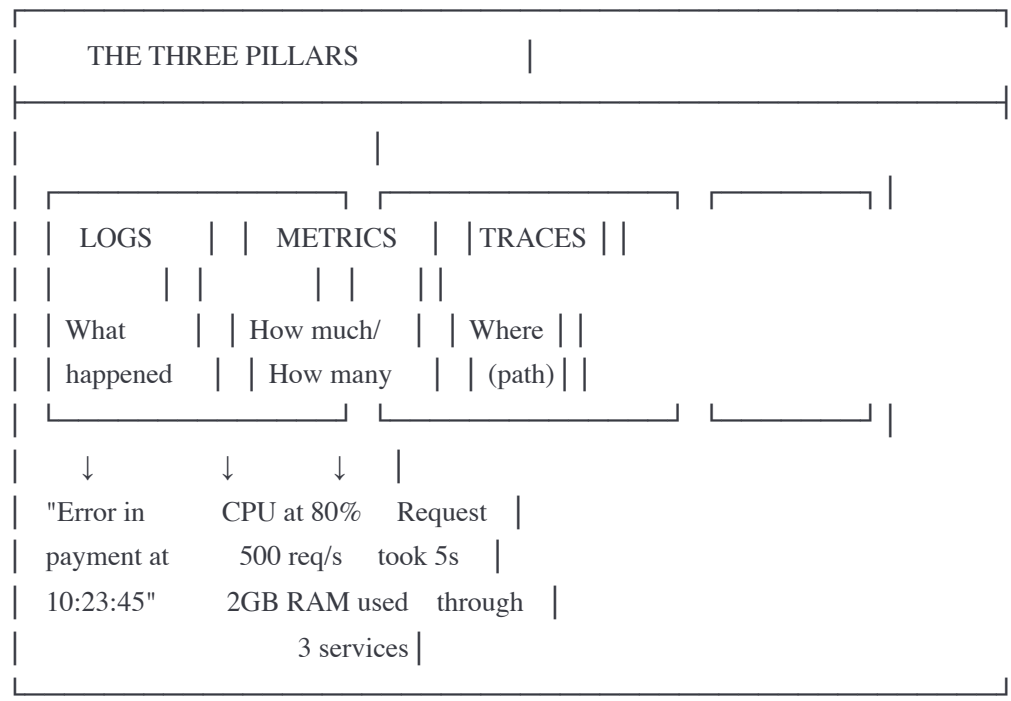
Chapter 14: Monitoring and Observability

Introduction: The Three Pillars of Observability

Observability: Understanding what's happening inside your system by examining its outputs.



Together they answer:
- What went wrong?
- When did it happen?
- Where did it happen?
- Why did it happen?

Monitoring vs Observability:

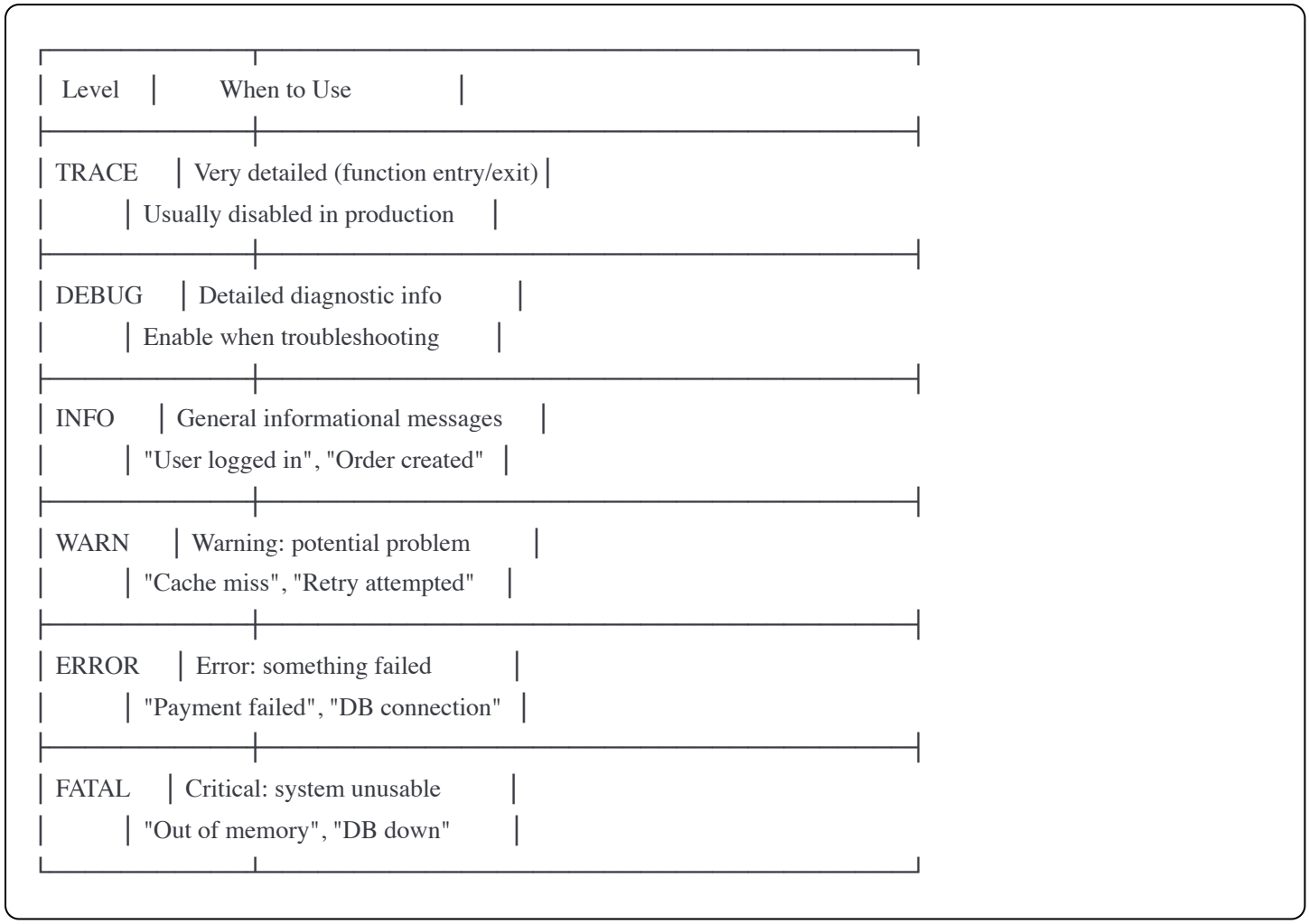
MONITORING (Known Unknowns):
- "Is the server up?"
- "Is CPU usage high?"
- "Are we getting errors?"
- Dashboard with predefined metrics

OBSERVABILITY (Unknown Unknowns):
- "Why is this specific user's request slow?"
- "What's the path this request took?"
- "What happened between 10:23:45 and 10:23:46?"
- Ability to ask arbitrary questions

---

1. Logging Best Practices

Log Levels



Structured Logging

Bad (Unstructured):

javascript

```
// ❌ Hard to parse and search
console.log('User John logged in from 192.168.1.1 at 10:30');
console.log('Order 123 created with total $99.99');
console.log('Payment processing failed - insufficient funds');
```

