

---

# TotalPerspectiveVortex Documentation

*Release 2.4.0*

**Galaxy and GVL projects**

**Apr 29, 2024**



**CONTENTS:**

<b>1</b>	<b>Shared database</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	TPV by example . . . . .	5
1.3	Advanced Topics . . . . .	14
1.4	Concepts and Organisation . . . . .	17
1.5	Inner workings . . . . .	20
1.6	Shell Commands . . . . .	22
1.7	Migration Guide . . . . .	24
1.8	FAQ . . . . .	25
1.9	Indices and tables . . . . .	25





# Total Perspective Vortex

## Dynamic rules for routing Galaxy entities to destinations

TotalPerspectiveVortex (TPV) is a plugin for the [Galaxy application](#) that can route entities (Tools, Users, Roles) to appropriate destinations with appropriate resource allocations (cores, gpus, memory), based on a configurable yaml file. For example, it could allocate 8 cores and 32GB of RAM to a bwa-mem job, and route it to a Slurm cluster, while allocating 2 cores and 4GB of RAM to an upload job, and route it to a local runner. These rules can also be shared community-wide, imported at runtime by any Galaxy deployment, and overridden locally when necessary.

TPV can be plugged into Galaxy via `job_conf.yaml`. TPVs configuration file specifies how entities (tools, users, roles) should be allocated resources (cores, gpus, memory) and in complex environments with multiple job destinations, where to map the resulting jobs to (through a flexible tagging system). Destinations can have arbitrary scheduling tags defined, and each entity can express a preference or aversion to specific scheduling tags. This tagging affects how jobs are routed to destinations. In addition, admins can also plugin arbitrary python based rules for making more complex decisions, as well as custom ranking functions for choosing between matching destinations.



## SHARED DATABASE

A shared database of TPV rules are maintained in: <https://github.com/galaxyproject/tpv-shared-database/> These rules are based on typical settings used in the usegalaxy.\* federation, which you can override based on local resource availability.

### 1.1 Installation

#### 1.1.1 Basic steps

1. `pip install total-perspective-vortex` into Galaxy's python virtual environment
2. Create the TPV job mapping yaml file, specifying resource allocation and job routing preferences
3. Configure Galaxy's `job_conf.yml` to use TPV
4. Submit jobs as usual

#### Configuring Galaxy

1. First install TPV into your Galaxy virtual environment.

TPV is a conditional dependency of Galaxy since Galaxy 22.05. If TPV is enabled in your Galaxy job configuration, it will automatically be installed into Galaxy's virtualenv. Otherwise, or if you wish to upgrade to a newer version of TPV, you can use the process below to install manually:

```
cd <galaxy_home>
source .venv/bin/activate
pip install --upgrade total-perspective-vortex
```

2. Edit your `job_conf.yml` in the `<galaxy_home>/config` folder and add the highlighted sections to it.

You can refer to a local file for the `tpv_config_files` setting, or alternatively, provide a link to a remote url.

```
1 runners:
2   local:
3     load: galaxy.jobs.runners.local:LocalJobRunner
4     workers: 4
5   drmaa:
6     load: galaxy.jobs.runners.drmaa:DRMAAJobRunner
7   k8s:
8     load: galaxy.jobs.runners.kubernetes:KubernetesJobRunner
```

(continues on next page)

(continued from previous page)

```
9
10 handling:
11   assign:
12     - db-skip-locked
13
14 execution:
15   default: tpv_dispatcher
16   environments:
17     tpv_dispatcher:
18       runner: dynamic
19       type: python
20       function: map_tool_to_destination
21       rules_module: tpv.rules
22       tpv_config_files:
23         - https://gxy.io/tpv/db.yml
24         - config/tpv_rules_local.yml
25   local:
26     runner: local
27   k8s_environment:
28     runner: k8s
29   docker_enabled: true
```

3. Add your own custom rules to your local `tpv_config_file`, following instructions in the next section.

## 1.1.2 Combining multiple remote and local configs

TPV allows rules to be loaded from remote or local sources.

```
1 tpv_dispatcher:
2   runner: dynamic
3   type: python
4   function: map_tool_to_destination
5   rules_module: tpv.rules
6   tpv_config_files:
7     - https://gxy.io/tpv/db.yml
8     - config/tpv_rules_australia.yml
```

The config files listed first are overridden by config files listed later. The normal rules of inheritance apply. This allows a central database of common rules to be maintained, with individual, site-specific overrides.

## 1.1.3 Standalone Installation

If you wish to install TPV outside of Galaxy's virtualenv (e.g. to use the `tpv lint` command locally or in a CI/CD pipeline), use the `[cli]` pip requirement specifier to make sure the necessary Galaxy dependency packages are also installed. **This should not be used in the Galaxy virtualenv:**

```
$ pip install 'total-perspective-vortex[cli]'
```



## 1.2 TPV by example

### 1.2.1 Simple configuration

The simplest possible example of a useful TPV config might look like the following:

```

1 tools:
2   toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
3     cores: 12
4     mem: cores * 4
5     gpus: 1
6
7 destinations:
8   slurm:
9     runner: slurm
10    max_accepted_cores: 16
11    max_accepted_mem: 64
12    max_accepted_gpus: 2
13  general_pulsar_1:
14    runner: pulsar_1
15    max_accepted_cores: 8
16    max_accepted_mem: 32
17    max_accepted_gpus: 1

```

Here, we define one tool and its resource requirements, the destinations available, and the total resources available at each destination (optional). The tools are matched by tool id, and can be a regular expression. Note how resource requirements can also be computed as python expressions. If resource requirements are defined at the destination, TPV will check whether the job will fit. For example, hisat2 will not schedule on *general\_pulsar\_1* as it has insufficient cores. If resource requirements are omitted in the tool or destination, it is considered a match. Note that TPV only considers destinations defined in its own config file, and ignores destinations in job\_conf.yml.

