Extreme Scale Image Simulation Workflows for LSST DESC

Antonia Sierra Villarreal, Postdoctoral Researcher
Argonne National Laboratory
May 26th, 2022

LSST DESC
Dark Energy Science Collaboration

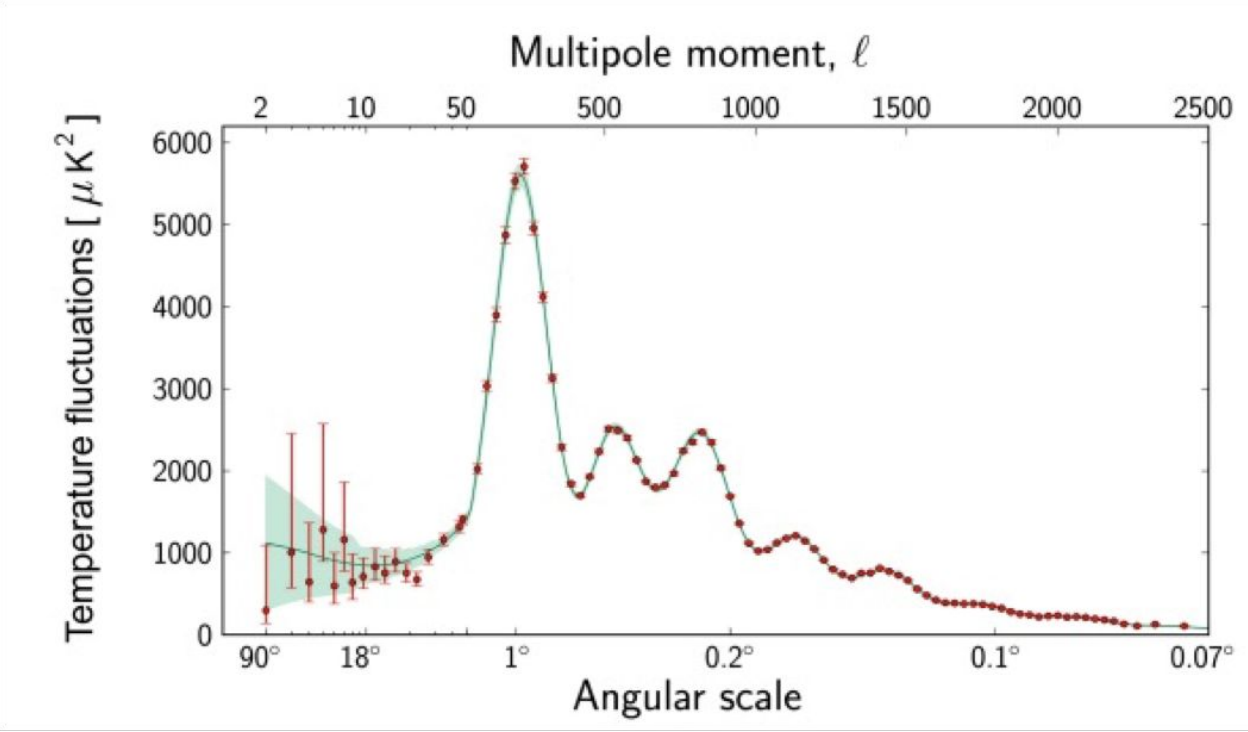
The **Big** Questions

- What is the Vera Rubin Observatory LSST DESC Second Data Challenge Simulation Campaign
  - Corollary: why is this such a long title?
- How do we start from nothing to generating simulated telescope quality images?
- How do we turn these into scalable and portable workflow?
- What lessons can we take away from this going forward?

What Even is the Vera Rubin Observatory LSST Second Data Challenge?

The “Standard” Cosmology

- “Lambda Cold Dark Matter Cosmology”
- Measured to very high precisions by multiple experiments
- Fit with just 7 cosmological parameters.
- While some tensions exist, competing models struggle



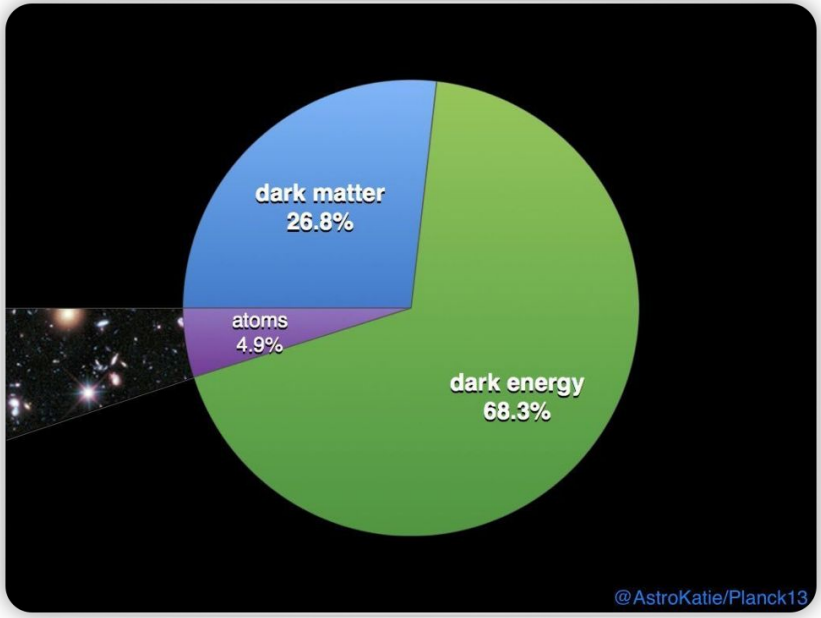
ESA and the Planck Collaboration

DC2 Image Simulation Campaign

- Vera Rubin Observatory LSST is a massive undertaking planning to observe 18,000 deg^2 of the night sky over the course of ten years.
- The Dark Energy Science Collaboration (DESC) is one of several science collaborations seeking to use this exciting data.
  - We want to constrain cosmological parameters
- This is an unprecedented amount of data in cosmology — effectively surveying the entire southern night sky once every three nights!