Good (Structured):

javascript

```javascript
// ✓ JSON format, easy to parse and query
const logger = require('winston');

logger.info('user_login', {
  userId: 123,
  username: 'john',
  ip: '192.168.1.1',
  timestamp: '2024-01-20T10:30:00Z',
  userAgent: 'Mozilla/5.0...'
});

logger.info('order_created', {
  orderId: 123,
  userId: 456,
  total: 99.99,
  itemCount: 3,
  timestamp: '2024-01-20T10:31:00Z'
});

logger.error('payment_failed', {
  orderId: 123,
  userId: 456,
  amount: 99.99,
  errorCode: 'INSUFFICIENT_FUNDS',
  timestamp: '2024-01-20T10:32:00Z'
});

// Output (JSON):
// {
//   "level": "info",
//   "message": "user_login",
//   "userId": 123,
//   "username": "john",
//   "ip": "192.168.1.1",
//   "timestamp": "2024-01-20T10:30:00Z"
// }
```

Complete Logging Implementation

javascript

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

// Create logger with multiple transports
const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  defaultMeta: {
    service: 'order-service',
    version: '1.0.0',
    environment: process.env.NODE_ENV
  },
  transports: [
    // Console output (for development)
    new winston.transports.Console({
      format: winston.format.combine(
        winston.format.colorize(),
        winston.format.simple()
      )
    }),

    // File output (all logs)
    new winston.transports.File({
      filename: 'logs/combined.log'
    }),

    // File output (errors only)
    new winston.transports.File({
      filename: 'logs/error.log',
      level: 'error'
    })
  ]
});

// Enhanced logging with context
class ContextLogger {
  constructor(baseLogger) {
    this.logger = baseLogger;
  }

  child(context) {
    """Create logger with additional context"""
    return this.logger.child(context);
```

```
}

logRequest(req, res, duration) {
  """Log HTTP request"""
  this.logger.info('http_request', {
    method: req.method,
    path: req.path,
    statusCode: res.statusCode,
    duration: duration,
    ip: req.ip,
    userAgent: req.get('user-agent'),
    userId: req.user?.id,
    requestId: req.id,
    query: req.query,
    body: this._sanitize(req.body)
  });
}

logDatabaseQuery(query, duration, rowCount) {
  """Log database query"""
  this.logger.debug('database_query', {
    query: query,
    duration: duration,
    rowCount: rowCount,
    slow: duration > 100  // Flag slow queries
  });
}

logError(error, context = {}) {
  """Log error with full stack trace"""
  this.logger.error('error_occurred', {
    errorMessage: error.message,
    errorCode: error.code,
    stack: error.stack,
    ...context
  });
}

logBusinessEvent(eventType, data) {
  """Log business events"""
  this.logger.info(`business_event:${eventType}`, {
    eventType: eventType,
    ...data
  });
}

_sanitize(data) {
```

```
    """Remove sensitive data from logs"""
    if (!data) return data;

    const sanitized = { ...data };

    // Remove sensitive fields
    const sensitiveFields = ['password', 'creditCard', 'ssn', 'token'];
    sensitiveFields.forEach(field => {
      if (sanitized[field]) {
        sanitized[field] = '***REDACTED***';
      }
    });

    return sanitized;
  }
}

// Usage
const contextLogger = new ContextLogger(logger);

// Request logging middleware
app.use((req, res, next) => {
  const startTime = Date.now();

  // Add request ID for tracing
  req.id = generateRequestId();

  // Log after response
  res.on('finish', () => {
    const duration = Date.now() - startTime;
    contextLogger.logRequest(req, res, duration);
  });

  next();
});

// Business event logging
app.post('/orders', async (req, res) => {
  const order = await createOrder(req.body);

  contextLogger.logBusinessEvent('order_created', {
    orderId: order.id,
    userId: req.user.id,
    total: order.total,
    itemCount: order.items.length
  });
```

```
  res.json(order);
});

// Error logging
app.use((err, req, res, next) => {
  contextLogger.logError(err, {
    requestId: req.id,
    path: req.path,
    method: req.method,
    userId: req.user?.id
  });

  res.status(500).json({ error: 'Internal server error' });
});
```

Correlation IDs (Request Tracing)

Problem: Request goes through multiple services
How do you correlate logs?

Request Flow:
User → API Gateway → User Service → Database
→ Order Service → Database
→ Payment Service

Without Correlation ID:
- API Gateway logs: "Request received"
- User Service logs: "User fetched"
- Order Service logs: "Order created"
How do you know these are from same request?

With Correlation ID:
- API Gateway: [req-abc-123] "Request received"
- User Service: [req-abc-123] "User fetched"
- Order Service: [req-abc-123] "Order created"
Now can trace entire request flow!

Implementation:

javascript

```
const { v4: uuidv4 } = require('uuid');

// Generate or propagate correlation ID
function correlationMiddleware(req, res, next) {
  // Get correlation ID from header or generate new one
  req.correlationId = req.get('X-Correlation-ID') || uuidv4();

  // Add to response headers
  res.set('X-Correlation-ID', req.correlationId);

  // Create logger with correlation ID
  req.log = logger.child({
    correlationId: req.correlationId,
    path: req.path,
    method: req.method
  });

  next();
}

app.use(correlationMiddleware);

// All logs now include correlation ID
app.get('/users/:id', async (req, res) => {
  req.log.info('Fetching user');

  const user = await db.query('SELECT * FROM users WHERE id = ?', req.params.id);

  req.log.info('User fetched', { userId: user.id });

  res.json(user);
});

// When calling other services, propagate correlation ID
async function callOrderService(correlationId, data) {
  return await fetch('https://order-service.example.com/orders', {
    method: 'POST',
    headers: {
      'X-Correlation-ID': correlationId  // Propagate!
    },
    body: JSON.stringify(data)
  });
}

// Now can search logs by correlation ID:
```

```
// grep "req-abc-123" *.log
// Shows complete request flow across all services!
```

Logging Best Practices

1. STRUCTURED LOGS (JSON)
   ✓ {"level":"info","msg":"user_login","userId":123}
   ✗ "User 123 logged in"

2. INCLUDE CONTEXT
   ✓ userId, orderId, correlationId, timestamp
   ✗ Just the error message

3. LOG LEVELS APPROPRIATELY
   ✓ INFO for business events
   ✓ ERROR for actual errors
   ✗ ERROR for expected conditions
   ✗ DEBUG in production (too verbose)

4. DON'T LOG SENSITIVE DATA
   ✗ Passwords, credit cards, SSNs, tokens
   ✓ Sanitize or hash sensitive fields

5. ADD CORRELATION IDS
   ✓ Trace requests across services
   ✓ Propagate in headers

6. LOG AT BOUNDARIES
   ✓ Request entry/exit
   ✓ External service calls
   ✓ Database queries (slow ones)

7. INCLUDE TIMING
   ✓ Duration of operations
   ✓ Timestamp of events

8. CENTRALIZE LOGS
   ✓ Send to central system (ELK, Datadog)
   ✗ Leave on individual servers

---

2. Metrics Collection and Monitoring

Types of Metrics

1. SYSTEM METRICS (Infrastructure)

```
┌──────────────────────────────────────────────────┐
│ • CPU usage (%)                                  │
│ • Memory usage (MB)                              │
│ • Disk usage (GB, IOPS)                          │
│ • Network I/O (Mbps)                             │
│ • File descriptors                               │
└──────────────────────────────────────────────────┘
```

2. APPLICATION METRICS (Business)

```
┌──────────────────────────────────────────────────┐
│ • Requests per second                            │
│ • Response time (p50, p95, p99)                  │
│ • Error rate (%)                                 │
│ • Active users                                   │
│ • Orders per minute                              │
│ • Revenue per hour                               │
└──────────────────────────────────────────────────┘
```

3. BUSINESS METRICS (KPIs)