### 1.2.2 Default inheritance

Inheritance provides a mechanism for an entity to inherit properties from another entity, reducing repetition.

```

1 global:
2   default_inherits: default
3
4 tools:
5   default:
6     cores: 2
7     mem: 4
8   params:
9     nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
10    ↪ {mem*1024}"
11   toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/2.1.0+galaxy7:
12     cores: 12
13     mem: cores * 4
14     gpus: 1

```

The *global* section is used to define global TPV properties. The *default\_inherits* property defines a “base class” for all tools to inherit from.

In this example, if the *bwa* tool is executed, it will match the *default* tool, as there are no other matches, thus inheriting its resource requirements. The *hisat2* tool will also inherit these defaults, but is explicitly overriding *cores*, *mem* and *gpus*. It will inherit the *nativeSpecification* param.

### 1.2.3 Explicit inheritance

Explicit inheritance provides a mechanism for exerting greater control over the inheritance chain.

```

1 global:
2   default_inherits: default
3
4 tools:
5   default:
6     cores: 2
7     mem: 4
8     params:
9       nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
    ↪ {mem*1024}"
10  toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
11    cores: 12
12    mem: cores * 4
13    gpus: 1
14  .*minimap2.*:
15    inherits: toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
16    cores: 8
17    gpus: 0

```

In this example, the *minimap2* tool explicitly inherits requirements from the *hisat2* tool, which in turn inherits the default tool. There is no limit to how deep the inheritance hierarchy can be.

### 1.2.4 Scheduling tags

Scheduling tags provide a means by which to control how entities match up, and can be used to route jobs to preferred destinations, or to explicitly control which users can execute which tools, and where.

```

1 tools:
2   default:
3     cores: 2
4     mem: 4
5     params:
6       nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
    ↪ {mem*1024}"
7   scheduling:
8     reject:
9       - offline
10  toolshed.g2.bx.psu.edu/repos/iuc/hisat2/.*:
11    cores: 4
12    mem: cores * 4
13    gpus: 1
14    scheduling:
15      require:
16      prefer:

```

(continues on next page)

(continued from previous page)

```

17     - highmem
18     accept:
19     reject:
20     toolshed.g2.bx.psu.edu/repos/iuc/minimap2/.*:
21     cores: 4
22     mem: cores * 4
23     gpus: 1
24     scheduling:
25     require:
26     - highmem
27
28 destinations:
29     slurm:
30     runner: slurm
31     max_accepted_cores: 16
32     max_accepted_mem: 64
33     max_accepted_gpus: 2
34     scheduling:
35     prefer:
36     - general
37
38     general_pulsar_1:
39     runner: pulsar_1
40     max_accepted_cores: 8
41     max_accepted_mem: 32
42     max_accepted_gpus: 1
43     scheduling:
44     prefer:
45     - highmem
46     reject:
47     - offline

```

In this example, all tools reject destinations marked as offline. The `hisat2` tool expresses a preference for `highmem`, and inherits the rejection of offline tags. Inheritance can be used to override scheduling tags. For example, the `minimap2` tool inherits `hisat2`, but now requires a `highmem` tag, instead of merely preferring it.

The destinations themselves can be tagged in similar ways. In this case, the `general_pulsar_1` destination also prefers the `highmem` tag, and thus, the `hisat2` tool would schedule there. However, `general_pulsar_1` also rejects the `offline` tag, and therefore, the `hisat2` tool cannot schedule there. Therefore, it schedules on the only available destination, which is `slurm`.

The `minimap2` tool meanwhile requires `highmem`, but rejects `offline` tags, which leaves it nowhere to schedule. This results in a `JobMappingException` being thrown.

A full table of how scheduling tags match up can be found in the Scheduling section.

These TPV defined scheduling tags should be contrasted with Galaxy's destination level handler tags: [https://github.com/galaxyproject/galaxy/blob/0a0d68b7feed5e303ed762f6586ea9757219c6f7/lib/galaxy/config/sample/job\\_conf.sample.yml#L1037](https://github.com/galaxyproject/galaxy/blob/0a0d68b7feed5e303ed762f6586ea9757219c6f7/lib/galaxy/config/sample/job_conf.sample.yml#L1037) Galaxy handler tags can be defined as simply *tags* at the destination.

## 1.2.5 Rules

Rules provide a means by which to conditionally change entity requirements.

```

1 tools:
2   default:
3     cores: 2
4     mem: cores * 3
5   rules:
6     - id: my_overridable_rule
7       if: input_size < 5
8       fail: We don't run piddling datasets of {input_size}GB
9   bwa:
10    scheduling:
11      require:
12        - pulsar
13    rules:
14      - id: my_overridable_rule
15        if: input_size < 1
16        fail: We don't run piddling datasets
17      - if: input_size <= 10
18        cores: 4
19        mem: cores * 4
20        execute: |
21          from galaxy.jobs.mapper import JobNotReadyException
22          raise JobNotReadyException()
23      - if: input_size > 10 and input_size < 20
24        scheduling:
25          require:
26            - highmem
27      - if: input_size >= 20
28        fail: Input size: {input_size} is too large shouldn't run

```

The if clause can contain arbitrary python code, including multi-line python code. The only requirement is that the last statement in the code block must evaluate to a boolean value. In this example, the *input\_size* variable is an automatically available contextual variable which is computed by totalling the sizes of all inputs to the job. Additional available variables include *app*, *job*, *tool*, and *user*.

