Automatically Detect the Performance & Scalability Issues in Distributed Architectures

“And integrate this in your delivery pipeline with 🌐 keptn”

Andreas Grabner
DevOps Activist at Dynatrace
DevRel for Keptn
@grabnerandi, https://www.linkedin.com/in/grabnerandi

Follow us @keptnProject
Star us @ https://github.com/keptn/keptn
Slack Us @ https://slack.keptn.sh
How distributed systems look like!

Or how they shouldn’t ...
Distributed Trace Example from StepStone (AWS Summit Berlin 2019)
Dependencies in the infrastructure: 323 k8s Nodes
Dependencies in the infrastructure: 323 k8s Nodes
4229 k8s Pods
Lesson Learned: When moving to a more distributed architecture ...

...you also grow your dependencies...
... and the potential impact of a failure grows!

![4 Impacted Services](image1)

<table>
<thead>
<tr>
<th>Service</th>
<th>Response Time Degradation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prep: svc</td>
<td>The current response time (350 ms) exceeds the auto-detected baseline (143 ms) by 144%</td>
</tr>
<tr>
<td>Prep: svc</td>
<td>The current response time (258 ms) exceeds the auto-detected baseline (3.1 ms) by 8,230%</td>
</tr>
<tr>
<td>Prep: service g2-4</td>
<td>The current response time (188 ms) exceeds the auto-detected baseline (3.02 ms) by 6,012%</td>
</tr>
</tbody>
</table>

![1 Bad Update](image2)

**Root Cause**

Based on our dependency analysis all incidents have the same root cause:

<table>
<thead>
<tr>
<th>Service</th>
<th>Failure Rate Increase by a Failure Rate Increase to 0.53%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prep: /products</td>
<td>Affected requests: 3,096 /min, All dynamic requests</td>
</tr>
<tr>
<td>Prep: svc</td>
<td>Affected requests: 407 /min, All dynamic requests</td>
</tr>
</tbody>
</table>

**Visual Resolution Path**

![Diagram](image3)

Because of all dependencies

In distributed architectures we need to answer: Who is depending on me? What is the risk of change?
Common Distributed Architectural Patterns

Patterns I’ve seen in > 90% of the problems I analyzed
There are more – and we only have time to cover some today

1. N+1 call
2. N+1 query
3. Payload flood
4. Granularity
5. Tight Coupling
6. Inefficient Service Flow
7. Timeouts, Retries, Backoff
8. Dependencies

More recorded presentations on problem patterns:
• Java and Performance: Biggest Mistake - https://www.youtube.com/watch?v=IBkxiWmjM-g (SFO Java Meetup)
• Top Performance Challenges: https://www.youtube.com/watch?v=QypHTQr2RXk (Confitura 2019)
N + 1 Call Pattern

Or better: 1 + N

1 initial call + 1 Call per N results
N+1 Call Pattern

Monolithic Code

public double getTotalQuote(Products[] products) {
    double quote = 0;
    for (Product product : products) {
        quote += product.getQuote();
    }
    return quote;
}

“Works” well within a single process

Extract into Service?
N+1 Call Pattern across distributed “Product Service”

1 call to Quote Service = 44 calls to Product Service
Subtotal: 243
N + 1 Query Pattern

Similar to N +1 Call Pattern but focused on database queries
N+1 Query Pattern

1 call to Quote Service = 87 calls to DB
Cascading N+1 Query Pattern: This is a single End-2-End Distributed Trace

26k Database Calls
Payload Flood

AKA – sending useless information across the network
Payload Flood: “Doc Creation” sequential across distributed services
Payload Flood in numbers: Full DOC sent between distributed services