```
┌──────────────────────────────────────────────────┐
│ • Conversion rate                                │
│ • Customer acquisition cost                      │
│ • Average order value                            │
│ • Churn rate                                     │
│ • User engagement                                │
└──────────────────────────────────────────────────┘
```

The Four Golden Signals (Google SRE)

1. LATENCY
   Time to serve a request

   Track:
   - p50 (median): 50% of requests faster than this
   - p95: 95% of requests faster than this
   - p99: 99% of requests faster than this

   Example:
   p50: 100ms (good)
   p95: 500ms (acceptable)

p99: 2000ms (some users have bad experience)

2. TRAFFIC
Demand on your system

Track:
- Requests per second
- Transactions per second
- Concurrent users

Example:
Normal: 1,000 req/s
Peak: 10,000 req/s
Alert if > 15,000 req/s (capacity limit)

3. ERRORS
Rate of failed requests

Track:
- Error rate (%)
- Errors per second
- By error type (4xx vs 5xx)

Example:
Normal: 0.1% error rate
Alert if > 1% (degraded)
Alert if > 5% (critical)

4. SATURATION
How "full" your service is

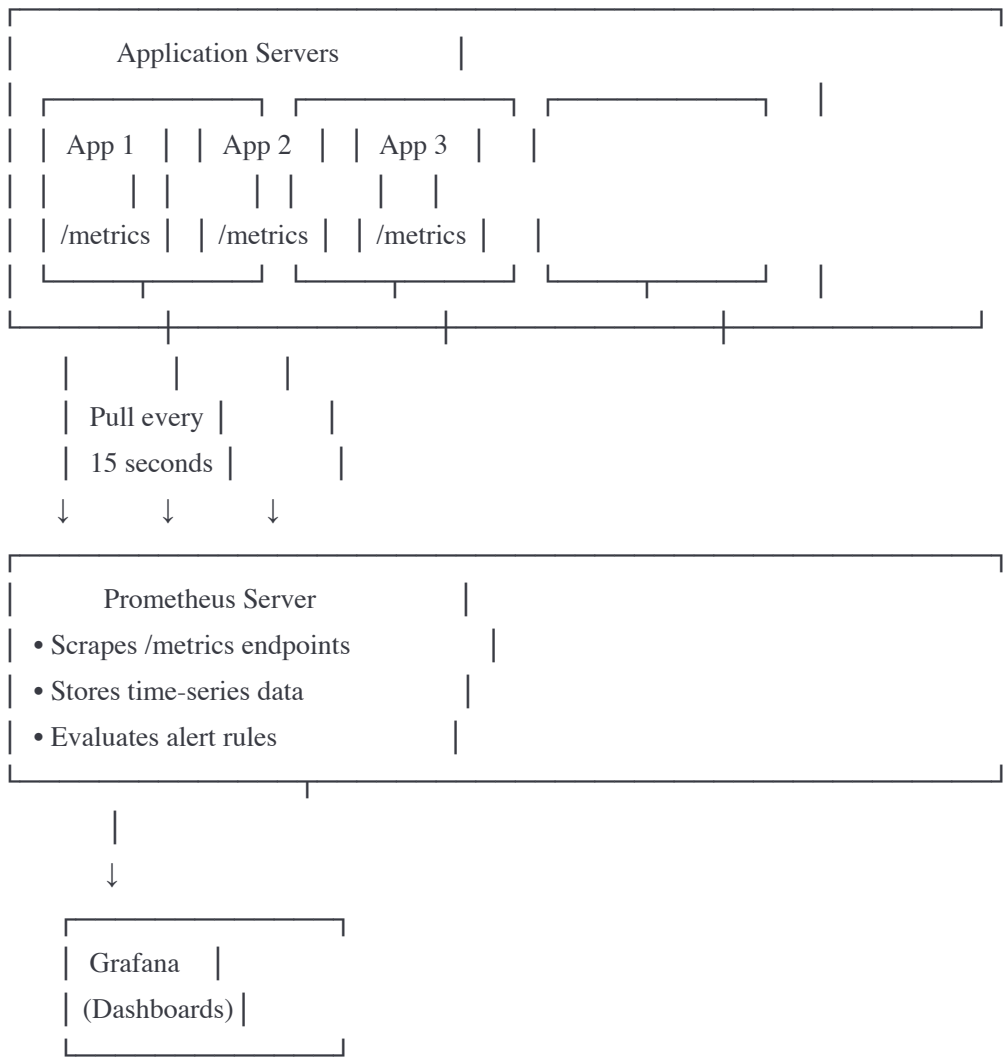
Track:
- CPU usage (%)
- Memory usage (%)
- Disk I/O (%)
- Connection pool usage (%)

Example:
CPU: 60% (normal)
Alert if > 80% (warning)
Alert if > 90% (critical)

Metrics with Prometheus

Prometheus: Pull-based monitoring system

Architecture:



Metrics Implementation (Node.js):

javascript

```
const express = require('express');
const promClient = require('prom-client');

const app = express();

// Create a Registry
const register = new promClient.Registry();

// Add default metrics (CPU, memory, etc.)
promClient.collectDefaultMetrics({ register });

// Custom metrics

// 1. Counter (only goes up)
const httpRequestsTotal = new promClient.Counter({
  name: 'http_requests_total',
  help: 'Total number of HTTP requests',
  labelNames: ['method', 'path', 'status'],
  registers: [register]
});

// 2. Gauge (can go up and down)
const activeUsers = new promClient.Gauge({
  name: 'active_users',
  help: 'Number of currently active users',
  registers: [register]
});

// 3. Histogram (distribution of values)
const httpRequestDuration = new promClient.Histogram({
  name: 'http_request_duration_seconds',
  help: 'Duration of HTTP requests in seconds',
  labelNames: ['method', 'path', 'status'],
  buckets: [0.1, 0.5, 1, 2, 5, 10],  // Buckets for histogram
  registers: [register]
});

// 4. Summary (similar to histogram, calculates percentiles)
const paymentProcessingTime = new promClient.Summary({
  name: 'payment_processing_seconds',
  help: 'Payment processing time',
  percentiles: [0.5, 0.9, 0.95, 0.99],
  registers: [register]
});

// Middleware to track metrics
```

```
app.use((req, res, next) => {
  const start = Date.now();

  res.on('finish', () => {
    const duration = (Date.now() - start) / 1000;

    // Increment request counter
    httpRequestsTotal.inc({
      method: req.method,
      path: req.route?.path || req.path,
      status: res.statusCode
    });

    // Record request duration
    httpRequestDuration.observe({
      method: req.method,
      path: req.route?.path || req.path,
      status: res.statusCode
    }, duration);
  });

  next();
});

// Business metrics
app.post('/orders', async (req, res) => {
  const start = Date.now();

  try {
    const order = await createOrder(req.body);

    // Track successful order
    const orderCounter = new promClient.Counter({
      name: 'orders_created_total',
      help: 'Total orders created',
      registers: [register]
    });
    orderCounter.inc();

    // Track order value
    const orderValue = new promClient.Histogram({
      name: 'order_value_dollars',
      help: 'Order value in dollars',
      buckets: [10, 50, 100, 500, 1000],
      registers: [register]
    });
    orderValue.observe(order.total);
```

```
    res.json(order);

  } catch (error) {
    // Track error
    const errorCounter = new promClient.Counter({
      name: 'orders_failed_total',
      help: 'Total failed orders',
      labelNames: ['reason'],
      registers: [register]
    });
    errorCounter.inc({ reason: error.code });

    res.status(500).json({ error: error.message });
  }
});

