parsl sustainability - interactions between humans and the codebase

Opinions of Ben Clifford <benc@hawaga.org.uk>
APeX 2026

4 Types of Research Software Engineer

Research programmer

Main job: prototype ideas very quickly

Essential to bootstrap new research software project, test initial ideas

might take shortcuts that harm project's later sustainability

Software developer

Main job: development professional-class research software

Focus on software itself & its users

Process may impede fast innovation

User/developer

Main job: a scientist or disciplinary researcher

Focus on their own usage of the software

Write code elsewhere for their real job; a power user of Parsl who can understand bugs and fix them

Collaborating developer

Main job: responsible for a collaborative project

Focus on their software working with or being "part of" Parsl

May follow different coding/software engineering style, potentially leading to culture challenges

Defining the interface to these developers is a key challenge for sustainability

copied + edited from https://archive.fosdem.org/2025/schedule/event/fosdem-2025-6244-research-software-sustainability-and-rses/ - Katz

UC and UIUC had some grants over the last 4 years that in part funded sustainability.

that funded for example, Sophie our former community manager organising a lot of things, joining numfocus, and stuff like that that I am not going to talk about.

In general, I pretty much want to be anchored around the codebase. but sustainability is a human thing. so I want to talk about some of the stuff we've done to and aroudn the codebase to support sustainability.

by sustainability I mean something like: parsl keeps being usable for the things people want it to be used for. (and there's a tie between the code base and the humans)

one thing thats come out of this and Daniel Katz has done a bunch of work on is identifying types of humans, in the context of research software engineering more generally. I like to think that some of that has been influenced by some private conversations we've had over the year so I like to imagine some of my spirit is infused there. for example this talk at fosdem https://archive.fosdem.org/2025/schedule/event/fosdem-2025-6244-research-software-sustainability-and-rses/

types of programers there:

I'm going to steal a couple of things from his talk to put in here.

here's four classifications of developer - I'm not just interested in developers but in all humans, but this is fine as a base to think about things:

1. 1. Research programmer
– Main job: prototype ideas very quickly
– Focus on what the software can do, more than how users will use it
– Essential to bootstrap new research software project, develop & test initial ideas; might take
shortcuts that harm project's later sustainability (adding technical debt)
– Stage: Initial development & new features in later stages

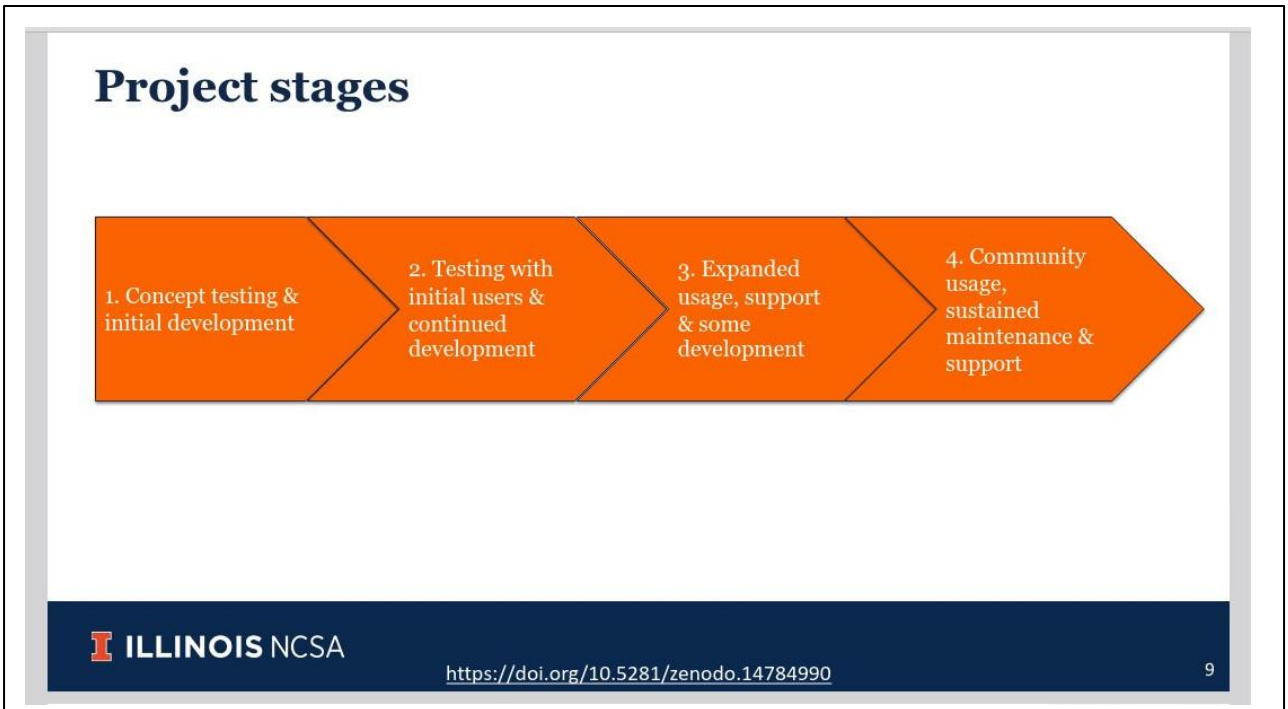
2. Software developer
– Main job: development professional-class research software
– Focus on software itself & its users
– Dedicated to making the software as useful as possible; making it clean & beautiful; increasing simplicity, compatibility, future maintainability; reducing technical debt
– Stage: Important to have involved in all but the initial stage of the project (where process
may impede fast innovation)

