Alerting Infrastructure

Improving Signal to Noise ratio.

Status: Complete (2019-11-22)

Authors: Filippo Giunchedi / Keith Herron

Introduction

Problem statement

As of August 2019 at the Wikimedia Foundation, we use a suite of tools to monitor the health of our services and infrastructure. This tooling is effective at generating alerts, but leaves much to be desired in the areas of alert volume and intelligibility. Alerts are frequently false positives, duplicates, or are sent in rapid succession from multiple co-dependent services. At the present time we are lacking…

- Policy
    - Guidelines describing which piece(s) of tooling should implement which types of alerting
    - Guidelines for alert thresholds
    - Well defined alert requirements and escalation strategies (e.g. alert must be actionable, and have clear initial remediation steps, ownership must be clear)
    - A clear and consistent alert delivery strategy
    - Clear and consistent documentation for alert remediation (runbooks)

    - Clear definition of appropriate alert granularity (e.g. alert should point clearly and directly to a problem location which as much granularity as possible, as opposed to “I feel a great disturbance in the force”)
    - A well defined mapping of services to maintainers/owners/sub-teams
    - Methods to distribute alert triage load, and associated hands-on experience/knowledge-transfer evenly across teams.
- Technical
    - Monitoring tooling (alert producers)
        - Robust alert/service dependency tracking
        - Multi-tenancy within the monitoring system
        - Uniform monitoring on a per-service and per-cluster level (as opposed to host based monitoring)
        - Tracking of production change events in monitoring tooling (annotations, etc.)
        - Integrations between configuration management and monitoring (e.g. squelch alerting on depooled host, prevent alerts during host (re)builds)
        - Long-term downtime capabilities
        - Forecast and anomaly based alerting
        - A testing environment for monitoring and alerting
    - Alert aggregation & delivery
        - An alert aggregation layer to apply “business logic” (e.g. team routing, scheduling, prioritization) to alerts generated by our various tooling. (e.g. understands timezones to avoid paging someone sleeping)
        - A mechanism to avoid delivering alerts to the entire SRE team by default, especially for minor or subteam-specific alerts.
        - Automatic creation of tasks (either not alerts, or auto-ack) for recurring, well understood issues
        - Push and other non-sms notification delivery
        - A mechanism to quickly acknowledge and escalate alerts from both computers and mobile devices

- Top-level monitoring dashboard (e.g. status page, or dashboard of dashboards)
- Self service to update contact method
- Easy way to set time off (PTO)

Scope

This document is focused on alerting infrastructure. As such the policy items above are considered out of scope. With that said, infrastructure and tooling need some amount of policy and direction in order to be configured and deployed. Our focus here is on building flexible tooling to address the technical shortcomings outlined above. This tooling should be able to adapt to policy changes as they are established and become more well defined. In terms of policies for incident response, see recommendations for incident response by ONFIRE.

Components & Concepts

This section provides an overview of the components and concepts involved in alerting, with a description of each component's role and its relation to other components. Some of these components are already deployed in production, sometimes under a different name but similar functionality and implemented as a single software/system. The intention of calling out each component separately is to make it easier to reason about them, and discuss their functions and interfaces.

