

Based upon agentic AI, Kraken is designed to answer queries from alerts or operators by applying the most suitable set of actions depending on the supercomputer state.

Why HPC systems need AI?

AI can help monitoring and managing HPC clusters which:

- increase in size
- diversify in terms of technology
- operate ever-longer and ever-bigger simulation codes

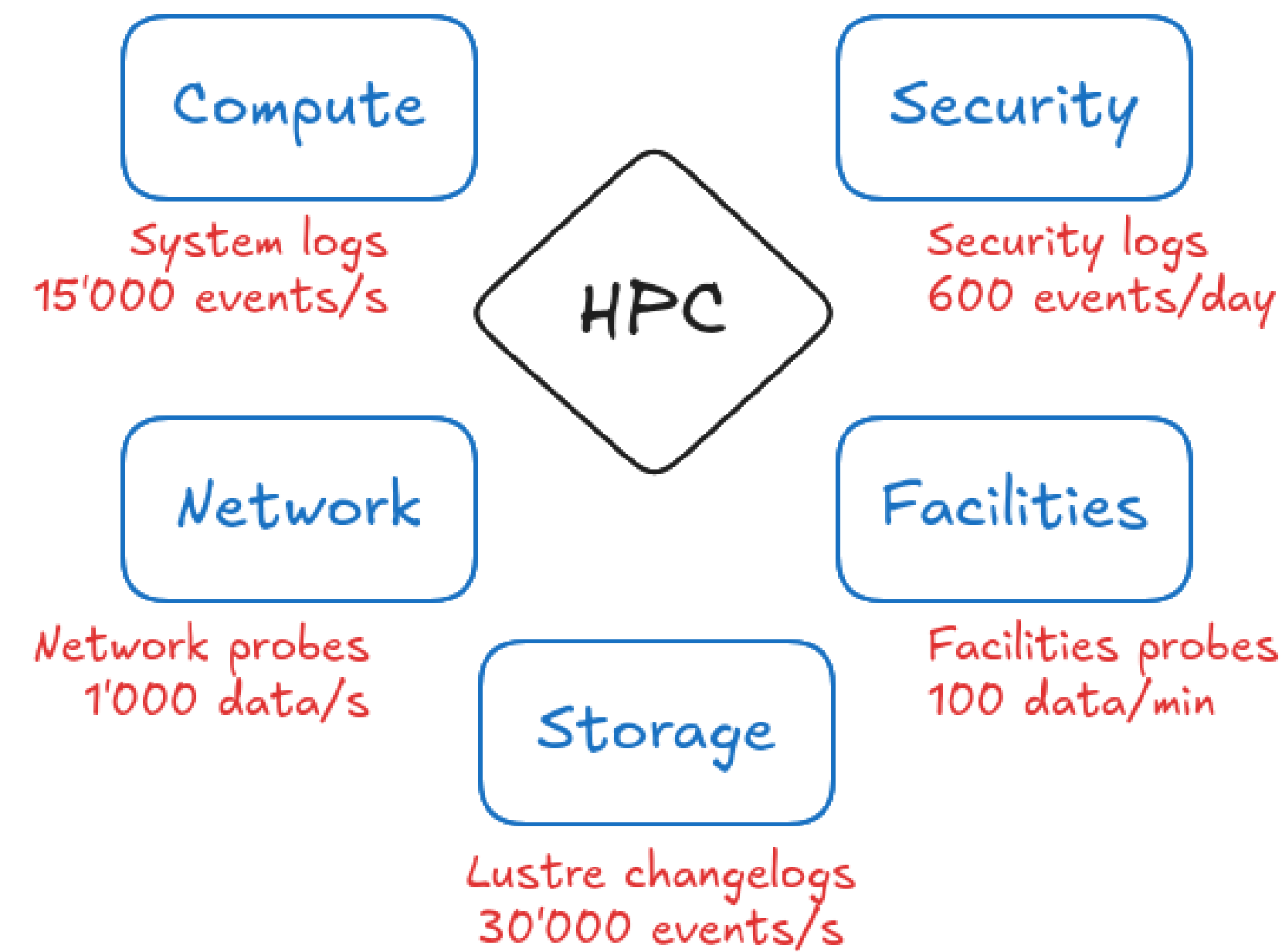


Fig 1. Order of magnitude of data generated by HPC cluster usage

This data is just a subset of what is actually captured. Other information, such as disk usage for storage, and security alerts, is also monitored and generates data.