3. User/developer
– Main job: a scientist or disciplinary researcher who also adds features relevant to their work
– Focus on their own usage of the software
– Typically write code elsewhere for their real job (research); not dedicated to Parsl development, but a
power user of Parsl who can understand bugs and fix them
– May take shortcuts that harm project's future sustainability
– Stage: all but initial stage

4. Collaborating developer
– Main job: responsible for a collaborative project
– Focus on their software working with or being "part of" Parsl
– In their own project, can may be any other developer type; may follow different coding/software
engineering style, potentially leading to integration and culture challenges
– Defining the interface to these developers is a key challenge for sustainability, as ideally, they will become
committed to Parsl's success as important to their own project's
– Stage: most helpful after the project has become initially proven and has demonstrated some success
(stages 3 & 4)

pretty much everyone here either is interacting with the parsl codebase in something

like one of those roles - maybe actually as a developer, maybe in some other role. so this isn't a perfect categorization for this talk but its good enough for now.

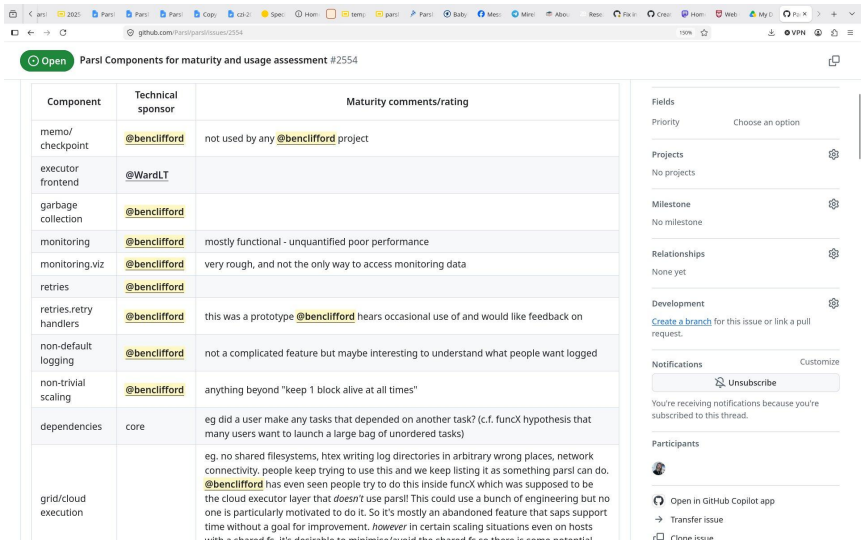


I'm also going to steal a slide from Dan's talk on project maturity, and repurpose it a bit. This slide is talking about the maturity journey of Parsl as a whole - going from a prototype to begin with to something with very different needs later on. But I want to repurpose it to think about if you divide parsl up, different pieces can have different levels of maturity, be at different stages on this journey, all inside the same codebase.