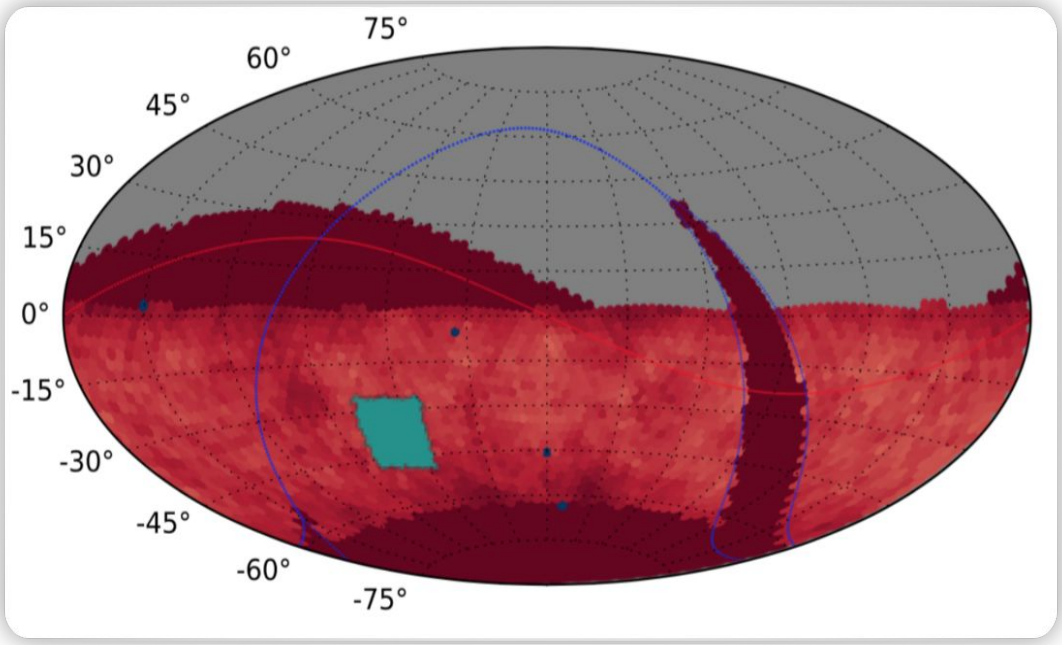
DC2 Image Simulation Campaign

- Our goal: understand those 7 parameters that define the universe
- Many probes: large-scale structure, weak lensing, type Ia supernovae, galaxy clusters, strong lensing…
- LSST will drive down statistical error
- DESC must drive down systematic error
- The key? Simulations!



Data Challenge 2 Specifications

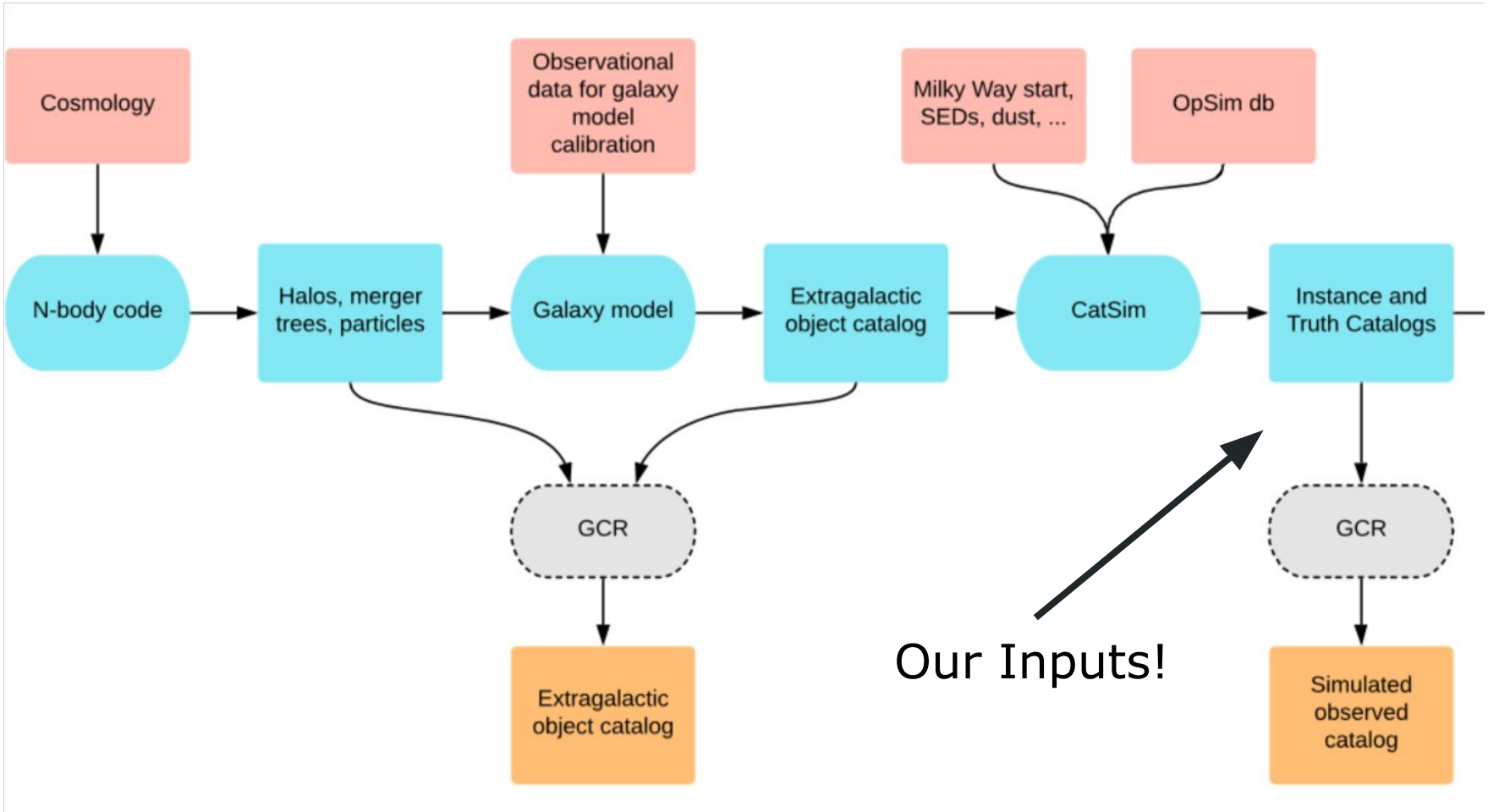
- Simulation 300 degrees^2 of the night sky for five years depth
- "Wide-Fast-Deep" component to enable large scale structure studies
- "Deep Drilling Field" component to enable transient studies
- Six color bands