// Update active users gauge
setInterval(async () => {
  const count = await getActiveUserCount();
  activeUsers.set(count);
}, 10000);  // Update every 10 seconds

// Expose metrics endpoint
app.get('/metrics', async (req, res) => {
  res.set('Content-Type', register.contentType);
  res.end(await register.metrics());
});

app.listen(3000);

// Metrics output (Prometheus format):
// # HELP http_requests_total Total number of HTTP requests
// # TYPE http_requests_total counter
// http_requests_total{method="GET",path="/users",status="200"} 1523
// http_requests_total{method="POST",path="/orders",status="201"} 342
//
// # HELP http_request_duration_seconds Duration of HTTP requests
// # TYPE http_request_duration_seconds histogram
// http_request_duration_seconds_bucket{method="GET",path="/users",le="0.1"} 1200
// http_request_duration_seconds_bucket{method="GET",path="/users",le="0.5"} 1500
// http_request_duration_seconds_sum{method="GET",path="/users"} 152.3
// http_request_duration_seconds_count{method="GET",path="/users"} 1523
```

Prometheus Configuration

yaml

```yaml
# prometheus.yml

global:
  scrape_interval: 15s     # Scrape every 15 seconds
  evaluation_interval: 15s  # Evaluate rules every 15 seconds

# Scrape configurations
scrape_configs:
  # Application servers
  - job_name: 'order-service'
    static_configs:
      - targets:
        - 'app-server-1:3000'
        - 'app-server-2:3000'
        - 'app-server-3:3000'
    scrape_interval: 10s

  # Database
  - job_name: 'postgres'
    static_configs:
      - targets: ['postgres-exporter:9187']

  # Redis
  - job_name: 'redis'
    static_configs:
      - targets: ['redis-exporter:9121']

  # Node exporters (system metrics)
  - job_name: 'node'
    static_configs:
      - targets:
        - 'server-1:9100'
        - 'server-2:9100'
        - 'server-3:9100'

# Alert rules
rule_files:
  - 'alerts.yml'

# Alertmanager
alerting:
  alertmanagers:
    - static_configs:
      - targets: ['alertmanager:9093']
```

Alert Rules (alerts.yml):

yaml

```yaml
groups:
  - name: application_alerts
    interval: 30s
    rules:
      # High error rate
      - alert: HighErrorRate
        expr: |
          sum(rate(http_requests_total{status=~"5.."}[5m]))
          / sum(rate(http_requests_total[5m])) > 0.05
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "High error rate detected"
          description: "Error rate is {{ $value | humanizePercentage }}"

      # High latency
      - alert: HighLatency
        expr: |
          histogram_quantile(0.95,
            sum(rate(http_request_duration_seconds_bucket[5m])) by (le)
          ) > 1
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High latency detected"
          description: "95th percentile latency is {{ $value }}s"

      # Service down
      - alert: ServiceDown
        expr: up == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Service {{ $labels.instance }} is down"

      # High CPU
      - alert: HighCPU
        expr: |
          100 - (avg by(instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100) > 80
        for: 5m
        labels:
          severity: warning
        annotations:
```

```
    summary: "High CPU on {{ $labels.instance }}"
    description: "CPU usage is {{ $value }}%"

  # Disk almost full
  - alert: DiskSpaceLow
    expr: |
      (node_filesystem_avail_bytes / node_filesystem_size_bytes) < 0.1
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "Disk space low on {{ $labels.instance }}"
      description: "Only {{ $value | humanizePercentage }} free"
```

PromQL Queries

promql

```
# Request rate (requests per second)
rate(http_requests_total[5m])

# Error rate (%)
sum(rate(http_requests_total{status=~"5.."}[5m]))
/ sum(rate(http_requests_total[5m])) * 100

# 95th percentile latency
histogram_quantile(0.95,
  rate(http_request_duration_seconds_bucket[5m])
)

# Average response time by endpoint
avg(rate(http_request_duration_seconds_sum[5m]))
  by (path)
/ avg(rate(http_request_duration_seconds_count[5m]))
  by (path)

# Memory usage
node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes

# Disk usage (%)
(1 - node_filesystem_avail_bytes / node_filesystem_size_bytes) * 100

# Active connections
sum(pg_stat_database_numbackends) by (datname)

# Queue depth
rabbitmq_queue_messages_ready

# Top 5 slowest endpoints
topk(5,
  avg(rate(http_request_duration_seconds_sum[5m])) by (path)
)
```

---

3. Distributed Tracing

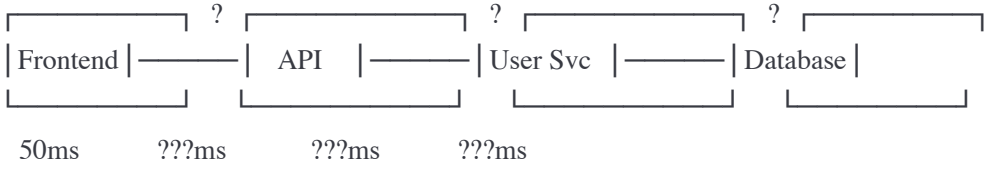
What is Distributed Tracing?

Problem: Request goes through multiple services

Without Tracing:

User reports: "Checkout is slow"

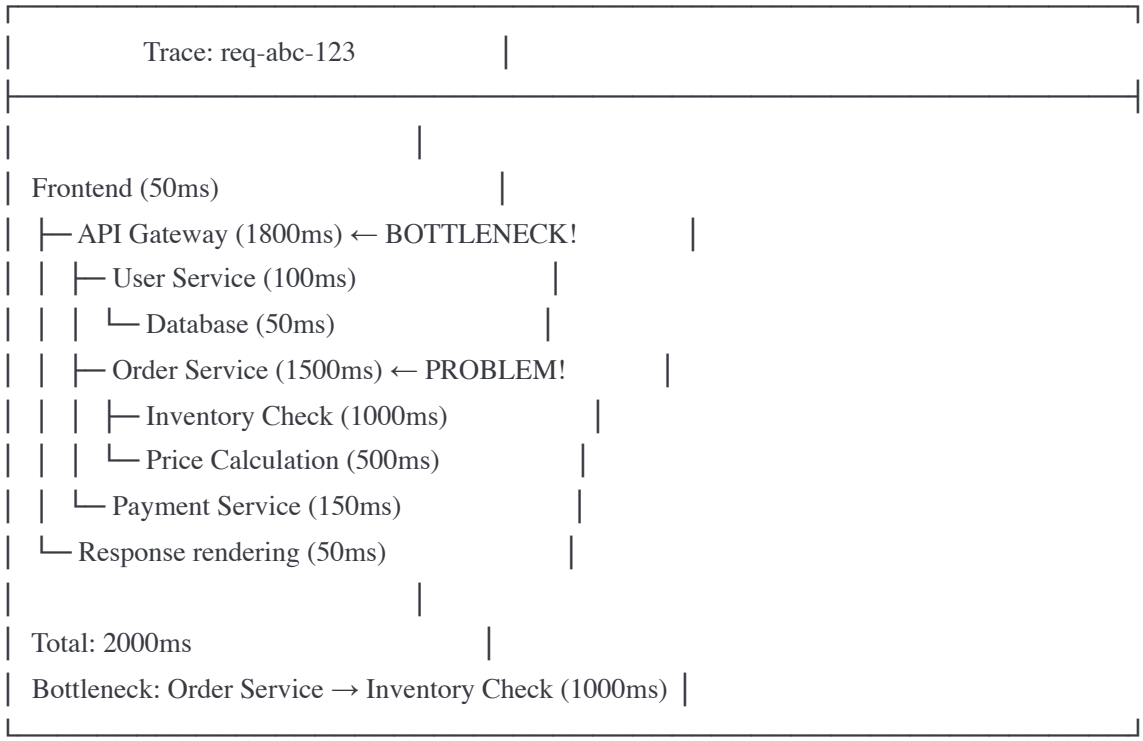
Where's the problem?



Total: 2000ms (slow!)

But which service caused it?

With Tracing:



Now we know exactly where to optimize!

Trace Structure

Trace: Complete request flow

