Making Research Data Flow with Python

Josh Borrow
University of Pennsylvania & Simons Observatory

Proceeding co-authors: Paul La Plante, James Aguirre, Peter K. G. Williams

Penn

SIMONS OBSERVATORY

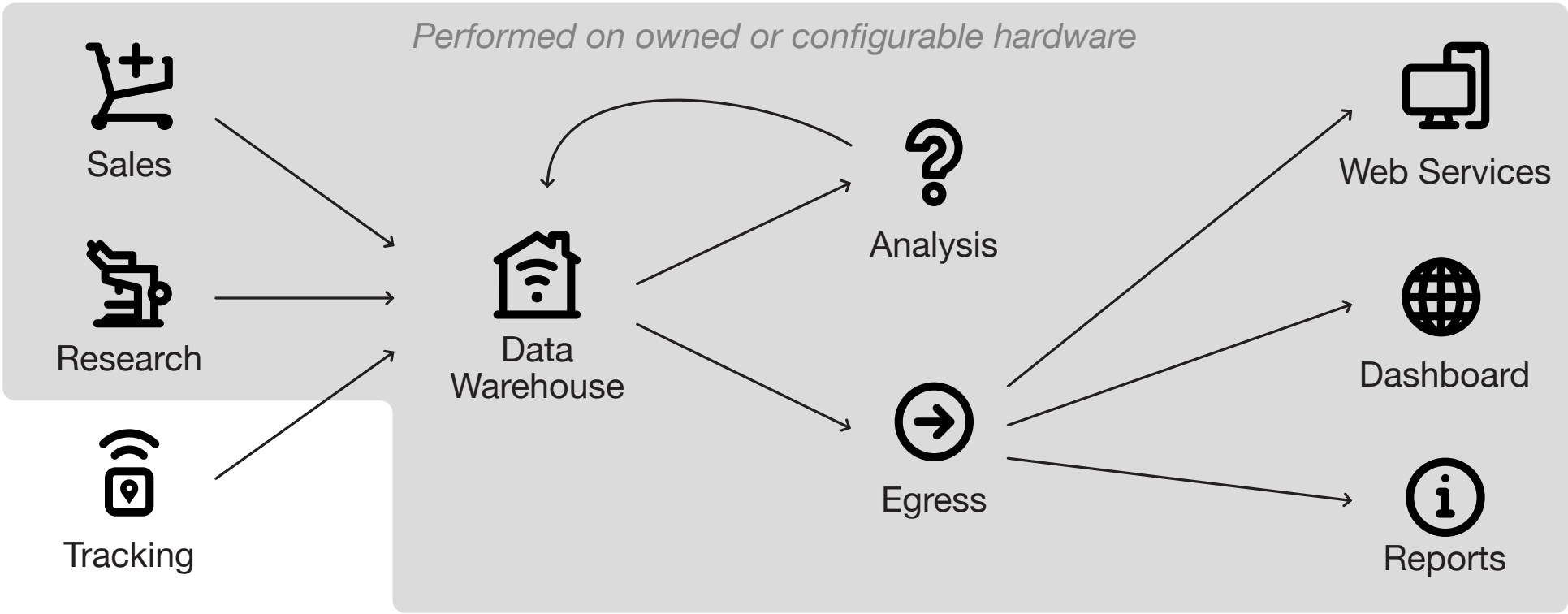
SciPy 2024


Gabriele Coppi, Rolando Dunner, Federico Nati, Matias Rojas

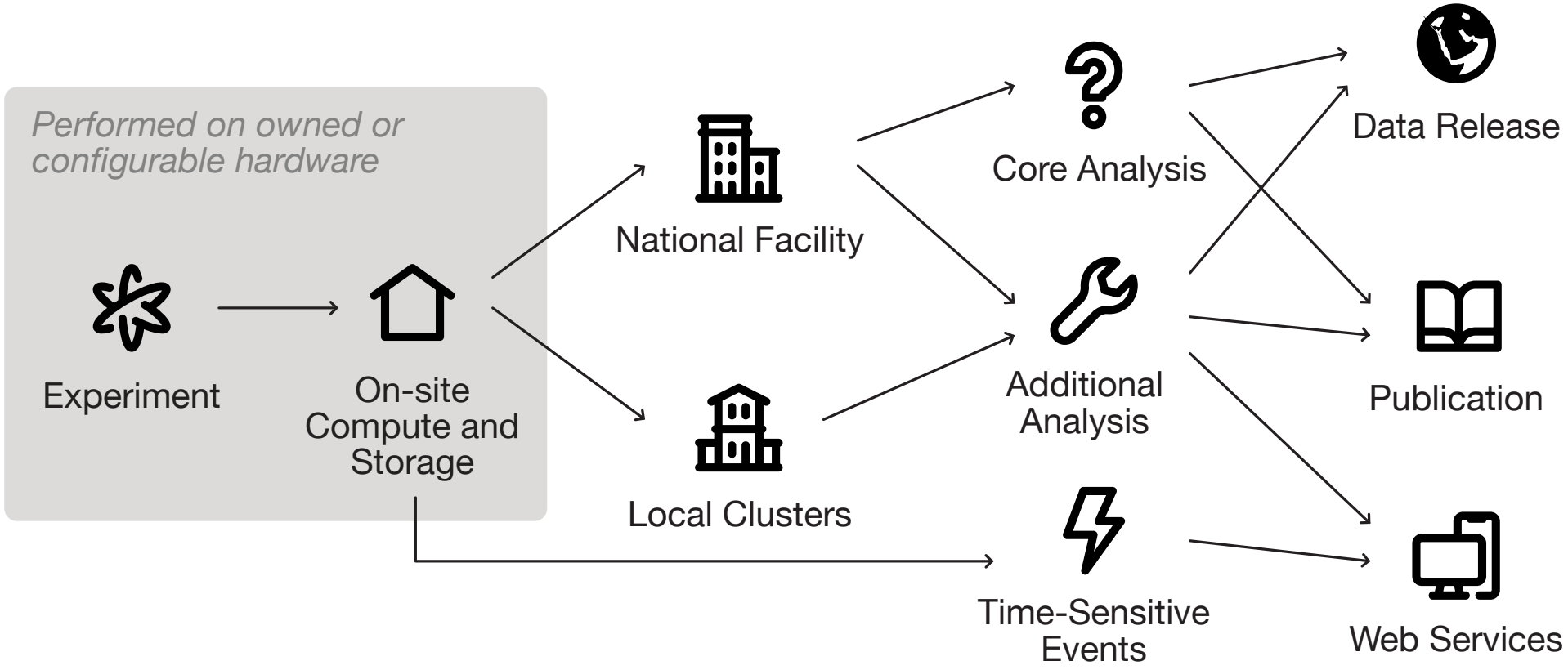
Data volumes and processing

- In sum, the observatory in full running mode produces ~20 TB of data a week (~1 PB/year).
- This data needs to be moved off-site for processing and long term storage.
  - Making one ‘map’ may use an entire year’s worth of data simultaneously.
  - Such map-making additionally needs high performance computing resources that are not available on-site.
  - Data is extremely precious: you can’t go back in time!

Differing hardware landscapes



Differing hardware landscapes

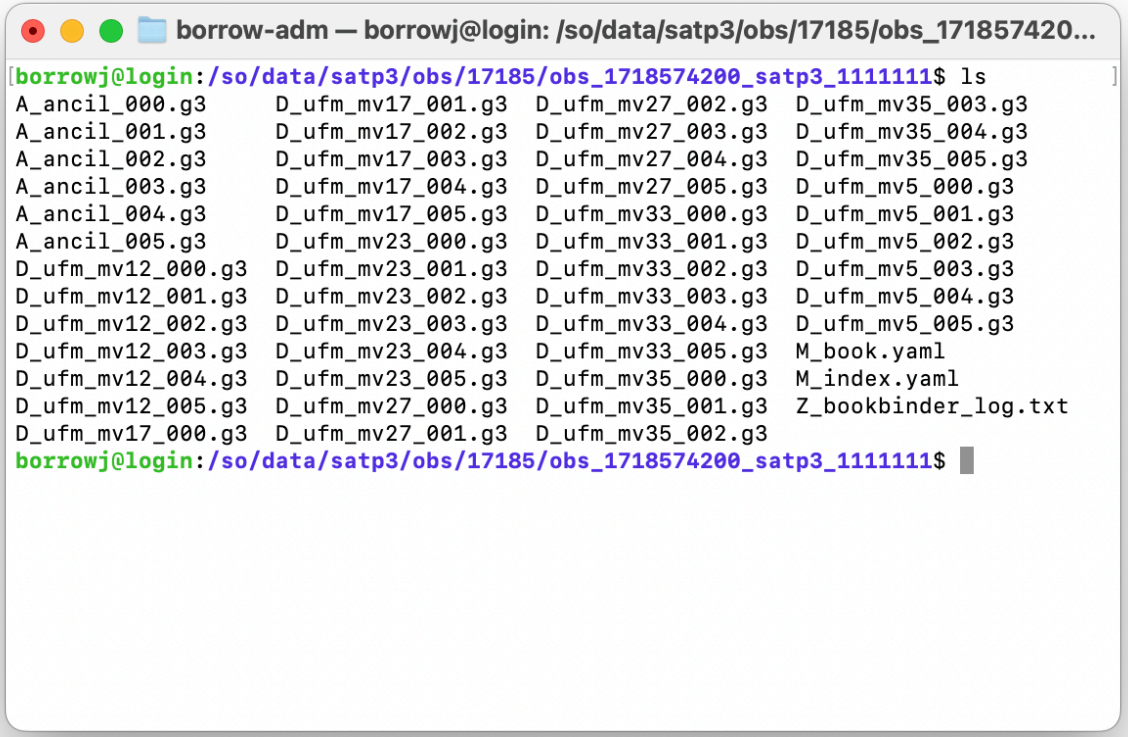


NERSC's Spin

- Spin is a Kubernetes cluster that can be used by NERSC projects to host persistent applications.
- These applications can have access to the high-performance filesystems at NERSC!
- We host our docker images here, alongside a number of other services.

What does that data look like..?

- Data is pre-processed at site into chunked ~10-20 GB ‘books’ (essentially a directory filled with raw data files).
- This is organized using a pre-determined filesystem structure.
- Our core goal: make this filesystem structure appear on HPC machines across the globe.



Preferred transfer: Globus

- Globus is a data transfer service designed primarily for high performance computing.
- It provides a very comprehensive python API for orchestrating specific transfers.
- Every system that we work with has globus installed.
- Think of globus as rsync but with some bells and whistles…

Troubles with mountains

- Being remote is great for astronomy, but is a problem when it comes to internet access!
- We (recently) gained a fiber link to North America, but for a very long time (and potentially in the future) we used a 50 Mbps radio link.