Example of DC2 Coverage from LSST DESC 2021 paper

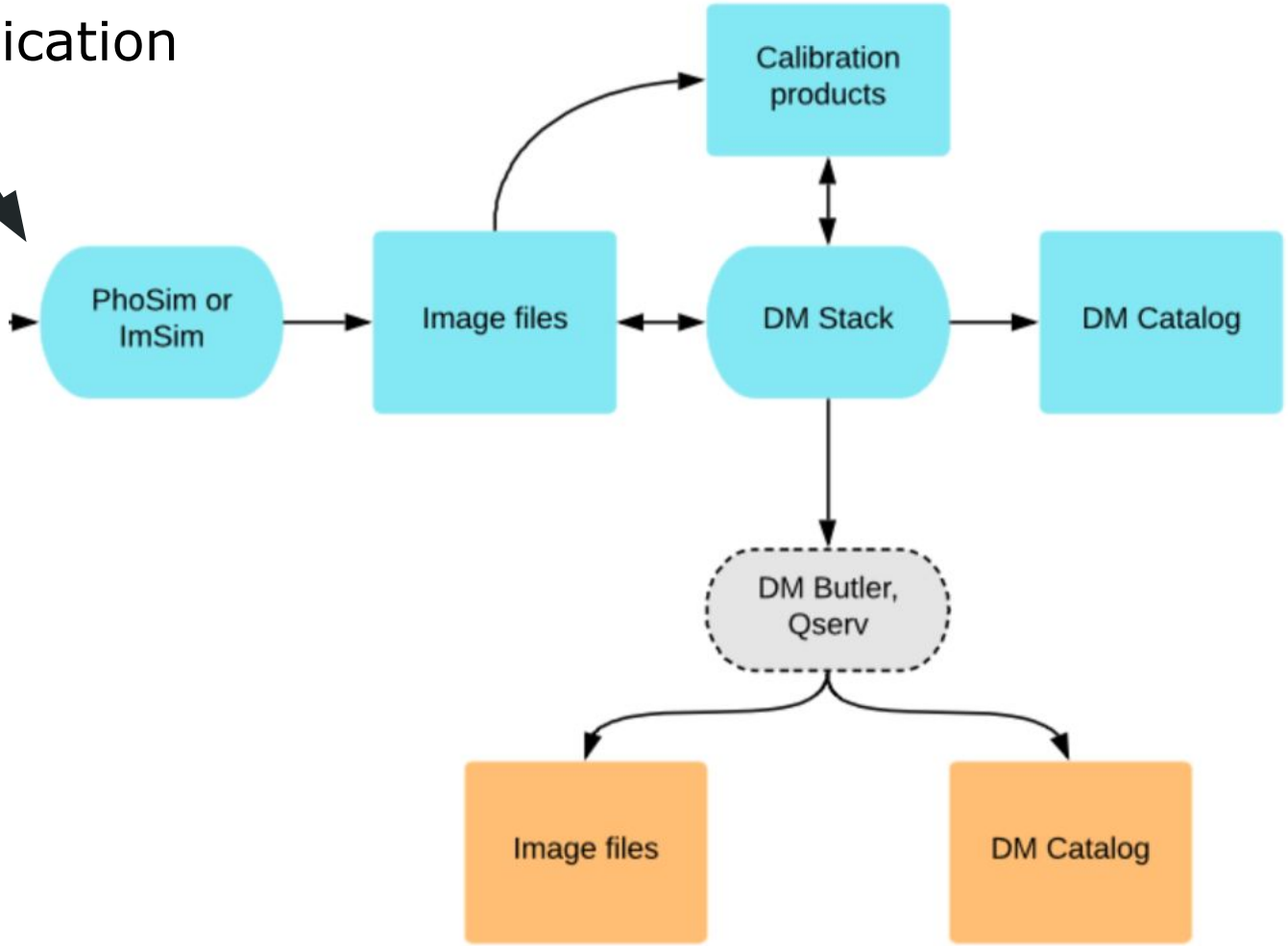
Starting from Nothing to Telescope Quality Images

A Long and Complicated Road…



…Build to an Answer

Our Core Application



So Many Contributors!

There are so many contributors to list, even working directly with me!

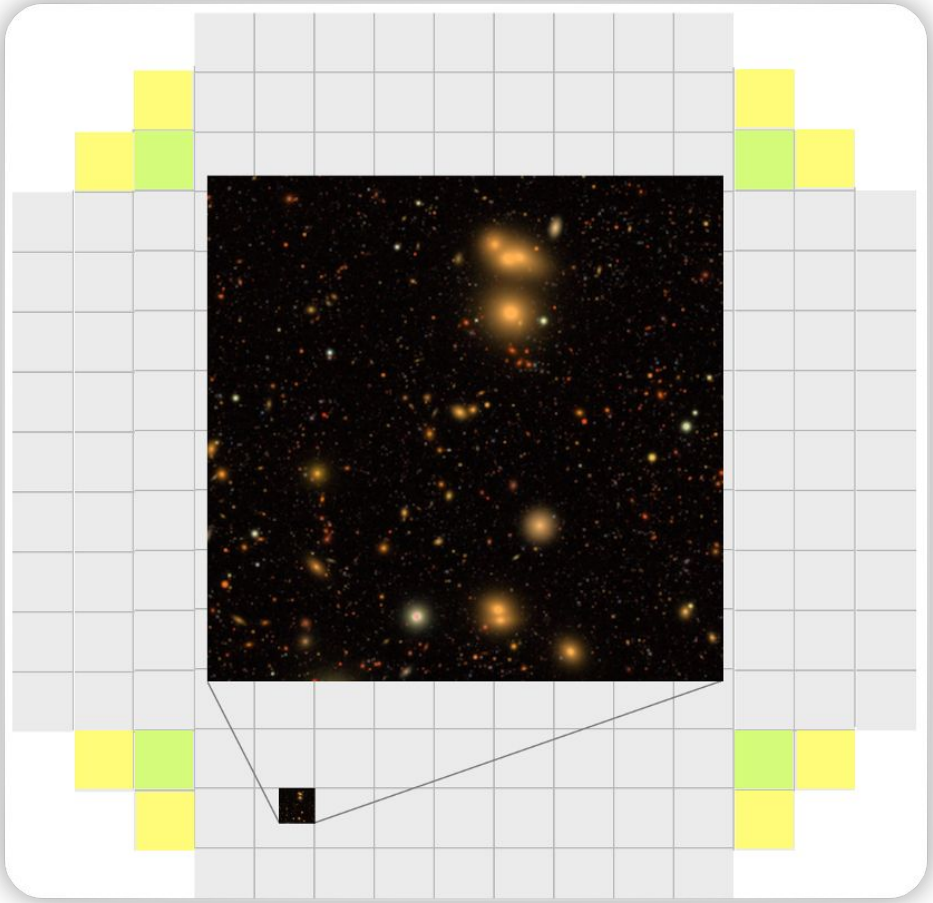
- Seriously, there are about ~90 authors on the DC2 paper.
- Full details of the effort: https://arxiv.org/abs/2010.05926

Some particular thanks (in no real order) go to:

- The imSim development team for iterating on the software throughout the runs
- The Parsl team for working exhaustively to make sure our pipelines ran on HPC resources
- The DC2 team behind me for working hard to get me inputs!