```
├─ Span 1: API Gateway (root span)
│  ├─ Span 2: User Service
│  │  └─ Span 3: Database query
│  ├─ Span 4: Order Service
│  │  ├─ Span 5: Inventory check
│  │  └─ Span 6: Price calculation
│  └─ Span 7: Payment Service
```

Each span contains:
- Span ID (unique)
- Trace ID (same for all spans in trace)
- Parent Span ID (to build hierarchy)
- Service name
- Operation name
- Start timestamp
- Duration
- Tags (metadata)
- Logs (events during span)

Tracing Implementation (OpenTelemetry)

javascript

```
const { NodeTracerProvider } = require('@opentelemetry/sdk-trace-node');
const { registerInstrumentations } = require('@opentelemetry/instrumentation');
const { HttpInstrumentation } = require('@opentelemetry/instrumentation-http');
const { ExpressInstrumentation } = require('@opentelemetry/instrumentation-express');
const { JaegerExporter } = require('@opentelemetry/exporter-jaeger');
const { Resource } = require('@opentelemetry/resources');
const { SemanticResourceAttributes } = require('@opentelemetry/semantic-conventions');

// Initialize tracer
const provider = new NodeTracerProvider({
  resource: new Resource({
    [SemanticResourceAttributes.SERVICE_NAME]: 'order-service',
    [SemanticResourceAttributes.SERVICE_VERSION]: '1.0.0'
  })
});

// Configure exporter (send to Jaeger)
const exporter = new JaegerExporter({
  endpoint: 'http://jaeger:14268/api/traces'
});

provider.addSpanProcessor(
  new SimpleSpanProcessor(exporter)
);

// Register provider
provider.register();

// Auto-instrument HTTP and Express
registerInstrumentations({
  instrumentations: [
    new HttpInstrumentation(),
    new ExpressInstrumentation()
  ]
});

// Get tracer for manual instrumentation
const tracer = provider.getTracer('order-service');

// Application code with tracing
app.post('/orders', async (req, res) => {
  // Create span for this operation
  const span = tracer.startSpan('create_order');

  // Add attributes (tags)
  span.setAttribute('user.id', req.user.id);
```

```
span.setAttribute('order.total', req.body.total);

try {
  // Step 1: Validate order
  const validateSpan = tracer.startSpan('validate_order', {
    parent: span
  });

  const order = await validateOrder(req.body);
  validateSpan.end();

  // Step 2: Check inventory
  const inventorySpan = tracer.startSpan('check_inventory', {
    parent: span
  });

  const available = await checkInventory(order.items);
  inventorySpan.setAttribute('inventory.available', available);
  inventorySpan.end();

  if (!available) {
    span.setAttribute('error', true);
    span.setAttribute('error.reason', 'out_of_stock');
    throw new Error('Out of stock');
  }

  // Step 3: Process payment
  const paymentSpan = tracer.startSpan('process_payment', {
    parent: span
  });

  const payment = await processPayment(order.total);
  paymentSpan.setAttribute('payment.id', payment.id);
  paymentSpan.end();

  // Step 4: Create order
  const dbSpan = tracer.startSpan('database_insert', {
    parent: span
  });

  const savedOrder = await db.createOrder(order);
  dbSpan.end();

  span.setAttribute('order.id', savedOrder.id);
  span.setStatus({ code: SpanStatusCode.OK });

  res.json(savedOrder);
```

```
  } catch (error) {
    // Record error in span
    span.recordException(error);
    span.setStatus({
      code: SpanStatusCode.ERROR,
      message: error.message
    });

    res.status(500).json({ error: error.message });

  } finally {
    span.end();
  }
});

// Propagate trace context to downstream services
async function callUserService(userId, traceContext) {
  const span = tracer.startSpan('call_user_service');

  try {
    const response = await fetch('https://user-service/users/' + userId, {
      headers: {
        // Propagate trace context
        'traceparent': traceContext.traceparent,
        'tracestate': traceContext.tracestate
      }
    });

    const user = await response.json();
    span.setAttribute('user.name', user.name);

    return user;

  } finally {
    span.end();
  }
}
```

Trace Analysis

Analyzing slow request:

Trace ID: abc-123-def-456
Total Duration: 2.5 seconds

Breakdown:

```
┌──────────────────────────────────────────────────────────┐
│ Service          Operation       Duration                │
├──────────────────────────────────────────────────────────┤
│ API Gateway      authenticate    50ms                    │
│ API Gateway      route           10ms                    │
│ User Service     get_user        80ms                    │
│ Order Service    create_order    1900ms ⚠                │
│ ├─ validate_items                100ms                   │
│ ├─ check_inventory               1500ms ⚠ ⚠              │
│ │  └─ database_query             1450ms ⚠ ⚠              │
│ └─ calculate_total               50ms                    │
│ Payment Service  charge          400ms                   │
│ API Gateway      format_response 60ms                    │
└──────────────────────────────────────────────────────────┘
```

Root cause found:
Order Service → check_inventory → database_query (1450ms)

Action items:
1. Add database index on inventory table
2. Add caching layer
3. Optimize query (was doing full table scan)

After optimization:
inventory check: 1500ms → 50ms
Total request: 2500ms → 600ms (4x faster!)

---

4. Alerting Strategies

Alert Design Principles

GOOD ALERTS:
✓ Actionable (can do something about it)
✓ Relevant (impacts users or will soon)
✓ Timely (detected quickly)
✓ Context-rich (enough info to debug)

BAD ALERTS:
✗ "Server CPU is high" (so what? is it affecting users?)
✗ Too many alerts (alert fatigue)
✗ Too sensitive (false positives)
✗ Not sensitive enough (miss real issues)

Alert Severity Levels

Severity	Condition	Response	Example
P0 Critical	User-facing outage	Wake up anyone immediately	Site down Payments failing
P1 High	Degraded service	Page on-call during work hours	Slow response times
P2 Medium	Warning sign	Create ticket	Disk 80% full
P3 Low	Informational	Review next day	Cache hit rate low

Alert Implementation

```javascript
```

```
class AlertManager {
  constructor() {
    this.alerts = new Map();
    this.alertChannels = {
      pagerduty: new PagerDutyClient(),
      slack: new SlackClient(),
      email: new EmailClient()
    };
  }

  async checkAndAlert(metricName, value, threshold, severity) {
    const alertKey = `${metricName}_${severity}`;

    // Check if threshold exceeded
    if (value > threshold) {
      // Check if already alerted (prevent spam)
      if (this.alerts.has(alertKey)) {
        const lastAlert = this.alerts.get(alertKey);
        const timeSinceLastAlert = Date.now() - lastAlert;

        // Don't re-alert within 15 minutes
        if (timeSinceLastAlert < 15 * 60 * 1000) {
          return;
        }
      }

      // Send alert
      await this.sendAlert({
        severity,
        metric: metricName,
        value,
        threshold,
        message: `${metricName} is ${value} (threshold: ${threshold})`
      });

