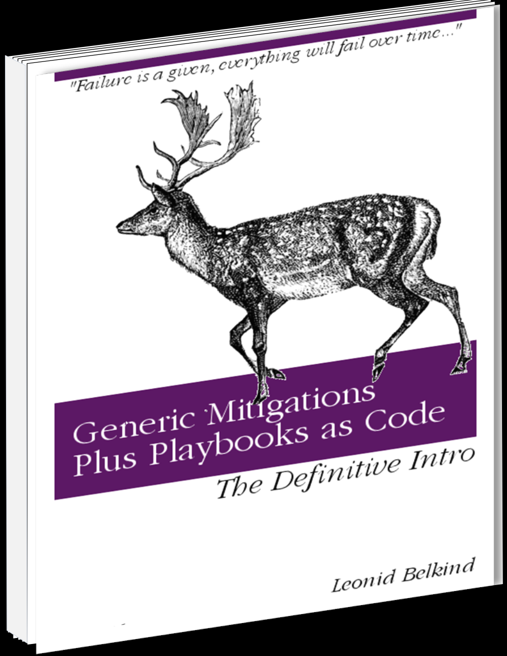
StackPulse

Using Automation for Generic Mitigations in Production

May 20, 2021

Presenter: *Leonid Belkind, Co-Founder & CTO, StackPulse*

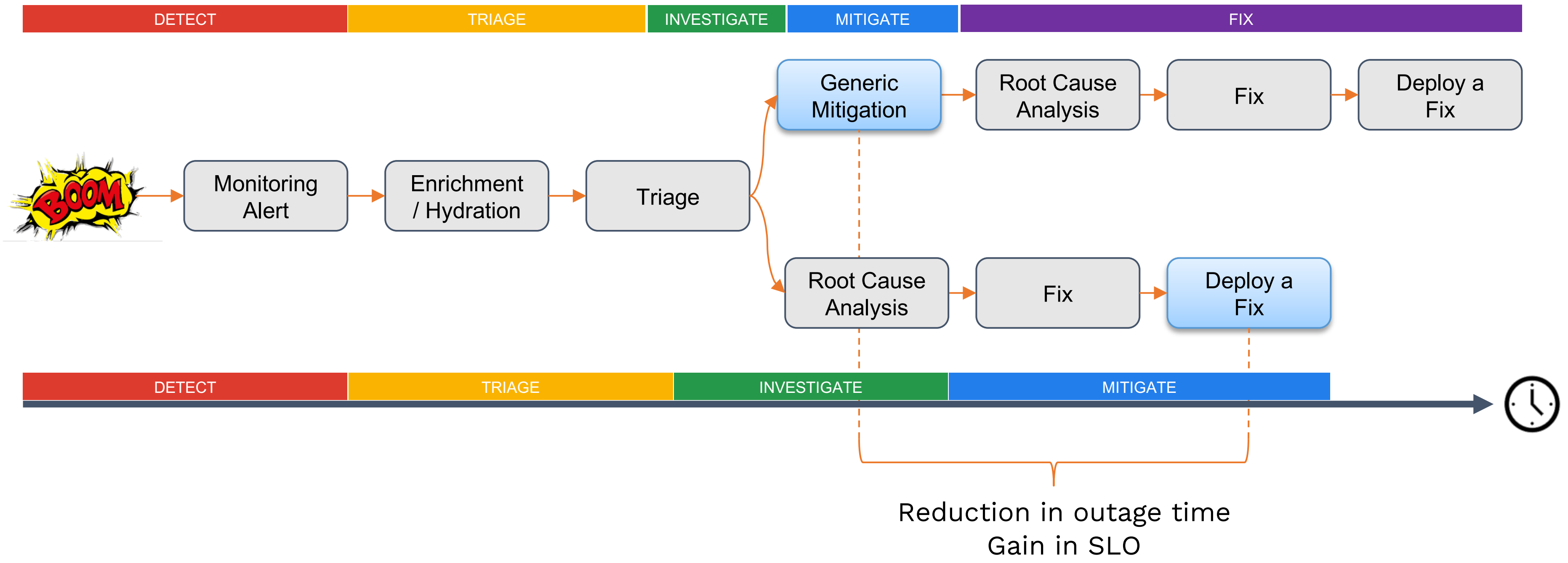


DevOps INSTITUTE

Agenda

- Generic Mitigations
- Automation: Playbooks-as-Code
- Bringing it all together...
  - Samples
  - Architecture Blueprint
  - Main principles

Typical timeline of an outage



So, what kind of Generic Mitigations do you have in mind?

