Galaxy in Production

Nate Coraor
Galaxy Team
Penn State University

Galaxy runs out of the box!

- Simple download, setup, and install design:

```
% hg clone http://bitbucket.org/galax...
% sh run.sh
```

- Great for development!
- Not designed to support multiple users in a production environment with default configuration

But more powerful scenarios are supported

- By default, Galaxy uses:
  - SQLite
  - Built-in HTTP server for all tasks
  - Local job runner
  - Single process
  - Simplest error-proof configuration

Groundwork for scalability

Start with a clean environment

- Galaxy becomes a managed system service
- Give Galaxy its own user
- Don't share database or db users
- Make sure Galaxy is using a clean Python interpreter: virtualenv, or compile your own
- Galaxy can be housed in NFS or other cluster/network filesystems (has been tested w/ GPFS)

Basic Configuration

Disable the developer settings

- use_interactive = False - Not even safe (exposes config)
- use_debug = False - You'll still be able to see tracebacks in the log file, doesn't load response in memory

Get a real database

- SQLite is serverless
- Galaxy is a heavy database consumer
- Locking will be an immediate issue
- Migrating data is no fun
- Setup is very easy: '`database_connection = postgres://`'

Offload the menial tasks: Proxy

- Directly serve static content faster than Galaxy's HTTP server
- Reduce load on the application
- Caching and compression
- Load balancing (more on that later)
- Hook your local authentication and authorization system