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>/docs/create/test</td>
<td>1</td>
<td>31</td>
<td>31</td>
<td>95</td>
<td>8132.53</td>
<td>8132.53</td>
</tr>
<tr>
<td>C/S</td>
<td><a href="http://127.0.0.1:45751/doc-ship/ship">http://127.0.0.1:45751/doc-ship/ship</a></td>
<td>6</td>
<td>23211833</td>
<td>139270997</td>
<td>21605081</td>
<td>8132.53</td>
<td>48795.20</td>
</tr>
<tr>
<td>C/S</td>
<td><a href="http://127.0.0.1:45748/doc-proc/processdoc">http://127.0.0.1:45748/doc-proc/processdoc</a></td>
<td>2</td>
<td>29175519</td>
<td>58351038</td>
<td>10725268</td>
<td>8132.53</td>
<td>16265.07</td>
</tr>
<tr>
<td>C/S</td>
<td><a href="http://127.0.0.1:45739/doc-sign/sign">http://127.0.0.1:45739/doc-sign/sign</a></td>
<td>6</td>
<td>23211833</td>
<td>139270997</td>
<td>20048787</td>
<td>8132.53</td>
<td>48795.20</td>
</tr>
<tr>
<td>C/S</td>
<td><a href="http://127.0.0.1:45776/doc-trans/transform">http://127.0.0.1:45776/doc-trans/transform</a></td>
<td>6</td>
<td>23211833</td>
<td>139270997</td>
<td>18526033</td>
<td>8132.53</td>
<td>48795.20</td>
</tr>
</tbody>
</table>
Refactor: Only send relevant data to specialized services

69MB vs 31.6MB
Inefficient Service Flow
drawing parallels to Web Performance Optimization
SFPO (Service Flow & Performance Optimization) has to teach us how to optimize (micro)service dependencies through Service Flows.
Especially useful to identify: inefficient 3rd party services, recursive call chains, N+1 Query Patterns, loading too much data, no data caching, ... -> sounds very familiar to WPO
Classical cascading effect of recursive service calls!
Common Distributed Architectural Patterns

Recap and overview of Metrics used for pattern detection!
Recap - Common Distributed Patterns + Metrics to look at

1. N+1 call: *same Service Invocations per Request*
2. N+1 query: *same SQL Invocations per Request*
3. Payload flood: *Transfer Size!*
4. Granularity: *# of Service Invocations across End-2-End Transaction*
5. Tight Coupling: *Ratio between Service Invocations*
6. Inefficient Service Flow: *# of Involved Services, # of Calls to each Service*
7. Timeouts, Retries, Backoff: *Pool Utilization, …*
8. Dependencies: *# of Incoming & Outcoming Dependencies*

More recorded presentations on problem patterns:
- Java and Performance: Biggest Mistake - [https://www.youtube.com/watch?v=IBkxiWmjM-g](https://www.youtube.com/watch?v=IBkxiWmjM-g) (SFO Java Meetup)
- Top Performance Challenges: [https://www.youtube.com/watch?v=QypHTQr2RXk](https://www.youtube.com/watch?v=QypHTQr2RXk) (Confitura 2019)
Can we automate pattern detection?

If we can detect them on a dashboard – we should be able to automate!
Keptn automates analysis through SLIs/SLOs

Instead of manually detecting patterns and comparing metrics, Keptn automates that process based on SLIs & SLOs.

Integrate in Testing, Delivery & Auto-Remediation.
Introducing Keptn

Declarative, extensible automation of SLO-driven delivery, quality gates & remediation

Keptn from 10000ft: Declarative Workflows + Event-Triggered Actions

**Application Plane (Process Definition)**
Define overall process for delivery and operations

**Control Plane**
Follow application logic and communicate/configure required services

**Execution Plane (Tool Definition)**

**Eventing**

- **config.change**: artifact:x:y
- **deploy.finished**: http://service1
- **tests.finished**: OK
- **evaluation.done**: 98% Score
- **problem.open**: High Failure

**Site Reliability Engineer**

DevOps

Developer

Artifact / Microservice

**shipyard.yaml**
- dev: direct, functional, SLO
- staging: B/G, perf, SLO
- prod: canary, real-user, SLA

**remediation.yaml**
- high-failure-rate:
- scaleup, rollback
- full-disk:
- cleandir, adjustlog-level

**uniform.yaml**
- config-change*: helm
- deploy*: JMeter
- deploy-finish: Lighthouse
- problem*: Remediation
  all: Slack, Dynatrace

**Execution Plane**

- **Config Service** (Git, ...)
- **Deploy Service** (Helm, Jenkins ...)
- **Test Service** (JMeter, Neotys, ...)
- **Validation Service** (Keptn Lighthouse ...)
- **Monitoring Service** (Prometheus, Dynatrace, ...)
- **Remediation Service** (Keptn Remediation, SNOW ...)