If the rule matches, the properties of the rule override the properties of the tool. For example, if the *input\_size* is 15, the *bwa* tool will require both *pulsar* and *highmem* tags.

Rules can be overridden by giving them an id. For example, the default for all tools is to reject input sizes < 5 by using the *my\_overridable\_rule* rule. We override that for the *bwa* tool by specifically referring to the inherited rule by id. If no id is specified, an id is auto-generated and no longer overridable.

Note the use of the *{input\_size}* variable in the fail message. The general rule is that all non-string expressions are evaluated as python code blocks, while string variables are evaluated as python f-strings.

The execute block can be used to create arbitrary side-effects if a rule matches. The return value of an execute block is ignored.

## 1.2.6 User and Role Handling

Scheduling rules can also be expressed for users and roles.

```

1 tools:
2   default:
3     scheduling:
4       require: []
5       prefer:
6         - general
7     accept:
8     reject:
9       - pulsar
10    rules: []
11  dangerous_interactive_tool:
12    cores: 8
13    mem: 8
14    scheduling:
15      require:
16        - authorize_dangerous_tool
17  users:
18    default:
19      scheduling:
20        reject:
21          - authorize_dangerous_tool
22    fairycake@vortex.org:
23      cores: 4
24      mem: 16
25      scheduling:
26        accept:
27          - authorize_dangerous_tool
28        prefer:
29          - highmem
30
31  roles:
32    training.*:
33      cores: 5
34      mem: 7
35      scheduling:
36        reject:
37          - pulsar

```

In this example, if user *fairycake@vortex.org* attempts to dispatch a *dangerous\_interactive\_tool* job, the requirements for both entities would be combined. Most requirements would simply be merged, such as env vars and job params. However, when combining gpus, cores and mem, the lower of the two values are used. In this case, the combined entity would have a core value of 4 and a mem value of 8. This allows training users for example, to be forced to use a lower number of cores than usual.

In addition, for these entities to be combined, the scheduling tags must also be compatible. In this instance the *dangerous\_interactive\_tool* requires the *authorize\_dangerous\_tool* tag, which all users by default reject. Therefore, most users cannot run this tool by default. However, *fairycake@vortex.org* overrides that and accepts the *authorize\_dangerous\_tool* allowing only that user to run the dangerous tool.

Roles can be matched in this exact way. Rules can also be defined at the user and role level.

## 1.2.7 Metascheduling

Custom rank functions can be used to implement metascheduling capabilities. A rank function is used to select the best matching destination from a list of matching destinations. If no rank function is provided, the default rank function simply chooses the most preferred destination out of the available destinations.

When more sophisticated control over scheduling is required, a rank function can be implemented through custom python code.

```

1 tools:
2   default:
3     cores: 2
4     mem: 8
5     rank: |
6       import requests
7
8       params = {
9         'pretty': 'true',
10        'db': 'pulsar-test',
11        'q': 'SELECT last("percent_allocated") from "sinfo" group by "host"'
12      }
13
14      try:
15        response = requests.get('http://stats.genome.edu.au:8086/query', params=params)
16        data = response.json()
17        cpu_by_destination = {s['tags']['host']:s['values'][0][1] for s in data.get(
18 ↪ 'results')[0].get('series', [])}
19        # sort by destination preference, and then by cpu usage
20        candidate_destinations.sort(key=lambda d: (-1 * d.score(entity), cpu_by_
21 ↪ destination.get(d.dest_name)))
22        final_destinations = candidate_destinations
23      except Exception:
24        log.exception("An error occurred while querying influxdb. Using a weighted random_
25 ↪ candidate destination")
26        final_destinations = helpers.weighted_random_sampling(candidate_destinations)
27      final_destinations

```

In this example, the rank function queries a remote influx database to find the least loaded destination. The matching destinations are available to the rank function through the *candidate\_destinations* contextual variable. Therefore, in this example, the candidate destinations are first sorted by the best matching destination (score is the default ranking function), and then sorted by CPU usage per destination, obtained from the influxdb query.

Note that the final statement in the rank function must be the list of sorted destinations.

## 1.2.8 Custom contexts

In addition to the automatically provided context variables (see *Concepts and Organisation*), TPV allows you to define arbitrary custom variables, which are then available whenever an expression is evaluated. Contexts can be defined both globally or at the level of each entity, with entity level context variables overriding global ones.

```

1 global:
2   default_inherits: default
3   context:
4     ABSOLUTE_FILE_SIZE_LIMIT: 100

```

(continues on next page)

(continued from previous page)

```

5     large_file_size: 10
6     _a_protected_var: "some value"
7
8 tools:
9     default:
10        context:
11            additional_spec: --my-custom-param
12        cores: 2
13        mem: 4
14        params:
15            nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
16    ↪ {mem*1024} {additional_spec}"
17        rules:
18            - if: input_size >= ABSOLUTE_FILE_SIZE_LIMIT
19              fail: Job input: {input_size} exceeds absolute limit of: {ABSOLUTE_FILE_SIZE_
20    ↪ LIMIT}
21            - if: input_size > large_file_size
22              cores: 10
23
24 toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/2.1.0+galaxy7:
25     context:
26         large_file_size: 20
27         additional_spec: --overridden-param
28     mem: cores * 4
29     gpus: 1

```

In this example, three global context variables are defined, which are made available to all entities. Variable names follow Python conventions, where all uppercase variables indicate constants that cannot be overridden. Lower case indicates a public variable that can be overridden and changed, even across multiple TPV config files. An underscore indicates a protected variable that can be overridden within the same file, but not across files.