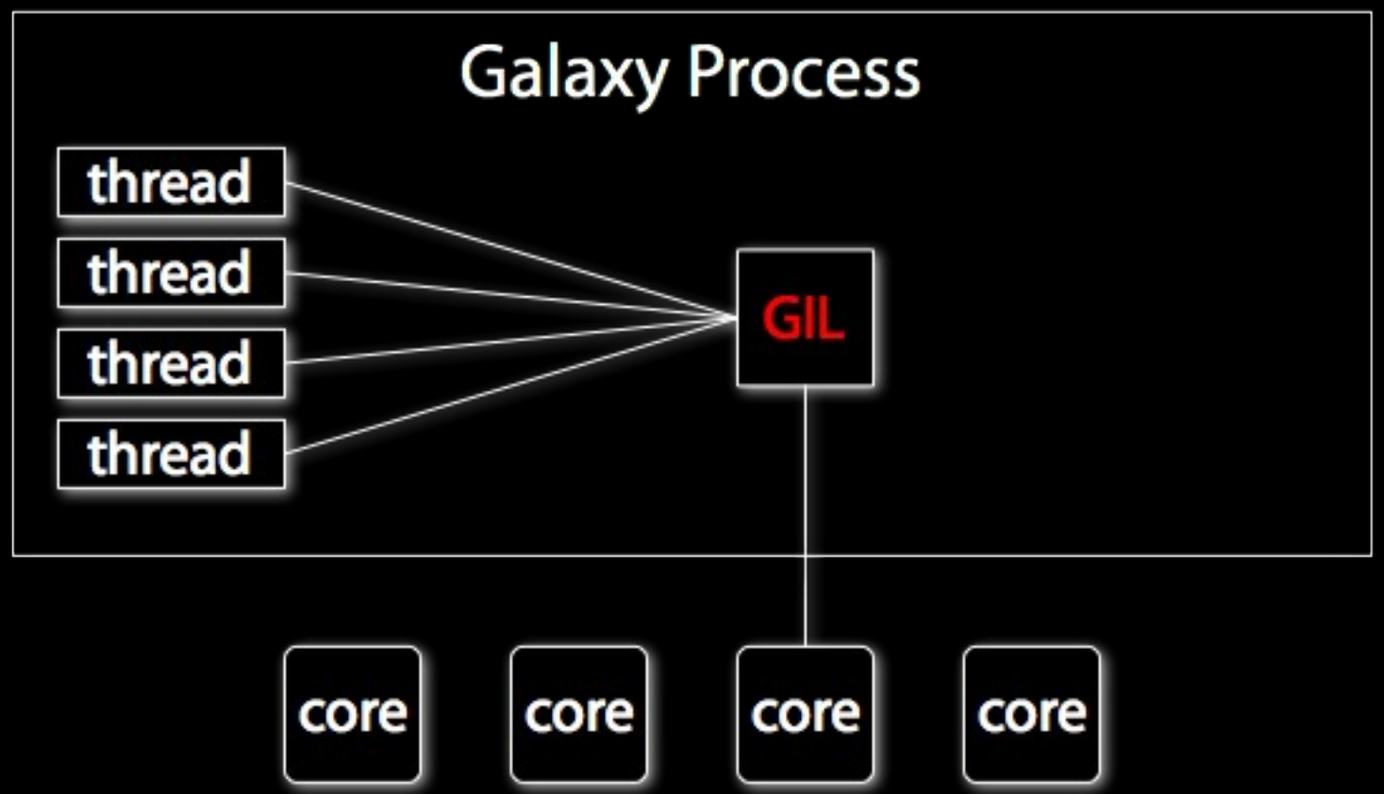
Metadata Detection

- Setting metadata is CPU intensive and will make Galaxy unresponsive
  - Make a new process (better yet, run on the cluster!)
    - All you need is: 'set_metadata_externally = True'
- Run the data source tools on the cluster if they have access to the Internet
  - Remove 'tool = local:///' from config file

Advanced Configuration

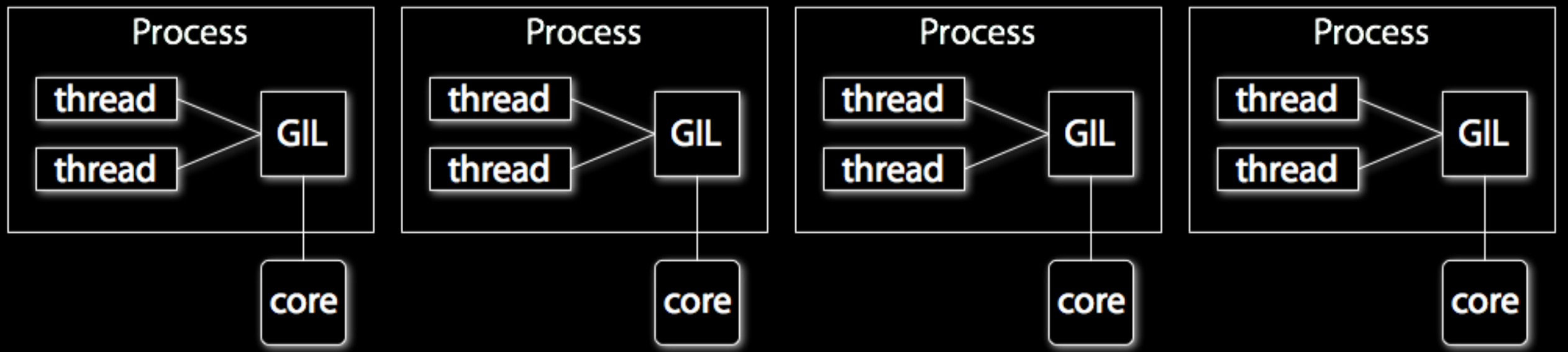
Python and threading

- Galaxy is multi-threaded. No problem, right?
- Problem... Enter the Global Interpreter Lock



- Guido says: "run multiple processes instead of threads!"

Galaxy's multiprocess model



- One job manager - responsible for preparing and finishing jobs, monitoring cluster queue
- Many web servers
- Doesn't really need IPC, job notification through database

Defining extra servers is easy

```
[server:web_0]
port = 8000
[server:web_1]
port = 8001
[server:web_2]
port = 8002
...
```

```
[server:runner_0]
port = 8100
```