**Site Reliability Engineer**

DevOps

Developer

Artifact / Microservice

**site reliability engineer**

DevOps

Developer

Artifact / Microservice
Use Case #1

Automated Architecture & Performance Validation

Through event-based SLI/SLO-based Quality Gates

Root Cause: Lengthy manual approval in existing delivery pipelines

Build Deploy to „Test“ Run Test In „Test“ Manual Approval ~30-60min Promote to „Staging“

Looking at all these dashboards and data points is time-consuming and slows down the process!

Identify / Optimize Architectural Patterns
Recursive Calls, N+1 Call Pattern, Chatty Interfaces, No Caching Layer ...

Identify Performance Hotspots
CPU, Memory, I/O, ...
Inspired by Dynatrace’s internal „Performance Signature as Code“

“Performance Signature” for Build Nov 16

“Performance Signature” for Build Nov 17

“Performance Signature” for every Build

“Multiple Metrics” compared to prev Timeframe

Simple Regression Detection per Metric

https://www.neotys.com/performance-advisory-council/thomas_steinmaurer
SLI/SLO-based evaluation implementation in Keptn

SLIs defined per SLI Provider as YAML
SLI Provider specific queries, e.g: Dynatrace Metrics Query

```yaml
indicators:
  error_rate: "builtin:service.errors.total.count:merge(0):avg"
  count_dbcalls: "calc:service.toptestdbcalls:merge(0):sum"
  jvm_memory: "builtin:tech.jvm.memory.pool.committed:merge(0):sum"
```

SLOs defined on Keptn Service Level as YAML
List of objectives with fixed or relative pass & warn criteria

```yaml
objectives:
  - sli: error_rate
    pass:
      criteria:
        - "<=1" # We expect a max error rate of 1%
    sli: jvm_memory
    sli: count_dbcalls
    pass:
      criteria:
        - "=+2%" # We allow a 2% increase in DB Calls to previous runs
    warning:
      criteria:
        - "<=10" # We expect no more than 10 DB Calls per TX
  total_score:
    pass: "90%"
    warning: "75%"
```

```
$ keptn start-evaluation 30m myservice sli.yaml slo.yaml
```

**Quality Gates**

1. Queries SLI Providers with SLI Definitions & Timeframe
2. Total Score
3. Scores SLIs
4. Total Score

<table>
<thead>
<tr>
<th>SLI Value</th>
<th>SLI Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 DB Calls</td>
<td>0.5</td>
</tr>
<tr>
<td>360MB</td>
<td>1.0</td>
</tr>
<tr>
<td>4.3%</td>
<td>0.0</td>
</tr>
<tr>
<td>123</td>
<td>info</td>
</tr>
</tbody>
</table>

Dysnatec, Prometheus, Neoload, Tool X
Solution: Automate Approval through SLI/SLO-based Quality Gates

**Build**

**Deploy to „Test”**

**Run Test In „Test”**

**Manual Approval**

~30-60min

**Promote to „Staging”**

**Deploy to „Test”**

**Build**

**Deploy to „Test”**

**Run Test In „Test” w Tagging**

**Trigger Quality Gate**

~1min

**Wait for Result**

**Promote to „Staging”**

**Deploy to „Test”**

**SLI & SLO**

- \( \text{Rt(p95)} < 500\text{ms} \)
- \( \# \text{ofSQLs} \leq 5 \)
- \( \text{cpu(max)} < 80\% \)
- \( \text{Java GC} < 2\% \)

...
Demo: Automated SLI/SLO Validation based on Dynatrace Dashboards

You: Just build a dashboard!

keptn: Automates the analysis!

15.5/16 (97%)
8/16 (50%)
User Example: Automating Build Approvals using Keptn’s SLIs/SLOs in GitLab

Automated SLI/SLO based Quality Gates

Trigger Evaluation

87.5%: passed
Use Case #2

Automated Remediation

Through a closed loop event-driven remediation workflow

Keptn – Closed-Loop Remediation with Keptn 0.7

Problem: Conversion Rate Dropped  
Root Cause: CPU Pressure

version: 0.2.0  
kind: Remediation
metadata:
  name: remediation-eCommerce
spec:
  remediations:
    - problemType: Conversion Rate Dropped
      actionsOnOpen:
        - name: Scaling ReplicaSet by 1
          action: scaling
          values:
            increment: +1
        - name: Stop Ad Campaign
          action: googleadtoggle
          values:
            enable: off
            campaign: $campaignid
Too risky? Start in Pre-Prod leveraging Chaos Engineering to define & test Auto-Remediation

