Observability in and around Academy

Ben Clifford
benc@hawaga.org.uk
2026-02

“Observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs.”

-- https://en.wikipedia.org/wiki/Observability

- what does that mean for academy?
- “a system” - academy the python library, the platforms it runs on (globus compute, parsl, linux kernel, slurm), the applications users build on top of academy
- “the external outputs” - deliberately generated outputs like log files, not the user’s application results
- “the internal states” - I take a question based approach: Q: why did this academy action take 12h to run? A: we can observe that the agent hadn’t started yet, was in a queue for a batch system for 11h59m10s, and then took 49.5s to execute and 0.5s to get result back to user... and then follow on questions.

what exists already?

- (not exhaustive attempt, some stuff from my own past and present)
- Parsl - monitoring, logs
- Academy - logs
- Globus Compute - some logs, some via human interaction: web portal, some ad-hoc via helpdesk
- slurm - squeue, sacct command lines (and other batch systems)
- linux kernel - dmesg and ebpf
- opentelemetry, and other “observability” oriented approaches
- (here or later flesh out user experience)

high level framework

- not a “product”
-
- create observability output
- => move&store that output to where it is useful
- => analyse that output

wide flat records

- recent latest cool in observability
- events happen
- they are repesented as a wide flat record of key/value pair - (with a database mindset, wide means many columns, one row per event). observability extremists might have 1000s of events per column!
- Python dict or JSON object

- python `logger` as a source of wide events:
- logger is well understood, widely used in Python code. very configurable.
- supports “extra” data dictionary to convey key/value pairs
- see merged PRs

other sources of events

- non-Python-logger, or complete non-Python
- log files from other systems
- ad-hoc format projected into dict or JSON
- eg. process monitor script interfacing to linux kernel
- can’t necessarily instrument everything to use what we want: HPC center won’t install custom slurm just for your app (that conflicts with the custom slurm it installed for someone else), and the kernel answer to process events is probably BPF not Python logging.

- no universal schema
- (so events are ad-hoc schema - doens’t mean no schema, means that infrastructure is not enforcing a schema)
- great way to sabotage concalls and create weekly meetings that go nowhere!
- if theres a “universal” schema that is universal enough for you - this infrastructure won’t stop you using it

moving and storing

- classic Python approach:
- logging.FileHandler
- each event becomes a line in a formatted text file
- (show an example here of such a log line)
- filesystem. or move between filesystems using file transfer tools.

moving and storing: over network

- parsl monitoring: pickle-over-UDP => shred into SQL DB
- chronolog, diaspora octopus - research projects
- open telemetry - moving to a distance place is baked right into the name!

analyse the output

- lots of different questions people might ask
- traditional approach of PIs focused on task execution systems is a fixed scope dashboard... because: cannot imagine serious questions, and cannot imagine interacting with data outside a dashboard
- Parsl user experience: people load the DB into Python and do numpy/matplotlib.

... but Parsl people also...

- Parsl people also ... parse logs using ad-hoc regexps on human readable text (me included)
- Why?
- Forgot to turn on monitoring, parsl.log is more default
- Parsl Monitoring doesn’t have the right data
  - fine tracing data is only in debug logs
  - some components make their own semi-structured logs
- (this feeds back to multiple event sources and broad schema)
- Do not pretend this is out of scope unless you want me to tell users they can never know the answers to their queries
- Not accepted this as a proper/non-guilty use of Parsl so not built infrastructure around it
- but this is extremely fragile

joining event sources together

- I’ve talked about a common wide event model, based on dicts
- but that doesn’t relate one event stream to another
- opentelemetry style observability has a span/trace approach focused on a hierarchy routed at a server request - that isn’t right for a lot of analyses here

opentelemetry hierarchical spans

- also a direction I pursed with parsl a few years ago
- hierarchical
- does an agent action live under an agent? does it live as part of a submit-side function that is running?
- yes

no universal ID

- we can’t instrument everything in the stack
- we can’t put a universal ID on everything
- (such as a uuid)
- e.g. linux is going to use PIDs, with its own uniqueness and non-uniqueness (not unique over time, not unique between hosts)
- but represent relationships relationally in events (see relational algebra or RDF triples)
- notions of keys - set of attributes that distinguish a “thing”. “thing” is deliberately vague: maybe its an agent. maybe its an action on an agent. these overlap! so the “thing” that you care about depends on your analysis/query. “thing” could call an object but don’t want to bring in peoples prejudices about what an “object” is.