      // Record alert time
      this.alerts.set(alertKey, Date.now());

    } else {
      // Metric back to normal
      if (this.alerts.has(alertKey)) {
        // Send recovery notification
        await this.sendRecovery({
          metric: metricName,
          message: `${metricName} is back to normal (${value})`
        });
```

```
      this.alerts.delete(alertKey);
    }
  }
}

async sendAlert(alert) {
  console.log(`🚨 ALERT [${alert.severity}]: ${alert.message}`);

  // Route based on severity
  switch (alert.severity) {
    case 'critical':
      // Page on-call engineer
      await this.alertChannels.pagerduty.trigger({
        title: alert.message,
        severity: 'critical',
        details: alert
      });

      // Also send to Slack
      await this.alertChannels.slack.send({
        channel: '#incidents',
        text: `🚨 CRITICAL: ${alert.message}`,
        color: 'danger'
      });
      break;

    case 'warning':
      // Send to Slack only
      await this.alertChannels.slack.send({
        channel: '#alerts',
        text: `⚠️ WARNING: ${alert.message}`,
        color: 'warning'
      });
      break;

    case 'info':
      // Just log it
      console.log(`ℹ️ INFO: ${alert.message}`);
      break;
  }
}

async sendRecovery(recovery) {
  console.log(`✅ RECOVERED: ${recovery.message}`);

  await this.alertChannels.slack.send({
```

```
      channel: '#alerts',
      text: `✅ RECOVERED: ${recovery.message}`,
      color: 'good'
    });
  }
}

// Usage with Prometheus metrics
const alertManager = new AlertManager();

setInterval(async () => {
  // Query Prometheus for metrics
  const errorRate = await getErrorRate();
  const latency = await getP95Latency();
  const cpu = await getCPUUsage();

  // Check thresholds and alert
  await alertManager.checkAndAlert('error_rate', errorRate, 5, 'critical');
  await alertManager.checkAndAlert('p95_latency', latency, 1000, 'warning');
  await alertManager.checkAndAlert('cpu_usage', cpu, 80, 'warning');

}, 30000);  // Check every 30 seconds
```

Alert Escalation

javascript

```
class AlertEscalation {
  async handleAlert(alert) {
    const severity = alert.severity;
    const metric = alert.metric;

    // Escalation levels
    const escalation = {
      critical: [
        { delay: 0, action: () => this.pageOnCall(alert) },
        { delay: 5 * 60, action: () => this.pageBackup(alert) },
        { delay: 15 * 60, action: () => this.pageManager(alert) }
      ],
      warning: [
        { delay: 0, action: () => this.slackAlert(alert) },
        { delay: 30 * 60, action: () => this.emailTeam(alert) }
      ]
    };

    const levels = escalation[severity] || [];

    for (const level of levels) {
      setTimeout(async () => {
        // Check if alert still active
        if (await this.isAlertStillActive(metric)) {
          console.log(`Escalating alert: ${metric} (${level.delay}s elapsed)`);
          await level.action();
        } else {
          console.log(`Alert resolved before escalation: ${metric}`);
        }
      }, level.delay * 1000);
    }
  }

  async pageOnCall(alert) {
    console.log('📟 Paging on-call engineer');
    // PagerDuty API call
  }

  async pageBackup(alert) {
    console.log('📟 Paging backup on-call');
    // Alert hasn't been acknowledged, escalate
  }

  async pageManager(alert) {
    console.log('📟 Paging engineering manager');
    // Still not resolved after 15 minutes!
```

```
  }

  async slackAlert(alert) {
    console.log('💬 Sending Slack alert');
  }

  async emailTeam(alert) {
    console.log('📧 Emailing team');
  }

  async isAlertStillActive(metric) {
    // Check if metric still above threshold
    return true;  // Simplified
  }
}
```

SLIs, SLOs, and SLAs

```
SLI (Service Level Indicator):
- Measurement of service behavior
- Examples: latency, error rate, availability

SLO (Service Level Objective):
- Target for SLI
- Examples: p95 latency < 200ms, 99.9% availability

SLA (Service Level Agreement):
- Contract with customers
- What happens if SLO not met (refund, credits)

Example:

SLI: Request success rate
  Measurement: successful_requests / total_requests

SLO: 99.9% of requests succeed
  Target: success_rate >= 0.999

SLA: If success rate < 99.9% for a month
  Penalty: 10% service credit to customer
```

Error Budget:

SLO: 99.9% availability
Allowed downtime: 0.1% = 43.2 minutes per month

Error Budget: 43.2 minutes/month

Usage:
Week 1: 10 minutes downtime → 33.2 minutes remaining
Week 2: 5 minutes downtime → 28.2 minutes remaining
Week 3: 30 minutes downtime → BUDGET EXCEEDED! ⚠️

Action when budget exceeded:
- Freeze feature development
- Focus on reliability
- No risky deployments
- Fix issues causing downtime

When budget healthy:
- Can take more risks
- Deploy faster
- Try new features

Implementation:

javascript

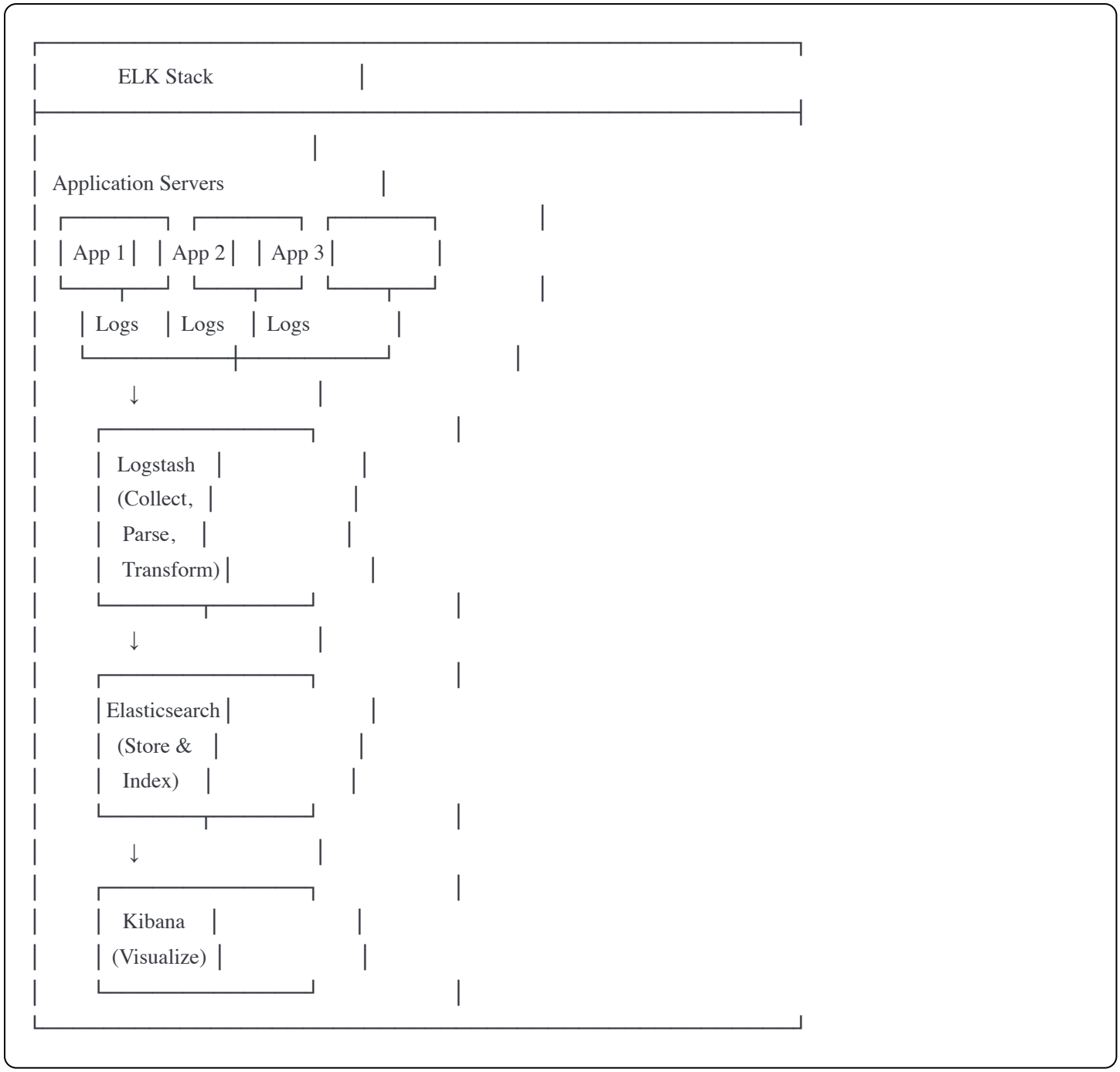
```
class ErrorBudget {
  constructor(slo, periodDays) {
    this.slo = slo;  // e.g., 0.999 for 99.9%
    this.periodDays = periodDays;
    this.incidents = [];
  }