Image Simulation

- Galsim is open source software for the simulation of astronomical objects
- imSim is a Python code handling:
  - the pointing of the telescope
  - simulation of lensing and dust
  - simulation of extragalactic objects and stars
- For each sensor (grey box on right), simulate one image



How Do We Scale This Up?

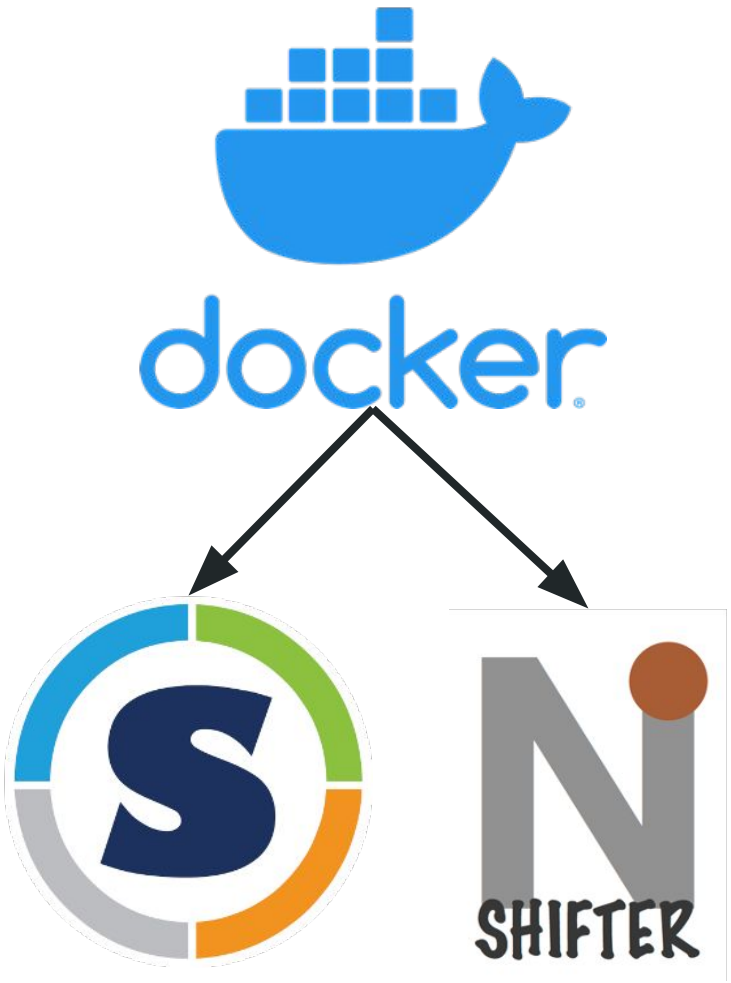
Step 1: Containers

Why even consider using containers?

- Underlying software is large, complex, and evolving
- Available compute resources were heterogeneous

Our solution? Docker!

- Underlying Docker container to run imSim
- Shifter and Singularity both convert Docker to their formats



Container Example

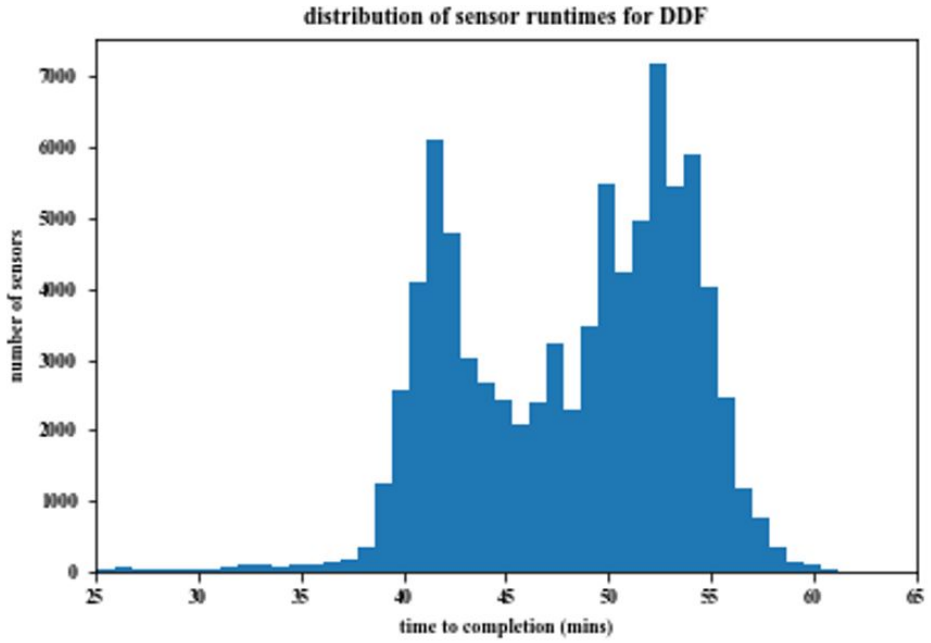
- LSST Software container as baseline
- Specific software added in
- Ideally tagged images for reproducibility

```
FROM lsstdesc/stack-sims:w_2019_42-sims_w_2019_42-v2

USER root
RUN mkdir -p /DC2 &&\
  chown lsst /DC2
USER lsst
RUN set +e &&\
  source scl_source enable devtoolset-8 &&\
  set -e &&\
  source /opt/lsst/software/stack/loadLSST.bash &&\
  setup lsst_sims -t sims_w_2019_42 &&\
  setup -t DC2production throughputs &&\
  setup -t DC2production sims_skybrightness_data &&\
  cd /DC2 &&\
  git clone https://github.com/LSSTDESC/imSim.git &&\
  git clone https://github.com/LSSTDESC/DESC_DC2_imSim_Workflow.git &&\
  setup -r imSim -j &&\
  cd imSim &&\
  git checkout v1.0.0 &&\
  scons &&\
  cd ../DESC_DC2_imSim_Workflow &&\
  git checkout Run2.2i-production-v2
ENTRYPOINT ["/DC2/DESC_DC2_imSim_Workflow/docker_run.sh"]
CMD ["echo You must specify a command to run inside the LSST ALCF container"]
```