Getting bits down the mountain: SneakerNet

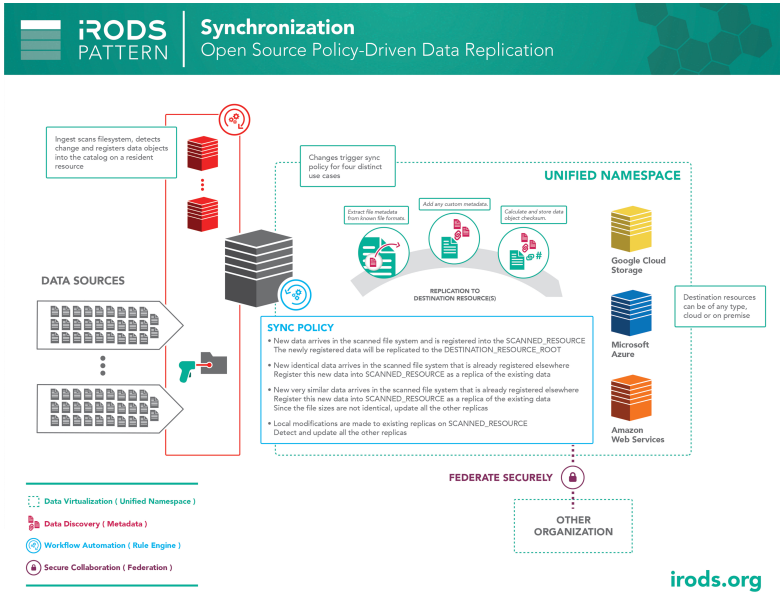
- Satellite internet is expensive, but grad students are cheap..?
- It’s everyone’s (even the PI at times!) job to carry disks down the mountain.
- We fill these up with the past n days of data, and hand-carry them to North America in hard cases.
- This mode of data movement doesn’t exactly have ‘built in’ support from most tools…

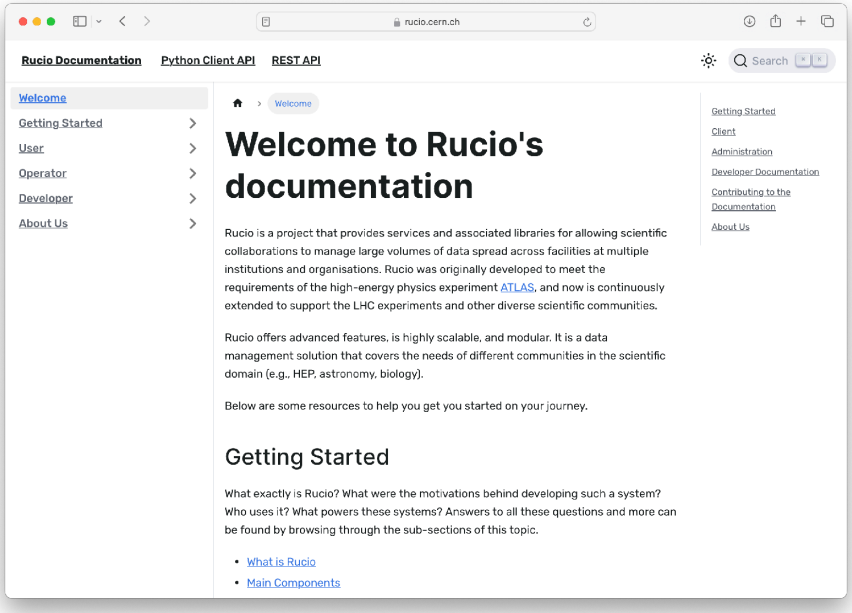
The affectionately-named Toaster

Existing tools for data management

iRODS®

RUCIO





The downfall of rule-based systems

- Rule-based systems are fantastic for allowing fast access to data and to ensure redundancy and failover.
- Users of these systems specify abstractions about the data; e.g. “I would like ten copies of this to exist at all times”. If a copy is lost to drive failure, a new one is automatically created at another site.
- This assumes available bandwidth >> system throughput!

Our service layout

We fly the disks from Chile to somewhere in the U.S. (here UCSD).

Data is then sent around the globe using globus.

At each site here, we have a librarian server running.

What makes a librarian?

- At the core of the librarian is a python-based web server.
- We chose FastAPI because of its ease of use, and its tight integration with pydantic.
- pydantic is a data validation library that helps you ‘strongly type’ your objects…

&

See also:

Pyramid

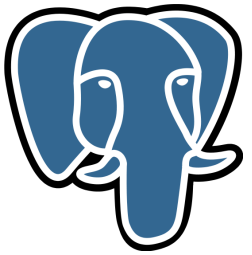
Anatomy of a FastAPI router

- FastAPI allows you to define ‘routers’ that allow you to respond to queries to specific page path.
- One object comes in, and another goes out.

```python
from pydantic import BaseModel
from fastapi import app

class InputModel(BaseModel):
    caller_name: str
    caller_id: int

class OutputModel(BaseModel):
    response_to_caller: str

@app.post("/hello/world")
def hello_world(request: InputModel) -> OutputModel:
    return OutputModel(
        response_to_caller=f"Hello {request.caller_name}!"
    )
```

Persistent metadata storage

- Databases, unsurprisingly, are our preferred metadata storage system.
- Combined with an ‘Object-Relational Mapping’ library (ORM), they allow you to store and retrieve python objects easily.