python implementations

- academy (and in progress in parsl) -- give examples
  - logger calls get extra parameter
  - logs as json files
  - logs into octopus prototype
  - configuration API for configuring the same logger in multiple environments, and as visitors (this could be its own slide?) - visitors is not complicated if you regard it as resource allocation and release rather than “I own this process, framework style”. “libraries should not be configuring logging. emphasise pluggability here (although pluggabilty/modularity is present throughout the framework)”
  - loggers by object, not by module (see Parsl prototype) - to attach object-related metadata more automatically.
  - anyone in python can log to a logger! so user agent code can also log. what does that look like?
- analysis side
  - common patterns “how many of this thing was happening at once, here’s how you identify a thing and when it started/stopped” - how many at once, histogram of duration, &c
  - tools to load json
  - tools to mutate json by potential query language-style helpers - review my big document and plotting code and see whats there. for example, widen by lambda, widen by joining. widening as a key operation.
  - examples: parsl lifetime plots from blog - “should” be adaptable to events in academy straightforwardly. academy message lifetime vs agents timeline plot i made before
  - question: what do “non-python” interfaces to this look like? for example, web UI (like the Parsl one) without it trying to be the universal solution? Analyses don’t have to look like log files or plots: for example, plenty of stuff in the Literature about machine learning on log/trace data. Are there uses here?

demo - perhaps scattered throughout the talk?

- fibonacci demo running on local system with agents running in parsl htex (so logs would be in a worker)
- see academy logs. see how parsl htex worker logs have not been damaged by academy observability reconfiguration
- see fibonacci logs - showing state of fibonacci algorithm at all times
- overlapping invocations, perhaps, or some other complication?
- do some analysis:
  - gather all logs into one python process - annotate by logfile source
  - pretty print logs from across system (filtered by something that isn't logfile identity)
  - some kind of plot, eg. agent communication plot? or activity start/end in different actors on a timeline? that shows unification of observability rather than any deep insight.
  - perhaps *two* different *conflicting* hierarchical view to emphasise "spans are in the eye of the beholder"
- compare to main branch and pre-my-PRs academy to highlight what I have contributed so far as user facing functionality: json logs. visitors? (vs forced log override? against a "pytest is a user representative" story for forcing overrides: logging one-shot logging.basicConfig call does nothing a lot of the time(!) vs force.
- compare to using eg. Diaspora Octopus - imagine what changes? (config... log retrieval...) and what stays the same. and what does an "Octopus component owner" need to provide?
- diagram of what is being initiaized in what process, and what is being uninitialized, including parsl-level logs.
- can also show submit process logging to console with colours, to demonstrate multiple configs and show how this work doesn't magically move logs around.
- opportunity to get the local redis factory logging some interesting stuff too... not sure if ti does at the moment. or perhaps locally hosted cloud http, because that's more interesting to see as a use case? it lives totally separate from the "workflow" configs because it is controlled by a different entity!
- for example, while we have "agents" as a primary concept, we also have Parsl workers, and various processes as other kinds of loci-of-execution that it would be interesting to understand.
- ... redis also makes (console) semi-structured logs and so that's another example of "somethign going on" that I probably care a bit about but doesn't fit in with the agents-as-primary-loggable-entity story

demo step 1

- fibonacci demo. we can see it uses a few agents. can see results happening. can’t see what else is happening inside the “system” (diagrammed, that is the agents and redis and parsl and various linux processes, etc)
- no observability but this defines the entire “user facing” app.

demo step 2

- console logging (with colours!)
- could also be to a file, which we could send to someone else / grep / shitty-parse - not too different either way.
- `main` branch way of configuring things (but perhaps using my modified syntax so that I don't have to switch branches?)
- can see three different "components" logging: __main__ from my application. academy.*, and parsl.*
- but only from the one "workflow" "submit" process...
- Q1: how can I do analysis of this data?
- Q2: where are the logs (shown in source code) for the agents doing things?

demo step 3 (q1)