Step 2: Parsl

- Parsl is a parallel scripting Python library.
- “Worker” processes are initialized on computed resources
- Python functions or arbitrary bash commands can be packaged as Parsl “apps”
- A driver process distributes “app” calls to workers



Some Parsl Examples

Python apps are a decorator around any Python function. Used for:

- pre-processing scripts
- bundling scripts
- archival scripts

Bash apps are also a decorator, but execute a string return on the system. Used for:

- container downloads
- container launches

```python
@python_app(executors=['login-node'])
def estimate_pi(n_points):
    import numpy as np
    x = np.random.uniform(0,1,n_points)
    y = np.random.uniform(0,1,n_points)
    dist = np.sqrt(x*x+y*y)
    n_circle = np.sum(dist <= 1)
    pi_est = 4*n_circle/n_points
    return pi_est
```

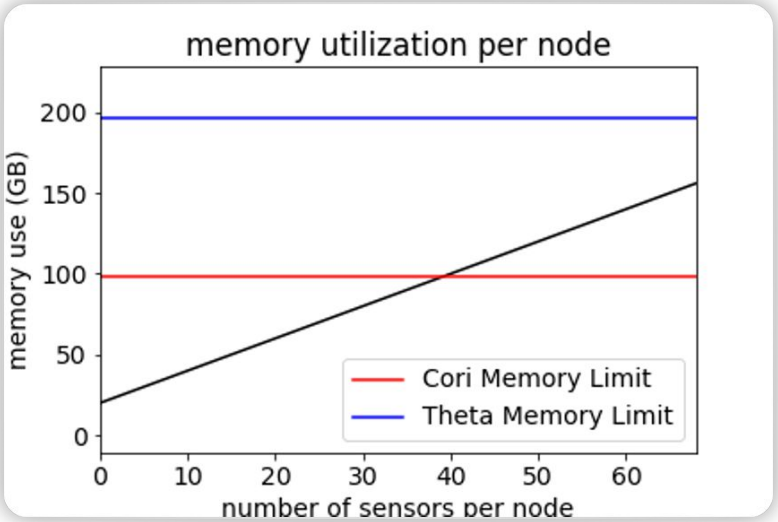
```python
@bash_app(executors=['login-node'])
def calc_pi(num_points, outputs=[]):
    return 'python /home/antoniov/repos/parsl-pi-test/calc_pi.py {} &> {}'.format(num_points, outputs[0])
```

Step 3: Bundler Scripts

We need to balance the following needs:

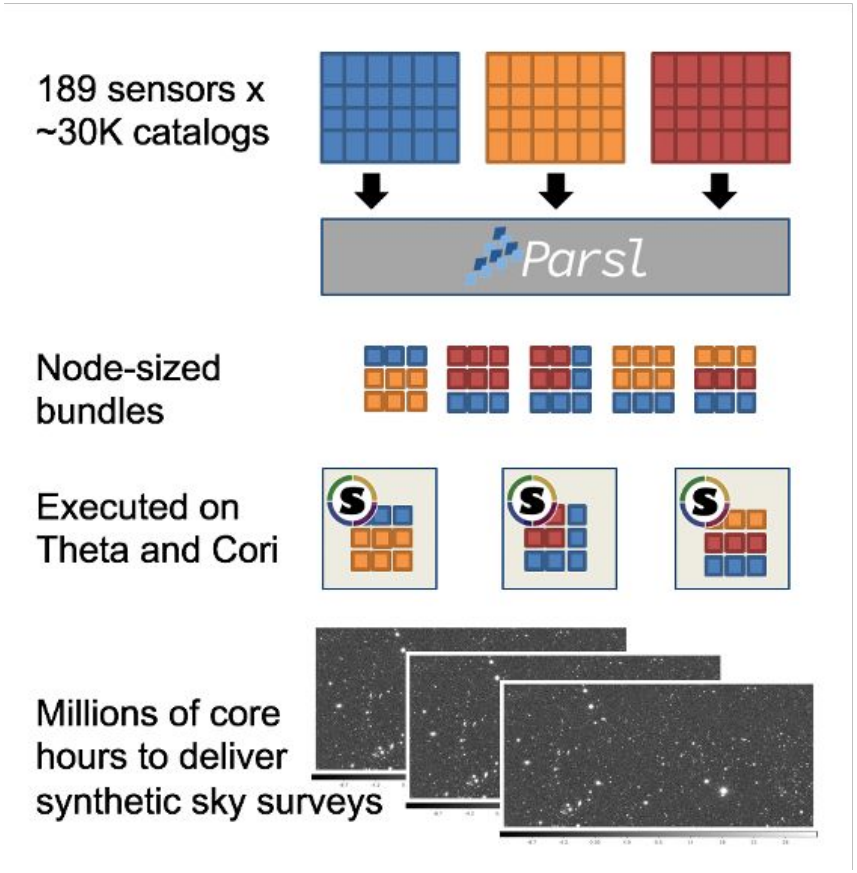
- Each container has an expected memory footprint
- Multiple sensors from the same visit can *share* some memory requirements
- Each sensor has an average memory footprint
- The sensor memory footprint varies by the object being drawn

Python scripts take all of this information into account, the architecture of the machine, and creates bundles of commands to be run as Parsl apps.