Additionally, the tool defaults section defines a context variable named *additional\_spec*, which is only available to inheriting tools.

If we were to dispatch a job, say bwa, with an input\_size of 15, the large file rule in the defaults section would kick in, and the number of cores would be set to 10. If we were to dispatch a hisat2 job with the same input size however, the large\_file\_size rule would not kick in, as it has been overridden to 20. The main takeaway from this example is that variables are bound late, and therefore, rules and params can be crafted to allow inheriting tools to conveniently override values, even across files. While this capability can be powerful, it needs to be treated with the same care as any global variable in a programming language.

### 1.2.9 Multiple matches

If multiple regular expressions match, the matches are applied in order of appearance. Therefore, the convention is to specify more general rule matches first, and more specific matches later. This matching also applies across multiple TPV config files, again based on order of appearance.

```

1 tools:
2     default:
3         cores: 2
4         mem: 4
5         params:

```

(continues on next page)

(continued from previous page)

```

6     nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
   ↪ {mem*1024}"
7
8     toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/.*:
9         mem: cores * 4
10        gpus: 1
11
12    toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/2.1.0+galaxy7:
13        env:
14            MY_ADDITIONAL_FLAG: "test"

```

In this example, dispatching a hisat2 job would result in a mem value of 8, with 1 gpu. However, dispatching the specific version of 2.1.0+galaxy7 would result in the additional env variable, with mem remaining at 8.

### 1.2.10 Job Environment

As seen in the previous example, it is possible to specify environment variables that will be set in the job's executing environment. It is also possible to source environment files and execute commands, using the same syntax as in Galaxy's job\_conf.yml, by specifying env as a list instead of a dictionary.

```

1  tools:
2    default:
3      cores: 2
4      mem: 4
5      params:
6      nativeSpecification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
   ↪ {mem*1024}"
7      env:
8        - execute: echo "Don't Panic!"
9
10     toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/.*:
11         mem: cores * 4
12         gpus: 1
13         env:
14             - name: MY_ADDITIONAL_FLAG
15               value: "arthur"
16             - file: /galaxy/tools/hisat2.env
17
18     toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/2.1.0+galaxy7:
19         inherits: toolshed.g2.bx.psu.edu/repos/iuc/hisat2/hisat2/.*:
20         env:
21             MY_ADDITIONAL_FLAG: "zaphod"

```

In this example, all jobs will execute the command echo "Don't Panic!". All versions of hisat2 will have \$MY\_ADDITIONAL\_FLAG set and will source the file /galaxy/tools/hisat2.env, but version 2.1.0+galaxy7 will have the value zaphod set for \$MY\_ADDITIONAL\_FLAG instead of the hisat2 default of arthur.



### 1.2.11 Job Resubmission

TPV has explicit support for job resubmissions, so that advanced control over job resubmission is possible.

```

1 tools:
2   default:
3     cores: 2
4     mem: 4 * int(job.destination_params.get('SCALING_FACTOR', 1)) if job.destination_
↪params else 1
5     params:
6       SCALING_FACTOR: "{2 * int(job.destination_params.get('SCALING_FACTOR', 2)) if job.
↪destination_params else 2}"
7     resubmit:
8       with_more_mem_on_failure:
9         condition: memory_limit_reached and attempt <= 3
10        destination: tpv_dispatcher

```

In this example, we have defined a resubmission handler that resubmits the job if the memory limited is reached. Note that the resubmit section looks exactly the same as Galaxy's, except that it follows a dictionary structure instead of being a list. Refer to the Galaxy job configuration docs for more information on resubmit handlers. One twist in this example is that we automatically increase the amount of memory provided to the job on each resubmission. This is done by setting the SCALING\_FACTOR param, which is a custom parameter which we have chosen for this example, that we increase on each resubmission. Since each resubmission's destination is TPV, the param is re-evaluated on each resubmission, and scaled accordingly. The memory is allocated based on the scaling factor, which therefore, also scales accordingly.

### 1.2.12 Using the shared database

A shared database of resource requirements and rules are maintained in:

<https://github.com/galaxyproject/tpv-shared-database/>

This shared database relieves you of the burden of figuring out what resources are typically required by tools, with recommended settings based on those used in the usegalaxy.\* federation. You can override these settings based on local resource availability. The shared database can be integrated through your local job\_conf.yml as follows:

```

1 tpv_dispatcher:
2   runner: dynamic
3   type: python
4   function: map_tool_to_destination
5   rules_module: tpv.rules
6   tpv_config_files:
7     - https://gxy.io/tpv/db.yml
8     - config/my_local_overrides.yml # optional

```

### 1.2.13 Clamping resources

Entities can define, `min_{cores|gpus|mem}` and `max_{cores|gpu|mem}` as a means of clamping the maximum resources that will be allocated to a tool, even if it requests a higher amount. For example, if a tool requests 16 cores, but a user is defined with `max_cores: 4`, then the tool's resource requirement would be clamped down to that maximum amount. This can be useful for allocating lower resources to training users for example, who only use toy datasets that do not require the full core allocation. Conversely, some users can be allocated more resources by using `min_cores`.

In addition, clamping resources can also be useful when using the TPV shared database. For example, the `canu` tool has a 96GB recommended memory requirement, which your local cluster may not have. However, you may still want to allow the tool to run, albeit with lower resources. You can of course, locally override the `canu` tool and allocated less resources, but this can be tedious to do for a large number of tools. All you may really want, is to restrict all tools to use the maximum your cluster can support. You can achieve that effect as follows:

```
1 destinations:
2   slurm:
3     runner: slurm
4     max_accepted_cores: 32
5     max_accepted_mem: 196
6     max_accepted_gpus: 2
7     max_cores: 16
8     max_mem: 64
9     max_gpus: 1
```

In the example above, we mark the slurm destination as accepting jobs up to 196GB in size, and therefore, the `canu` tool, which required 96GB, would successfully schedule there. However, we forcibly clamp the job's `max_mem` to 64GB, which is the actual memory your cluster can support. In this way, all tools in the shared database can still run, provided they do not exceed the specified `max_accepted` values.