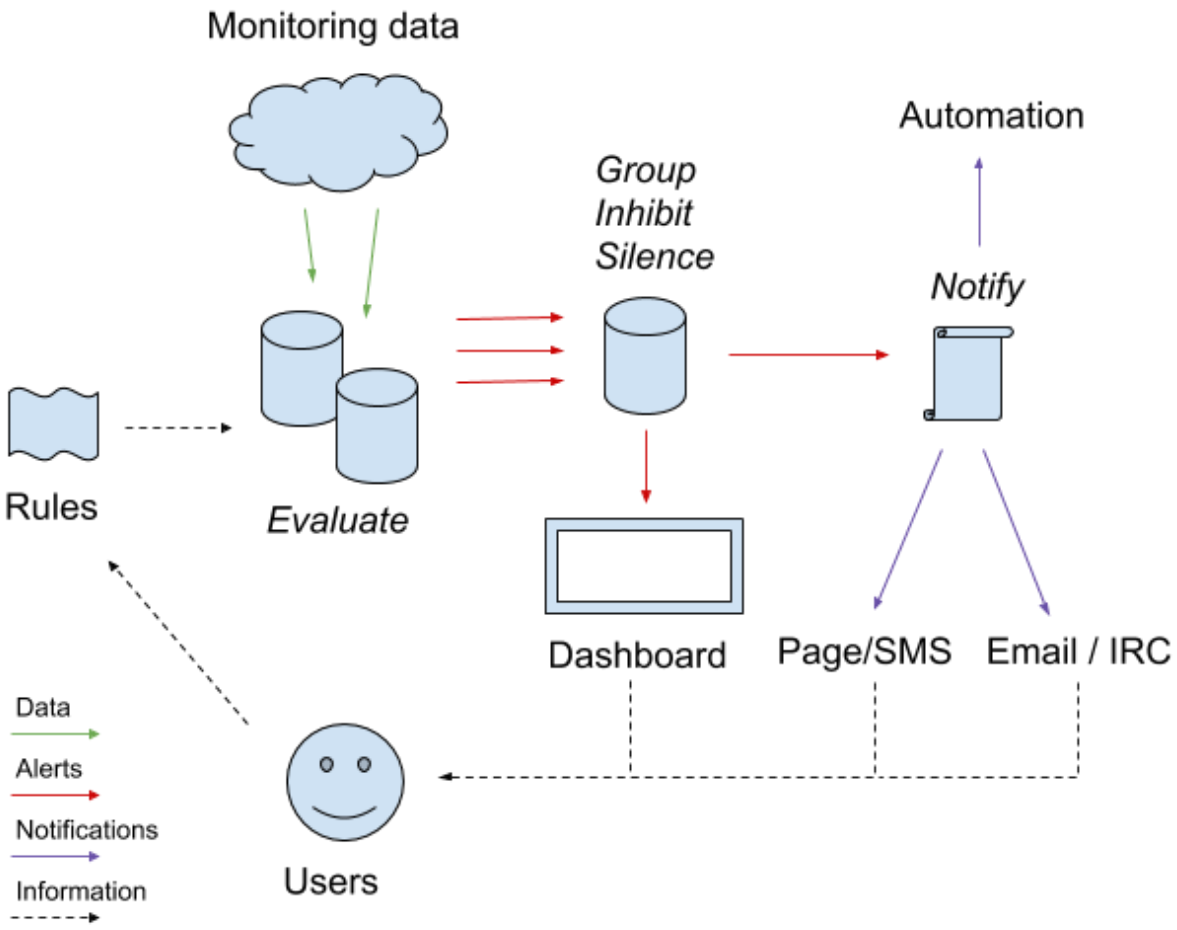
- **Rule** conditions that when true will generate an alert, the specific logic is dependent on the monitoring system used. Typically a rule also specifies how long to wait before considering sending out notifications. There can be metadata attached to the rule to help with understanding more about its context (e.g. dashboards, runbooks, etc) and scope (what is the rule affecting: host, site, etc).
- **Alert** whenever a rule's conditions become true an alert is generated, said alert will then notify users, invoke automatic remediation, open tasks and so on.
- **Page** indicates a rule serious enough that a human must be notified and act with urgency on the alerts the rule generates.
- **Severity** describes the level of urgency for an alert. (e.g. warning, critical)