- Rollback (Business Logic, Configuration, Data)

  The ability to safely return to a working state. This might sound simple, rolling back a deployment of a single component, but, actually, performing this in a multi-component environment, with dependencies and evolving data schemas is not straight forward at all.

- Upsize / Downsize

  The ability to increase / decrease amount of replicas of a certain component, while continuously handling the production traffic. Ability to do that controlling the system externally without invoking infrastructure and application specialists to perform dangerous changes in production.

- Drain and Flip Traffic

  The ability to gradually drain the connections from a specific instance / site / cluster (experiencing errors) and transfer them to another one. Doing so safely, without involving ad-hoc operations in production.

Is that it? .... Of course not

- Quarantine

  After identifying a “bad” instance in a cluster, remove it from rotation ensuring that the other instances continue handling traffic without impact on the users. Then, investigate the root cause of the problem.

- Block List

  Block a specific user / account / session / external source of problematic requests to make sure that it doesn’t impact the overall delivery of your service to the rest. Potentially, add specific quotas / guardrails on this particular source rather than just blocking it.

- Disable a Noisy Neighbour

  Identify the source of “noisy neighbour” (for example, in a database, sending long queries that require too much resources) and terminate the queries / sessions that impact others.

TL;DR

“**Generic Mitigations**” is a practice of improving SLOs and returning the service to operational state faster, without compromising on Root Cause Analysis and good software engineering practices.

Building Generic Mitigations and testing them (to build confidence to apply them to production) is a very important aspect of becoming proficient in building resilient systems.

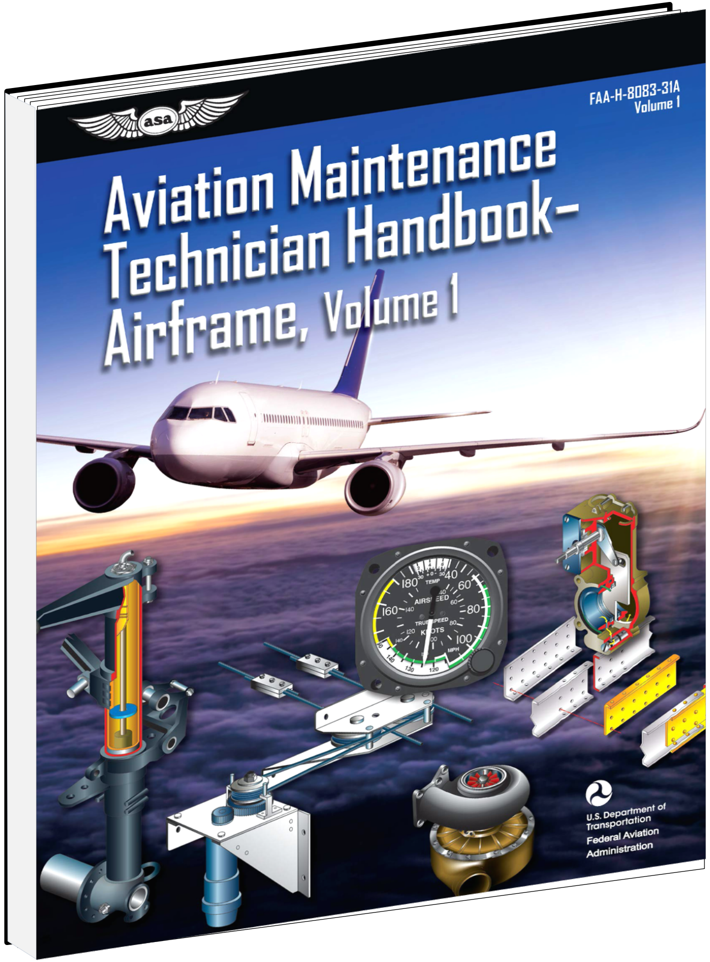
Playbooks
(a.k.a. Runbooks)

A playbook includes process workflows, standard operating procedures, and cultural values that shape a consistent response — the play.