&

SQLAlchemy

See also:

Beanie

SQLModel

For tighter pydantic integration

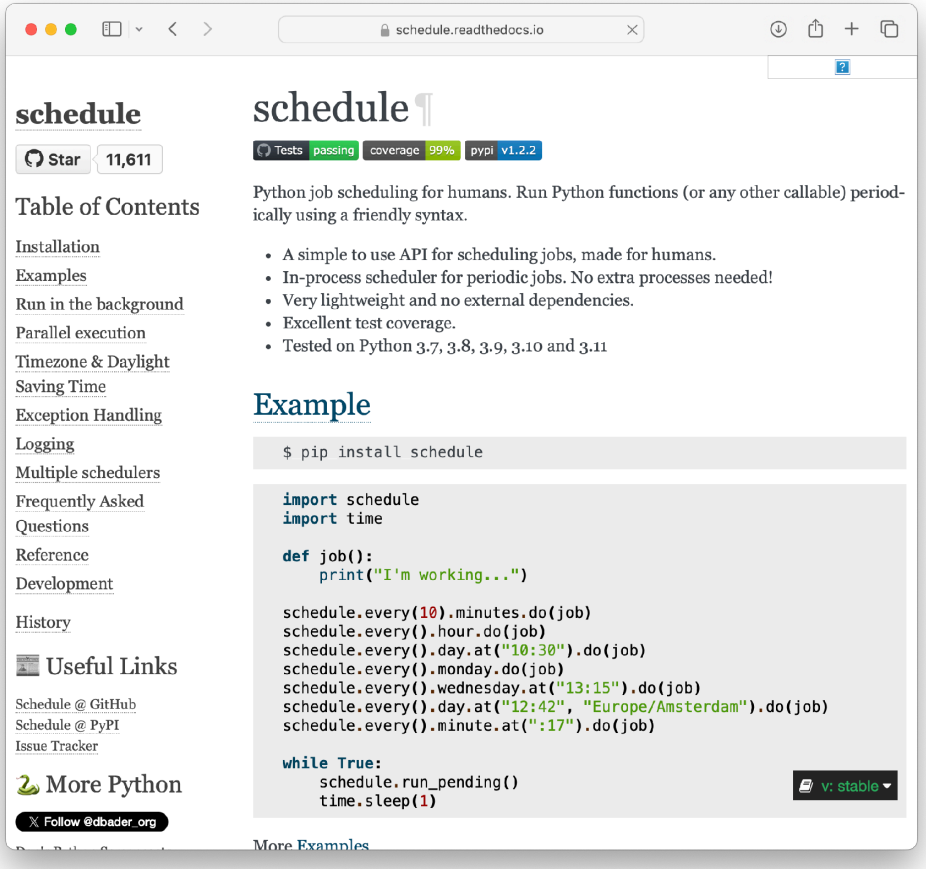
Anatomy of an ORM class

- Our ORM classes look pretty much exactly like a regular python class, with some additional magic added on to indicate their types in the database.
- A key note: relationships in databases are very useful!

```python
class Instance(db.Base):
    """
    Represents an instance of a file on a Store. Files are unique, Instances are not;
    there may be many copies of a single 'File' on several stores.
    """

    __tablename__ = "instances"

    # NOTE: SQLite does not allow autoincrement PKs that are BigIntegers.
    id = db.Column(db.Integer, primary_key=True, autoincrement=True, unique=True)
    "The unique ID of this instance."
    path = db.Column(db.String(256), nullable=False)
    "Full path on the store."
    file_name = db.Column(db.String(256), db.ForeignKey("files.name"), nullable=False)
    "Name of the file this instance references."
    file = db.relationship(
        "File",
        back_populates="instances",
        primaryjoin="Instance.file_name == File.name",
    )
    "The file that object is an instance of."
    store_id = db.Column(db.Integer, db.ForeignKey("store_metadata.id"), nullable=False)
    "ID of the store this instance is on."
    store = db.relationship(
        "StoreMetadata", primaryjoin="Instance.store_id == StoreMetadata.id"
    )
```

Background tasks

- schedule is an extremely lightweight python scheduling library.
- It allows you to run python functions on a fixed cadence.
- We use it on the web server to check the database frequently for new ingested files, and send them out to other librarians.



Configuring it all…

- pydantic-settings makes configuration easy. You can read your config from:
  - A structured file (e.g. JSON)
  - Environment variables
- Your settings are all validated and made the correct types (thanks pydantic!) on startup.