The Resulting Workflow

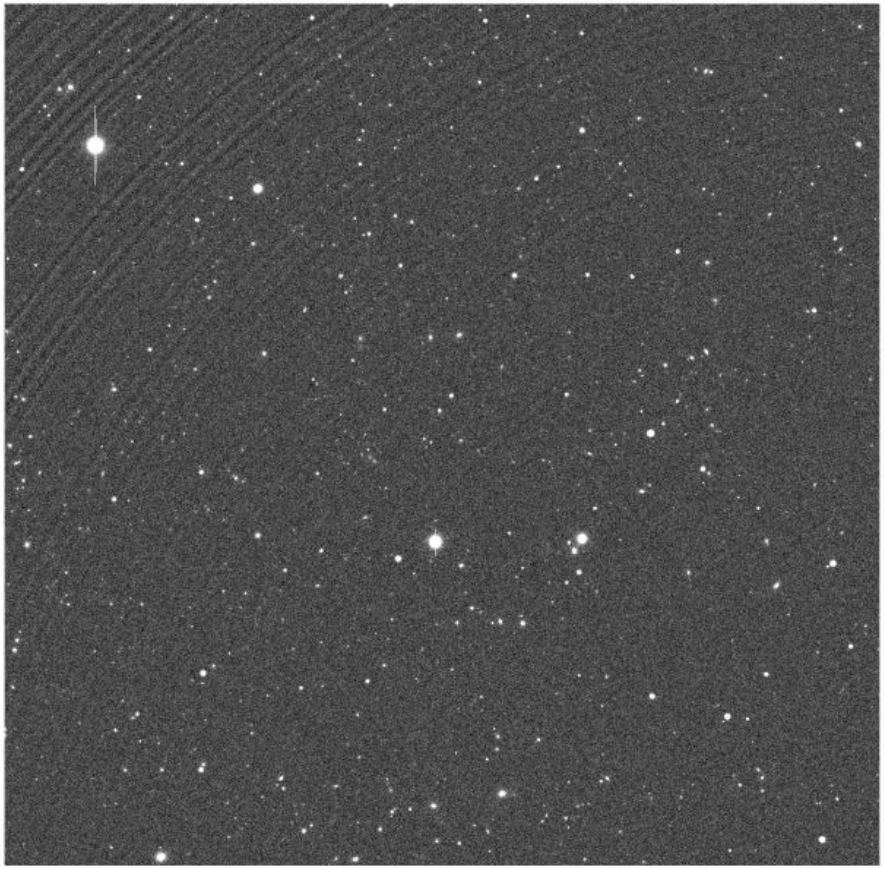
- Parsl driver started with a directory of raw inputs to cycle through
- Preprocessing scripts determine the optimal work configuration
- Parsl driver submits a job request to compute resources
- Once resources are available, workers are started on compute nodes and connect to the Parsl driver
- Each worker runs a container that launches imSim for a given sensor list
- If a worker finishes one container, it cycles in a new one from the overall task list

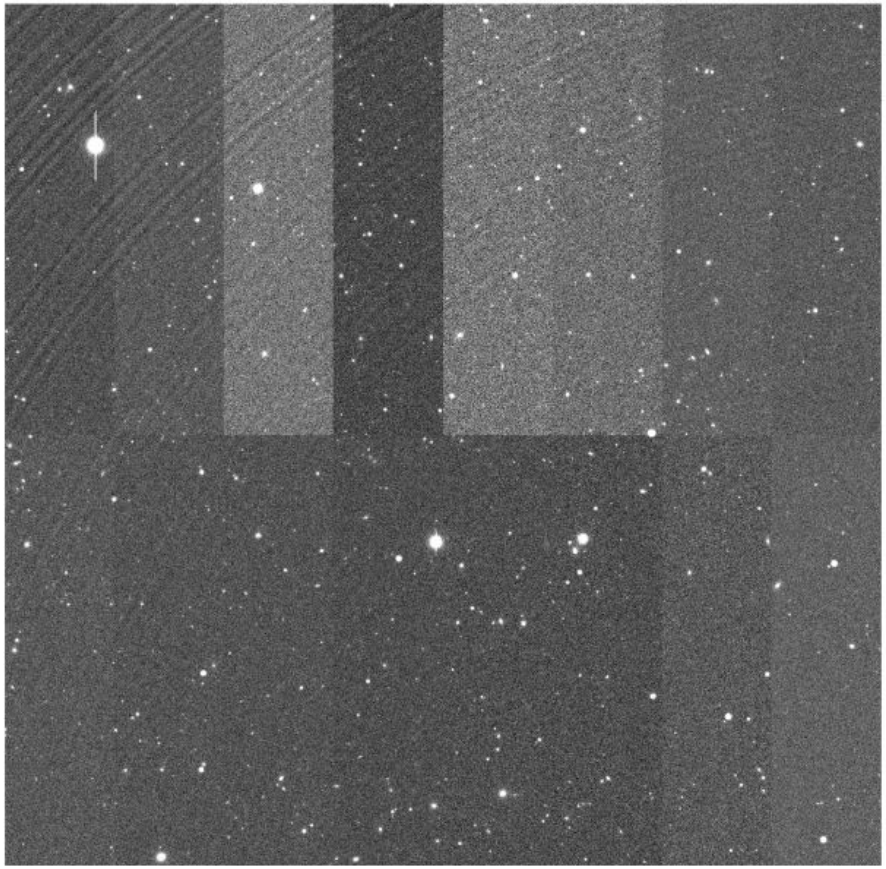


The Final Results?