accenture

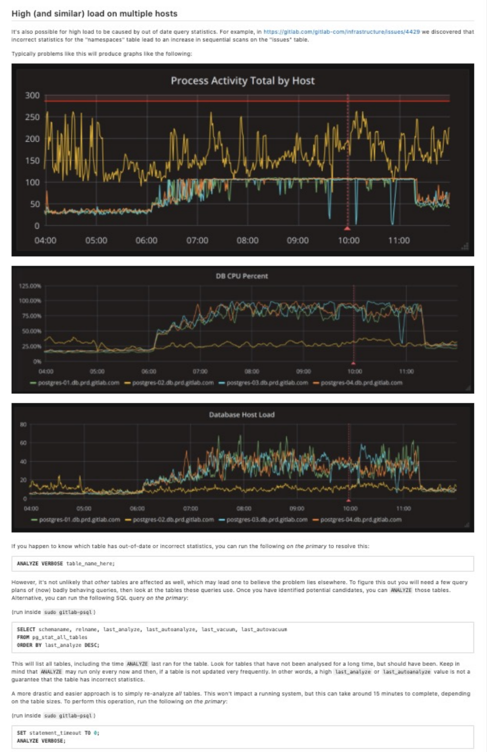
Enable consistent and prompt responses to failure scenarios by documenting the investigation process in playbooks. Playbooks are the predefined steps to perform to identify and resolve an issue.

amazon web services™



StackPulse

Sample Playbooks



High (and similar) load on multiple hosts

It's also possible for high load to be caused by out of date query statistics. For example, in https://gitlab.com/gitlab-com/infrastructure/issues/4429 we discovered that incorrect statistics for the "namespaces" table lead to an increase in sequential scans on the "issues" table.

Typically problems like this will produce graphs like the following:

If you happen to know which table has out-of-date or incorrect statistics, you can run the following on the primary to resolve this:

```
ANALYZE VERBOSE table_name_here;
```

However, it's not unlikely that other tables are affected as well, which may lead one to believe the problem lies elsewhere. To figure this out you will need a few query plans of (now) badly behaving queries, then look at the tables these queries use. Once you have identified potential candidates, you can ANALYZE those tables. Alternatively, you can run the following SQL query on the primary:

(run inside sudo gitlab-psql)

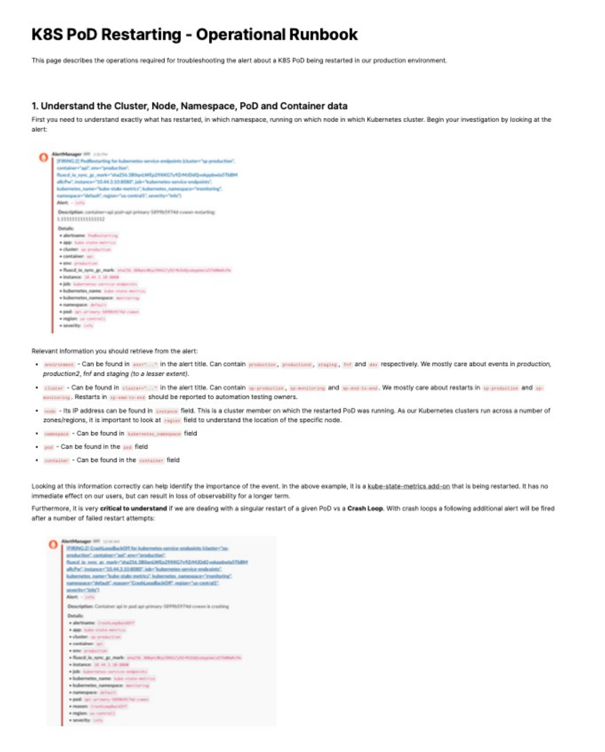
```
SELECT schemaname, relname, last_analyze, last_autoanalyze, last_vacuum, last_autovacuum
FROM pg_stat_all_tables
ORDER BY last_analyze DESC;
```

This will list all tables, including the time ANALYZE last ran for the table. Look for tables that have not been analysed for a long time, but should have been. Keep in mind that ANALYZE may run only every now and then, if a table is not updated very frequently. In other words, a high last_analyze or last_autoanalyze value is not a guarantee that the table has incorrect statistics.