### 1.2.14 Giving a parameterized, custom name to a destination

If you need to provide a parameterized name for a destination, you can do so by using the `destination_name_override` property.

```
1 destinations:
2   slurm:
3     runner: slurm
4     destination_name_override: "my-dest-with-{cores}-cores-{mem}-mem"
```

## 1.3 Advanced Topics

### 1.3.1 Expressions

Most TPV properties can be expressed as Python expressions. The rule of thumb is that all string expressions are evaluated as python f-strings, and all integers or boolean expressions are evaluated as python code blocks. For example, `cpu`, `cores` and `mem` are evaluated as python code blocks, as they evaluate to integer/float values. However, `env` and `params` are evaluated as f-strings, as they result in string values. This is to improve the readability and syntactic simplicity of TPV config files.

At the point of evaluating these functions, there is an evaluation context, which is a default set of variables that are available to that expression. The following default variables are available to all expressions:

## Default evaluation context

Variable	Description
app	the Galaxy App object
tool	the Galaxy tool object
user	the current Galaxy user object
job	the Galaxy job object
mapper	the TPV mapper object, which can be used to access parsed TPV configs
entity	the TPV entity being currently evaluated. Can be a combined entity.
self	an alias for the current TPV entity.

## Custom evaluation contexts

These are user defined context values that can be defined globally, or locally at the level of each entity. Any defined context value is available as a regular variable at the time the entity is evaluated.

## Special evaluation contexts

In addition to the defaults above, additional context variables are available at different steps.

*gpu, core and mem expressions* - these are evaluated in order, and thus can be referred to in that same order. For example, gpu expressions cannot refer to core and mem, as they have not been evaluated yet. cpu expressions can be based on gpu values. mem expressions can refer to both cores and gpus.

*env and param expressions* - env expressions can be based on gpu, cores or mem. param expressions can additionally refer to evaluated env expressions.

*rank functions* - these can refer to all prior expressions, and are additionally passed in a *candidate\_destinations* array, which is a list of matching TPV destinations.

## Properties that do not support expressions

Some properties do not support expressions. These are primarily:

- max\_accepted\_cores, max\_accepted\_mem and max\_accepted\_gpus, which can only be defined on destinations. This is because when a combined entity is matched with a destination, concrete values are required.
- tags defined on entities

## Evaluation by expression type

The simple rule of thumb here is that all string expressions are evaluated as python f-strings, and all integers or boolean expressions are evaluated as python code blocks. If evaluated as an f-string, the expressions must be a single line and must evaluate to a string. If evaluated as a code-block, expressions may span multiple lines of arbitrary Python code, but the last line must be an expression that evaluates to the expected return type (The return statement should not and cannot be used)

Field	Evaluated As	Expected type
gpus	code block	float
cores	code block	float
mem	code block	float
env	f-strings	string
params	f-strings	string
min_gpus	code block	float
min_cores	code block	float
min_mem	code block	float
max_gpus	code block	float
max_cores	code block	float
max_mem	code block	float
rank	code block	list of destinations
context	not evaluated	string
scheduling tags	not evaluated	string
inherits	not evaluated	string
max_accepted_gpus	not evaluated	float
max_accepted_cores	not evaluated	float
max_accepted_mem	not evaluated	float
if	code block	bool
rules	not evaluated	list of rules
execute	code block	void
fail	f-string	string
resubmit	f-strings	string

### 1.3.2 Scheduling

TPV offers several mechanisms for controlling scheduling, all of which are optional. In its simplest form, no scheduling constraints would be defined at all, in which case the entity would schedule on the first available destination. Admins can use scheduling tags to exert additional control over which destinations jobs can schedule on. Scheduling tags fall into one of four categories, (required, preferred, accepted, rejected), ranging from indicating a requirement for a particular entity, to indicating complete aversion.

Tag Type	Description
re-require	required tags must match up for scheduling to occur. For example, if a tool is marked as requiring the <i>high-mem</i> tag, only destinations that are tagged as requiring, preferring or accepting the <i>high-mem</i> tag would be considering for scheduling.
prefer	prefer tags are ranked higher than accept tags when scheduling decisions are made.
accept	accept tags can be used to indicate that an entity can match up or support another entity, even if not preferentially.
reject	reject tags cannot be present for scheduling to occur. For example, if a tool is marked as rejecting the <i>pulsar</i> tag, only destinations that do not have that tag are considered for scheduling. If two entities have the same reject tag, they still repel each other.

## Scheduling tag compatibility table

Tag Type	Require	Prefer	Accept	Reject	Not Tagged
Require	✓	✓	✓	×	×
Prefer	✓	✓	✓	×	✓
Accept	✓	✓	✓	×	✓
Reject	×	×	×	×	✓
Not Tagged	×	✓	✓	✓	✓

## Scheduling by tag match

Scheduling tags can be used to model anything from compatibility with a destination, to permissions to execute a tool. (e.g. a tool can be tagged as requiring the “restricted” tag, and users can be tagged as rejecting the “restricted” tag by default. Then, only users who are specifically marked as requiring, tolerating, or preferring the “restricted” tag can execute that tool. Of course, the destination must also be marked as not rejecting the “restricted” tag.