  recordIncident(durationMinutes) {
    this.incidents.push({
      duration: durationMinutes,
      timestamp: new Date()
    });

    // Remove incidents outside the period
    const cutoff = new Date();
    cutoff.setDate(cutoff.getDate() - this.periodDays);

    this.incidents = this.incidents.filter(
      incident => incident.timestamp > cutoff
    );
  }

  getStatus() {
    const totalMinutesInPeriod = this.periodDays * 24 * 60;
    const allowedDowntime = totalMinutesInPeriod * (1 - this.slo);

    const actualDowntime = this.incidents.reduce(
      (sum, incident) => sum + incident.duration,
      0
    );

    const remaining = allowedDowntime - actualDowntime;
    const percentUsed = (actualDowntime / allowedDowntime) * 100;

    return {
      slo: this.slo * 100,
      period: `${this.periodDays} days`,
      allowedDowntime: allowedDowntime.toFixed(2),
      actualDowntime: actualDowntime.toFixed(2),
      remaining: remaining.toFixed(2),
      percentUsed: percentUsed.toFixed(2),
      status: percentUsed > 100 ? 'EXCEEDED' : 'OK'
    };
  }

  canDeploy() {
```

```
    const status = this.getStatus();

    // If used > 80% of budget, be cautious
    if (status.percentUsed > 80) {
      console.log('⚠️  Error budget almost exhausted!');
      console.log('Consider delaying risky deployments');
      return false;
    }

    return true;
  }
}

// Usage
const errorBudget = new ErrorBudget(0.999, 30);  // 99.9% over 30 days

// Record incidents
errorBudget.recordIncident(10);  // 10 minute outage
errorBudget.recordIncident(5);   // 5 minute outage

// Check status
const status = errorBudget.getStatus();
console.log('Error Budget Status:', status);

// Output:
// {
//   slo: 99.9,
//   period: '30 days',
//   allowedDowntime: '43.20',
//   actualDowntime: '15.00',
//   remaining: '28.20',
//   percentUsed: '34.72',
//   status: 'OK'
// }

// Before deployment
if (errorBudget.canDeploy()) {
  console.log('✓ Safe to deploy');
  deploy();
} else {
  console.log('✗ Focus on reliability first');
}
```

5. Popular Tools

ELK Stack (Elasticsearch, Logstash, Kibana)

Architecture:



Logstash Configuration:

```ruby
```

```
# logstash.conf

input {
  # Read from application log files
  file {
    path => "/var/log/app/*.log"
    start_position => "beginning"
    type => "application"
  }

  # Read from syslog
  syslog {
    port => 5000
    type => "syslog"
  }

  # Read from beats (lightweight shippers)
  beats {
    port => 5044
  }
}

filter {
  # Parse JSON logs
  if [type] == "application" {
    json {
      source => "message"
    }

    # Add hostname
    mutate {
      add_field => { "hostname" => "%{[host][name]}" }
    }

    # Parse timestamp
    date {
      match => [ "timestamp", "ISO8601" ]
      target => "@timestamp"
    }
  }

  # Grok parsing for unstructured logs
  grok {
    match => {
      "message" => "%{TIMESTAMP_ISO8601:timestamp} %{LOGLEVEL:level} %{GREEDYDATA:message}"
    }
```

```
  }

  # GeoIP lookup
  if [ip] {
    geoip {
      source => "ip"
      target => "geoip"
    }
  }

  # Drop debug logs in production
  if [level] == "debug" {
    drop { }
  }
}

output {
  # Send to Elasticsearch
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "logs-%{+YYYY.MM.dd}"
  }

  # Also output to console (for debugging)
  stdout {
    codec => rubydebug
  }

  # Send errors to Slack
  if [level] == "error" {
    slack {
      url => "https://hooks.slack.com/services/..."
      channel => "#errors"
      username => "Logstash"
    }
  }
}
```

Elasticsearch Queries

```
# Search for errors in last hour
GET /logs-*/_search
{
  "query": {
    "bool": {
```

```
      "must": [
        { "match": { "level": "error" } },
        { "range": { "@timestamp": { "gte": "now-1h" } } }
      ]
    }
  },
  "sort": [ { "@timestamp": { "order": "desc" } } ]
}

# Count errors by service
GET /logs-*/_search
{
  "query": {
    "match": { "level": "error" }
  },
  "aggs": {
    "by_service": {
      "terms": { "field": "service.keyword" }
    }
  }
}

# Find slow requests (> 1 second)
GET /logs-*/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "message": "http_request" } },
        { "range": { "duration": { "gte": 1000 } } }
      ]
    }
  }
}

# Trace request by correlation ID
GET /logs-*/_search
{
  "query": {
    "match": { "correlationId": "req-abc-123" }
  },
  "sort": [ { "@timestamp": { "order": "asc" } } ]
}
```

Grafana Dashboard

Dashboard Configuration (JSON):

```
json
```

```
{
  "dashboard": {
    "title": "Application Performance",
    "panels": [
      {
        "title": "Request Rate",
        "targets": [{
          "expr": "sum(rate(http_requests_total[5m]))"
        }],
        "type": "graph"
      },
      {
        "title": "Error Rate",
        "targets": [{
          "expr": "sum(rate(http_requests_total{status=~\"5..\"}[5m])) / sum(rate(http_requests_total[5m])) * 100"
        }],
        "type": "graph",
        "alert": {
          "conditions": [{
            "evaluator": {
              "params": [5],
              "type": "gt"
            }
          }]
        }
      },
      {
        "title": "Response Time (p95)",
        "targets": [{
          "expr": "histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (le))"
        }],
        "type": "graph"
      },
      {
        "title": "Active Users",
        "targets": [{
          "expr": "active_users"
        }],
        "type": "stat"
      }
    ]
  }
}
```

Datadog Implementation

javascript

```
const StatsD = require('node-dogstatsd').StatsD;

// Initialize Datadog client
const dogstatsd = new StatsD('localhost', 8125);

