Supporting failure analysis with discoverable, annotated log datasets

Steve Leak*, Annette Greiner*, Jim Brandt†, and Ann Gentile†

*National Energy Research Scientific Computing Center (NERSC), Berkeley, CA USA 94720
Email: (sleak|amgreiner)@lbl.gov
†Sandia National Laboratories (SNL), Albuquerque, NM USA 87123
Email: (brandt|gentile)@sandia.gov

Abstract—Detection, characterization, and mitigation of faults on supercomputers is complicated by the large variety of interacting subsystems. Failures often manifest as vague observations like “my job failed” and may result from faults in system hardware/firmware/software, filesystems, networks, resource manager state, and more. Data such as system logs, environmental metrics, job history, cluster state snapshots, published outage notices and user reports are routinely collected. These data are typically stored in different locations and formats for specific use by targeted consumers. Combining data sources for analysis generally requires a consumer-dependent custom approach. We present a vocabulary for describing data, including format and access details, an annotation schema for attaching observations to a dataset, and tools to aid in discovery and publication of system-related insights. We present case studies in which our analysis tools utilize information from disparate data sources to investigate failures and performance issues from user and administrator perspectives.

I. INTRODUCTION

As systems have become larger and more complex, the volume and variety of log data has become formidable. Also increasing is the need to find insights in that data to troubleshoot system problems and identify root causes and propagation paths of faults, in support of improving system resilience.

To aid efforts to extract meaningful information from log data, the Holistic Measurement Driven Resilience (HMDR) [1] project has as a goal the publication of annotated datasets for resilience research. Making log datasets available to researchers is itself a significant challenge: the data is large, and diverse in terms of format, storage and access. Data beyond system logs is also valuable: contextual information provided by batch system history, maintenance logs and user error reports.

In this paper we address this challenge by first exploring the context of the problem to understand why it is difficult. From this we infer the requirements of a solution, outlined in section III. Our solution is comprised of a machine-readable vocabulary for describing relevant data, a schema for collating and using expert and machine-generated annotations about log data, and some tools to make these accessible to researchers. We describe this in detail in section IV. We describe methods used to populate a sample annotation database in section V, and use it for some illustrative case studies described in section VI.

II. CONTEXT

To illustrate the opportunities and challenges posed by the assortment of log-and-related data, consider the NERSC and ACES (LANL/SNL) large-scale computational environments, represented by the authors:

NERSC’s Cori system is a Cray XC40 with 2388 Xeon nodes, 9688 Xeon Phi nodes, a host of service nodes providing I/O forwarding, a Datawarp burst-buffer filesystem, Lustre networking and system management, and an Aries high-speed network in a Dragonfly topology. Cori has a large external Lustre filesystem also cross-mounted on Edison - another large Cray system at NERSC - via Infiniband. It has external login nodes and shares GPFS filesystems with other NERSC servers. There are air and water cooling components and UPS power circuits. Along with Cray system software and programming environments, Cori runs multiple compilers and MPI stacks and hundreds of software packages. Its 7,000 users run tens of thousands of jobs per day via the Slurm batch scheduler.

ACES Trinity is a Cray XC40 with 9436 Xeon and over 9500 Xeon Phi nodes, a DataWarp burst buffer, an Aries HSN, and a Lustre filesystem. Five months of system log data, not including job log data, from only the Xeon Phi section of Trinity when it operated in a stand-alone mode, contained 4.5 billion lines. Trinity serves the needs of the National Nuclear Security Administration and has far stricter access limitations than does Cori.

The associated centers support multiple production and test-bed systems including bleeding-edge technologies and with many staff members who support various aspects of each facility, the several hundred projects and the several thousand users they collectively support.

Data such as system logs, environmental metrics, job history, filesystem state, outage notices and user reports are already routinely collected, but the extraction of useful insights from these requires a customized solution for each investigation and is limited by several constraints:

• Data is collected by different people in different security domains - for example, system logs for each subsystem are typically available only to the system administrators for that subsystem.
• Data is collected in different formats, usually based on the output format of specific data sources and the needs of a specific investigation.
• Data is stored in different places with different access mechanisms - flat text files, binary formats including HDF5, SQL and NoSQL databases, JSON via a RESTful interface, etc.
• Not all data is suitable for publication. Log files intersperse entries relating to events contributing to some failure with unrelated entries that may contain sensitive information such as user login details. The difficulty of log anonymization and risk of mistakenly releasing sensitive information discourages data owners from exposing data to the wider community.
• The volume of data is very large, requiring an analyst to comb through many unrelated log entries to identify the significant entries.
• Much of the collected data requires domain expertise to interpret, and this expertise is distributed across many people. For example: at the 2016 Cray User’s Group Monitoring BoF [2], the community, comprised largely of people with substantial Cray experience, concluded that it would be valuable to have
In the next section we explore the requirements for a solution to limited our ability to extract useful insights from the data we collect. Expert commentary on the meaning of log messages has been identified as a desirable resource but expertise is domain-specific and distributed across many individuals, all of whom have other, primary responsibilities. The solution must therefore allow for domain experts to independently contribute advice in the format they already use (requirement 5) - for example a spreadsheet of log message definitions or a ticketing system with maintenance requests and notes.