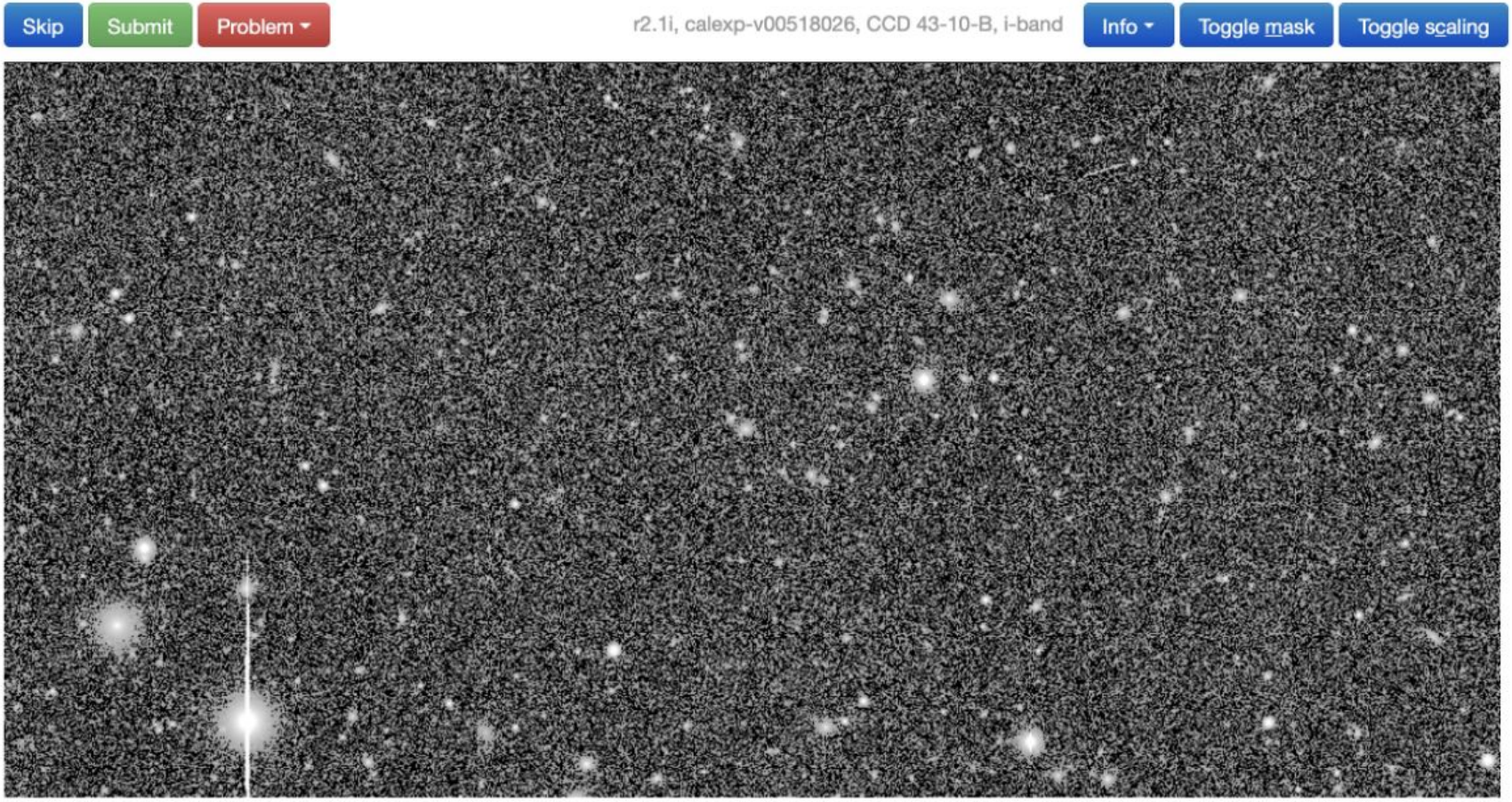
- 100M+ compute hours used
- Up to 2000 nodes used simultaneously on NERSC Cori
- Up to 4000 nodes on ALCF Theta
- Five years of simulated LSST depth
- Multiple DESC projects using these outputs