## Scheduling by rules

Rules can be used to conditionally modify any entity requirement. Rules can be given an ID, which can subsequently be used by an inheriting entity to override the rule. If no ID is specified, a unique ID is generated, and the rule can no longer be overridden. Rules are typically evaluated through an *if* clause, which specifies the logical condition under which the rule matches. If the rule matches, cores, memory, scheduling tags etc. can be specified to override inherited values. The special clause *fail* can be used to immediately fail the job with an error message. The *execute* clause can be used to execute an arbitrary code block on rule match.

## Scheduling by custom ranking functions

The default rank function sorts destinations by scoring how well the tags match the job’s requirements. As this may often be too simplistic, the rank function can be overridden by specifying a custom rank clause. The rank clause can contain an arbitrary code block, which can do the desired sorting, for example by determining destination load by querying the job manager, influx statistics etc. The final statement in the rank clause must be the list of sorted destinations.

# 1.4 Concepts and Organisation

## 1.4.1 Object types

Conceptually, TPV consists of the following types of objects.

1. **Entities** - An entity is anything that will be considered for scheduling by TPV. Entities include Tools, Users, Groups, Rules and Destinations. All entities have some common properties (id, cores, mem, env, params, and scheduling tags).
2. **Scheduling Tags** - Entities can have scheduling tags defined on them that determine which entities match up, and which destination they can schedule on. Tags fall into one of four categories, (required, preferred, accepted, rejected), ranging from indicating a requirement for a particular destination, to indicating complete aversion.
3. **Loader** - The loader is responsible for loading entity definitions from a config file. The loader will parse and validate entity definitions, including compiling python expressions, and processing inheritance, to produce a list of

entities suitable for mapping. The loader is also capable of loading config files from multiple sources, including https urls.

4. **Mapper** - The mapper is responsible for routing a Galaxy job to its destination, based on the current user, tool and job that must be scheduled. The mapper will respect the scheduling constraints expressed by the loaded entities.

## 1.4.2 Operations

When a mapper routes jobs to a destination, it does so by applying 5 basic operations on entities.

### 1. Inherit

The inherit operation enables an entity to inherit the properties of another entity of the same type, and to override any required properties. While a Tool can inherit another tool, which can in-turn inherit yet another tool, it cannot inherit a User, as it is a different entity type. It is also possible to globally define a *default\_inherits* field, which is the entity that all entities will inherit from should they not have an *inherits* tag explicitly defined. Inheritance is generally processed at load time by the *Loader*, so that there is no cost at runtime. However, the *Mapper* will process default inheritance, should the user, role or tool that is being dispatched not have an entry in the entities list.

When inheriting scheduling tags, if the same tag is defined by both the parent and the child, only the child's tag will take effect. For example, if a parent defines *high-mem* as a required tag, but a child defines *high-mem* as a preferred tag, then the tag will be treated as a preferred tag.

### 2. Combine

The combine operation matches up the current user, role and tool entities, and creates a combined entity that shares all their respective preferences. If the same property is defined on both entities, the entity with the higher merge priority will override the other. The priority order is fixed in the following way: Destination > User > Role > Tool. For example, if a tool specifies *cores*, and a user also specifies *cores*, the user's *cores* value will take precedence. Properties defined on destinations have the highest priority of all. The combine operation follows the following additional rules:

#### Combining scheduling tags

When combining scheduling tags, if a role expresses a preference for tag *training* for example, and a tool expresses a requirement for tag *high-mem*, the combined entity would share both preferences. This can be used to route certain roles or users to specific destinations for example.

However, if the tags are mutually exclusive, then an *IncompatibleTagsException* is raised. For example, if a role expresses a preference for training, but the tool rejects tag *training*, then the job can no longer be scheduled. If the tags are compatible, then the tag with the stronger claim takes effect. For example, if a tool requires *high-mem* and a user prefers *high-mem*, then the combined entity will require *high-mem*. An example of using this property would be to restrict the availability of dangerous tools only to trusted users.

## Combining envs and params

In this case, these requirements are simply merged, with duplicate envs and params merged in the following order: Destination > User > Role > Tool.

### 3. Evaluate

This operation evaluates any python expressions in the combined entity. At this point, rules are also evaluated. After evaluation, expressions such as `cores`, `mem`, `max_cores`, `min_gpus` etc., will all have concrete values. During evaluation, the `cores`, `mem` and `gpu` values are clamped between `min_cores`, `min_mem`, `min_gpus` and `max_cores`, `max_mem`, `max_gpus`. Afterwards, these values can be compared with a destination's values, as described in the *match* step next.

### 4. Match

The match operation is used to find matching destinations for the combined, evaluated entity. This step ensures that the destination has sufficient `gpu`, `cores` and `mem` to satisfy the entity's request. The maximum size of a job that a destination can accept can be defined using the `max_accepted_cores`, `max_accepted_mem` and `max_accepted_gpus` fields. If these are not defined, a match is assumed. In addition, all destinations that do not have scheduling tags required by the entity are rejected, and all destinations that have scheduling tags rejected by the entity are also rejected. Preference and acceptance is not considered at this stage, simply compatibility with available destination based on the tag compatibility table documented later.

### 5. Rank

After the matching destinations are short listed, they are ranked using a pluggable rank function. The default rank function simply sorts the destinations by tags that have the most number of preferred tags, with a penalty if preferred tags are absent. However, this default rank function can be overridden per entity, allowing a custom rank function to be defined in python code, with arbitrary logic for picking the best match from the available candidate destinations.

## 1.4.3 Job Dispatch Process

When a typical job is dispatched, TPV follows the process below.