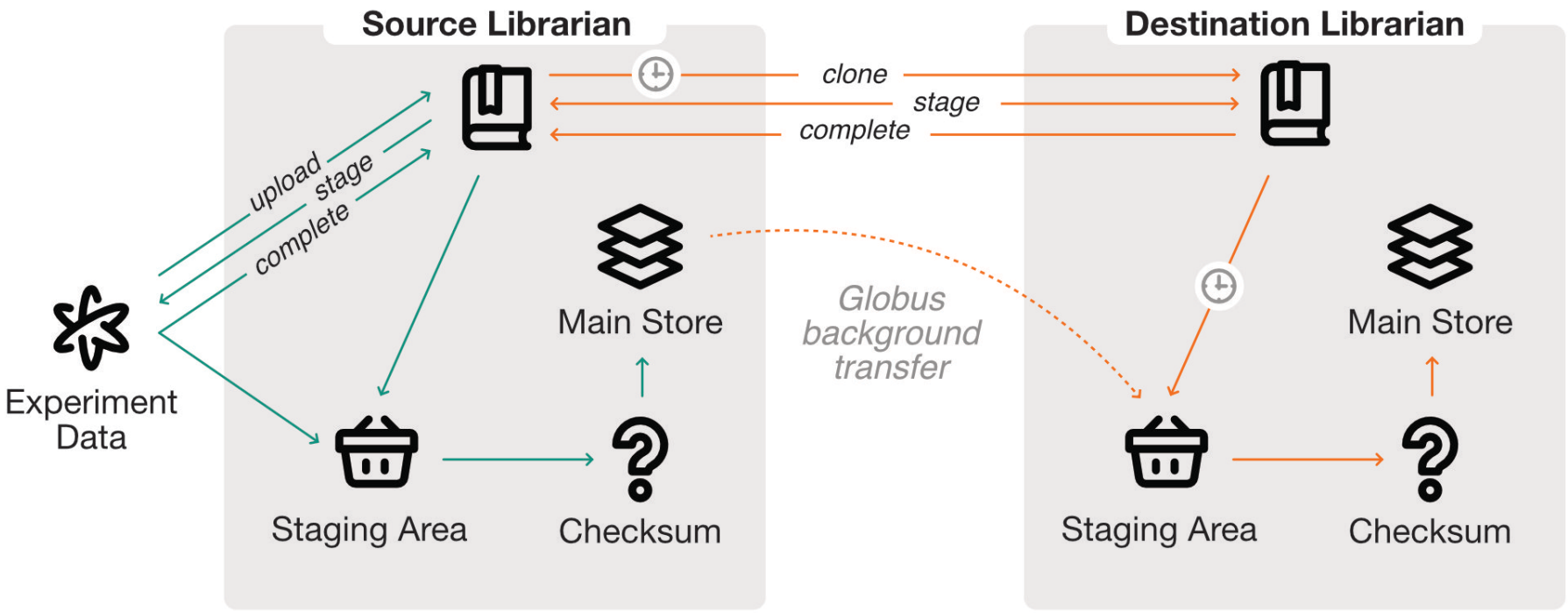
```python
class ServerSettings(BaseSettings):
    """
    Settings for the librarian server. Note that because this is a BaseSettings
    object, you can overwrite the values in the config file with environment
    variables.
    """

    # Top level name of the server. Should be unique.
    name: str = "librarian_server"

    # Whether to enable debugging features, like the API docs and OpenAPI schema.
    debug: bool = True

    # Encryption key for the server, for connecting to other librarians.
    # Don't write this in the config file, read it from a file whose name
    # you set as the encryption_key_file.
    encryption_key: Optional[str] = None
    encryption_key_file: Optional[Path] = None

    # Database settings.
    database_driver: str = "sqlite"
    database_user: Optional[str] = None
    database_password: Optional[str] = None
    database_host: Optional[str] = None
    database_port: Optional[int] = None
    database: Optional[str] = None

    database_password_file: Optional[Path] = None

    log_level: str = "DEBUG"
```

Getting data into the librarian

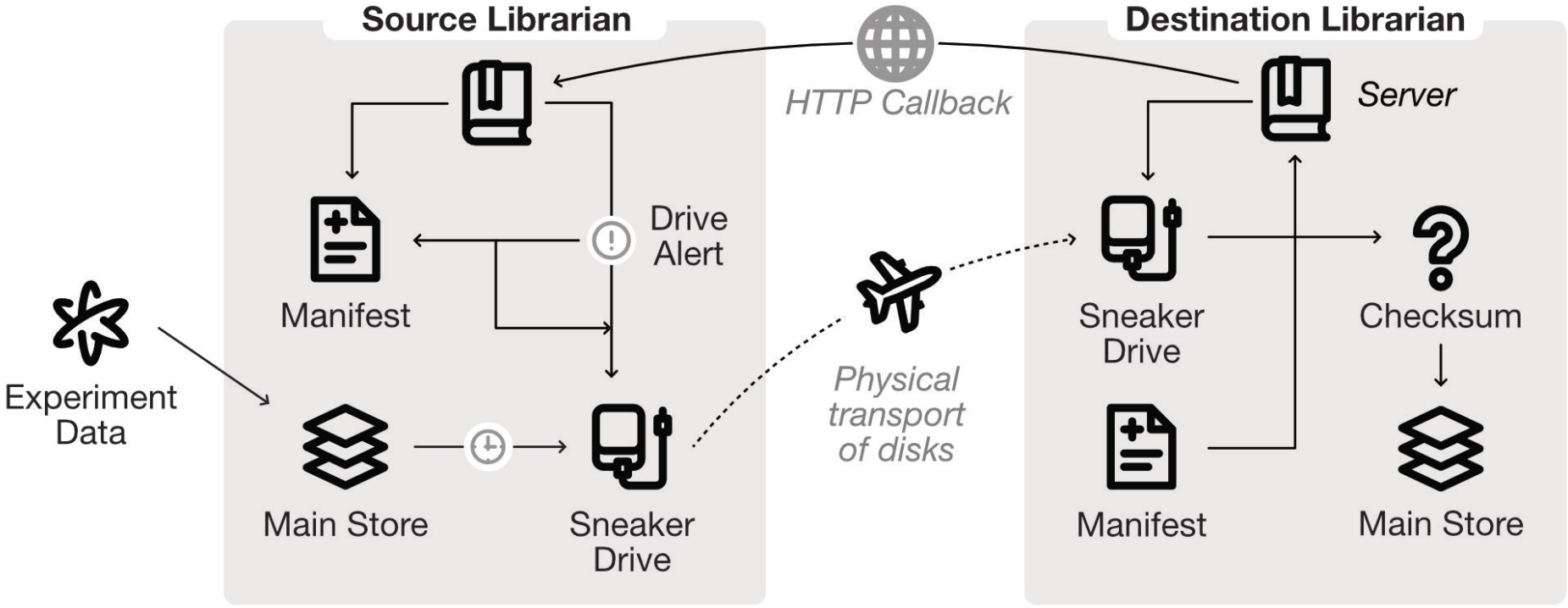
- Uploading from python is designed to be very easy.
- This is mainly performed from our data packaging prefect workers.
- Once data is in the system, and placed in its rightful place in the filesystem, the background tasks take over.

```python
from hera_librarian import LibrarianClient
from hera_librarian.settings import client_settings
from pathlib import Path

conn = client_settings.connections.get(
    "my_librarian_name"
)

conn.upload(
    Path(local_file),
    Path("/hello/world/this/is/a/file.txt")
)
```