- how can I do analysis of this data?
  - parsl monitoring dabase is great letting people analyse parsl behaviour - because it is structured data (but, also bad, because too restricted...)
- log events to file not console
- avoid parsing human messages
- in academy main already, but I'll use my configuration syntax here.
- jsonpool FilePoolLog
  - makes a "run directory" and in there one json-object-per-line log file. view with jq to see it nicely coloured and formatted.
- a few things:
- json rather than column format: nothing particularly magic about json, but structured key-value pairs rather than line formatting. hard to read as human, easy to read into a table. extensible.
- extensible because... there are extra entries with academy meta data

- {
- "formatted": "Registered UserId<229f8512> in exchange",
- "name": "academy.exchange.redis",
- "msg": "Registered %s in exchange",
- "args": "(UserId(uid=UUID('229f8512-87c1-4e29-add0-4b821b3a6c82'), name=None),)",
- "levelname": "INFO",
- "levelno": "20",
- "pathname": "/home/benc/parsl/academy/academy/academy/exchange/redis.py",
- "filename": "redis.py",
- "module": "redis",
- "exc_info": "None",
- "exc_text": "None",
- "stack_info": "None",
- "lineno": "132",
- "funcName": "new",
- "created": "1770216523.5438404",
- "msecs": "543.0",
- "relativeCreated": "1676.015845",
- "thread": "140438027604224",
- "threadName": "MainThread",
- "processName": "MainProcess",
- "process": "3963",
- "taskName": "Task-1",
- "academy.mailbox_id": "UserId<229f8512>",
- "message": "Registered UserId<229f8512> in exchange",
- "asctime": "2026-02-04 14:48:43"
- }
-

- ... and that comes from Python logging extras...
- like this:

```
        logger.info(
            'Registered %s in exchange',
            mailbox_id,
            extra={'academy.mailbox_id': mailbox_id},
        )
```

- if you’ve done any observability work, this should not be surprising: key-value wide events with attributes are normal things
- so I’m not proposing anything new here, i am proposing that we should treat this as a best practice

demo step 4 (Q2)

- Q2: where are the logs for agents doing things?
- we configured logging for a Python interpreter (roughly: a unix process)
- that is both too much (also saw Parsl logs)
- and too little (didn't see agent logs)
- show file logging (to ~/.academy)
- ... but show it being distributed across multiple processes
- the agents are running in worker processes... and the workers are configured to at least log parsl.* to (e.g.) runinfo/135/htex_Local/block-0/f87139072c28/worker_*log
- ... but not any academy.* or my application logs
- `main` academy lets you specify a log configuration... which uses logging.basicConfig with two modes that are bad:
  force=False - if someone else has already configured a root logger, do nothing - someone else can be any other library running in this process, if you claim libraries can do this, and any other library can do this too and delete your agent config... including (!) other academy agents starting up in the same process(!)
  force=True - delete what someone else has already configured, and use our own config
  - this is also a place to mention pytest showing the badness but "grumble, pytest" rather than "pytest as user representative"
  - oldnewthing thing about imagining what if everyone else did this? "what if two programs did this?" https://devblogs.microsoft.com/oldnewthing/20050607-00/?p=35413
  - why do we care? well if we don't care that our log configuraiton is beign deleted, why bother turning it on in the first place?
    what will Parsl support team (i.e. me) say when I ask for your parsl logs and you say "haha no i turned those off, I'm not giving you any clues on how to help me"

log configuration for visitors

- we are a visitor in someone elses process
  - academy as a library, even in the submit process, is not the owner of the process
  - see in Parsl, assumptions of being “in charge” conflict when Parsl is used as a library inside some other code that also believes it is in charge
  - academy “force” mode inside pytest: academy wants to be in charge, but it conflicts with pytest which is also in charge
- we might be a visitor in several processes
  - agent runs elsewhere, not in the submitting process
  - that elsewhere may have its own complex logging needs - see Parsl htex

log configuration for visitors

- log configuration that we can serialize to a different process (just like we do with agents and parameters and return values)
- log configuration that respects owner and other visitors:
  - initialize in a minimally interfering way at start
  - uninitialize at end, also in a minimally interfering way

lets use that json pool logger for our agents too!