- **Notification** the delivery mechanism to notify a human about an alert, could be more than one for a single alert (e.g. IRC and SMS)
- **Users** define alerting rules, add silences when needed
- **Contacts** a notification recipient's (usually an SRE or administrative user) preferred method of contact for notifications (for example, a cell number for SMS, IRC channel, push notification, etc.)
- **Contact groups** Collection of contacts, used for notifications
- **Evaluation** checks rules against current state, emit alerts when rules match.
- **Alert deduplication, inhibition, silencing** receives alerts, possibly from multiple sources, and applies deduplication (grouping) based on defined criteria. (Groups of) alerts can be inhibited when other related alerts are firing (e.g. cluster-level alerts inhibit host-level alerts). Users can add silences to stop a particular set of alerts to notify. Alerts that are not inhibited and not silenced are then sent out for notification(s).
- **Escalation** A structure comprised of contact(s), severities and schedule(s) that defines what alerts should be sent to what group, when, and what happens if an alert is not acknowledged.
- **Schedule** A time and day (MTWTFSS) based schedule that describes how alerts should be routed for a given service and severity.
- **Dashboard** provides an interface to see alerts at a glance, with filtering capabilities and possibility to act on alerts (e.g. silence). Information gathered from the dashboard can be also used for wider audiences/distribution, for example high impact alerts displayed on a status page.

Overview

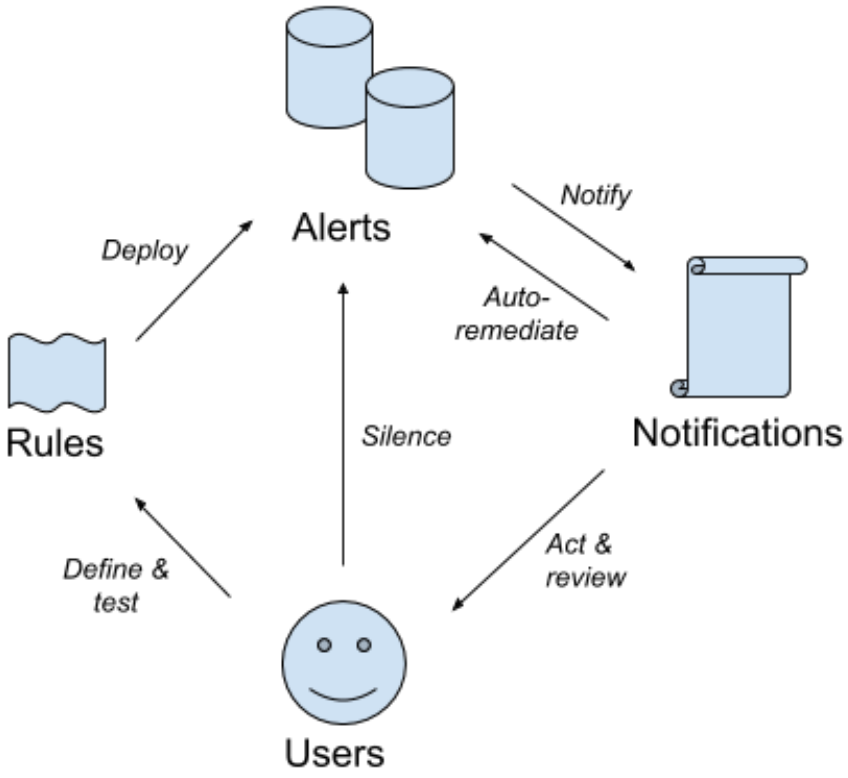
The diagram below provides a representation of the components and data flow involved in alerting. Some of these components are already deployed in production, sometimes under a different name but similar functionality and implemented as a single software/system.

One of the fundamental functions that alerting must be able to perform is turning *data* (a lot of it, generated by monitoring) into *information* (i.e. few alerts: actionable, relevant, etc).



Lifecycle

This section gives a description of the lifecycle of an alert, and what interactions happen while said alert is deployed.



Users of the system define their own rules, each rule contains a set of criteria plus metadata like links to relevant dashboards, owner team, etc. Rules should be also tested against dummy data when possible, to make sure the expected behaviour is achieved; in other words help answer questions like “why did this alert fire?”.

Once a rule is deployed it gets periodically evaluated against monitoring data and if found true for a certain (rule-specific) amount of time then an alert will be generated and sent out as a notification for further processing.