This work involves **processing Lustre changelogs** when the system is overused.

How to help Lustre?

- Lustre is a distributed, parallel file system used in HPC storage systems that provides good performance and reliability.
- Each user action on the file system generates an event. These events are stored and monitored.
- If a certain number of events are observed, an alert is triggered. We need to know whether this is due to misuse or a combination of circumstances.

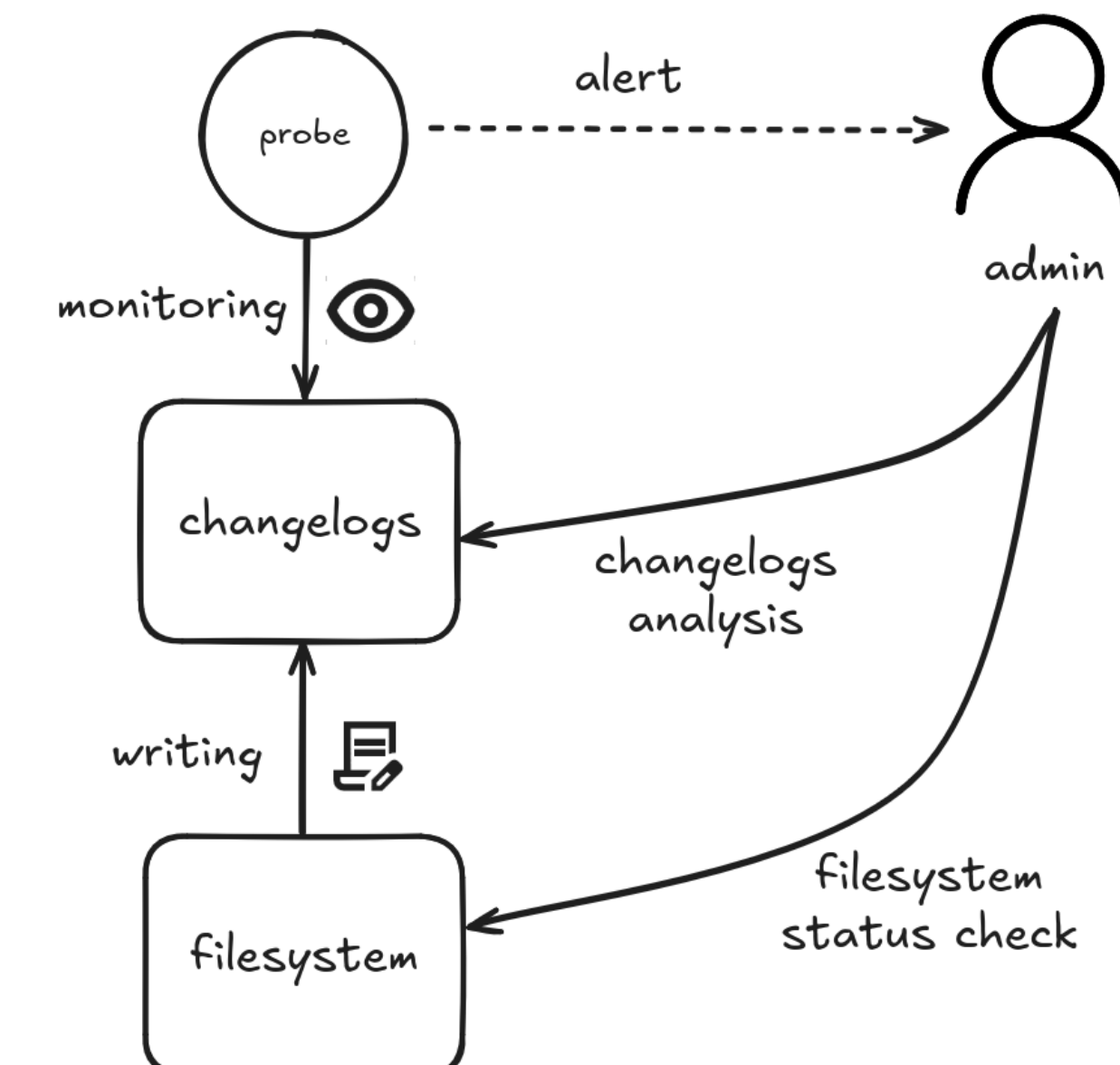


Fig 2. Lustre changelogs analysis by administrators

Currently, an alert is only processed by an operator, who needs to retrieve all available data. Then, they seek the root cause of the alert and estimate the necessary action to resolve it.