Dan presents this as a journey towards sustainability in a diagram which makes this look kinda inevitable, but my perspective is that it isn't mandatory for code to flow along this arrow - maybe something is a prototype and we just stop there.

whats important for me in distinguishing any of these roles or human types is that they all have different goals. some of which are aligned with moving along that sustainability chart, some not. for example, someone working towards a paper on some technique is unlikely to put 10x more effort into making code that is really production quality.

Maturity model



Open **Parsl Components for maturity and usage assessment** #2554

Component	Technical sponsor	Maturity comments/rating
memo/ checkpoint	@benclifford	not used by any @benclifford project
executor frontend	@WardLT	
garbage collection	@benclifford	
monitoring	@benclifford	mostly functional - unquantified poor performance
monitoring.viz	@benclifford	very rough, and not the only way to access monitoring data
retries	@benclifford	
retries.retry handlers	@benclifford	this was a prototype @benclifford hears occasional use of and would like feedback on
non-default logging	@benclifford	not a complicated feature but maybe interesting to understand what people want logged
non-trivial scaling	@benclifford	anything beyond "keep 1 block alive at all times"
dependencies	core	eg did a user make any tasks that depended on another task? (c.f. funcX hypothesis that many users want to launch a large bag of unordered tasks)
grid/cloud execution		eg. no shared filesystems, htex writing log directories in arbitrary wrong places, network connectivity. people keep trying to use this and we keep listing it as something parsl can do. @benclifford has even seen people try to do this inside funcX which was supposed to be the cloud executor layer that *doesn't* use parsl! This could use a bunch of engineering but no one is particularly motivated to do it. So it's mostly an abandoned feature that saps support time without a goal for improvement. *however* in certain scaling situations even on hosts

Fields
Priority Choose an option
Projects
No projects
Milestone
No milestone
Relationships
None yet
Development
Create a branch for this issue or link a pull request.
Notifications Customize
Unsubscribe
You're receiving notifications because you're subscribed to this thread.
Participants
Open in GitHub Copilot app
Transfer issue

So now I want to talk about some of the things we've done to the codebase, that mostly I've pushed on I think, really thinking about the kind of humans I've just talked about, and that maturity journey - I'll present a few topics, that isn't intended to be exhaustive.

* Maturity model

at the start, I (and others) wanted to have some of maturity model for components and think we were imagining a more documentation-oriented model there: label components with how much they're expected to work, who (if anyone) is first responder to bugs, imagine how far along that chart we genuinely believe the component is (and I want to be careful to distinguish that from wishful thinking of where we would like a component to be)

it didnt really play out that way - no one really wants to adopt components in some official sense like that, it turns out.

and for me this has played out more in a testing sense:

how much is our automated testing demonstrating that certain functionality really works in the codebase today. tested by machine. so a first approach response to any question can be "do I expect this to work in the sense that there is a test that shows it works in the latest release?"