Users receive notifications from alerts (after deduplication and grouping) and can decide to act on the problem and/or on the alert itself, namely to silence the alert. Similarly automation can

receive notifications from alerts and act on them, for example to auto-remediate the problem, open/update tasks, remove silences for resolved tasks, etc.

Alerts are reviewed periodically to assess their importance and relevance, when an alert isn't deemed useful anymore its rule is removed from deployment.

Architecture & Components

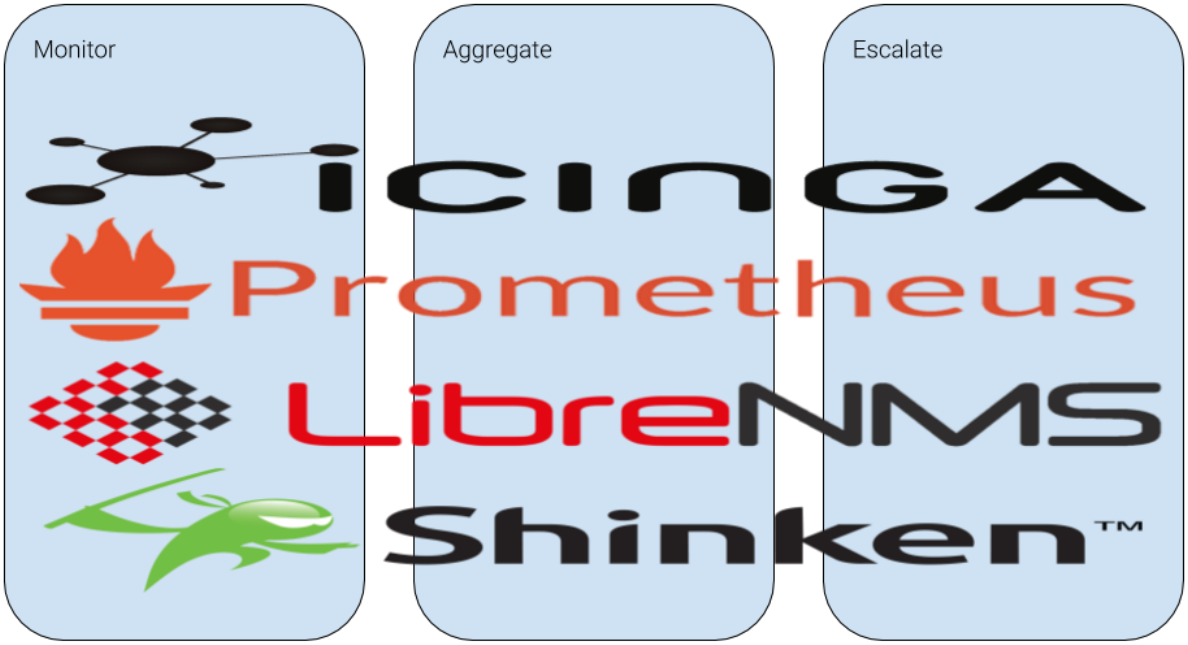
Our proposed future state combines a new set of architecture guidelines, which intends to organize supporting tooling into categories. These categories are monitoring, aggregation, and escalation.