Problem: Slow ReportGen Service
Root Cause: High CPU on host

version: 0.2.0
kind: Remediation
metadata:
  name: remediation-ecommerce
spec:
  remediations:
    - problemType: High CPU on ReportGen
      actionsOnOpen:
        - name: Stop Traffic
          action: configureLoadBalancer
          values:
            action: stopTraffic
            ip: $problem.hostIp
        - name: Restart Process
          action: executeAnsible
          values:
            script: restartProcess
            process: $problem.processID
To wrap it up ...

What you should have learned today is that
Automate Distributed Problem Detection & Remediation

#1 Understand your Patterns & Drive Metrics

#2 Derive and monitor your metrics (SLIs/SLOs)

#3 Let Keptn automate the analysis

#4 Integrate Keptn into Delivery & Operations
“And integrate this in your delivery pipeline with 🔄 keptn”
More examples
Example #1: Building Monitoring for AWS

- **AWS CloudWatch API**
  - **Single Fetch**
    - 907 Calls
    - 41 sec
    - 97 threads
    - $0.01 / 1000 Calls
  - **Bulk Fetch**
    - 104 Calls
    - 21 sec
    - 92 threads
    - $$

---

DynamoDB overview

- 2 Availability Zones
- 12 Instances
- 12 Tables
- 11 EBS volumes
- 222 S3 buckets
- 1 Auto Scaling group
- 1 LB instance
- 29 Lambda functions
Tight Coupling
When “Breaking the Monolith” be aware ...

Granularity
Granularity: Encryption carved out into separate service

Documents

Doc Processor
Doc Transformer
Doc Signer
Doc Shipments

Doc Encryption

316

118
Dependencies
Look beyond the “Tip of the Iceberg”: Understanding Dependencies is critical!
Example from StepStone (AWS Summit Berlin 2019)
The services and applications listed below make calls to this service. The tree view represents the sequence of services and application user actions that led to this service call, beginning with the page load or user action in the browser that triggered the sequence. Click to see which specific requests and user actions called this service.

### Incoming requests to this service

<table>
<thead>
<tr>
<th>Service</th>
<th>Requests</th>
<th>Failed Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>JourneyService</td>
<td>6.6%</td>
<td>0</td>
</tr>
<tr>
<td>eT-demo-1-BusinessBackend</td>
<td>6.6%</td>
<td>0</td>
</tr>
<tr>
<td>EasyTravelBackend/Webservice</td>
<td>6.8%</td>
<td>0</td>
</tr>
<tr>
<td>nginxForMicroservices</td>
<td>5.0%</td>
<td>0</td>
</tr>
<tr>
<td>MicroJourneyService</td>
<td>5.1%</td>
<td>0</td>
</tr>
<tr>
<td>nginxForCustomerFrontend</td>
<td>5.2%</td>
<td>0</td>
</tr>
<tr>
<td>easyTravel Customer Frontend</td>
<td>1.2%</td>
<td>0</td>
</tr>
<tr>
<td>Varnish:8079</td>
<td>1.2%</td>
<td>0</td>
</tr>
<tr>
<td>easyTravel Customer Frontend</td>
<td>0.3%</td>
<td>0</td>
</tr>
<tr>
<td>Varnish Cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dotNetFrontend/easyTravel_x64</td>
<td>1.8%</td>
<td>0</td>
</tr>
<tr>
<td>Application</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Timeouts, Retries & Backoff

Credits go to Adrian Hornsby (@adhorn)
Bad Timeout & Retry Settings

Timeout client side = 10s

Timeout backend = default (e.g: 60s)

User 1

Retry

Retry

Retry

Retry

INSERT

INSERT

INSERT

ERROR: Failed to get connection from pool

Backoff between Retries

Timeout client side = 10s
Wait 2s before Retry
Wait 4s before Retry
Wait 8s before Retry
Wait 16s before Retry

Timeout backend = 10s – time elapsed

Simple Exponential Backoff is not enough: Add Jitter

From Adrian Hornsby (@adhorn): https://speakerdeck.com/adhorn/resiliency-and-availability-design-patterns-3742b5ba-e013-4f50-8512-00a65775f478?slide=34