A more drastic and easier approach is to simply re-analyze all tables. This won't impact a running system, but this can take around 15 minutes to complete, depending on the table sizes. To perform this operation, run the following on the primary:

(run inside sudo gitlab-psql)

```
SET statement_timeout TO 0;
ANALYZE VERBOSE;
```



K8S PoD Restarting - Operational Runbook

This page describes the operations required for troubleshooting the alert about a K8S PoD being restarted in our production environment.

1. Understand the Cluster, Node, Namespace, PoD and Container data

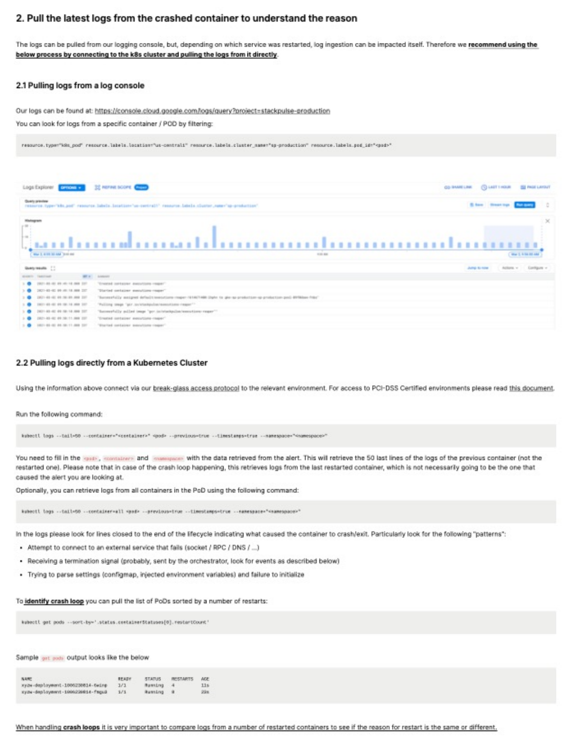
First you need to understand exactly what has restarted, in which namespace, running on which node in which Kubernetes cluster. Begin your investigation by looking at the alert:

Relevant information you should retrieve from the alert:

- environment - Can be found in env="..." in the alert title. Can contain production, production2, staging, fnf and dev respectively. We mostly care about events in production, production2, fnf and staging (to a lesser extent).
- cluster - Can be found in cluster="..." in the alert title. Can contain sp-production, sp-monitoring and sp-end-to-end. We mostly care about restarts in sp-production and sp-monitoring. Restarts in sp-end-to-end should be reported to automation testing owners.
- node - Its IP address can be found in instance field. This is a cluster member on which the restarted PoD was running. As our Kubernetes clusters run across a number of zones/regions, it is important to look at region field to understand the location of the specific node.
- namespace - Can be found in kubernetes_namespace field
- pod - Can be found in the pod field
- container - Can be found in the container field

Looking at this information correctly can help identify the importance of the event. In the above example, it is a kube-state-metrics add-on that is being restarted. It has no immediate effect on our users, but can result in loss of observability for a longer term.

Furthermore, it is very **critical to understand** if we are dealing with a singular restart of a given PoD vs a **Crash Loop**. With crash loops a following additional alert will be fired after a number of failed restart attempts:



2. Pull the latest logs from the crashed container to understand the reason

The logs can be pulled from our logging console, but, depending on which service was restarted, log ingestion can be impacted itself. Therefore we **recommend using the below process by connecting to the k8s cluster and pulling the logs from it directly**.

2.1 Pulling logs from a log console

Our logs can be found at: https://console.cloud.google.com/logs/query?project=stackpulse-production

You can look for logs from a specific container / POD by filtering:

2.2 Pulling logs directly from a Kubernetes Cluster

Using the information above connect via our break-glass access protocol to the relevant environment. For access to PCI-DSS Certified environments please read this document.

Run the following command:

You need to fill in the <pod>, <container> and <namespace> with the data retrieved from the alert. This will retrieve the 50 last lines of the logs of the previous container (not the restarted one). Please note that in case of the crash loop happening, this retrieves logs from the last restarted container, which is not necessarily going to be the one that caused the alert you are looking at.