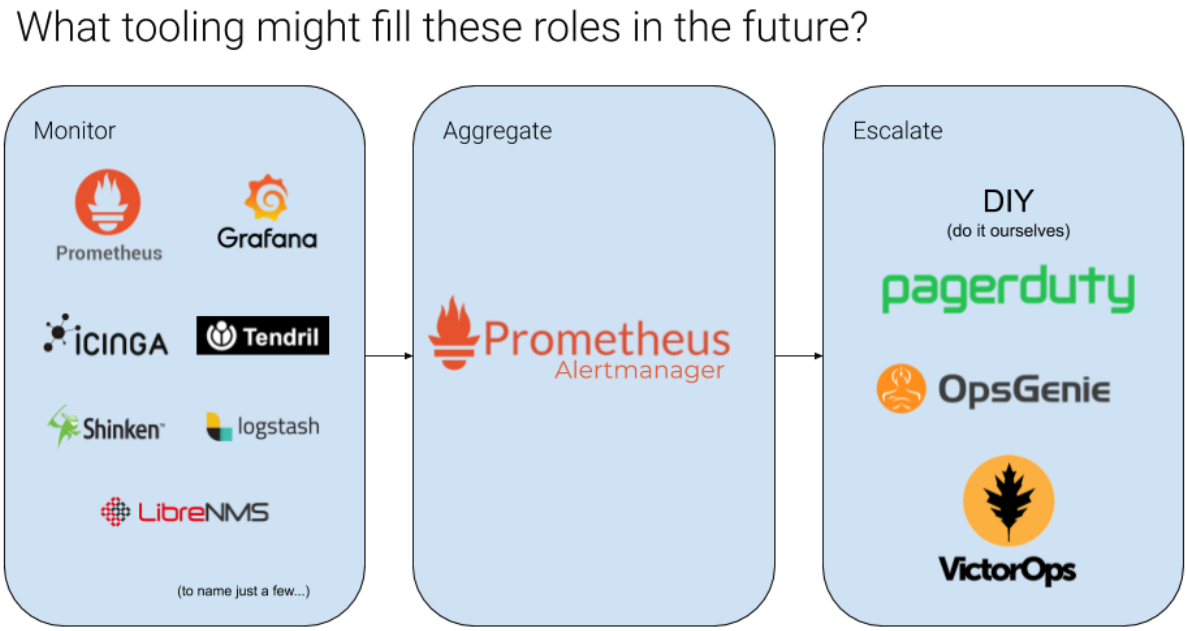
- **Monitor** - Tool(s) which monitor resources, maintain health/status information, evaluate rules and produce alerts.
- **Aggregate** - Tool(s) which ingests alerts from one or more monitors, deduplicates, categorizes, assigns severity, and generates actionable alerts for human consumption.
- **Escalate** - Tool(s) which transforms alerts into notifications (e.g. IRC messages, Push notifications, SMS, Phone calls, Phabricator tasks, Log messages, etc.). Notifications are routed based on numerous attributes (service ownership, severity and schedule) to service owners via a configurable mechanism.

In order to move forward with this structure, we must deploy new tooling to support this model, rationalize the functionality of our current monitoring systems, and move these functions to the appropriate tool(s).

First and foremost will be Icinga. In the future state proposed above Icinga will reside in the "Monitor" category. However, today Icinga stretches across all three. To move forward we must deprecate, move, and remove, the aggregation and escalation aspects of Icinga to tooling built for these purposes. Such categorization will help in the future when choosing to add new tools, especially in the monitor category such as generating alerts from logs (e.g. Elastalert) or from any kind of stream (e.g. with Modern Event Platform / Spark).

Where do our tools fit in this model today?





Implementation

The goal of this section is to present an implementation plan towards an alerting infrastructure that will fit our **requirements**, including:

- **Multi-tenancy** teams other than SRE need to manage their alerting in autonomy
- **Autonomy** said autonomy must not impact alerting for other teams
- **Reliability** same or better reliability than the current infrastructure
- **Simplicity** alerting should be as simple as possible to manage and understand, and more flexible and future proof than what's currently implemented
- **Scalability** the implementation must be able to handle at least one order of magnitude more than current scale (e.g. number of hosts, services, alerts, etc)

The plan is split into phases with increasingly bigger areas of concern; the idea is to start small, gain confidence in the new systems, and iron out the kinks as they come up.

Phase 1 - Introduce Prometheus Alertmanager

The first phase involves introducing components alongside the existing production infrastructure, specifically Alertmanager will receive all Icinga outstanding alerts in the form

of Prometheus metrics and will provide basic deduplication/inhibition and notify all resulting alerts to IRC. The alerts dashboard will also be deployed to see alerts at a glance but no interaction (i.e. creating silences) will be allowed, making the whole infrastructure effectively pass through with respect to the alerts coming from Icinga. Alertmanager has been chosen as the aggregation component because it fits all requirements and all of our major monitoring software has support for it (Prometheus, Grafana, Librenms), integration of new clients sending alerts is simple, and generic notifications integrations are possible through webhooks.