Inter-Librarian sharing with Globus



Inter-Librarian sharing SneakerNet



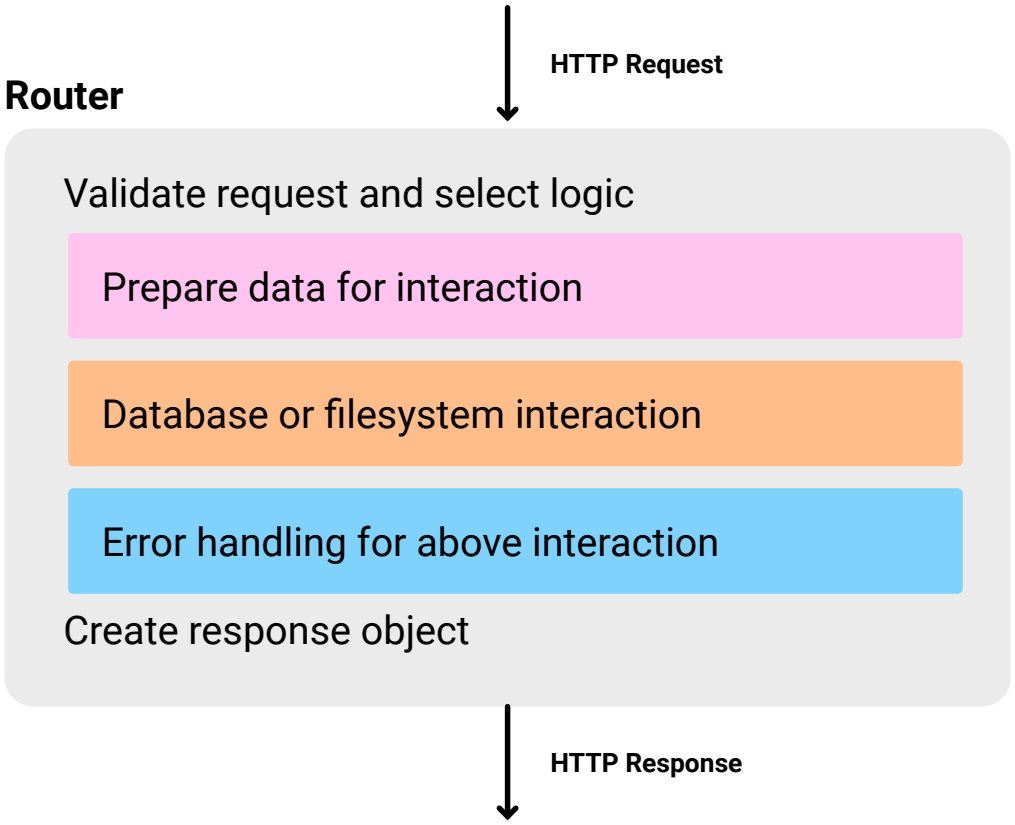
Lessons learned: Dependency injection

- Allowing different providers to conform to the same code structure means you can ‘inject’ them into your program as arguments.
- Example: globus transfer versus a simple local copy. They both just provide a `transfer` function that copies a file from A to B…

```python
class CoreTransferManager(BaseModel, abc.ABC):
    @abc.abstractmethod
    def transfer(self, local_path: str, remote_path: str):
        """
        Transfer a file from the local machine to the store.

        Parameters
        ----------
        local_path : str
            Path to the local file to upload.
        store_path : str
            Path to store file at on destination host.
        """
        raise NotImplementedError

    @property
    @abc.abstractmethod
    def valid(self) -> bool:
        """
        Whether or not this transfer manager is valid for the
        current system we are running on.
        """
        raise NotImplementedError
```