Some Example Raw Outputs

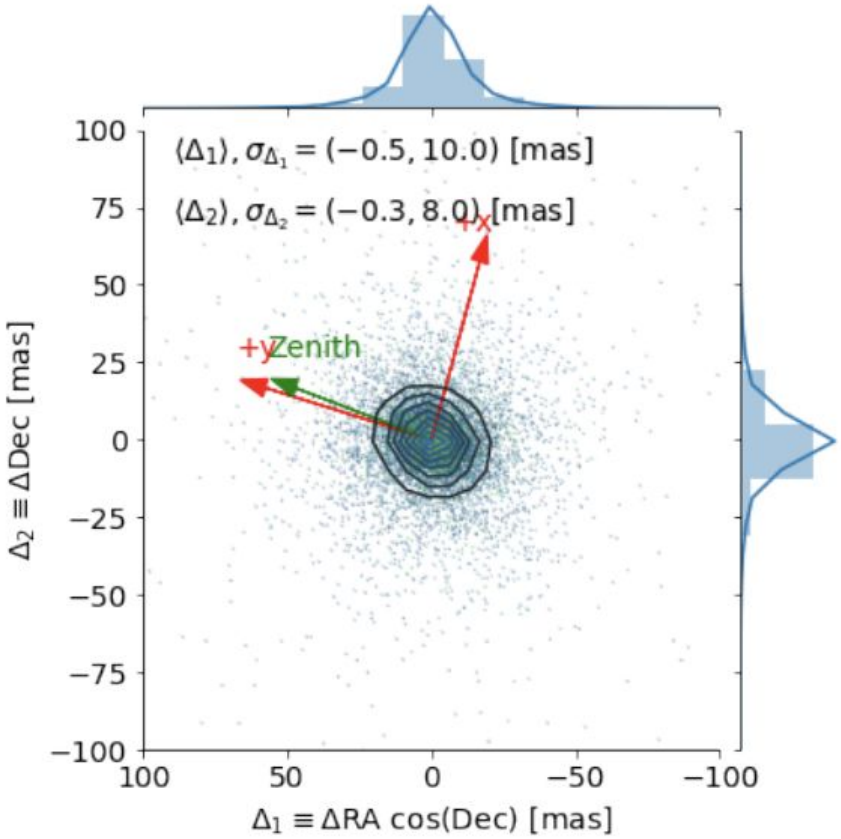


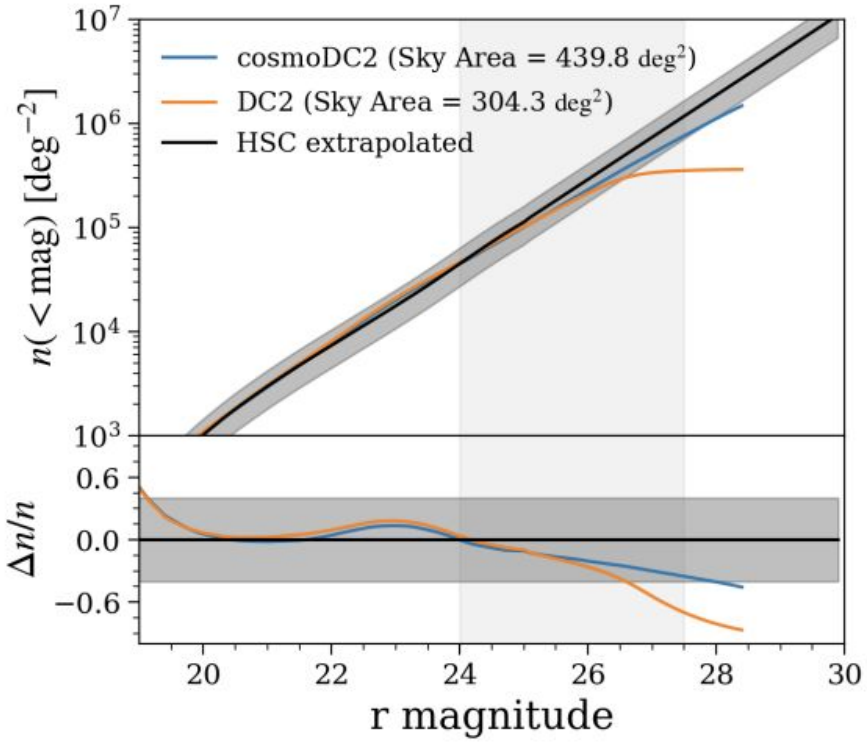


Validating through Many Eyeballs

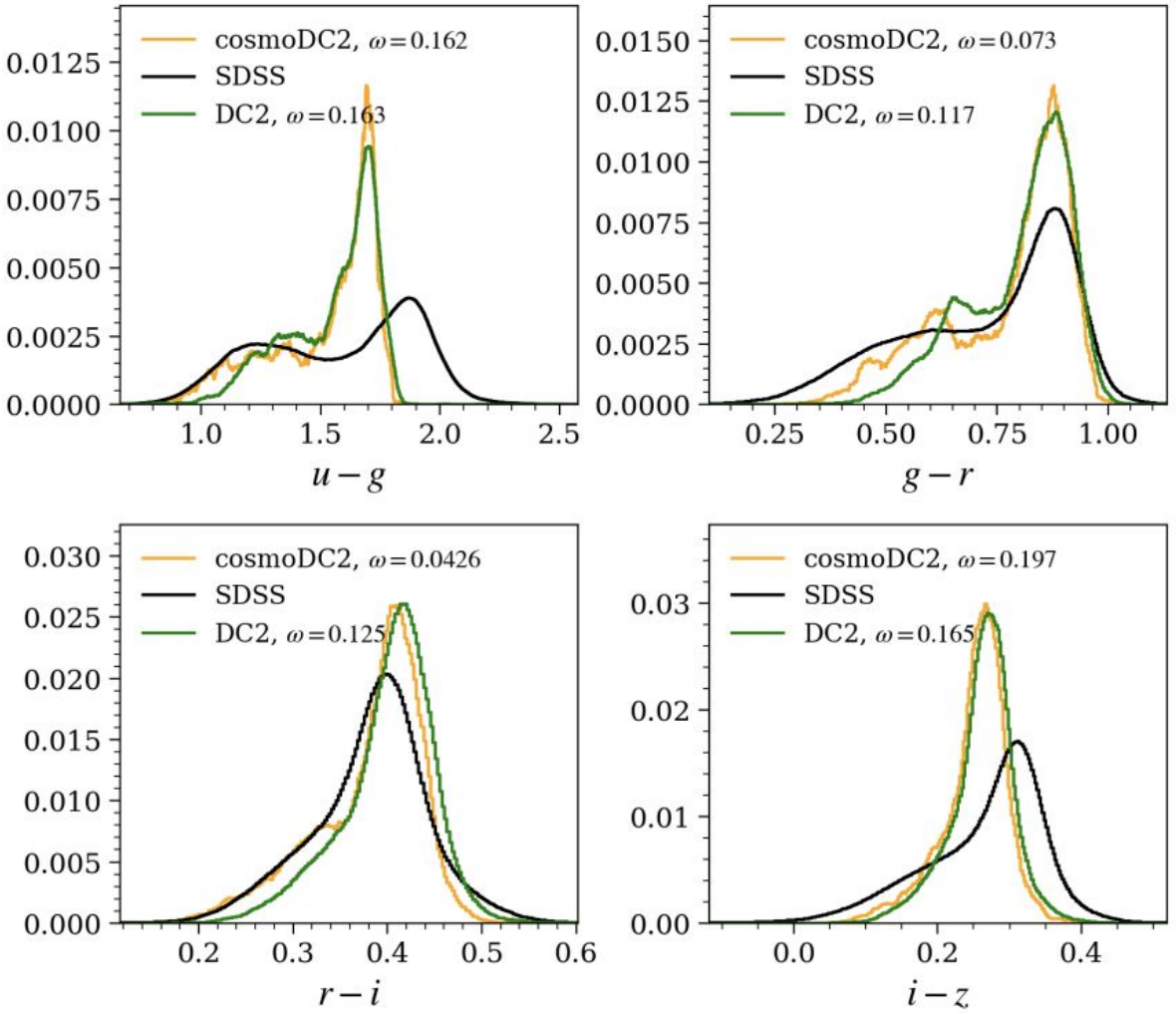


Some Quantitative Validation





And Compare to Observations



What Can We Take Forward?

Hard Earned Lessons

1. Parsl driver needs to be run on a stable and persistent resource — or else lose the connection to potential workers
    - The Parsl driver can be manually reconnected, but this can be rough on whoever is stuck monitoring things (**me**)
2. Containers work, but may have some unquantified costs
    - How much do you lose from a more generic Python installation versus something architecture specific?
    - How much do container start-up times matter for your application? What about the memory footprint?
3. Parallelizing by visit means slow sensors may lose compute time
    - Parsl only replaces work based on a complete container being finished, not per sensor
    - Sensors with more objects may delay a node substantially

Potential Improvements

- Utilize cloud resources to run our Parsl driver on
  - could potentially leverage multiple compute resources if authentication can be worked out
  - probably at least as stable as some HPC resources
  - Minimizing a need to overload input files may be tricky
- Further container optimizations
  - can likely pass Python library information into the container to leverage architecture specific builds
- Code refactoring for scaling
  - imSim developers are exploring leveraging GPUs as well as refactoring the code around individual sensors rather than visits

What does the future bring?

- Still to be seen!
- DESC members are using DC2 products to validate science pipelines
- Parsl being used in multiple ways within DESC
  - image processing
  - theory pipelines
- Smaller, more complex simulations may be needed going forward

Related Challenge: Data Serving

One main advantage to DC2 is it taught us a lot about data management.

- Large amount of inputs and outputs to track provenance of across multiple sites
- Complicated science workflow to document
- Use of PostgreSQL and Apache Spark for data exploration
- Internal Python based reader developed to go through inputs

A lot of these lessons will carry forward, especially as we work with our own In-Kind Contributions, where we may leverage compute resources outside of the US.

Thank you for listening!