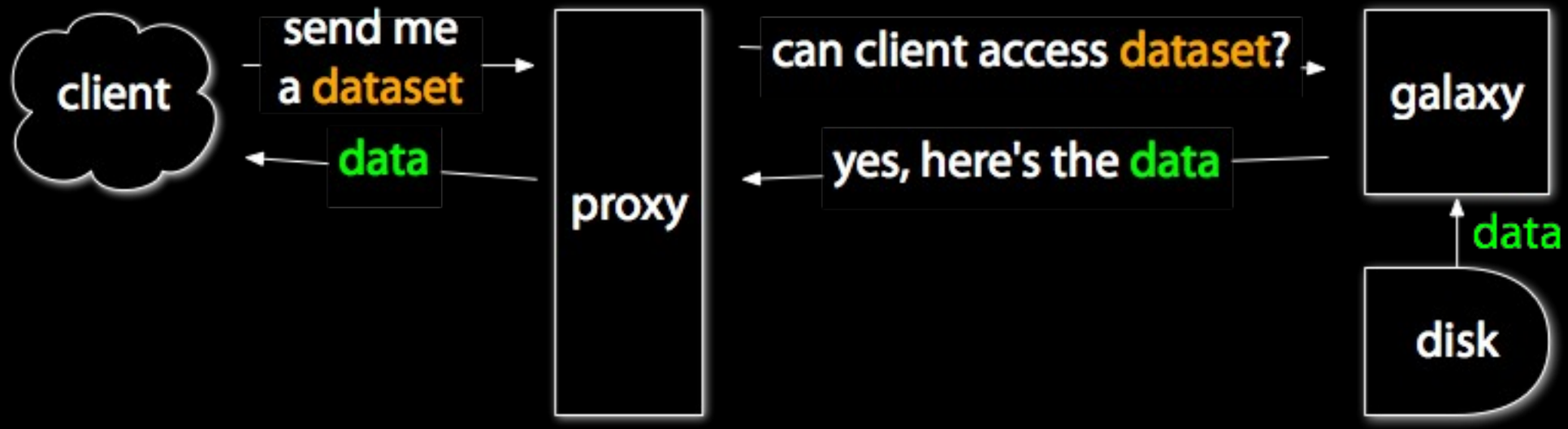
Let your tools run free: Cluster

- Move intensive processing (tool execution) to other hosts
- Frees up the application server to serve requests and manage jobs
- Utilize existing resources
- No job interruption upon restart
- Per-tool cluster options
- Generic DRMAA support: SGE (and derivatives), LSF, PBS Pro, Condor?
- It's easy: Set 'start_job_runners' and 'default_cluster_job_runner' and go!
- If your cluster has Internet access, run upload, UCSC, etc. on the cluster too

Tune Database Parameters

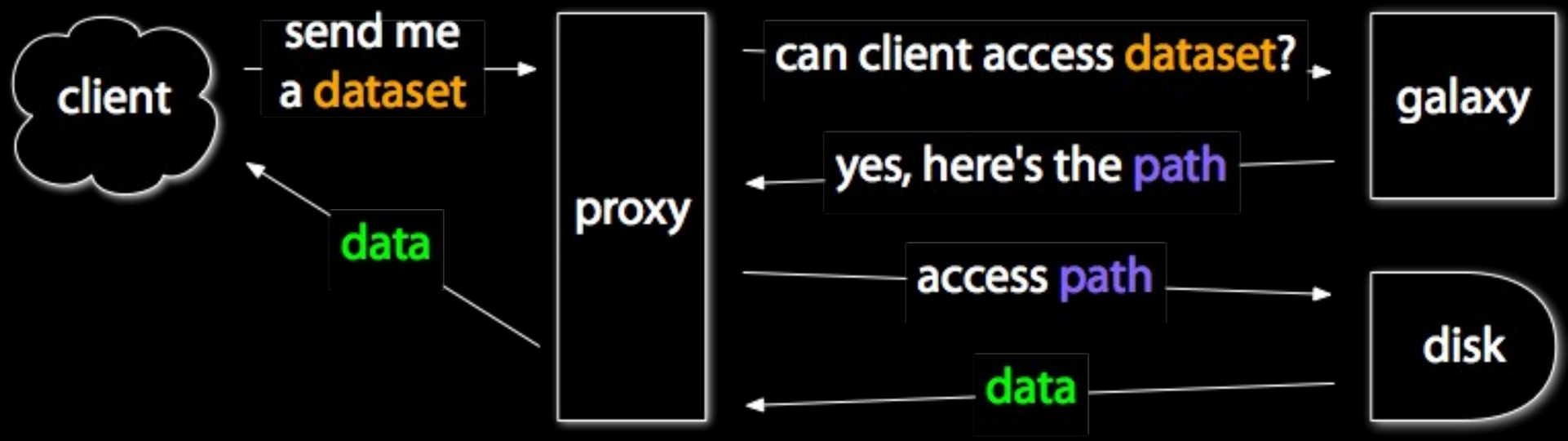
- Let Postgres (not Galaxy) keep the result in memory.
  'database_engine_option_server_side_cursors = True'
- Allow more database connections.
  'database_engine_option_pool_size = 10'
  'database_engine_option_max_overflow = 20'
- Don't create unnecessary connections to the database. 'database_engine_option_strategy = threadlocal'