Failure analysis research often looks for relationships between and sequences in log records, and might include comparisons against “control” logs from a different period or system. Operational troubleshooting seeks to identify possibly-contributing events in the lead up to an identified failure.

Failures at one component may be triggered by events at a connected component so an ideal search would extend to logs from related components. These relationships may not be a simple hierarchy - for example the failure of a link might affect two “peer” nodes. But an overly-wide search will return an intractable volume of data, so the solution should support some means of filtering data as well as of discovering non-obviously-related data (requirement 6).

In summary, the solution should:

1) Be agnostic towards the format and storage of data.
2) Not require a priori knowledge of existing collections.
3) Be decentralized.
4) Allow publishing of data to be low effort and low risk.
5) Allow domain experts to independently contribute advice in a format convenient for them.
6) Support data discovery and filtering across related components.

In our solution, metadata provides that level of indirection. Our solution integrates three contributions to address these requirements, with the key idea being the decoupling of publication from data access:

1) An RDF vocabulary for describing log and monitoring data collections in terms of the subject being monitored, the time period covered, the type and format of the data and details for accessing the data or contacting its curator. The vocabulary allows construction of a distributed graph that can be queried to discover relevant collections of log-like data and annotations on that data.

2) A schema for publishing annotations for log data, which can be represented in an SQL database (or otherwise used to define the query and return fields of an API of a more complex representation) also in terms of subject and time period. Annotations databases fit neatly into the RDF graph and might contain:
   - Expert commentary, such as “this log message means that down links caused the network to be quesiced while recalculating routing tables”
   - Human observations, such as “during this period our engineer was replacing some failed nodes, so the network was probably disrupted”
   - Machine-generated observations, such as message-pattern frequencies computed by running certain logs through an analysis tool such as Baler.

3) A collection of tools to make the RDF vocabulary and annotation schema accessible to users, system administrators and
A. RDF Vocabulary

A mechanism for decentralized description and discovery of data was proposed over two decades ago [5] and exists now in the form of Linked Data tools and technologies. The specification of RDF [6] as a data interchange format for the World Wide Web is particularly relevant to our identified requirements. Decoupling publication from access meets many of these requirements (1, 4 and 5) and using RDF to describe log data collections provides decentralization and discovery without requiring a priori knowledge of other collection efforts.

In RDF things (concepts or concrete items) are represented as URIs and arranged in triples of a subject, a predicate and an object.

For example, we wish to state that NERSC is an Organization. We have a subject (NERSC), a predicate (“is an”) and an object (Organization). In a manner of pulling oneself up by one’s bootstraps, the W3C [8] publishes some standard vocabularies in the form of URIs that have a well-defined and documented meaning, including that <http://www.w3.org/2000/01/rdf-schema#type> refers to the predicate “is a”. Another vocabulary, known as the Friend-of-a-friend vocabularies, associates <http://xmlns.com/foaf/0.1/Organization> with the concept of an organization. In this spirit we write in Figure 1 by associating a URI we’ve chosen: <http://portal.nersc.gov/project/mpcc/sleak/nersc\#nersc> with the organization we know as “NERSC”.

A single triple tells us very little, but a collection of many triples forms a graph representing almost arbitrary knowledge graph. We get decentralization by the use of URIs as graph elements - any contributor can publish a set of triples and, so long as somebody is aware of it, it can be incorporated into a global graph.

The other Linked Data element key to our requirements is the SPARQL graph query language. A SPARQL query arranges variables into a set of triples and returns nodes for which the triples form a true statement. For example, the following SPARQL query will return the name and interest for each node whose type is a subclass of true statement. For example, the following SPARQL query will return

```
SELECT ?name ?interest
WHERE {

  ?type rdfs:subClassOf foaf:Agent .
  ?uri foaf:name ?name .
  ?uri foaf:interest ?interest .
}
```

Fig. 2. Example of a SPARQL query

Figure 3 illustrates how this query might act on a graph: the first statement locates nodes - colored blue - from which one can traverse rdfs:subClassOf edges and reach a foaf:Agent. The second statement locates nodes (shown in red) in triples with an rdfs:type predicate whose object is one found by the first statement. Thus far we have found Jim, Ann, Annette and Steve. Next we look for triples whose subject is one of those nodes and whose predicate is foaf:name, reducing the set to Jim, Ann and Steve, then again for predicate foaf:interest. Now only Steve matches all of the criteria. Finally, we return the nodes associated with the name and interest variables, which in this case are the nodes show in purple.

