



## 12. How do you implement custom metrics and alarms for Fargate services?

**Answer:**

To track application-specific metrics beyond default CPU/Memory:

- **Use CloudWatch Embedded Metrics:**
  - Inside the app code, use AWS SDK or CloudWatch agent to push metrics like request count, latency, etc.
  - Example: Log structured JSON like { "\_aws": { ... }, "requests": 12 } in CloudWatch Logs; AWS automatically extracts metrics.
- **Create CloudWatch Alarms:**
  - Based on custom metrics (e.g., Errors > 5/min).
  - Alarms can trigger **SNS notifications**, **Lambda functions**, or **auto-scaling**.

🔑 Tip: Use tools like **Prometheus exporters** or **StatsD agents** inside containers to send metrics.

### Examples of Application-Specific Metrics in Fargate

Metric Name	Description	Example Use Case
http_request_count	Number of HTTP requests received by the app	Monitor traffic volume
http_5xx_errors	Number of server errors (HTTP 5xx)	Detect backend failures
request_latency_ms	Time taken to process a request (in milliseconds)	Identify performance bottlenecks
db_query_duration_ms	Time taken to execute a database query	Spot slow DB interactions
user_signup_total	Number of users signed up	Track growth over time
payment_failure_count	Count of failed payment transactions	Monitor payment gateway issues

Here's a **simple Node.js app** example that captures **custom application-specific metrics** (like request count and latency) and sends them to **Amazon CloudWatch using Embedded Metric Format (EMF)** — ideal for AWS Fargate.

### Step 1: Install AWS SDK

bash

CopyEdit

npm install aws-sdk

---

## Step 2: Sample Express App (with custom metrics)

js

CopyEdit

// app.js

```
const express = require('express');
```

```
const app = express();
```

```
const port = 3000;
```

// For logging to CloudWatch

```
const winston = require('winston');
```

```
const { CloudWatchTransport } = require('winston-aws-cloudwatch');
```

// Setup logger for CloudWatch

```
const logger = winston.createLogger({
```

```
  transports: [
```

```
    new CloudWatchTransport({
```

```
      logGroupName: 'FargateAppMetrics',
```

```
      logStreamName: 'AppStream',
```

```
      createLogGroup: true,
```

```
      createLogStream: true,
```

```
      awsRegion: 'us-east-1',
```

```
    }),
```

```
  ],
```

```
});
```

```
app.get('/', (req, res) => {
```

```
  const startTime = Date.now();
```

// Simulate processing

```
  setTimeout(() => {
```

```
    const latency = Date.now() - startTime;
```

// Log custom metric in EMF format

```
  const metricLog = {
```

```
    _aws: {
```

```
      Timestamp: Date.now(),
```

```
      CloudWatchMetrics: [
```

```
        {
```

```
          Namespace: 'MyApp/Fargate',
```

```
          Dimensions: [['ServiceName']],
```

```
          Metrics: [
```

```
            { Name: 'RequestCount', Unit: 'Count' },
```

```
            { Name: 'Latency', Unit: 'Milliseconds' },
```

```
          ],
```

```
        },
```

```

    ],
  },
  ServiceName: 'FargateSampleApp',
  RequestCount: 1,
  Latency: latency,
};

logger.info(JSON.stringify(metricLog));

res.send(` Hello! Latency: ${latency}ms `);
}, Math.random() * 200);
});

app.listen(port, () => {
  console.log(` App running on port ${port} `);
});

```

---

### Step 3: View Metrics in CloudWatch

- Go to **CloudWatch** → **Metrics** → **MyApp/Fargate**
- You'll see RequestCount and Latency per ServiceName

---

### Optional: Create Alarm

- Go to **CloudWatch** → **Alarms** → **Create Alarm**
- Select metric RequestCount or Latency
- Define threshold (e.g., Latency > 500ms for 3 datapoints)
- Set up **SNS** or **Lambda** as alarm action

---

## 13. What are the considerations for running stateful applications on Fargate?

Feature	Stateless Application	Stateful Application
Definition	Does not retain user/session data between requests	Remembers past interactions (state) across sessions
Example	REST APIs, static websites	Databases, chat apps, shopping carts

### Answer:

Fargate is designed for stateless workloads, but you *can* run stateful apps with care:

- **Storage:**
  - Use **Amazon EFS** for persistent, shared storage across tasks.
  - Avoid using container local storage—it's **ephemeral** (lost when task restarts).
- **Sticky Sessions:**
  - Use **Application Load Balancer (ALB)** with **session stickiness** if needed.
- **Backup & Sync:**
  - Ensure data is backed up or replicated to avoid loss during scaling or task termination.