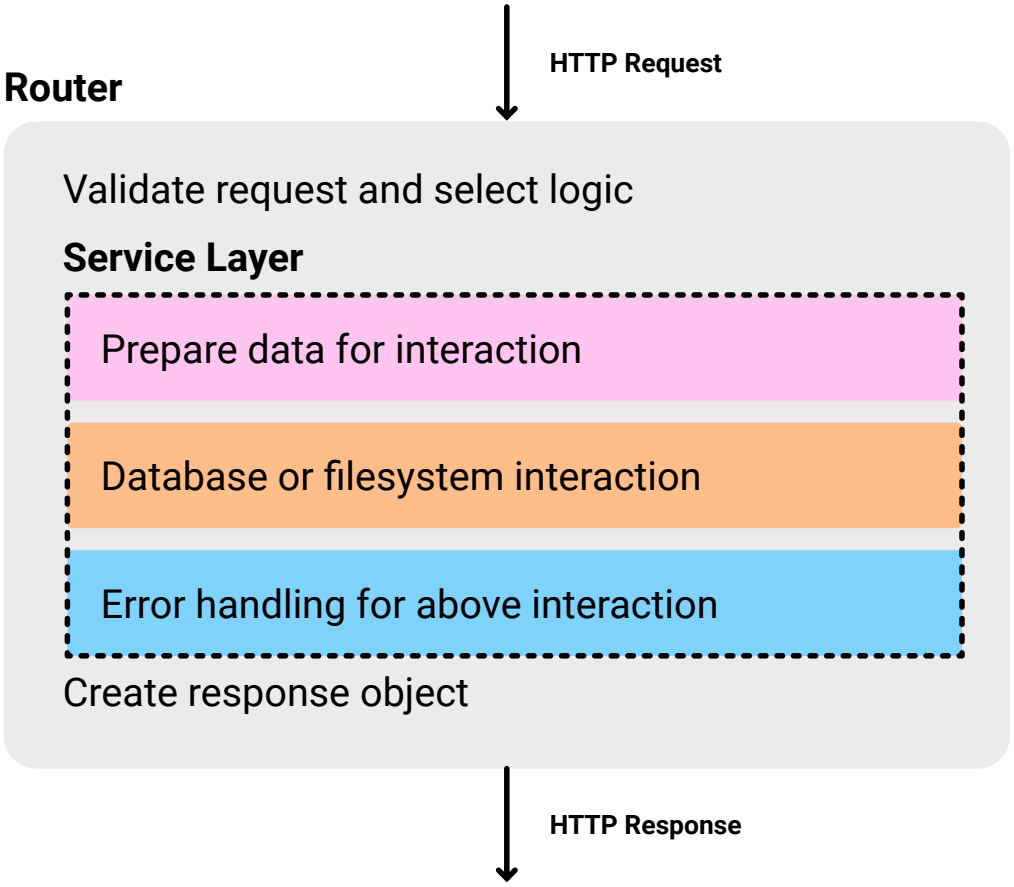
Lessons learned: Service layers

- Having a service layer in your application increases code volume but ends up decreasing code complexity.
- Core benefit: this keeps your routing functions simple.
- Can easily re-factor for different dependencies without touching every router.
- Can re-use service layer in jobs…



Lessons learned: Service layers

- Having a service layer in your application increases code volume but ends up decreasing code complexity.
- Core benefit: this keeps your routing functions simple.
- Can easily re-factor for different dependencies without touching every router.
- Can re-use service layer in jobs…



Lessons learned: Testing

- Testing is really hard for these kind of data management applications.
- Unit testing is helpful but only to a point (even more limited than usual!); you simply depend on too many components.
- Integration testing and end-to-end testing are much more helpful.
- Massive pain point: Globus is a nightmare to test with - need to use dependency injection.

Lessons learned: Grab a full server to test with

- pytest-xprocess is amazing for end-to-end testing.
- You can spin up an entire web-server as a pytest fixture, and interact with it.
- Strongly recommend this, alongside testcontainers for ‘real’ dependencies.

```python
@pytest.fixture(scope="package")
def server(xprocess, tmp_path_factory, request) -> Server:
    """
    Starts a single server with pytest-xprocess.
    """

    setup = server_setup(tmp_path_factory, name="live_server")

    class Starter(ProcessStarter):
        pattern = "Uvicorn running on"
        args = [sys.executable, shutil.which("librarian-server-start"), "--setup"]
        timeout = 10
        env = setup.env

    for label, key in setup.env.items():
        if key is None:
            raise ValueError(f"Environment variable {label} is None.")

    xprocess.ensure("server", Starter)

    setup.process = "server"
    yield setup
```

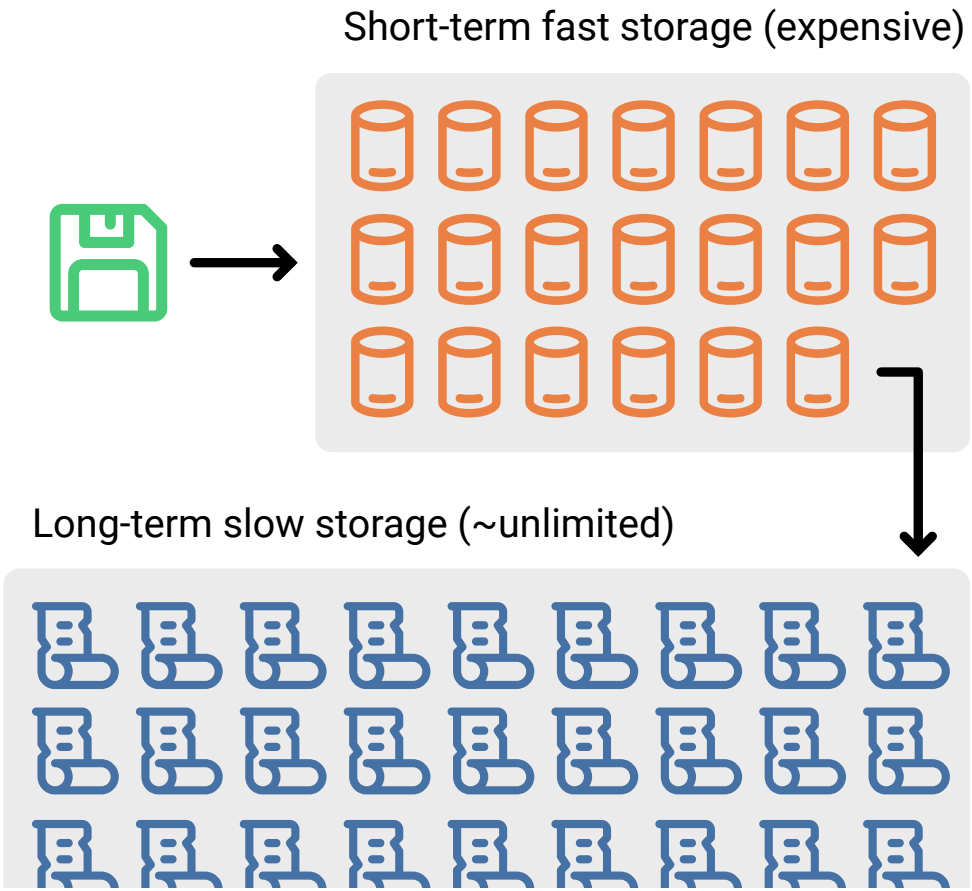
Summary & Takeaways

- The librarian is our simple open-source data orchestration framework.
- By putting together some simple components you can create a very flexible system tailored to your exact needs.
- Give web frameworks a go for your scientific workloads! This is all easier than you think…
- Your web applications do not need to be polished, perfect, things for them to be incredibly useful and solve real problems!