Notification wise, IRC will be the only channel implemented in this phase, with email and paging coming later. The goal of this phase is to validate the basic functionality of the infrastructure and its components without affecting existing workflows.

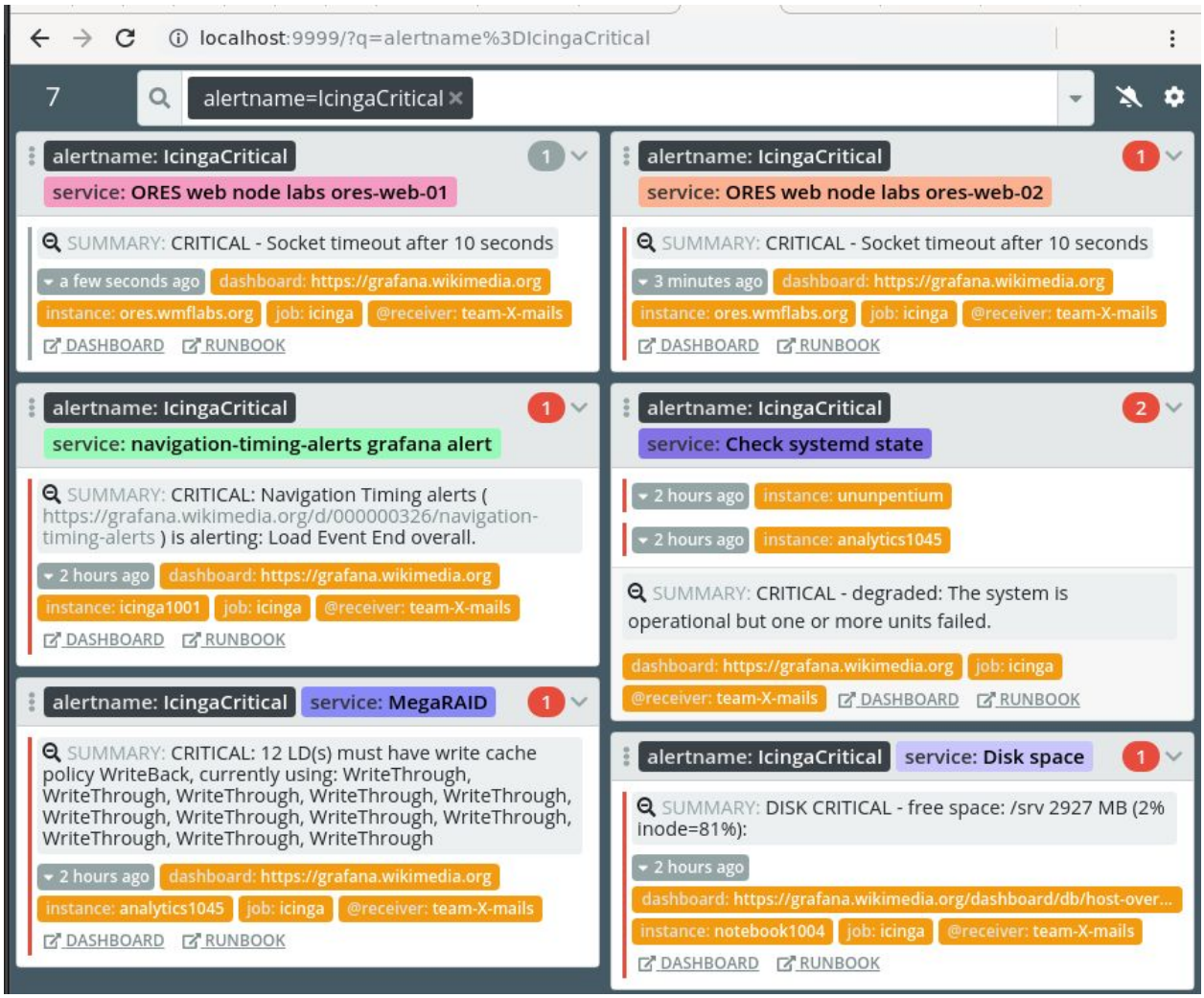
Having a fully functional parallel infrastructure also opens the door to experiments such as replacing WMCS Shinken with Prometheus alerts, e.g. to replace beta/deployment-prep alerting.