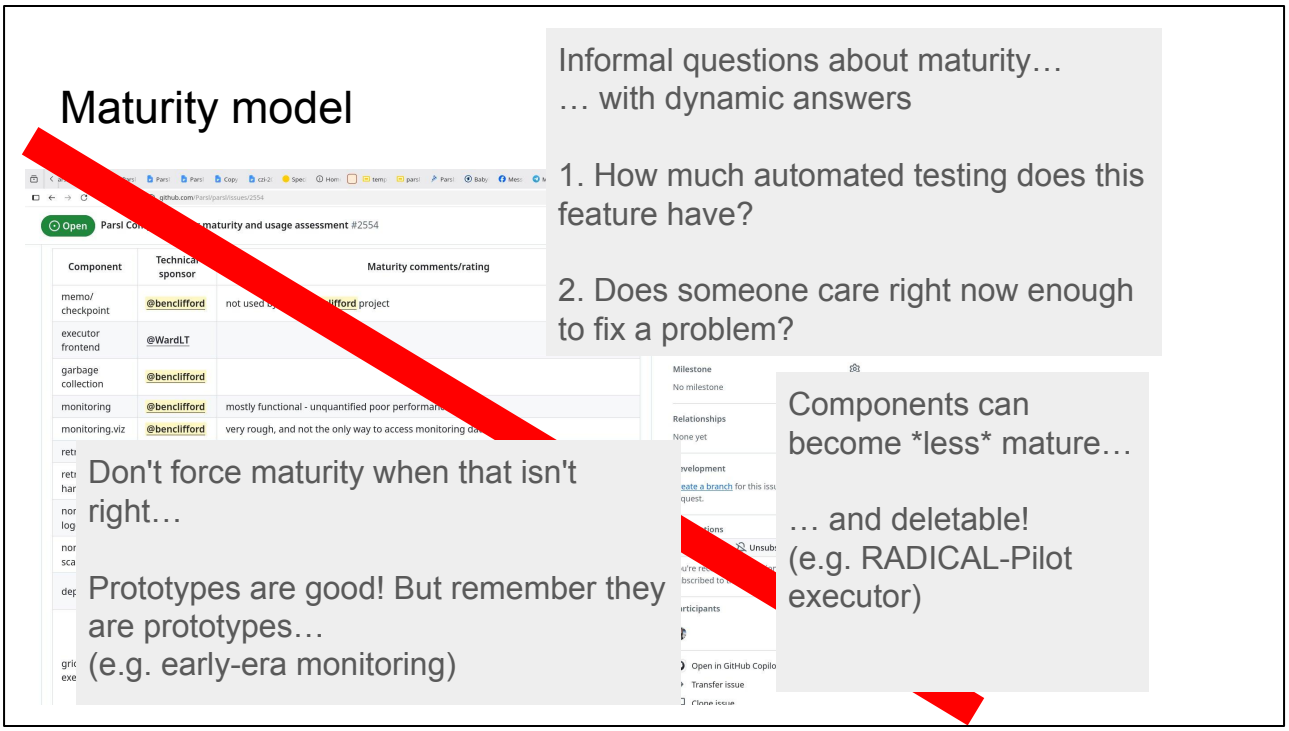
the other approach has been having more empathy for people wanting to work at different places on that code maturity diagram, trying to make them think about the place of their code on that diagram.

let everyone be clear if you are producing a prototype: it's a prototype. are you intending to support it for users for the next year, for example? often no.

but prototypes *are* worth doing and sharing - so don't be discouraged there. an example is Parsl monitoring, which was implemented as a summer project and turned out to be something lots of users wanted but weren't specifically saying. so that was a lot of frustration for me to get working properly. but it has gone on to influence the approach to observability in Academy which several people will talk about tomorrow.

features will look after themselves, work on the other stuff - i couldn't find the actual quote i wanted for this but its something like this.

stuff can go backwards in maturity, and be abandoned, and for some of that, it's the right thing to do to delete a feature entirely - for example, channels which were for remote execution. thats what globus compute is for, and globus compute has shown it is a serious amount of work we shouldn't be trying to do a sneaky lame reimplementation of.
natural attrition in the sense of "we interface to something else. that something else is gone now." for example radical pilot is end-of-life, and so the radical pilot executor kinda automatically becomes end of life there. the testing for it was disabled because of that and really the code should be deleted from the release, I think.



So now I want to talk about some of the things we've done to the codebase, that mostly I've pushed on I think, really thinking about the kind of humans I've just talked about, and that maturity journey - I'll present a few topics, that isn't intended to be exhaustive.