Downloading data
from Galaxy



Downloading data

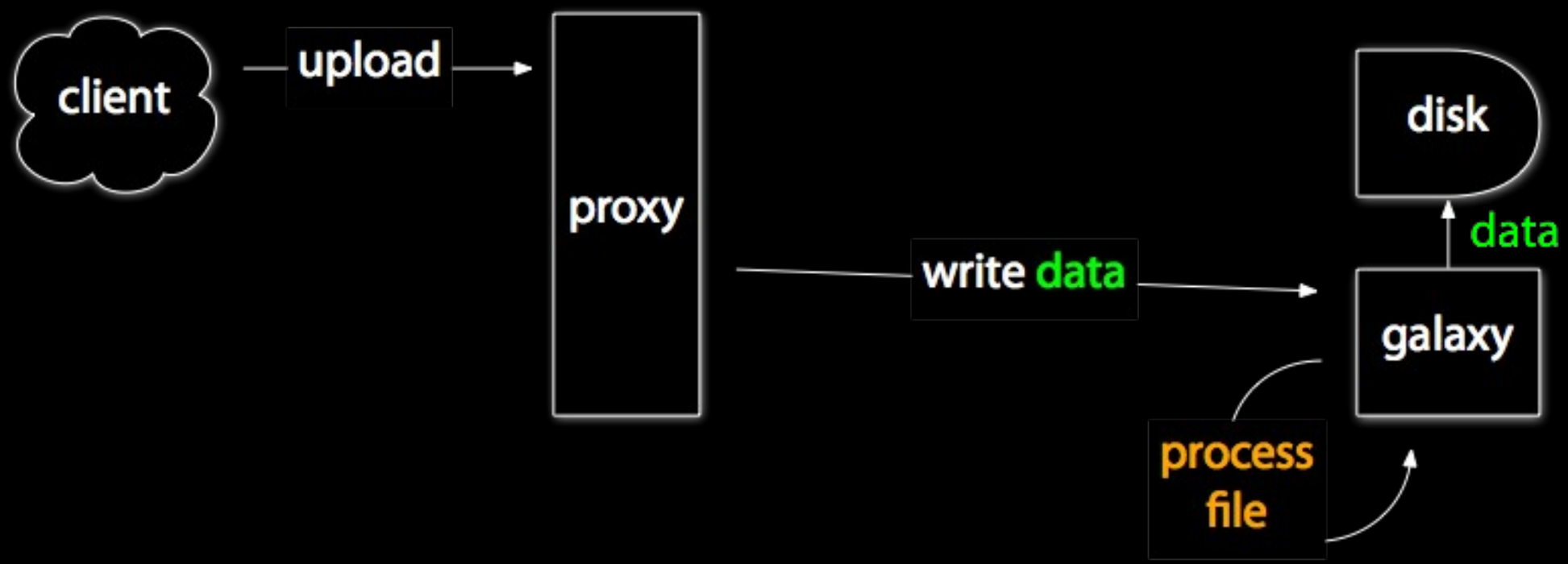
from the proxy



Downloading data
from the proxy

- The proxy server can send files much faster than Galaxy's internal HTTP server and file I/O methods
- Reduce load on the application, free the process
- Restartability
- Security is maintained: the proxy consults Galaxy for authZ
- Proxy server requires minimal config and then:
  - nginx: 'nginx_x_accel_redirect_base = /_download'
  - Apache: 'apache_xsendfile = True'