1) The vocabulary: The key classes and predicates forming our vocabulary are illustrated in Figure 4. Figure 5 provides examples of nodes in a graph corresponding to each class, and Figure 6 shows how RDF descriptions of different LogSets published in different places form a single, global graph. Figure 6 also shows how data dictionaries describing different SubjectTypes and LogSeries can be published and become part of the global graph when used.

The vocabulary is extended and specialized from the Data Catalog Vocabulary [9]. The meaning, reason and usage of some key classes and properties are:

- **Catalog**
  - The dcat:Catalog class connects LogSets and also, via rdfs:seeAlso, other catalogs. This is the primary mechanism for linking sites into a global graph: we anticipate that each site will publish a catalog to which its own staff can contribute LogSets, and which is linked to a few others via rdfs:seeAlso.

- **LogSet**
  - A collection of logs related in system and access and timespan, for example the logs collected in a p0- directory in the SMW of a Cray XC for a single boot session. The LogSet should provide a description of the data and contact information and is an entry point to metadata for the ConcreteLogs. Temporal and subject information for the LogSet can be inferred from those properties of its ConcreteLogs.
  - A LogSet might be a closed archive or might be “open”, acquiring new logs over time.

- **ConcreteLog**
  - A ConcreteLog describes a specific, concrete source of log entries. This will often be a log file but could also be, for example, a Slurm instance from which job data can be obtained.

  The accessURL and downloadURL have subtly different uses, inherited from dcat:Distribution. Where security or practical constraints preclude direct download of data, accessURL can be used instead to find more information (such as how to request access).

  The ConcreteLog should also include the start and (optional) end dates encompassed by the log. Inclusion of information about the size and number of records is also recommended, as a means of avoiding download of excessively large data volumes.

- **LogSeries**
  - Most log data can be classified into a few series, such as “console log files” or “Slurm job records”. ConcreteLogs within a LogSeries have the same structure but span different times or subjects. A LogSeries is often common to systems from a given vendor and is expected to be published in a common dictionary.

- **LogFormatType**
  - The LogFormatType gives hints to tools about how a particular LogSeries should be handled. For example, many logs are in the form of a timeStampedLogFile. Series-specific details such as how to identify the timestamp
@prefix nersc: <http://portal.nersc.gov/project/mpcc/sleak/nersc#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

nersc:nersc rdfs:type foaf:Organization .

Fig. 1. A triple of (subject, predicate, object) describes an edge in an RDF graph. The Turtle [7] syntax shown here aids human readability by condensing URIs into a prefix and a suffix, so for example rdfs:type expands as <http://www.w3.org/2000/01/rdf-schema#type>.

B. Annotation Schema

The primary goal of annotation is to provide a reduced set of searchable, understandable data that can hint at relationships between logged events and guide the user to further searches and to interesting locations in the raw log data. This differs from tools such as SEC [10], which is intended to enable action upon the run time occurrence of a log line matching a regex (e.g., notification of failed component), or Splunk, which is intended to facilitate knowledge of the occurrences of pre-defined events with accompanying statistical plots. Along with our identified requirements, this goal provides some design requirements for the schema:

- Temporal and subject (system and component) information are the key fields, along with a human readable description, the annotation itself.
- The raw logfiles that the annotation concerns should be identified, if possible.
- Multiple people should be able to annotate the same underlying log data, and users of the annotations should be able to identify annotators to request more information, if necessary.
- The architectural relationships between components indicated in annotations should be accessible, in order to facilitate traversal from an annotation of interest to annotations relating to components that may have impacted it.
- Annotation fields should support searching based on the subject type of an annotated event (such as a network event) or other related information.

1) The schema: Our annotation schema can be represented as an SQL database definition with a central “annotations” table:

CREATE TABLE 'annotations' (  
id integer,  
authorid char(3) NOT NULL,  
description text NOT NULL,  
-- timespan of the action or event:  
starttime datetime NOT NULL,  
endtime datetime NOT NULL,  
-- impact of the action or event:  
startstate text,  
endstate text,  
systemdown boolean,  
system text,  
components text,  
-- was the event manually induced?  
manual boolean,  
-- subject type and annotation context:  
LDcategory text,  
LDtag text,  
balerpatternid integer,  
-- event source:  
logfiles text,  
PRIMARY KEY('id','authorid')  
);

Fig. 8. The central annotations table definition.
Fig. 4. Key classes and predicates in the logset vocabulary.

Fig. 5. Examples of nodes in the graph and how they relate to vocabulary classes.

Fig. 6. Examples of some nodes and relationships published in different places from different sites (indicated via color), forming a single global graph.
The id, source, description and temporal fields are self-explanatory. The impact fields are designed to aid exploratory analysis, for example startstate and endstate can be used to indicate that a component that was in a faulty state was either repaired or flagged by the end of the event. A systemdown flag allows search for events resulting in a full system failure.