1. `lookup` - Looks up Tool, User and Role entity definitions that match the job.
2. `combine()` - Combines entity requirements to create a merged entity.
3. `evaluate()` - Evaluates expressions in combined entity.
4. `match()` - Matches the combined entity requirements with a suitable destination.
5. `rank()` - Rank the matching destinations using a pluggable rank function.
6. `choose` - The entity is combined with the best matching destination and any expressions on the destination are evaluated, with the first non-failing match chosen (no rule failures).

## 1.5 Inner workings

### 1.5.1 Types of objects

Conceptually, TPV consists of the following types of objects.

1. **Entities** - An entity is anything that will be considered for scheduling by TPV. Entities include Tools, Users, Roles, Rules and Destinations. All entities have some common properties (id, cores, mem, env, params, and scheduling tags).
2. **Scheduling Tags** - Entities can have scheduling tags defined on them that determine which entities match up, and which destination they can schedule on. Tags fall into one of four categories, (require, prefer, accept, reject), ranging from indicating a requirement for a particular destination, to indicating complete aversion.
3. **Loader** - The loader is responsible for loading entity definitions from a config file. The loader will parse and validate entity definitions, including compiling python expressions, and processing inheritance, to produce a list of entities suitable for mapping. The loader is also capable of loading config files from multiple sources, including https urls.
4. **Mapper** - The mapper is responsible for routing a Galaxy job to its destination, based on the current user, tool and job that must be scheduled. The mapper will respect the scheduling constraints expressed by the loaded entities.

### 1.5.2 Operations

When a mapper routes jobs to a destination, it does so by applying 5 basic operations on entities.

#### 1. Inherit

The inherit operation enables an entity to inherit the properties of another entity of the same type, and to override any required properties. While a Tool can inherit another tool, which can in-turn inherit yet another tool, it cannot inherit a User, as it is a different entity type. It is also possible to globally define a *default\_inherits* field, which is the entity that all entities will inherit from should they not have an *inherits* tag explicitly defined. Inheritance is generally processed at load time by the *Loader*, so that there is no cost at runtime. However, the *Mapper* will process default inheritance, should the user, role or tool that is being dispatched not have an entry in the entities list.

When inheriting scheduling tags, if the same tag is defined by both the parent and the child, only the child's tag will take effect. For example, if a parent defines *high-mem* as a required tag, but a child defines *high-mem* as a preferred tag, then the tag will be treated as a preferred tag.

#### 2. Combine

The combine operation matches up the current user, role and tool entities, and creates a combined entity that shares all their respective preferences. If the same property is defined on both entities, the entity with the higher merge priority will override the other. The priority order is fixed in the following way: Destination > User > Role > Tool. For example, if a tool specifies *cores*, and a user also specifies *cores*, the user's *cores* value will take precedence. Properties defined on destinations have the highest priority of all. The combine operation follows the following additional rules:



## Combining scheduling tags

When combining scheduling tags, if a role expresses a preferences for tag *training* for example, and a tool expresses a requirement for tag *high-mem*, the combined entity would share both preferences. This can be used to route certain roles or users to specific destinations for example.

However, if the tags are mutually exclusive, then an `IncompatibleTagsException` is raised. For example, if a role expresses a preference for training, but the tool rejects tag *training*, then the job can no longer be scheduled. If the tags are compatible, then the tag with the stronger claim takes effect. For example, if a tool requires *high-mem* and a user prefers *high-mem*, then the combined entity will require *high-mem*. An example of using this property would be to restrict the availability of dangerous tools only to trusted users.

## Combining envs and params

In this case, these requirements are simply merged, with duplicate envs and params merged in the following order: Destination > User > Role > Tool.

## 3. Evaluate

This operation evaluates any python expressions in the combined entity. At this point, rules are also evaluated. After evaluation, expressions such as `cores`, `mem`, `max_cores`, `min_gpus` etc., will all have concrete values. During evaluation, the `cores`, `mem` and `gpu` values are clamped between `min_cores`, `min_mem`, `min_gpus` and `max_cores`, `max_mem`, `max_gpus`. Afterwards, these values can be compared with a destination's values, as described in the *match* step next.

## 4. Match

The match operation is used to find matching destinations for the combined, evaluated entity. This step ensures that the destination has sufficient gpus, cores and mem to satisfy the entity's request. The maximum size of a job that a destination can accept can be defined using the `max_accepted_cores`, `max_accepted_mem` and `max_accepted_gpus` fields. If these are not defined, a match is assumed. In addition, all destinations that do not have scheduling tags required by the entity are rejected, and all destinations that have scheduling tags rejected by the entity are also rejected. Preference and acceptance is not considered at this stage, simply compatibility with available destination based on the tag compatibility table documented later.

## 5. Rank

After the matching destinations are short listed, they are ranked using a pluggable rank function. The default rank function simply sorts the destinations by tags that have the most number of preferred tags, with a penalty if preferred tags are absent. However, this default rank function can be overridden per entity, allowing a custom rank function to be defined in python code, with arbitrary logic for picking the best match from the available candidate destinations.

### 1.5.3 Job Dispatch Process

When a typical job is dispatched, TPV follows the process below.

1. lookup - Looks up Tool, User and Role entity definitions that match the job.
2. combine() - Combines entity requirements to create a merged entity.
3. evaluate() - Evaluates expressions in combined entity.
4. match() - Matches the combined entity requirements with a suitable destination.
5. rank() - Rank the matching destinations using a pluggable rank function.
6. choose - The entity is combined with the best matching destination and any expressions on the destination are evaluated, with the first non-failing match chosen (no rule failures).