Uploading data
to Galaxy



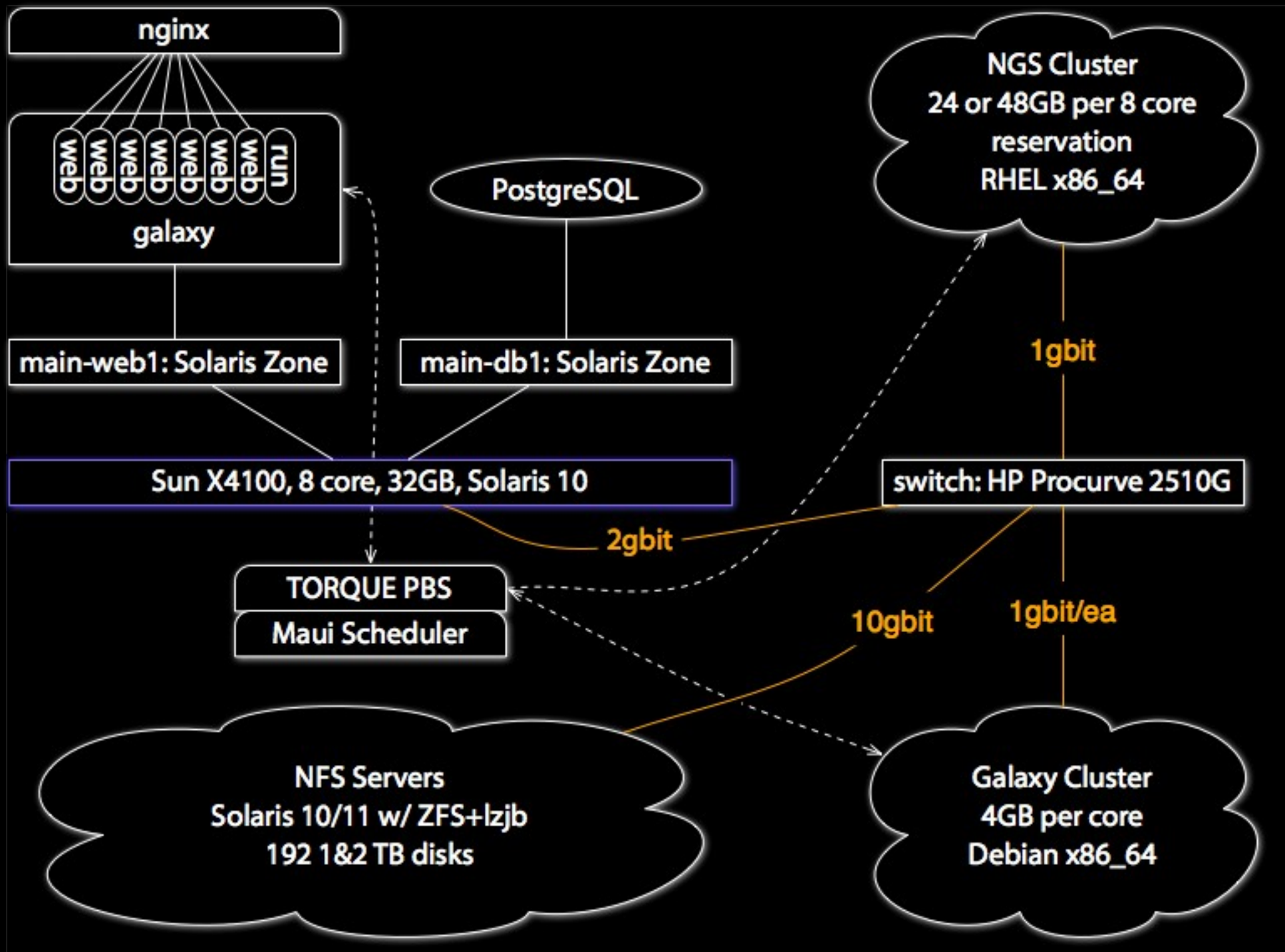
Uploading data

to the proxy

Uploading data
to the proxy

- The proxy is also better at receiving files than Galaxy
- Again, reduce load on the application, free the process
- Again, restartability
- More reliable
- Slightly more complicated to set up, and nginx only

How do we run Galaxy Main?


nginx
web
web
web
web
web
web
web
run
galaxy
PostgreSQL
NGS Cluster
24 or 48GB per 8 core
reservation
RHEL x86_64
main-web1: Solaris Zone
main-db1: Solaris Zone
1gbit
Sun X4100, 8 core, 32GB, Solaris 10
switch: HP Procurve 2510G
2gbit
TORQUE PBS
Maui Scheduler
10gbit
1gbit/ea
NFS Servers
Solaris 10/11 w/ ZFS+lzjb
192 1&2 TB disks
Galaxy Cluster
4GB per core
Debian x86_64

All the details:

usegalaxy.org/production

Automating Galaxy

Dannon Baker
Galaxy Team
Emory University

RESTful API

- Simple URIs
  - Collections: http://example.org/resources/
  - Elements: http://example.org/resources/42
- The method performed defines the operation
  - HTTP GET/PUT/POST/DELETE

Galaxy's REST Overview

- Uses generated API keys for per-user authentication
  - No username/password
  - No credential caching (not REST!)
- Request parameters and responses are in JSON (JavaScript Object Notation)
- Maintains security
- Enable with `enable_api = True` in config

GET Example

```
» GET /api/libraries?key=966354fc14c9e427cee380ef50a72a21
  « [
  «   {
  «     'url': '/api/libraries/f2db41e1fa331b3e',
  «     'id': 'f2db41e1fa331b3e',
  «     'name': 'Library 1'
  «   }
  « ]
```

Modules

- Libraries
- Users and Roles
- Sample Tracking
- Forms
- Workflows
- Histories *

Scripted Usage

```
import os, sys, traceback

from common import update

try:
    data = {}
    data[ 'update type' ] = 'request state'
except IndexError:
    print 'usage: %s key url' % os.path.basename( sys.argv[0] )
    sys.exit( 1 )
update( sys.argv[1], sys.argv[2], data, return_formatted=True )
```

- More in galaxy/scripts/api
- Share yours - Galaxy tool shed

Extending the API

- It's still an early beta
- Wrap controller methods
- It's easy!