The system and components fields correspond to the Subject class of the RDF vocabulary. For the Cray system we define node, blade, chassis, cabinet, router, tile, link, nic, smw, and other, all of which (except other) are also SubjectTypes described in RDF in data dictionaries and can, in combination with the system, identify a specific component.

The components may be identified with finer granularity than is represented in the RDF graph - for example c0-1c0s4n0. The LDcategory includes node.blade, scheduler, storage, network, cooling/facilities/sensors, power, system software, datawarp, and unknown, which also correspond (except for unknown) to SubjectTypes in an RDF dictionary. The name of this field and its categories are inspired by LogDiver [11], a tool developed by UIUC which associates regular expressions defining events in log files of interest with categorizations including these. We envisage LogDiver as an option used to identify log entries for annotation.

The manual flag indicates that an event was manually induced, such as an administrator action to take down a node as opposed to the system taking down a node because it failed a health check. Knowledge of this can be used to more accurately determine the number of true failure events and for assessing the effectiveness and availability of system resilience mechanisms.

2) Other tables: The schema includes some supporting tables:

Meta Annotations
The metaannotations table supports annotation sets, wherein a collection of annotations relates to a single larger event

Component Aliases
Different components are identified differently in different log series, for example the same node on a Cray XC might be called nid00001 by the scheduling system, c0-0c0s0n1 by the boot system and even its IP address by another log source.

To support this we define an aliases table, mapping equivalent component names. This is generally populated via /etc/hosts, or on Cray systems, the output from rtr --system-map

Architectures
The topologies of current clusters are more complex than a simple hierarchy, so we support the definition of multiple architectures to describe relationships between components. We have identified three architectures for Cray systems, these are described below. We chose to separate architectures in order to enable different search and interpretation of events which affect components with different association with each other. Associations include parent-child: component-component as opposed to parent-child: router-tile, or peer: HSN link as opposed to peer: Router-NIC or peer: NIC-Proc.

3) Component hierarchies: We have identified three distinct architectures within a Cray cluster

1) The physical architecture consists of parent-child or container-contained associations, such as a cabinet - chassis. blade - router, router - link and router - nic. This table is populated from the system definition - for example on a Cray XC each cabinet holds three chassis, so an entry might look like:

   id  type  name  children
   1   CAB   c0-0  types
   2   RTR   c0-0cs0a0  c0-0cs0a0
   3   RTR   c0-0cs11a0  c0-0cs11a0

   The physical architecture supports discovery of faults propagating from child to parent components and vice versa.

2) The router architecture describes network topology from the perspective of the router - for example an Aries may be specified by its (blue:black:green) identifiers while a Gemini by its X.Y.Z. This supports determination of proximity of events in the network topology.

   id  type  name  children
   11  RTR   c0-0cs0a0  c0-0cs0a0
   12  RTR   c0-0cs1s0a0  c0-0cs1s0a0
   13  RTR   c0-0cs1s1a0  c0-0cs1s1a0

3) The link architecture supports discovery of faults propagating across connected components, such as a node failure disabling a link and contributing to congestion conditions on its peer node.

   id  type  name  children
   16  GRE   c0-0cs0s0a0120  c0-0cs0s4a0122

A few components, such as the SMW, affect everything in the system. To represent such global associations we also define a supremum relationship.

C. Tools
There is no technical constraint preventing someone from cataloging, annotating and investigating a dataset by manually writing RDF files, populating an annotation database and running SQL and SPARQL queries against them. However, the schema and vocabulary are made accessible to a much wider audience via tools that ease the learning curve and automate many of the processes.

In the course of this work we have developed a small collection of tools for constructing and querying graphs and annotation databases.

1) Cataloging log sources and archives: For easier use of the RDF graph, we have developed a framework, in the form of a python package for constructing and querying the graph. The package is still pre-release and in active development; so far, its capability is limited

Fig. 7. Annotations provide reduced, tractable view of significant events in a much larger volume of log data.
to walking a directory tree to discover and catalog log files – with some constraints on the types of log files it can inspect – and some basic querying. The design is extensible: a small interface is defined for handlers of file and log format types and additional handlers are easily added to the toolkit.

Much information must be collected to create useful metadata: most of this can be automatically inferred by querying the graph and inspecting the file (for example, the filename hints at the likely LogSeries, temporal boundaries can be discovered by reading the first and last records and the subjects for a tree of log data can be inferred from a coarse-grained subject (such as the cluster) and information held by each LogSeries. Details requiring human input are acquired through a simple question-and-answer interface that presents the user with reasonable guesses and prompts editing and confirmation. In this manner thousands of logs can be described based on only a handful of user interactions.

2) Querying Annotations: A stated goal of our project is to enable annotation of log data so that researchers can analyze it with sufficient context to understand what was happening on the system when a given set of loglines were written. However, such a tool would also enable consultants and users to make sense of log data in order to better address questions about job failures. Neither use case would be addressed, however, without an efficient means of querying the annotations. We have therefore created a python-based query engine for that purpose.