* Maturity model

at the start, I (and others) wanted to have some of maturity model for components and think we were imagining a more documentation-oriented model there: label components with how much they're expected to work, who (if anyone) is first responder to bugs, imagine how far along that chart we genuinely believe the component is (and I want to be careful to distinguish that from wishful thinking of where we would like a component to be)

it didnt really play out that way - no one really wants to adopt components in some official sense like that, it turns out.

and for me this has played out more in a testing sense:

how much is our automated testing demonstrating that certain functionality really works in the codebase today. tested by machine. so a first approach response to any question can be "do I expect this to work in the sense that there is a test that shows it works in the latest release?"

the other approach has been having more empathy for people wanting to work at different places on that code maturity diagram, trying to make them think about the place of their code on that diagram.

let everyone be clear if you are producing a prototype: it's a prototype. are you intending to support it for users for the next year, for example? often no.

but prototypes *are* worth doing and sharing - so don't be discouraged there. an example is Parsl monitoring, which was implemented as a summer project and turned out to be something lots of users wanted but weren't specifically saying. so that was a lot of frustration for me to get working properly. but it has gone on to influence the approach to observability in Academy which several people will talk about tomorrow.

features will look after themselves, work on the other stuff - i couldn't find the actual quote i wanted for this but its something like this.

stuff can go backwards in maturity, and be abandoned, and for some of that, it's the right thing to do to delete a feature entirely - for example, channels which were for remote execution. thats what globus compute is for, and globus compute has shown it is a serious amount of work we shouldn't be trying to do a sneaky lame reimplementation of.
natural attrition in the sense of "we interface to something else. that something else is gone now." for example radical pilot is end-of-life, and so the radical pilot executor kinda automatically becomes end of life there. the testing for it was disabled because of that and really the code should be deleted from the release, I think.

Hackability

Understandability

Parsl source code as textbook (for a human to read) on how to implement Parsl / how Parsl works

(vs I hacked out the prototype feature, it seems to work, move on to next feature

e.g. block scaling strategy)

Fragility

If I make a change, do Mysterious Unrelated Problems happen?

(e.g. Python fork-multiprocessing vs almost everything interesting: logs, C libraries, global state)

* Hackability

hackability: understandability and fragility - both as relates to humans

notion that code is *understandable* to a human e.g. like the textbook on how parsl works or on how to write a workflow system (vs "it works for me in my test cases and on one application i put a lot of effort into supporting")
that's distinct from the notion that code *works*
readability also can lead to maintainability and reliability: some parts of the codebase are hard to reason about because they're tangled. easier to reason about, easier to see problems.

fragility: eg. race conditions that "usually" dont' fire. but for example, at one time (now fixed) if you added a log line at the wrong part of startup, parsl would hang almost always. thats a strange/sad message to tell to users: "you can't add log lines in these somewhat mysterious places"
story of telling student basically half their project was going to always hit some misarchitecture of python process forking problems… so was in some sense "impossible" for them.

how to address fragility: more automated testing pushes against this by triggering sporadic bugs more (although you then need a human to care about test failures - in

some cases, what we've done is realised that the tested feature is less mature than we expected and downgraded its implicit maturity by disabling test) - who has responsibility to fix a test that is showing occasional failures in the tested feature?

Releases and Testing

c. 2022 support cycle - no release for 13 months

Q. I have a problem

A. Please try again with the latest `master` branch

Our (implict) policy is you should be using the latest master, not a 13 month old release…

Make this happen for real!

Culture: master should always be releasable

My work is done … but I need to fix something before release

My work is not done.

If we believe this… release without further testing, weekly in **cronjob**

* Releases and testing

Who does releases? Most people don't want to perform releases. There are no "permanent" or "pure" parsl devs to do this. History: we ended up so far behind in releases (13 months) that the default response to almost any bug report was "please try master from github". if that's our answer, then lets get users actually installing the latest from master by default.
lets keep master in a state that it can always be released. and if it is in such a state, we can release it automatically. and that helps nudge people out of "later" thinking.
testing is on PR, not on release: if you broke something, I want it found before you wander off. and its your responsibility to fix it. and to fix it now, not some abstract "later, before release".
testing gives a better machine-checkable description to everyone of "do we basically expect this code to work?". and lack of automated testing lets us (me) rapidly answer support questions with "its not tested, i have no reason to believe it works".

broadening testing:
unit tests
integration tests
pushed on type annotations and type checking - when I started on this, it was a bit of a fringe notion in the python world. It has subsequently become mainstream. a nice complement to unit and integration tests.

contributions from "people who care" about particular components - for example, NERSC uses SLURM and implemented a container based SLURM provider test that runs in our CI, so now we have more confidence in that provider compared to many others.

release versioning - lets be clearer about what compatiblity we expect. which is that all your versions should be consistent. that's caused some grumbles in globus compute - but the point is to be clearer here about what parsl core should do, so it is more apparent that globus compute needs to implement "more compatibility" as a feature in itself - which is what chris will talk about in this session.