- **Scaling Challenges:**
  - Stateful apps are harder to scale horizontally; you need mechanisms to maintain consistency.


---

#### 14. How do you implement zero-downtime deployments with Fargate?

**Answer:**

Zero-downtime means users never face an outage during a new deployment. I use:

- **Blue/Green Deployment (via CodeDeploy):**
  - Run old version (blue), then deploy new version (green).
  - Shift traffic gradually from blue → green.
  - Rollback if issues occur.
- **Rolling Updates** in ECS:
  - ECS replaces tasks gradually (based on minimumHealthyPercent and maximumPercent settings).
- **Health Checks:**
  - ALB should only route traffic to **healthy tasks** based on container health check.

 Test new version with a small portion of traffic before full rollout.

---

#### 15. Explain the security model of Fargate and how isolation is achieved.

**Answer:**

Fargate provides strong **security and isolation by design**:

- **Each task runs in its own lightweight VM** (Firecracker MicroVM) — not shared with others.
- **Task-level ENI (Elastic Network Interface)** ensures network isolation.
- **IAM Roles for Tasks:**
  - Assign least-privilege policies for accessing AWS services.
- **Security Groups + Private Subnets:**
  - Control traffic at VPC level.
- **No Host Access:**
  - Since there's no EC2 instance, there's no SSH access, reducing attack surface.


This isolation makes Fargate ideal for multi-tenant or sensitive workloads.

---

#### 16. How do you handle long-running vs short-lived workloads on Fargate?

**Answer:**

- **Long-running Tasks:**
  - Run as ECS **Services** (e.g., web apps, APIs).
  - Auto-restart on failure, load-balanced, and can scale horizontally.
- **Short-lived Jobs:**
  - Run as **One-time Tasks** using ECS RunTask API or Scheduled Tasks (like cron).
  - Ideal for batch jobs, data processing, or event-driven Lambda triggers.

 Use **EventBridge or Step Functions** to trigger short-lived tasks on schedule or events.

---

#### 17. What are the best practices for container image optimization for Fargate?

**Answer:**

1. **Use Minimal Base:**
  - Prefer alpine, distroless, or slim variants to reduce size.
2. **Multi-stage Builds:**
  - Compile in one stage, copy only artifacts to final image.
3. **Layer Caching:**
  - Order Dockerfile commands **Images** to maximize layer reuse (e.g., dependencies first).
4. **Scan for Vulnerabilities:**
  - Use ECR scanning or tools like Trivy.
5. **Avoid Unnecessary Packages:**
  - Keep images clean and focused.
6. **Tag Images Properly:**
  - Use semantic tags (v1.0, latest, commit-sha) for clarity and rollback.

⚡ Smaller, secure images lead to **faster startup, lower storage costs, and better security.**

**What is the purpose of using Embedded Metric Format (EMF) in a Fargate application?**

- A. To reduce memory usage of the container
- B. To automatically scale the task count
- C. To structure application logs so CloudWatch can extract metrics
- D. To encrypt application logs

✅ **Correct Answer: C**

📄 **Explanation:** EMF allows you to send structured logs (JSON) that CloudWatch can parse and turn into metrics.

---

**Which of the following tools can also be used inside Fargate containers to send custom metrics?**

- A. CodeDeploy
- B. CloudTrail
- C. StatsD or Prometheus exporters
- D. ECS Exec

✅ **Correct Answer: C**

📄 **Explanation:** StatsD agents and Prometheus exporters are common tools used for exporting app-specific metrics.