The query engine offers a set of command-line flags to customize a query. Users can combine them for highly specific queries without resorting to complex SQL. A user can refine a query by start (-s or –start) and/or end (-e or –end) time, by component name (e.g., node or slot name, -c or –component), by type (descriptive phrase, such as "link down", -t or –type), or by any of the columns in the database schema (-a or –type). In addition, a user can specify a job ID (-j or –job) to retrieve annotations on logs written during the time of the job and affecting any of the nodes on which it ran. Adding the -a or –after flag enables the user to specify a complex query with a single timestamp and retrieve the next single instance of a match for the query.

In addition to specifying a single component, a user can retrieve annotations for related components, traversing the architecture of the system to a specified depth (-d or –depth), or number of hops. For example, a query for a component specified to depth 2 would retrieve parent and grandparent components as well as child and grandchild components. For any depth search, any supremum (eg SMW) or unknown components are also queried. This flag leverages a table of physical components in the database that reflects the architecture of the system for which the annotations were made.

The user also has options for viewing the retrieved annotations. Formatting options include a table (-f table), a JSON array (-f json), or the default textual listing Figure 11. The user may choose to increase verbosity (+v, -vv) or to limit the maximum number of retrieved annotations (+limit).

An additional query for jobs (-j or –jobs) allows the user to enter one or more annotation identifiers and retrieve the list of job IDs that could be affected by the annotation, based on the time and the nodes on which they ran.

The current query engine retrieves annotations only, though the annotation metadata provides sufficient information to locate relevant log files. We plan to enable retrieval of subsets of the logs through the tool in the future.

3) Creating Annotations: Annotations can be added to a database by several means. The simplest approach would be with an SQL insert at an sqlite command line, but this method can be tedious and relies on manual input. One simple way of adding a large batch of annotations is to enter them as a spreadsheet in CSV format, using a small uploader script which we provide. These approaches work well for human-generated annotations, such as individual observations from a system administrator or output from a ticketing system.

To facilitate the automated creation of log line annotations and the identification of the occurrences of events to be annotated, we have been using two external tools, Baler and LogDiver (described in more detail below). In support of these we have developed tools to create annotation using output from these.

Note that, while we use SQLite in our prototype, SQL databases are not required by our design. A plugin interface to a component presenting an API which could return similar annotation information would also be appropriate (e.g., one could front a tool with its own datastore). However, to enable a simple, consistent format for portable general release of an annotated dataset we do so, as described in the next section.
Fig. 11. Annotations as returned by the query engine in default format.

V. PROOF OF CONCEPT

To facilitate the creation of log line annotations and the identification of the occurrences of events to be annotated, we have been using two tools.

LogDiver [11] is a tool developed by UIUC which includes a set of regular expressions defining events in log files of interest; the regular expressions are associated with categorizations which are a subset of those described in the previous section; the category name, LogDiver category, was chosen to reflect our intention to map to the LogDiver categorizations where possible. LogDiver itself is used to discover the occurrences of the regular expressions in the logs and to determine statistics and information about event sequences such as statistics of failure events, or of timings of failures and recoveries. LogDiver, or any such regex-based tool (e.g., SEC), can be used to efficiently extract events for subsequent annotation, based on the intention for the existence of the regex.

For the dataset described in this work, we principally used Baler [4] for identifying the log lines to be annotated and for extracting them from the dataset. Baler extracts patterns from log files without requiring apriori knowledge of regex of interest. Rather, Baler takes dictionaries of "words"; words appearing in the log lines are the passed through to the pattern and non-words become a wildcard in the pattern. Wildcards of certain formats, for example numbers, hex dumps, char arrays, hostnames and link names (in cname format for Cray systems) are represented as that formatted type in the pattern. For example, every instance of the log message mutrino-smw 24626 found_critical_aries_error: Processing ''PCI-e CMPL_TIMEOUT’’ critical error (0x660e) is represented by the pattern <host> nlrd <pid> found_critical_aries_error: Processing ‘’* _TIMEOUT’’ critical error (<num>). This illustrates where words, formatted wildcards, and unformatted wildcards (represented by *) appear in the pattern.

For Cray systems, we augment the dictionary with an architecture specific dictionary of about 100 words (e.g., Lustre, DIMM). For 3 months of data [12] from our Trinity test system, Mutrino, a 100 node XC 40, we had over 120 million text log lines which were reduced to 15,500 patterns 1. To further identify patterns of interest, we weight the patterns by the occurrence of 50 weighted keywords (e.g., congestion = 1.5, error = 1.5, degrade = 0.75). This further reduced the patterns to 2,500 significantly weighted patterns. For example the pattern <host> nlrd <pid> ***ERROR***: Link recovery operation failed; error <num> has an aggregate weight of 5.5. From those, we chose 150 patterns (about 1 percent of the total) to annotate with enhanced descriptions. This resulted in about 860,000 annotated log line instances.