- instead of only initializing_logging(), we can do other things with the log configuration object
- like send it alongside an agent, and let academy do something useful at the remote end
- ah = await m.launch(a, log_config=lc)
- now see the run directory has multiple json files in it:
- $ ls ~/.academy/logs/87fd6a71-32b0-40ef-a766-1cd9ea2bdb09/
- 722b0f65-947e-4cdd-b3fb-e2bd744e10df.jsonlog
- e92a7250-9539-48e5-b4ef-368f51487001.jsonlog
- one file is the submit process, as seen before
- the other file comes from the agent, running in some parsl worker (without interfering too much with parsl logging in the same process)
- so now we have a pool of json records from various pieces of academy that we can do some mechanical analysis of

q3: what about when there is no shared filesystem?

- classic grid computing situation from 1990s or contemporary cloud computing situation from 2020s
- distributed telemetry - eg opentelemetry, hosted cloud logging services
- octopus, chronolog - research ideas
- need to be open to research ideas, while acknowledging they are not core products
- so these can plug in through the same mechanism
- i won’t explore these any futher in this talk other than as a direction for hacking and discovering what works
- again this is not “new stuff” that needs reinventing, despite the desire to reinvent.

“log analyses”

- parsl monitoring web UI shows a (much unloved) attempt at a GUI for some Parsl data
- lots of log parsing for other stuff, across components - with regexp
- expect many of our users to be Python-data literate - expectation from observing Parsl users
- pursue “non-Python non-data literate” avenue - sure, but it needs serious use-case analysis: syntax is probably not the barrier here
  - vs “what does this data *mean*” and “what are useful questions to ask?”

performance analysis of demo code

- we should “naively expect” this run in ((500ms + 500ms) x 16 = 16 seconds)
  - (expectation comes from 2 sleep calls per iteration)
- On my laptop, actually takes 24.7 seconds
- Q4: where does that extra 8.7 seconds come from?
- (to me) this is more interesting than debugging “failures” because there is nothing like an Exception to start at - the execution “succeeded” in that sense even though it “failed” my performance expectation

“is this working?” = performance study

- is this performing? is a similiar question to “is this working?”
- “is this hung?” “is ‘performance’ happening?”
-

jq

- we can ask to see all the message-related logs from all (both) log files in a run. message-related in this sense means “has a message tag”
- cat ~/.academy/logs/1865863e-556e-4d21-ae5f-6f809f01132b/* | jq '.| select(.["academy.message_tag"])'
- or we can pick a message from that output and match exactly that message tag:
- cat ~/.academy/logs/1865863e-556e-4d21-ae5f-6f809f01132b/* | jq '.| select(.["academy.message_tag"] == "9bfde10f-ff05-4c7c-a44f-e9eb0da00be4")'
- this is the “grepping logs” equivalent but with a bit more structure:
  - academy.message_tag should always be in the same namespace of messages
  - the identifier space is UUIDs which means identical UUIDs should refer to the same “thing”.
  - but: string comparison on string UUID representation is not a valid comparison!
- for this I see 5 messages. in a weird order. (it’s actually the order they were read out of log files, rather than “time” order)
- get entries and sort by date:
  cat ~/.academy/logs/1865863e-556e-4d21-ae5f-6f809f01132b/* | jq --slurp 'map(select(.["academy.message_tag"] == "9bfde10f-ff05-4c7c-a44f-e9eb0da00be4")) | sort_by(.["created"] | tonumber)'
- (to me vaguely reminiscent of xpath)
- but I’m not suggesting this is how you interact with logs... just one example of using existing tooling
-
- datatypes: see how I turned created (which is a string unix time stamp) into number, so that it sorts numerically not alphabetically. c.f. uuid comparison.
-
- this is using ~/.academy as a log store. if you were experimenting with some other log tech, you might query records in a totally different way. for example, logs in cloudwatch can be queried in a web portal with a different query language.
- if you were doing Parsl Monitoring style stuff, you’d put this in an SQL database - eg shredded or JSON-enabled sqlite3
- (research area, at least in the past: what are nice query languages for this?)