Kraken workflow

The LLM used to build the Kraken agent is **Devstral Small** from Mistral AI, an agentic LLM designed for software engineering tasks. Specialization is achieved using an **in-context few-shot learning** approach with tool-use demonstrations, structured as five problem-solutions pairs.

The report includes:

- a breakdown of the reasoning steps leading to the agent's conclusion
- a description of all actions performed

Sent through tool-calling

- `send_admin_report(report)`

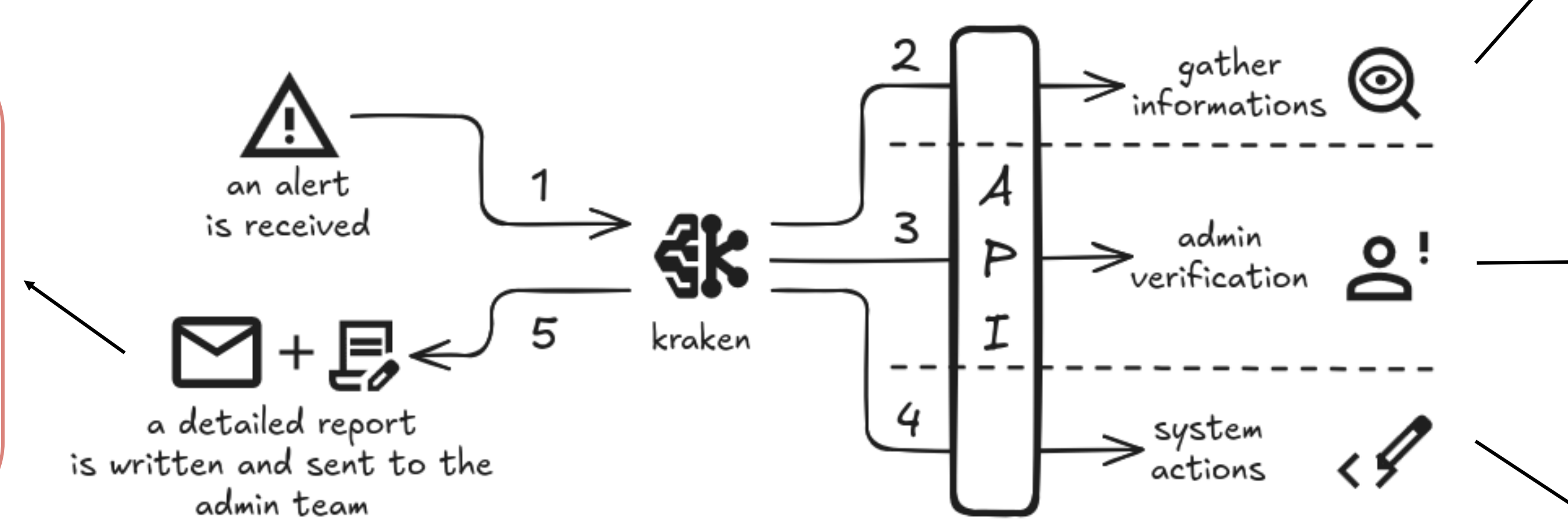


Fig 3. Kraken agent workflow

This workflow represents a typical treatment of an alert received from probes. Each external interaction corresponds to a query sent through the **tool-calling** mechanism.