It is our intention to build a plugin to interface with Baler, and support the annotations there, however in the prototype, we merely annotated the extracted patterns from Baler and loaded them into a single database. We include the Baler pattern id in the annotation fields for reference ease; only the annotation description, not the original log line nor the pattern, is stored in the annotation database.

Some example patterns, from which the originating log line will be obvious, and the resulting annotations used in this work, are given in Figure 12.

We fed Baler with the Cray logs in the format in which they reside on the smw. This required us to do some file-specific format extraction of messages, timestamps, and components (e.g., netwatch, hwerrlog) which we may not have had to do if we had fed it raw syslog versions of the files or datastream; however, this also enabled us to include the log file type (e.g., nlrd, hwerrlog) in the pattern metadata, which aids the log file look-up.

For many cases, messages are reported on the smw with the component association of the smw. For some cases, we can extract the actual component of interest. For example, from the Baler pattern <host> nlrd <pid> found_critical_aries_error: handling failed * link on <host> (node ) we can infer the fields from which to extract the host to which the annotation should be associated. Other messages refer to actions by the SMW for which the component cannot be inferred. In these cases the component assignment will either be the smw or ‘unknown’.

We used Baler for all major log processing, except for the command log, which required us to associate START and END of events, for which we used a perl script. This log includes both manually initiated and automatically invoked commands of interest such as warmswaps, boots, etc. This was particularly useful for determining manual actions that may not have been well documented by the system administrators. This resulted in another 2,000 annotations. In addition, we extracted the times of reboots from the datetime in the name of the p0-XXX directories. All annotations from these two sources are attributed as manually induced.

1This differs from some previously published [13] numbers due to some new pattern formatting features and the lack of metachroning in this work.
A. Job Impact

We have annotated messages relating to potentially performance impacting conditions, including thermal throttling events, power budgets exceeded, and memory errors. Example annotation descriptions include:

- Correctable memory error. This may result in degraded performance.
- Blade or Cabinet controller taking correctable memory errors. This may affect performance.
- Package temperature above threshold (too hot). The CPU clock has been throttled. Should result in all threads for all cores will be throttled. This may affect application performance.

Of particular interest in XC systems is the ability to power cap. In such cases, not only is it of interest when the power budget is exceeded, but also when caps are applied, or perhaps fail to be applied. Our annotation system enables such events to be exposed to the user. Many of these messages are identified by commands in the commands file or in the controller logs and therefore are not typically released to users. Since our move to SLURM, there has been no automatic reporting of cap settings to the user. Currently, our users must manually cat /sys/cray/pm_counters/power_cap on each node to see current cap setting, or poll it to catch dynamic changes during runtime.

Examples include commands annotated as:

- applying a power profile
- enforcing a power limit

error messages annotated as:

- Error getting initial node power status. This may affect power capping.
- Error disabling power monitor. This may affect power capping.
- Node Error setting power budget. This may affect power capping.

A fundamental goal of the annotations to expose information to users that will enable understanding of why performance and power limits did not perform as expected. Basic capabilities enabled by our infrastructure include the ability to determine annotations occurring during a given job and jobs running while an annotation occurred. Examples of each are given in Figure 13. In the top of the figure, all annotations during a job are queried – it is seen that multiple times on multiple nodes, the power budget is exceeded, which may result in performance impact and may be unanticipated by the user who set the cap. The capping mechanism is intended to keep the average power draw at or below the cap over a few second window or below the cap; however instantaneous values may exceed the cap. Excursions over the cap are not indicated to the user. In the bottom of the figure jobs running while an annotation occurs are queried. In this case,
while the annotation was specific to node c0-0c1s3n2, the job itself was running on 96 nodes. Facilitating tracing the propagation of impact of an event is one of the design goals of this work.

B. HSN Congestion

In order to investigate network conditions, we chose to search for annotations involving the word "congest". Query results are given in Figure 14. While this looks like the expected output of response to a congestion conditions, with candidate nodes and applications as computed by the system, the occurrences were surprising on a system of this size.

These annotations guided us to the nlr/d in search of applications of interest. The link from annotation to relevant slice of the source logfile is a planned capability of the graph-query tool, however that functionality was not yet available so a visual inspection of the relevant file was used.

There were, however, no applications listed. As a result, we then queried for all annotations around this time window to find indication of a non-application congestion cause.