Log quality

- here’s sorted events for one academy message tag with a few fields extracted:
- $ cat ~/.academy/logs/1865863e-556e-4d21-ae5f-6f809f01132b/* | jq --slurp 'map(select(.["academy.message_tag"] == "9bfde10f-ff05-4c7c-a44f-e9eb0da00be4")) | sort_by(.["created"] | tonumber) | map("\(.created) \(.processName) \(.formatted)")'
- [
- "1770228140.9842227 MainProcess Sent ActionRequest to AgentId<16f14515>",
- "1770228140.9845004 HTEX-Worker-0 Received ActionRequest from UserId<616d4ec5> for AgentId<16f14515>",
- "1770228140.9845388 MainProcess Sent action request from UserId<616d4ec5> to AgentId<16f14515> (action='__anext__')",
- "1770228141.4881656 HTEX-Worker-0 Sent ActionResponse to UserId<616d4ec5>",
- "1770228141.48863 MainProcess Received ActionResponse from AgentId<16f14515> for UserId<616d4ec5>"
-]
- Two processes: the submit side process and an htex worker (where the agent happens to be running)
- can pretty much see the 500ms deliberate sleep in action processing (503.6ms ... 3.6ms coming from somewhere else ... probably not a big problem in our Q4 analysis)
- can see submit side is logging “sent action*request” twice in different forms (from different layers of Academy) - maybe this is confusing.
- there is some interesting ordering around the messages from two distributed components - this is a distributed system so we can only expect messages to be partially ordered - is one of these messages “I have sent” and another “i am about to send”?
- Caring about what is logged is something people have to care about. These techniques won’t magically make logs good.

keys to “things”

- in previous side, used “ProcessName”. That doesn’t uniquely identify processes at all! so it’s useful for humans but not ok for (eg) mechanical equality
- what are “things” - i vaguely mean “objects” without bringing in peoples individual prejudices about what an “object” is.
- different at different times -- previous slides cared about academy messages, other times we care about academy agents. not necessarily heirarchical - e.g. stuff to do with an agent (including on the invoking side) overlaps unix processes in a non-heirarchical way.

- back to Q4
- we’ve got lots of event records. we can identify various “things” which those event records are about.
- lets use Python to plot some graphs

A web UI of common queries

- What are the common analyses/queries someone might want to make?
- that would be worth packaging in a GUI (probably a local web UI like parsl?)
- ... to avoid everyone doing Secret Parsing

- build the web UI in slides, but only showing the data processing parts, without showing the web UI parts. so that these examples are all valid for interactive/notebook use

data model

- ... especially joins vs hierachical span IDs (opentelemetry style)
- use “broader application” and “tighter subsystem” examples. Parsl is a tighter subsystem. A “user application” is something hypothetical, but perhaps I can phrase it in my fibonacci system with something that uses fibonaccis?
- “transform” / query operators from my notes here as part of describing visualization analysis?
- just to give a taste of what might be here, not prescriptive/exhaustive

other data sinks

- but lets not forget our users are often skilled data manipulators and also want to do their own processing.
- by definition, Python programmers (like it or not)
- script however they want... notebooks, ad-hoc scripts, ...
- or any other (non-Python) tools... even a spreadsheet in libreoffice (demo via CSV of something simple - eg a simple plot)
- opentelemetry (one actual user) - this one you'd probably want to attach quite early in the pipeline rather than at the post-facto analysis stage... but conceptually "the same"

Other data sources

- How does this relate to other things?
- translate into wide events (jsonish/dict-ish) from other formats
- translate wide events into other formats - perhaps you have some other monitoring/event processing/tracing system to integrate into?
- Is there an example I can easily do for this demo?
  - some kind of redis debug mode? doesn't look much correlatable info except by time. where we can see, for example, the number of entries increasing.
  - or kernel? (bpf on processes? joins onto process IDs!)
  - or parsl monitoring DB - very different looking and I've already done stuff importing it as events! we'd see the blocks (for example) starting up, and that would help us dig into the startup time a bit more. and some interesting notions of how to tie up the blocks and the agents (perhaps by worker process ID?) - two ways, could either talk to the monitoring DB using SQL, or could convert the whole DB into events (because it's quite eventy)
- goal: facilitate users integrating with other systems in a loose framework
- non-goal: universal monitoring system that aims (and fails) to do it all
- non-goal: simple system that doesn't let you dig in enough with other correlated data, self-defeating mindsets like: "we can see there's a problem here but 'nobody' is qualified enough to understand it" - we are doing serious work here.
- there's always going to be some impedence mismatch here - let's not pretend otherwise. user has to deal with that, if they're using multiple data sources.

what does future development look like?

- who wants to actually do stuff rather than express vague noncomittal interest in the end product?
- whoever wants to do stuff ... can ... do that stuff