Tool-calling to **extract informations** from the changelogs database:

- `get_last_alerts(seconds)`
- `get_jobs_count_by_type(event_type)`
- `get_users_by_jobs(jobs_id)`

➤ enables **informed reasoning** to trace the alert's origin

Returns control and pauses until administrator approval is received before performing any system action.

Tool-calling to **realise actions** on the system:

- `ban_jobs_by_id(jobs_id)`
- `send_mail_to_user(user_id)`

➤ carries out validated actions on the system

Is it efficient enough?

Experimentation is done in a simulated environment where:

- jobs consist of a set of I/O accesses converted into Lustre changelogs
- changelogs are read and processed in real time
- 3 different scenarii are considered: bad job, bad user, and shared overusage

The inference time of the agent is nearly 45 seconds with 2 NVIDIA A100 GPUs using the **Devstral Small** model from Mistral AI.

- **Fine-tuning:** improved efficiency and the ability to manage more various scenarii
- **API improvement:** support for additional metrics, such as disk usage
- **Kraken project:** the changelog agent is the first agent of Kraken, which aims to manage the entire supercomputer

Fig 4. Screen capture of Kraken agent solving an issue

The agent interacts both with the incoming alert (treated as a user request) and with the tools it invokes.

```

The function returns a list containing tuples in the format (timestamp, alert). Here, MAX_OICREAT means that the alert
received is related to too many OICREATs. This alert is important because too many occurrences can lead to a significant
increase in the number of inodes, which can be dangerous for the file system. Now I need to count the frequency of e
ach operation to find out the exact number of OICREATs. To do this, I use the 'get_types_count' tool, which takes a du
ration in seconds as an argument. Let's look at the number of each operation for the last 10 seconds.

'''python
get_types_count(("seconds": 10))
'''

{
  "tool": "tool",
  "name": "get_types_count",
  "context": {"OICREAT": 134, "R2MWRITE": 14, "STRWRITE": 13, "IICLOSE": 14, "I3TRUNC": 3},
  "tool_call_id": "951e933c040b40f0b2fe417095e0f772"
}

I will now analyse the results of the function. I have a dictionary with each operation as a key and its frequency of
occurrence in the last 10 seconds as a value. Here, there are several operations present, but we are only interested i
n OICREAT. Let's now see if it is possible to target one or more jobs responsible for these OICREAT operations. To do
this, I use the 'get_jobs_count_by_type' tool to retrieve the distribution of jobs responsible for the OICREAT operati
on. This tool takes as arguments a duration in seconds to retrieve the distribution over that period and the metadata
operation we want to target. Here, I will look at the last 10 seconds and only for the OICREAT operation.

'''python
get_jobs_count_by_type(("seconds": 10, "event_type": "OICREAT"))
'''

{
  "tool": "tool",
  "name": "get_jobs_count_by_type",
  "context": {"1100000001": 4, "1100000007": 7, "1100000008": 124, "1100000008": 1},
  "tool_call_id": "a014ec37b9a94740b4a40f0b4c952801"
}

The function returns a dictionary with the job_ids responsible for the operation passed as an argument as keys and the
amount of metadata generated by that job as values. Here, we have four keys, i.e. four jobs that are responsible for
OICREAT operations. Now let's retrieve the threshold for the OICREAT operation to see if all jobs exceed this thresho
ld, if only some exceed it, or if none exceed it. To do this, I use the 'get_config' tool, which takes as an argument t
he name of the variable for which we want to know the threshold. Here, I pass the name of the alert received as the va
riable name: MAX_OICREAT.

'''python
get_config(("threshold_name": "MAX_OICREAT"))
'''

{
  "tool": "tool",
  "name": "get_config",
  "context": "50",
  "tool_call_id": "04b0c32e03f145db0f703baaf3e0ad9"
}

```