Optionally, you can retrieve logs from all containers in the PoD using the following command:

In the logs please look for lines closed to the end of the lifecycle indicating what caused the container to crash/exit. Particularly look for the following "patterns":

- Attempt to connect to an external service that fails (socket / RPC / DNS / ...)
- Receiving a termination signal (probably, sent by the orchestrator, look for events as described below)
- Trying to parse settings (configmap, injected environment variables) and failure to initialize

To **identify crash loop** you can pull the list of PoDs sorted by a number of restarts:

Sample get pods output looks like the below

When handling **crash loops** it is very important to compare logs from a number of restarted containers to see if the reason for restart is the same or different.

StackPulse

Playbooks – The essense

Playbooks provide (hopefully) **step-by-step guides for human operators** to ensure repeatable and consistent response to incident situations.

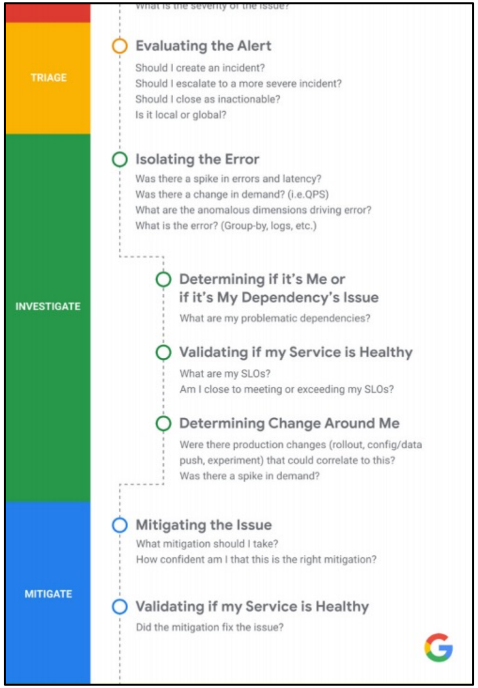
But why?

If we have repetitive technical step-by-step procedures, isn't **automation** the better option to **ensure efficiency and consistency**?

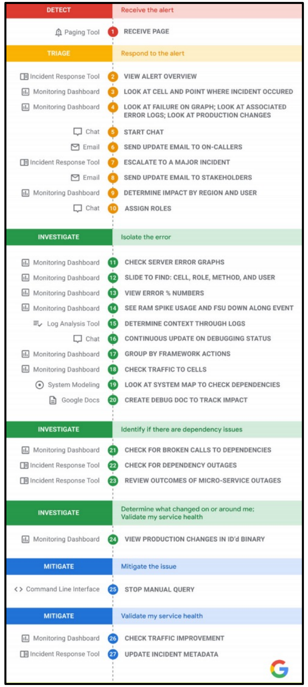
Enter Playbooks-as-Code

Google Research on Incident Response

Building Blocks



User Journey



StackPulse

* From "Debugging Incidents in Google's Distributed Systems, Charisma Chan and Beth Cooper, 2020

What are playbooks-as-code

Playbooks-as-code are deterministic manual operator instructions **converted into** automation processes

Chat 5 START CHAT	Automate Chat Channels, Notifications
Email 6 SEND UPDATE EMAIL TO ON-CALLERS	Automate Messaging / Paging to Stakeholders
Incident Response Tool 7 ESCALATE TO A MAJOR INCIDENT	Automatically Update Incident Based on Enriched Data
Monitoring Dashboard 9 DETERMINE IMPACT BY REGION AND USER	Automate Pulling Impact Data from Logs/Context
Monitoring Dashboard 11 CHECK SERVER ERROR GRAPHS	Automate Diagnostics, Look for Suspicious Signals
Monitoring Dashboard 13 VIEW ERROR % NUMBERS	Automate Log Queries, Calculate Data on Pulled Logs
Monitoring Dashboard 24 VIEW PRODUCTION CHANGES IN ID'd BINARY	Automatically Connect to SCM/CI Pipeline
Command Line Interface 25 STOP MANUAL QUERY	Automate Remediative Actions / CLI / API / ...

Playbooks-as-Code → The Approach

- Similar to CI/CD Pipelines or Automated Tests, define playbook workflows as high-level code/configuration

- Apply Software Engineering principles to Incident Response Playbooks
  - Break down into modules
  - Handle specific tasks, as a part of Incident handling
  - Think of arguments / parameters and encapsulation allowing re-use
  - Consider sharing between teams in the organization and between organizations
  - Visualize / Troubleshoot / Audit execution
  - Apply SDLC, GitOps, ...