Software

- Prometheus Alertmanager https://github.com/prometheus/alertmanager
- IRC bot to receive AM notifications via webhooks, https://github.com/google/alertmanager-irc-relay
- Alerts dashboard for Alertmanager, https://github.com/prymitive/karma
- icinga_exporter extended to export alert status as Prometheus metrics

Deliverables

- Alertmanager deployed in HA/clustered mode in two sites
- Alerts dashboard deployed and available behind HTTP authentication
- IRC bot deployed alongside Alertmanager and sending notifications to a test channel
- All Icinga outstanding alerts show up in alerts dashboard
- Grouping and inhibition of alerts is working as expected



Sample alerts dashboard with Icinga criticals deduplicated and grouped

Phase 2 - Initial Onboarding of Email-Only alerts to Alertmanager

During phase two we'll be enabling email notifications and onboard more alerts producers, namely Grafana and Librenms, both of which support sending alerts to Alertmanager. Silencing alerts for Grafana and Librenms through the alerts dashboard will also be supported. Onboarding Grafana will be particularly effective since it paves the road to alert self-management from teams and we can stop funneling Grafana alerts into Icinga. With the

experience and insights gained during this phase we'll be also producing guidelines and documentation for service owners on how to deploy and design alerts[1].

Deliverables

- Grafana sends its alerts to Alertmanager, resulting in email notifications routed to the appropriate teams
- LibreNMS sends its alerts to Alertmanager, resulting in email notifications to network operations
- Silencing alerts through the alerts dashboard is supported and functional
- Alert design guidelines for teams are produced

Phase 3 - Alert Scheduling and Paging

Scheduled paging will be implemented during this phase. The exact implementation is TBD but at least two ways seem possible: external service like Pagerduty to receive pages from Alertmanager and schedule/escalate accordingly or keep the email interface to pages and integrate schedules during mail transport (e.g. exim aliases/transports).

As a baseline this phase intends to deliver:

- Abstractions to generate notifications including SMS, phone, IRC, Phabricator tasks
- Per-recipient notification schedule
- Addition of paging alerts to Alertmanager

By introducing a 3rd party service like Pagerduty we can include the below as in-scope for this phase as well.

- Notification routing by customizable per-team/per-service schedules
- Provide a mechanism for issue acknowledgement, communication and escalation to teammates from smartphone app, SMS, phone, and web interface.
- Per-user contact method preferences (push, SMS, call, etc)
- Track open alert status and escalate to another notification method (e.g. if unresolved for a long period of time)

[1] The document My Philosophy on Alerting for example contains a lot of good starting points

Phase 4 - Initial onboarding of non-SRE Alertmanager Tenants

In this phase we'll be onboarding teams with Icinga alerts to use metrics-based Prometheus alerting rules and Alertmanager.

Identifying teams responsible for existing alerts will require an audit of alerts in Icinga and assigning ownership. This will be executed during the beginning of this phase, and will help identify which teams can be readily onboarded. For example, teams with services living exclusively in Kubernetes already have their service metrics available in Prometheus and thus can move from Icinga checks to Prometheus alerting rules from the get go.

Lastly, a system for non-SRE users to deploy and test rules will be devised and implemented, e.g. a separate git repository with alerting rules that auto-deploys to production Prometheus and has wider write access than Puppet for multi tenancy purposes.