Releases and Testing

Culture: master should always be releasable

If we believe this… release without further testing, weekly in **cronjob**

All testing should be automated testing

One off testing invalidated 1 commit later

Sustainability effort => build more automated testing

Clear value in automated testing contributions
(e.g. NERSC SLURM test)

* Releases and testing

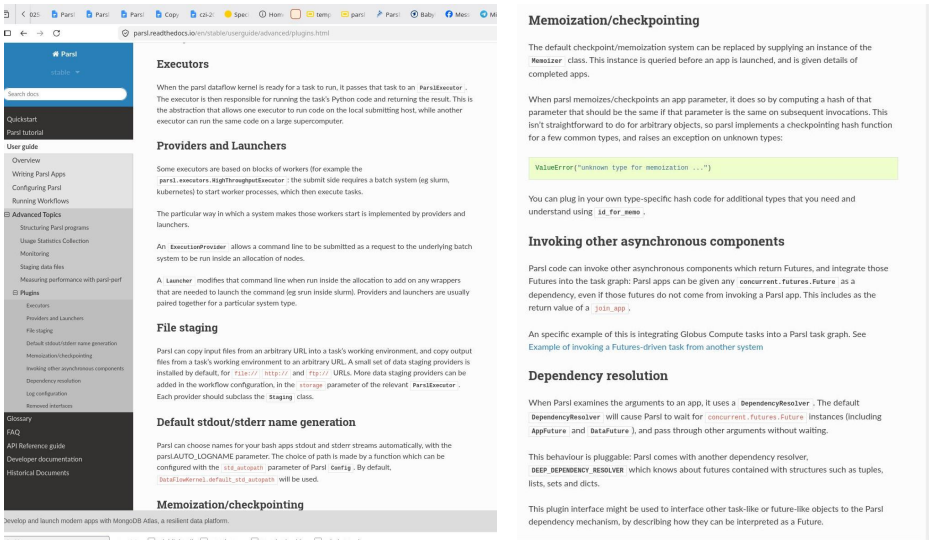
Who does releases? Most people don't want to perform releases. There are no "permanent" or "pure" parsl devs to do this. History: we ended up so far behind in releases (13 months) that the default response to almost any bug report was "please try master from github". if that's our answer, then lets get users actually installing the latest from master by default.
lets keep master in a state that it can always be released. and if it is in such a state, we can release it automatically. and that helps nudge people out of "later" thinking.
testing is on PR, not on release: if you broke something, I want it found before you wander off. and its your responsibility to fix it. and to fix it now, not some abstract "later, before release".
testing gives a better machine-checkable description to everyone of "do we basically expect this code to work?". and lack of automated testing lets us (me) rapidly answer support questions with "its not tested, i have no reason to believe it works".

broadening testing:
unit tests
integration tests
pushed on type annotations and type checking - when I started on this, it was a bit of a fringe notion in the python world. It has subsequently become mainstream. a nice complement to unit and integration tests.

contributions from "people who care" about particular components - for example, NERSC uses SLURM and implemented a container based SLURM provider test that runs in our CI, so now we have more confidence in that provider compared to many others.

release versioning - lets be clearer about what compatiblity we expect. which is that all your versions should be consistent. that's caused some grumbles in globus compute - but the point is to be clearer here about what parsl core should do, so it is more apparent that globus compute needs to implement "more compatibility" as a feature in itself - which is what chris will talk about in this session.

Plugins and Modularity



Executors

When the parsl dataflow kernel is ready for a task to run, it passes that task to an `ParslExecutor`. The executor is then responsible for running the task's Python code and returning the result. This is the abstraction that allows one executor to run code on the local submitting host, while another executor can run the same code on a large supercomputer.