## 1.6 Shell Commands

### 1.6.1 lint

TPV config files can be checked for linting errors using the `tpv lint` command.

```
tpv lint <url_or_path_to_config_file>
```

If linting is successful, a lint successful message will be displayed with an exit code of zero. If the linting fails, a lint failed message with the relevant error will be displayed with an exit code of 1. For example:

```
$ cat >good.yml <<EOF
tools:
  default:
    cores: 1
    mem: cores * 3.9
    context:
      partition: normal
    params:
      native_specification: "--nodes=1 --ntasks={cores} --ntasks-per-node={cores} --mem=
↪{round(mem*1024)} --partition={partition}"
    scheduling:
      reject:
        - offline
    rules: []
EOF
$ tpv lint good.yml
INFO : tpv.commands.shell: lint successful.
$ echo $?
0

cat >bad.yml <<EOF
tools:
  - default:
    cores: 1
EOF
$ tpv lint bad.yml
```

(continues on next page)

(continued from previous page)

```
INFO : tpv.commands.shell: lint failed.
$ echo $?
1
```

To display the reasons for the failure, use the `-v` option to increase verbosity, with ↵ each additional `v` increasing log level.

## 1.6.2 dry-run

You can test that your TPV configuration returns the expected destination for a given tool and/or user using the `tpv dry-run` command.

```
tpv dry-run --job-conf <path_to_galaxy_job_conf_file> [--tool <tool_id>] \
  [--user <user_name_or_email>] [--input-size <size_in_gb>] \
  [tpv_config_file ...]
```

If no TPV config files are specified on the command line, they will be read from the `tpv_dispatcher` execution environment (destination) definition in the specified Galaxy job configuration file.

For example:

```
$ tpv dry-run --job-conf /srv/galaxy/config/job_conf.yml
!!python/object:galaxy.jobs.JobDestination
converted: false
env:
- {name: LC_ALL, value: C}
id: slurm
legacy: false
params: {native_specification: --nodes=1 --ntasks=1 --ntasks-per-node=1 --mem=3994
  --partition=normal, outputs_to_working_directory: true, tmp_dir: true}
resubmit: []
runner: slurm
shell: null
tags: null
url: null
```

```
$ tpv dry-run --job-conf /srv/galaxy/config/job_conf.yml --tool trinity --input-size 40 ↵
↵ *.yml
!!python/object:galaxy.jobs.JobDestination
converted: false
env:
- {name: LC_ALL, value: C}
- {name: TERM, value: vt100}
- {execute: ulimit -c 0}
- {execute: ulimit -u 16384}
id: pulsar
legacy: false
params:
  default_file_action: remote_transfer
  dependency_resolution: remote
  jobs_directory: /scratch/pulsar/staging
```

(continues on next page)

(continued from previous page)

```
outputs_to_working_directory: false
remote_metadata: false
rewrite_parameters: true
submit_native_specification: --nodes=1 --ntasks=20 --ntasks-per-node=20 --
↪partition=xlarge
transport: curl
resubmit: []
runner: pulsar
shell: null
tags: null
url: null
```

## 1.7 Migration Guide

### 1.7.1 Migrating from v1.x to v2.x

TPV v2.0.0 introduces a number of changes to improve simplicity by disambiguating overloaded terms and reducing code complexity. (xref: <https://github.com/galaxyproject/total-perspective-vortex/pull/58>). This has resulted in the following breaking changes.

1. *cores*, *mem* and *gpus* on destinations have been renamed to *max\_accepted\_cores*, *max\_accepted\_mem* and *max\_accepted\_gpus*. While *cores*, *mem* and *gpus* can still be defined on a destination, the result will be that all tools at that destination would be forcibly changed to use those core, mem and gpu values, which is probably not what is desired. Instead, define *max\_accepted\_{cores|mem|gpus}* to preserve previous behaviour.
2. In TPV 1.x, *cores*, *mem* and *gpus* defined on Users or Roles were used as the *max\_cores* to allocate to a user or role. For example, if a tool defines *cores* as 4, but a user defines *cores* as 2, the lower of the two values would be used. This overloaded terminology has been disambiguated in TPV 2.x, by introducing several additional properties. All entities can now define: *min\_cores*, *min\_mem* and *min\_gpus* as well as *max\_cores*, *max\_mem* and *max\_gpus*. No matter how many cores a tool requests, they will be clamped between these specified min and max values. Therefore, in TPV 2.x, *cores* defined on users or roles will need to be renamed to *max\_cores* to preserve earlier semantics.
3. The *runner* parameter is now required on destinations. TPV no longer reads destinations defined in *job\_conf.yml*, and instead, only uses destinations defined in its own configuration files. The linter has been updated to warn you if the *runner* parameter is not defined.
4. The *destination\_name\_override* property is no longer an extra param on the destination. It is instead, a top-level property of a destination. The *destination\_name\_override* can be used to dynamically generate a custom name for the destination.
5. Any custom Python code that refers to scheduling tags through *entity.tags* should now use *entity.tpv\_tags* on Tool, User, and Role entities. Destination entities now have the property *entity.tpv\_dest\_tags*.

## 1.8 FAQ

1. How can I perform some debug logging within a code block?

You can simply import the logging module within a code block. For example:

**linenos**

**tools:**

default: cores: 2 mem: cores \* 3 rules:

- if: True execute: |

```
import logging log = logging.getLogger(__name__) log.debug(f'Here's what the entity
looks like: {entity}')
```

2. How can I import custom libraries or code?

As long as the code is available within Galaxy's virtualenv or the python path, it can be imported like any other package. For example:

**linenos**

**tools:**

default: cores: 2 mem: cores \* 3 rules:

- if: True execute: |

```
import my_custom_module my_custom_module.my_func()
```

## 1.9 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)