// Add default tags
dogstatsd.globalTags = [
  `service:order-service`,
  `env:${process.env.NODE_ENV}`,
  `version:1.0.0`
];

class DatadogMetrics {
  constructor(client) {
    this.client = client;
  }

  // Increment counter
  incrementCounter(metric, value = 1, tags = []) {
    this.client.increment(metric, value, tags);
  }

  // Set gauge (current value)
  gauge(metric, value, tags = []) {
    this.client.gauge(metric, value, tags);
  }

  // Record histogram
  histogram(metric, value, tags = []) {
    this.client.histogram(metric, value, tags);
  }

  // Timing (convenience for duration)
  timing(metric, duration, tags = []) {
    this.client.timing(metric, duration, tags);
  }

  // Set (count unique values)
  set(metric, value, tags = []) {
    this.client.set(metric, value, tags);
  }
}

const metrics = new DatadogMetrics(dogstatsd);

// Track HTTP requests
```

```
app.use((req, res, next) => {
  const start = Date.now();

  res.on('finish', () => {
    const duration = Date.now() - start;

    const tags = [
      `method:${req.method}`,
      `path:${req.route?.path || 'unknown'}`,
      `status:${res.statusCode}`
    ];

    // Increment request counter
    metrics.incrementCounter('http.requests', 1, tags);

    // Record duration
    metrics.histogram('http.request.duration', duration, tags);

    // Track errors separately
    if (res.statusCode >= 500) {
      metrics.incrementCounter('http.errors', 1, tags);
    }
  });

  next();
});

// Business metrics
app.post('/orders', async (req, res) => {
  try {
    const order = await createOrder(req.body);

    // Track order creation
    metrics.incrementCounter('orders.created', 1, [
      `user_tier:${req.user.tier}`
    ]);

    // Track order value
    metrics.histogram('orders.value', order.total);

    // Track active orders
    const activeOrders = await getActiveOrderCount();
    metrics.gauge('orders.active', activeOrders);

    res.json(order);

  } catch (error) {
```

```
    metrics.incrementCounter('orders.failed', 1, [
      `reason:${error.code}`
    ]);

    res.status(500).json({ error: error.message });
  }
});

// Custom business metrics
setInterval(async () => {
  // Revenue metrics
  const revenue = await getHourlyRevenue();
  metrics.gauge('revenue.hourly', revenue);

  // User metrics
  const activeUsers = await getActiveUserCount();
  metrics.gauge('users.active', activeUsers);

  // Database metrics
  const dbConnections = await getDBConnectionCount();
  metrics.gauge('database.connections', dbConnections);

}, 60000);  // Every minute
```

Complete Monitoring Setup

javascript

```
class MonitoringService {
  constructor() {
    // Metrics
    this.prometheus = this.initPrometheus();
    this.datadog = this.initDatadog();

    // Logging
    this.logger = this.initLogger();

    // Tracing
    this.tracer = this.initTracer();
  }

  initPrometheus() {
    const register = new promClient.Registry();
    promClient.collectDefaultMetrics({ register });

    return {
      register,
      counter: (name, help, labels) =>
        new promClient.Counter({ name, help, labelNames: labels, registers: [register] }),
      gauge: (name, help, labels) =>
        new promClient.Gauge({ name, help, labelNames: labels, registers: [register] }),
      histogram: (name, help, labels, buckets) =>
        new promClient.Histogram({ name, help, labelNames: labels, buckets, registers: [register] })
    };
  }

  initDatadog() {
    return new StatsD('localhost', 8125);
  }

  initLogger() {
    return winston.createLogger({
      level: 'info',
      format: winston.format.json(),
      transports: [
        new winston.transports.File({ filename: 'error.log', level: 'error' }),
        new winston.transports.File({ filename: 'combined.log' })
      ]
    });
  }

  initTracer() {
    // OpenTelemetry setup
    return provider.getTracer('my-service');
```

```
}

// Unified interface
trackRequest(req, res, duration) {
  const tags = {
    method: req.method,
    path: req.path,
    status: res.statusCode
  };

  // Log
  this.logger.info('http_request', {
    ...tags,
    duration,
    correlationId: req.correlationId
  });

  // Metrics (both Prometheus and Datadog)
  this.prometheus.counter('http_requests_total', 'HTTP requests', ['method', 'path', 'status'])
    .inc(tags);

  this.prometheus.histogram(
    'http_request_duration_seconds',
    'Request duration',
    ['method', 'path'],
    [0.1, 0.5, 1, 2, 5]
  ).observe({ method: tags.method, path: tags.path }, duration / 1000);

  this.datadog.increment('http.requests', 1, [
    `method:${tags.method}`,
    `path:${tags.path}`,
    `status:${tags.status}`
  ]);

  this.datadog.histogram('http.duration', duration, [
    `method:${tags.method}`,
    `path:${tags.path}`
  ]);
}

trackError(error, context) {
  // Log with full context
  this.logger.error('error', {
    message: error.message,
    stack: error.stack,
    ...context
  });
```

```
    // Increment error counter
    this.prometheus.counter('errors_total', 'Errors', ['type'])
      .inc({ type: error.code || 'unknown' });

    this.datadog.increment('errors', 1, [
      `type:${error.code || 'unknown'}`
    ]);
  }

  trackBusinessEvent(eventType, data) {
    // Log
    this.logger.info(eventType, data);

    // Metrics
    this.prometheus.counter('business_events_total', 'Business events', ['type'])
      .inc({ type: eventType });

    this.datadog.increment('business.events', 1, [`type:${eventType}`]);
  }

  async getMetrics() {
    return await this.prometheus.register.metrics();
  }
}

// Global monitoring instance
const monitoring = new MonitoringService();

// Use throughout application
app.use((req, res, next) => {
  const start = Date.now();

  res.on('finish', () => {
    monitoring.trackRequest(req, res, Date.now() - start);
  });

  next();
});

app.post('/orders', async (req, res) => {
  try {
    const order = await createOrder(req.body);

    monitoring.trackBusinessEvent('order_created', {
      orderId: order.id,
      total: order.total
```

```
    });

    res.json(order);

  } catch (error) {
    monitoring.trackError(error, {
      userId: req.user.id,
      path: req.path
    });

    res.status(500).json({ error: error.message });
  }
});

// Expose metrics
app.get('/metrics', async (req, res) => {
  res.set('Content-Type', 'text/plain');
  res.send(await monitoring.getMetrics());
});
```

Key Takeaways

1. **Three Pillars:**
   - Logs: What happened (events)
   - Metrics: How much (numbers)
   - Traces: Where (request path)
   - Use all three together!
2. **Logging:**
   - Structured (JSON)
   - Correlation IDs
   - Appropriate levels
   - Sanitize sensitive data
3. **Metrics:**
   - Four Golden Signals (latency, traffic, errors, saturation)
   - Counter, Gauge, Histogram
   - Prometheus or Datadog
4. **Tracing:**

- Distributed tracing across services
- Find bottlenecks
- OpenTelemetry standard

5. **Alerting:**
   - SLIs, SLOs, SLAs
   - Error budgets
   - Severity levels
   - Escalation policies
6. **Tools:**
   - Prometheus + Grafana: Open source, powerful
   - ELK Stack: Log aggregation and search
   - Datadog: All-in-one, managed

Practice Problems

1. Design a monitoring strategy for a microservices architecture with 10 services. What metrics would you track?
2. You have 99.9% SLO. You've had 30 minutes downtime this month. Can you deploy a risky feature?
3. Design an alerting policy for an e-commerce site. What alerts and thresholds?

Ready for the next chapter or want to explore monitoring deeper?