A query for annotations between 10:00 and 10:32 on that day resulted in 300 annotations, with only 7 distinct ones. The reduction in log lines to annotation instances makes investigation of time ranges tractable and eases discovery of similar event instances. Other than the ones in Figure 14, the rest dealt with problems with a single component c0-0c1s8a0n0 and system response to congestion:

- 192 occurrences for c0-0c1s8a0n0 of Correctable memory error. This may result in degraded performance.
- 47 occurrences for c0-0c1s8a0n0 of Component failed.
- Telling all blades to throttle network bandwidth. This should result in decreased network injection.
- Telling all blades to unthrottle network bandwidth. This should enable increased network injection.
- Unthrottling the service blades only

While we cannot be sure from the annotations alone that failure in this component was the cause of the congestion, it is clearly a strong suspect.

Querying for annotations for c0-0c1s8a0n0 revealed that the component's problems of these 2 types started on 2015-04-02 10:22:47 and ended on 2015-04-30 at 07:16:52. Narrowing down the root cause of the problems is difficult, however, because of a number of deliberately induced failures during facilities testing which occurred that day. While we have put in a manual annotation for Facilities testing, the distributed system which we envision would have enabled the Facilities staff to annotate in more detail the exact testing which occurred. Currently the facilities testing annotation has to serve as an indicator to examine the logs in which indications of induced fan and power failures occurred.

The resolution of the problem is discovered by using a depth search to query annotations of related components: here -d 2 includes two levels of parents (c0-0c1s8a0 and c0-0c1s8) children (none), and any unknown/supremum components. The depth was chosen with the expectation that resolution would occur due to actions at the Aries or blade level.

While roughly 500 annotations occurred in response to the query, only about 30 distinct annotations occurred. This is in contrast to the raw log files in which over 153,000 log lines occurred. The annotations make it easy to understand the sequence of events. Extracted annotations are in Figure 15. First an annotation of a system administrator, 'abc', action, generically assigned to the day (green) confirms that the blade is being reseated in response to errors. Warmswaps of the blade occur (cyan), however, while the warmswaps report as successful, timeouts waiting for items in the Outstanding Request Buffer (ORB) result in the ORB being 'scrubbed', delaying the recovery (red). The annotations help with the understanding of the ORB scrub related events. Eventually the blade is added back (green) successfully and the blade is then booted.

The annotations additionally make it easy to compare and investigate timescales of similar events. For instance, a recovery analysis might be based on the occurrences and durations of the warmswap sequences. This case might appear of longer duration that others, and the intervening ORB scrubbing events that needed to be handled for full recovery would be easily apparent.

C. Root Cause Diagnosis

Another network investigation started with a search for annotations involving the word "route". We were expecting to see events about reroutes triggered as a result of component failure. More interesting are cases where the reroutes failed. Query output is shown in Figure 16.

Each annotation Setting alerts due to failures. A network reroute is required makes clear that there has been a failure and what the next step in the response will be. It appears that 3 failures occurred that require network reroutes to recover and two of the computations of those route computations failed (red).

To understand why the route computation failed, we query for annotations during a time frame preceding the event, limiting the options to network (NE) annotations only. Query output is shown in Figure 17. There are only 24 total annotations as opposed to the raw log lines which total over 238,000. It is clear from the annotations, that the component triggering the problem was c0-0c0s9a0 and that while the recovery operation for a failed link was successful (green), the failure of the reroute was due to a problem in adding the blade back to the HSN (to include it in the routing) (orange).

This drives us to investigate problems with c0-0c0s9. Query output is shown in Figure 18. The full output has 90 annotations, but there are only a few distinct ones (shown). There is an out-of-memory killer annotation (red) that occurs repeatedly (repeats suppressed in the figure). Of particular interest is that the out of memory problem is reported by the blade controller (file is controllermessages and component is the blade), as opposed to a user process being killed by the OOM killer on a node.

While determination of the exact cause of the problem may necessitate involvement of a vendor, it is at least obvious from this that the blade controller operating system believes it is experiencing a low memory condition and taking active measures to prevent complete failure. This could be due to a variety of reasons such as a memory leak, a communication problem causing buffering of messages to fill memory, etc. Unfortunately the processes it kills may be needed for proper operation and the problem being fixed by a complete reboot seems to validate that the problem was a software/firmware state issue and not hardware failure.

Finally, we are interested in determining if this problem got resolved and how. We utilize the depth search -d 1 to query parents (c0-0c0), children (the nodes and Aries), and any unknown/supremum components. The depth and time range currently are chosen by trial and error, however from output in Figure 19 it is clear that the OOM messages continue until an unsuccessful attempt is made to power down the blade (orange), and a few attempts are necessary to reboot the system (red) and clear the alert (green).

This case also illustrates the endstate field (which is automatically populated with the end state of commands in the command file (described in Section V)). Note that erroneous commands, for
Annotations provide access to power state information in unavailable logs. Basic implementation capabilities include discovery of annotations during a job (top) and jobs running while an annotation occurred (bottom).

Annotations about the time frame for the interest name and any components within a depth of 2
python get.py -c c0-0c1s8a0n0 -d 2 -s "2015-04-30 07:00:00" -e "2015-04-30 10:00:00" -f table annotations

Logs file, but no applications were listed.

