Up to 15 minutes execution time
- Memory allocation from 128MB to 10GB

Event Sources

- Amazon S3, Amazon DynamoDB, AWS EventBridge, API Gateway, Custom events

Common Data Processing Patterns:

ETL Pipelines: Extract data from source, Transform using Lambda, Load to destination, Orchestrate with Step Functions

Real-time Analytics: Capture events in Kinesis, Process with Lambda, Store results in DynamoDB, Visualize with QuickSight

Data Validation: Trigger on data arrival, Validate against schema, Route valid/invalid data, Generate quality metrics

```python
import boto3

def lambda_handler(event, context):
    # Get the S3 bucket and object key from the event
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']

    # Process the file
    s3 = boto3.client('s3')
    response = s3.get_object(Bucket=bucket, Key=key)
    data = response['Body'].read().decode('utf-8')

    # Transform data (e.g., CSV to JSON)
    processed_data = process_data(data)

    # Write results back to S3
    s3.put_object(
        Bucket=bucket,
        Key=f"processed/{key}",
        Body=processed_data
    )

    return {
        'statusCode': 200,
        'body': f"Successfully processed {key}"
    }
```

Major Serverless Computing Services: Google Cloud Functions

Overview

Supports Node.js, Python, Go, Java, .NET, Ruby, PHP

Integrated with Google Cloud services

1st gen: Up to 9 minutes execution time

2nd gen: Up to 60 minutes execution time

Memory allocation from 128MB to 8GBEvent Sources

Event Sources

Cloud Storage, Firestore, HTTP requests, Cloud Scheduler, Eventarc, Custom events

Common Data Processing Patterns

Data Transformation: Trigger on file upload, Transform data format, Store processed results, Log processing metadata

Event-driven Analytics: Capture events from Pub/Sub, Process with Cloud Functions, Store in BigQuery, Analyze with Data Studio

Webhook Processing: Receive webhook via HTTP trigger, Validate payload, Process data, Send confirmation response

```python
from google.cloud import storage

def process_file(event, context):
    """Background Cloud Function to process a file.
    Args:
         event (dict): The dictionary with data specific to this type of event.
         context (google.cloud.functions.Context): Metadata of triggering event.
    """
    bucket_name = event['bucket']
    file_name = event['name']

    # Create a client
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(file_name)

    # Download the file
    data = blob.download_as_string().decode('utf-8')

    # Process the data
    processed_data = process_data(data)

    # Upload the processed file
    output_blob = bucket.blob(f"processed/{file_name}")
    output_blob.upload_from_string(processed_data)

    print(f"File {file_name} processed successfully")
```

Major Serverless Computing Services: Azure Functions

Overview

- Microsoft's serverless compute service
- Supports C#, JavaScript, Python, PowerShell, Java
- Integrated with Azure ecosystem
- Consumption plan: Up to 10 minutes execution
- Premium plan: Up to 60 minutes execution
- Memory allocation up to 14GB

Event Sources

Blob Storage, Cosmos DB, Event Hub, HTTP, Queue Storage, Service Bus, Timer, Custom triggers

Common Data Processing Patterns

Orchestrated Processing: Coordinate multiple functions, Maintain processing state, Handle failures and retries, Track progress

Event Grid Integration: Capture system events, Filter relevant events, Process with Functions, Store results in appropriate sinks

Hybrid Connections: Connect to on-premises data sources, Process data in Functions, Store results in cloud services, Bridge cloud and on-premises systems

```
[FunctionName("ProcessDataOrchestrator")]
public static async Task<List<string>> RunOrchestrator(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var outputs = new List<string>();

    // Get the list of files to process
    var files = await context.CallActivityAsync<List<string>>("GetFilesToProcess", null);

    // Process each file in parallel
    var tasks = new Task<string>[files.Count];
    for (int i = 0; i < files.Count; i++)
    {
        tasks[i] = context.CallActivityAsync<string>("ProcessFile", files[i]);
    }

    // Wait for all tasks to complete
    await Task.WhenAll(tasks);

    // Aggregate results
    for (int i = 0; i < tasks.Length; i++)
    {
        outputs.Add(tasks[i].Result);
    }

    return outputs;
}
```

Participation Question

Imagine a photo-sharing app where, when a user uploads an image, a serverless function automatically processes the image (e.g., resizing, filtering) and stores the processed version for display.

Instructions:

1. Quickly review the scenario and the key elements from the lecture—event triggers, serverless functions (like AWS Lambda), auto-scaling, and the pay-per-use model.
2. In your group of 3–4 students, discuss and draft a simple diagram addressing:
    1. **Event Trigger:** How does the system detect the image upload?
    2. **Serverless Function:** Which service (AWS Lambda, Google Cloud Functions, or Azure Functions) processes the image?
    3. **Storage:** Where will both the original and processed images be stored?
    4. **Challenge & Mitigation:** Identify one potential challenge (e.g., cold start latency) and briefly suggest a mitigation strategy.

Summary

Serverless computing eliminates infrastructure management, enabling developers to focus on code

The evolution from on-premises to IaaS to PaaS to FaaS represents increasing levels of abstraction

Key characteristics include no server management, auto-scaling, pay-per-use pricing, and event-driven architecture

Serverless data processing offers advantages in scalability, cost efficiency, and operational simplicity

Common patterns include event-driven processing, scheduled processing, and stream processing

Major providers (AWS, Google Cloud, Azure) offer robust FaaS platforms with unique features

Effective serverless architectures require careful consideration of event sources, state management, and orchestration