Providers and Launchers

Some executors are based on blocks of workers (for example the `parsl.executors.HighThroughputExecutor`): the submit side requires a batch system (eg slurm, kubernetes) to start worker processes, which then execute tasks.

The particular way in which a system makes those workers start is implemented by providers and launchers.

An `ExecutionProvider` allows a command line to be submitted as a request to the underlying batch system to be run inside an allocation of nodes.

A `Launcher` modifies that command line when run inside the allocation to add on any wrappers that are needed to launch the command (eg srun inside slurm). Providers and launchers are usually paired together for a particular system type.

File staging

Parsl can copy input files from an arbitrary URL into a task's working environment, and copy output files from a task's working environment to an arbitrary URL. A small set of data staging providers is installed by default, for `file://`, `http://` and `ftp://` URLs. More data staging providers can be added in the workflow configuration, in the `storage` parameter of the relevant `ParslExecutor`. Each provider should subclass the `Staging` class.

Default stdout/stderr name generation

Parsl can choose names for your bash apps stdout and stderr streams automatically, with the parsl.AUTO_LOGNAME parameter. The choice of path is made by a function which can be configured with the `std_autopath` parameter of Parsl `Config`. By default, `DataFlowKernel.default_std_autopath` will be used.

Memoization/checkpointing

The default checkpoint/memoization system can be replaced by supplying an instance of the `Memoizer` class. This instance is queried before an app is launched, and is given details of completed apps.

When parsl memoizes/checkpoints an app parameter, it does so by computing a hash of that parameter that should be the same if that parameter is the same on subsequent invocations. This isn't straightforward to do for arbitrary objects, so parsl implements a checkpointing hash function for a few common types, and raises an exception on unknown types:

```
ValueError("unknown type for memoization ...")
```

You can plug in your own type-specific hash code for additional types that you need and understand using `id_for_memo`.

Invoking other asynchronous components

Parsl code can invoke other asynchronous components which return Futures, and integrate those Futures into the task graph: Parsl apps can be given any `concurrent.futures.Future` as a dependency, even if those futures do not come from invoking a Parsl app. This includes as the return value of a `join_app`.

An specific example of this is integrating Globus Compute tasks into a Parsl task graph. See Example of invoking a Futures-driven task from another system

Dependency resolution

When Parsl examines the arguments to an app, it uses a `DependencyResolver`. The default `DependencyResolver` will cause Parsl to wait for `concurrent.futures.Future` instances (including `AppFuture` and `DataFuture`), and pass through other arguments without waiting.

This behaviour is pluggable: Parsl comes with another dependency resolver, `DEEP_DEPENDENCY_RESOLVER` which knows about futures contained with structures such as tuples, lists, sets and dicts.

This plugin interface might be used to interface other task-like or future-like objects to the Parsl dependency mechanism, by describing how they can be interpreted as a Future.

* Plugins and modularity

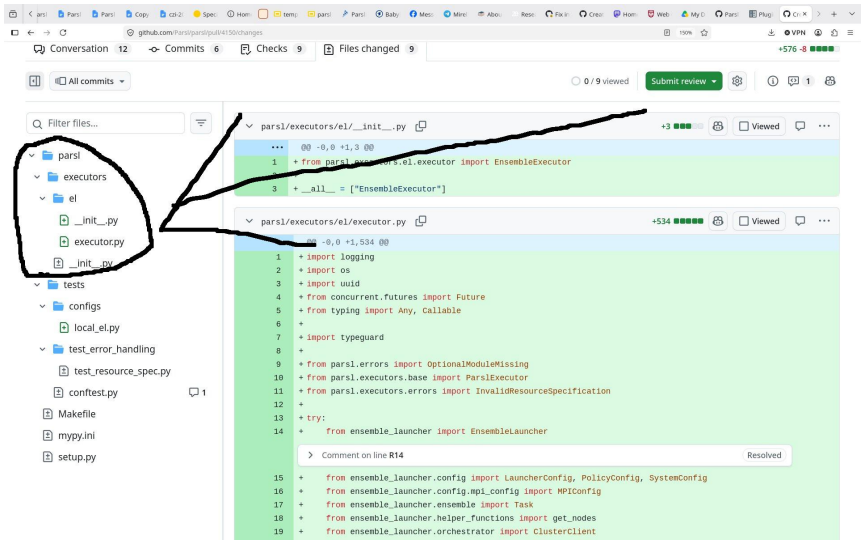
globus compute wants to use only some components, it never initialises parsl. so to support that kind of use case, the code needs to be modular enough that you can initialise the pieces separately.