Fig. 13. Annotations provide access to power state information in unavailable logs. Basic implementation capabilities include discovery of annotations during a job (top) and jobs running while an annotation occurred (bottom).

Annotations about the time frame for the interest name and any components within a depth of 2
python get.py -t congest -f table annotations

Fig. 14. Congestion response annotations occur 5 times within 15 minutes. The annotation regarding candidate applications drove investigation of the nlrd log file, but no applications were listed.

Fig. 15. A blade reseating was performed to resolve blade problems which led to the congestion event. Multiple iterations of scrubbing the Outstanding Request Buffer (ORB) were needed which delayed resolution. The annotation of the system administrator (identified by 'abc') action supports the diagnosis.
Fig. 16. Output of query for route annotations. Complete output = 15 annotations. Occurrences of network reroutes and failures in the rerouting process are of interest.

Fig. 17. Output of query for network related annotations to investigate the cause of the failed routes. Complete output = 24 annotations. A search of the raw log lines would be much more labor intensive ~ 238,000 raw log lines occurred during this period.

Fig. 18. Output of query for annotations to investigate the cause of the component failure. Complete output = 90 annotations, about 10 of which are distinct. For example, the node-related annotations occur for each node on the blade and many repeat in time and are suppressed in the figure. An OOM killer event occurs which is reported by the blade controller, not a node.

example xtcli power with incorrect argument or target specified at 11:05:57, result in error. Also illustrated are manual attribution of any annotations of events from the command file and p0-XXX directories. The latter are attributed to a generic system administrator, a chm, for the annotation, as opposed to the human annotation of the blade reseating in the previous example.

VII. CONCLUSIONS

The Holistic Measurement Driven Resilience (HMDR) project seeks to characterize faults in modern large-scale systems in terms of root and/or most probable cause, likelihood of detection, frequency of occurrence, timescales for resultant system impact, and efficiency of error recovery. In addition we seek to determine instrumentation that can be used for fault detection, characterization, and triggering of response mechanisms.

In this work we developed a basis - in the form of a machine-readable vocabulary and an annotation schema - for cataloging and discovering collections of log-like data and for annotating them to expose a tractable view of significant events, expert commentary and contextual notes. We also developed tools that use this basis to find and filter log data in support of bring failure analysis to a tractable scope.

Our annotations of key events enable more efficient search and identification of events, locations, and timescales of interest. Further, the annotations enable identification of external events, such as fault injections and component replacements, that will enable more
# query for annotations between the time frame of interest for the named component and any components within a depth of 1
python get.py -s "2015-05-08 08:35:00" -e "2015-05-08 23:35:00" -c c0-0c0s9 -d 1 -f table annotations

<table>
<thead>
<tr>
<th>id</th>
<th>authorid</th>
<th>starttime</th>
<th>endtime</th>
<th>endstate</th>
<th>description</th>
<th>manual logfiles</th>
<th>lcategory</th>
<th>components</th>
<th>balerpatternid</th>
</tr>
</thead>
<tbody>
<tr>
<td>865800</td>
<td>acg</td>
<td>2015-05-08 08:40:24</td>
<td>2015-05-08 08:40:54</td>
<td>0</td>
<td>xtcli power down 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>866017</td>
<td>acg</td>
<td>2015-05-08 08:51:18</td>
<td>2015-05-08 08:52:39</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865959</td>
<td>acg</td>
<td>2015-05-08 08:50:09</td>
<td>2015-05-08 08:51:05</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865426</td>
<td>acg</td>
<td>2015-05-08 10:15:04</td>
<td>2015-05-08 10:15:21</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865420</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865429</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865422</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865423</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865424</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865425</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865426</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865427</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865428</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>865429</td>
<td>acg</td>
<td>2015-05-08 10:15:07</td>
<td>2015-05-08 10:15:08</td>
<td>0</td>
<td>xtcli power up 1 commands PW</td>
<td>&quot;c0-0c0s9&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 19. Output of query for annotations to investigate the resolution of the component failure. Attempts to address the blade itself were unsuccessful, and several reboots were required before the alert cleared.

accurate characterization of fault occurrences and impact.

In addition, HMDR releases datasets for resilience research. We will be releasing the annotations in this work to augment the dataset, which is currently released. In addition, we will be releasing an annotated dataset from controlled, complex single and multi fault injection tests [14] on the (now retired) 9000 node Cielo Cray XE system. The annotations will facilitate understanding of the datasets for these two different generations of Cray systems.

ACKNOWLEDGMENTS

The authors would like to thank Saurabh Jha (UIUC), Tom Tucker (Open Grid Computing), Kevin Pedretti (SNL), Jason Repik (SNL), and the HMDR team for useful conversations.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number 2015-02674.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, and Sandia National Laboratories, a multidisciplinary laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525. The views expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

REFERENCES