- Have a Playbooks-as-Code orchestrator separate from our application infrastructure

Playbook-as-Code Sample

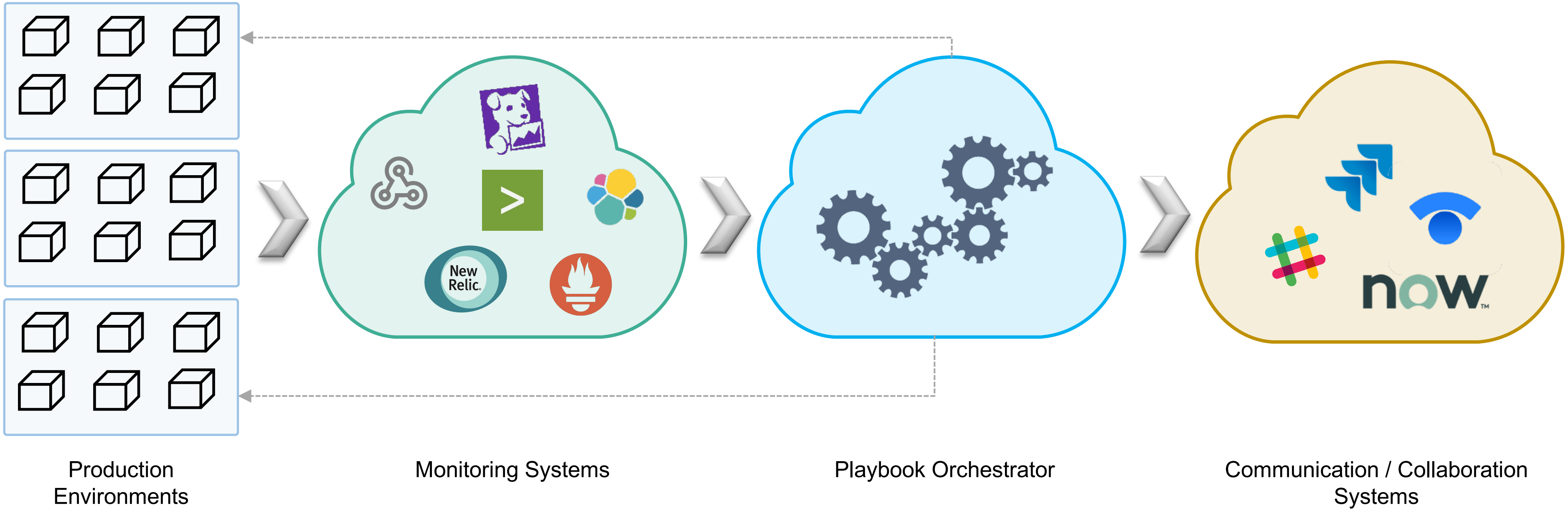
<> Command Line Interface 25 RETRIEVE DATA FROM VIRTUAL MACHINE

```
- name: stackpulse/public/ssh/command
  id: ssh_command
  env:
     USERNAME: "{{  $.params.UserName }}”
     HOSTNAME: "{{  $.params.ServerAddress }}”
     COMMAND: df -k
     PRIVATE_KEY: '{{ secret "SSH_KEY"}}’
```

Chat 5 PROVIDE INFORMATION TO RESPONDERS

````
- name: stackpulse/public/slack/message
  id: slack_send_message
  env:
     MESSAGE_TEXT: |
       The filesystems for server {{ $.params.ServerAddress }}
       ```
       {{ $.ssh_command.output }}
       ```
     RECIPIENTS: "alerts"
````

Typical Architecture for Playbooks-as-Code Orchestration



Summary

- Well-defined, pre-rehearsed and deterministic processes are a MUST to ensure efficient handling of incidents

- A “library” of Generic Mitigations ensures ability to reduce outages

- “Documenting Step-by-Step Directions for Human Operators” is not the way to go. There is a better alternative, as proven by:
  - Automated Testing
  - Automated Integration / Deployment
  - Infrastructure-as-Code
  - ...

- Think of actions taking place during incidents / alerts just as of another aspect of “code” and act accordingly

Thank You!

Questions?
leonid@stackpulse.com

StackPulse