someone wants to implement an executor. that's relatively common - ~10 executors over lifetime of parsl. previous talk was on adding an executor - if you look at the pull request, it's all nicely in one directory. when ALCF decides to no longer support it, its easy to remove. which makes it easier to add in the first place: it's got a nice isolated bubble of maturity-level, so its easy to accept into the codebase now even though it isn't even tested in CI, at that expectation/maturity level, without that immaturity affecting the more stable parts of parsl.

looking at these interfaces as "plugin interfaces" is a different perspective on what I regard as pretty much the same thing - clean interfaces between components. these interfaces are often a good place to think about differences in maturity.

why should the piece of code that deals with task dependencies have anything to do with batch job submission? (as an example tidyup)

Plugins and Modularity



* Plugins and modularity

globus compute wants to use only some components, it never initialises parsl. so to support that kind of use case, the code needs to be modular enough that you can initialise the pieces separately.

someone wants to implement an executor. that's relatively common - ~10 executors over lifetime of parsl. previous talk was on adding an executor - if you look at the pull request, it's all nicely in one directory. when ALCF decides to no longer support it, its easy to remove. which makes it easier to add in the first place: it's got a nice isolated bubble of maturity-level, so its easy to accept into the codebase now even though it isn't even tested in CI, at that expectation/maturity level, without that immaturity affecting the more stable parts of parsl.

looking at these interfaces as "plugin interfaces" is a different perspective on what I regard as pretty much the same thing - clean interfaces between components. these interfaces are often a good place to think about differences in maturity.

why should the piece of code that deals with task dependencies have anything to do with batch job submission? (as an example tidyup)

Contributions "welcome(*)"

"Big C" Contributions

Over the fence

PR is the *end* of the work, not the *start* of the work (Hollywood wedding)

Very exciting at the prototype stage
Less exciting for mature mode code

"Good first issues"
are mostly fixed after all this time

Student internships

Cost of mentorship is massive

Taught/learned skills evaporate from the project after 10 weeks - teaching skills is valuable to Society but not to Parsl

End of project - so we need to be clear about maturity

So what is the right kind of sustainability contribution?
Probably around code you're going to keep using?
e.g. NERSC => Parsl SLURM tests

* Contributions

we can all dream of a thousand people showing up to make glorious high quality commits to the code base.

that's not the reality.

callback to the types of developer - with their own types of contribution to the codebase.

Big C Contributions:

"Contribution" with a capital-C very much can have a "throw this over the wall" or "add trash to the dumpster fire" model if we don't push back on that.

over-the-fence contributions

student skill - costs of mentorship that is evaporated at end of project - this is a poor way to get code for Parsl (separately, it is a good way to give people experience, but that's not part of sustainability work) - its not the job of the (non-existent) Parsl core devs to train you to be a junior programmer for free, or to bring you up to speed on the Parsl codebase for free.

But conversely I'd had student say that me being clear something is a prototype has helped their guilt for not being in a position to produce production quality code.
and if the end goal is to demonstrate a point, maybe it doens't need to go into the main parsl codebase at all - write your paper, or your report, or your poster. don't try to encumber the few parsl maintainers with your demo code. the paper, report, poster is a perfectly legitimate output by itself. that's an alignment issue.

for Big contributions:
are you expecting to support your contribution for the next year?
is this the *end* of your project, or the *start* of your project? (hollywood wedding style)

- fin -

Tension and disappointments from not thinking about:

* what other people are aiming for

* how mature different pieces of the code are

Think about these things

Make better Parsl

Make more human happiness

a lot of tension and disappointments and wrong expectations have arisen (in my opinion) from not thinking about these two different axes of whats going on - the different goals of different people involved, and the different maturity of code.

so by thinking about this more actively, we can have better Parsl and more human happiness.