Deliverables

- All Icinga alerts have an owner associated
- At least one team with alerts in Icinga has been fully onboarded onto Prometheus alerting rules
- Prometheus alerting rules can be changed by teams and deployed to production in autonomy
- Prometheus alerting rules are unit tested before being deployed to production

Phase 5 - Reduce Icinga' scope

This phase will see the number of checks performed by Icinga reduced even more. The idea being to have Icinga as a purely monitoring tool: running checks and forwarding alerts to Alertmanager, with no UI being used and no contact features. At the same time we'll be adding features and tighter integrations between the wider infrastructure and alerting, e.g. reacting to alerts by opening tasks.

Deliverables

- All human contacts removed from icinga
- Icinga alerts paired down to a minimal set

Cross cutting concerns

This section outlines considerations common to all services SRE operates, as such are not included in the main design.

Backups

Backing up data is not being directly considered in this document, instead we intend to leverage our existing infrastructure-as-code tooling, primarily Puppet, to enable re-deployment of alerting infrastructure when necessary.

Scalability

We should be able to seamlessly add capacity at each layer with no downtime, and without requiring major config changes.

We should document estimated upper boundaries at each layer and set up mechanisms to provide actionable warnings with adequate advance notice.

High Availability

To ensure reliability, and maximize flexibility in maintenance work, all alerting infrastructure should be deployed with redundancy and HA in mind.

Monitoring since the aggregation layer is capable of deduplicating alerts, monitoring tools should be deployed in an active/active N+1 configuration with multiple instances actively sending alerts to the aggregation layer for processing.

Aggregation the aggregation layer will be deployed in an active/active configuration, multiple Alertmanager instances will be deployed and Prometheus will send alerts to all of them. Notifications (below) are handled by Alertmanager to avoid multiple notifications for the same problem

Notification the notification/escalation layer achieves HA by providing different contact mechanisms per-person, and notifying multiple people via multiple mechanisms (either via schedules and escalations, or by simple distribution list).

Monitoring the monitors We must ensure we have adequate "nothing shared" external baseline monitoring in place to notify us should we experience a significant monitoring service impacting outage. Such a system must be simple and reliable, we can adapt and extend the Icinga meta-monitoring we have in place today.

Alert Logging we should make reasonable efforts to ensure our alerts are logged in ELK, possibly in a long-term storage index. This will be useful for looking back at past alerts and better understanding overall alert volumes and trends.

Deprecation and Simplification

It is clear that a successful alerting strategy will require a suite of tooling. This section outlines some scenarios for deprecation and/or simplification of existing components used in our alerting infrastructure. There is an important distinction between the two.

Deprecation a piece of tooling has been superseded entirely by something new, and as such is marked for removal. Nothing new should be added to the deprecated tool, and efforts are actively under way to remove it entirely.

Simplification one or more pieces of functionality of a given tool a tool have been superseded by something new, but a subset of features will remain. Therefore the configuration of the tool will be simplified and optimized for its strong suits. E.g. Decoupling notification and scheduling functionality from Icinga (in favor of a new tool for this purpose) while some core check execution remains.

Monitoring with notifications offloaded to the escalation tier, monitoring tool configuration may be simplified, and the process of deprecating one tool in favor of another is more straightforward as there are less pieces of functionality to migrate. E.g. after offloading notifications and scheduling, an upgrade of Icinga should be less daunting.

Closing Thoughts

As a whole, this alerting infrastructure roadmap aims to simplify our individual monitoring tools, while at the same time improving our capabilities, by focusing on their strengths. This provides several benefits:

Speed by providing re-usable mechanisms to aggregate and notify, monitoring tools only need to handle the core task of evaluating and reporting service health.

Simplicity by clarifying what a monitoring tool should and should not do, our configurations become much simpler, and we can avoid areas of overlap (e.g. multiple tools implementing contact lists, schedules, irc bots, etc.)

Extensibility narrowing the scope of a monitoring tool lowers the bar of entry and maintenance required, allowing us to extend our current offerings and introduce new tools without adding significant complexity or duplicating efforts.

Redundancy by embracing a suite of tools we inherit a level of redundancy not possible with a single system.