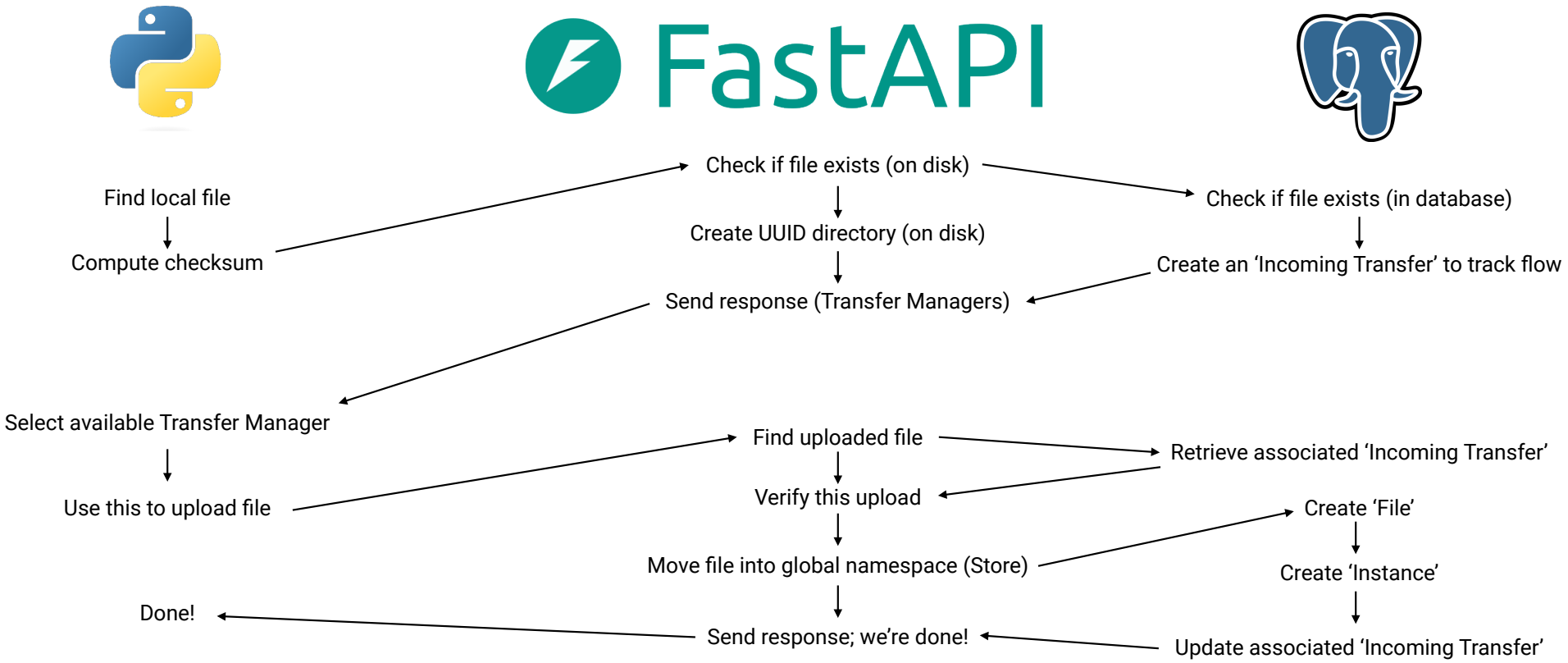
Backup Slides

Storage tiering

- Yet another complication: we do not have unlimited fast storage space (i.e. space that can be used for analysis).
- Data can come in to this fast disk, but at some point must be sent off to a slower storage tier.
- Each site may have different policies, and this is a manual process.



Anatomy of an upload

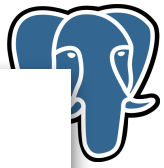


Anatomy of an upload

Find local file

↓

Compute checksum



```
response: UploadInitiationResponse = self.post(
    endpoint="upload/stage",
    request=UploadInitiationRequest(
        upload_size=get_size_from_path(local_path),
        upload_checksum=get_md5_from_path(local_path),
        upload_name=dest_path.name,
        destination_location=dest_path,
        uploader=self.user,
    ),
    response=UploadInitiationResponse,
)
```

Anatomy of an upload

```python
# Check that the upload is not already on the librarian.
if File.file_exists(request.destination_location):
    log.debug(
        f"File {request.destination_location} already exists on librarian. Returning error."
    )
    response.status_code = status.HTTP_409_CONFLICT
    return UploadFailedResponse(
        reason="File already exists on librarian.",
        suggested_remedy=(
            "Check that you are not trying to upload a file that already exists on the librarian, "
            "and if it does not choose a unique filename that does not already exist."
        ),
    )
```

Anatomy of an upload

Find local fil

↓

Compute check

```
# Stage the file.
def stage(self, file_size: int, file_name: Path) -> tuple[Path]:
    if file_size > self.free_space:
        raise ValueError("Not enough free space on store")

    stage_path = Path(f"{uuid.uuid4()}")

    # Create the empty directory.
    resolved_path = self._resolved_path_staging(stage_path)

    if self.group_write_after_stage:
        resolved_path.mkdir(mode=0o775)
        os.chmod(resolved_path, 0o775)
    else:
        resolved_path.mkdir()

    return stage_path, resolved_path / file_name
)
```

sts (in database)

↓

Transfer' to track flow

Anatomy of an upload

Find local file

↓

Compute checksu

Select available Transfer

↓

Use this to upload f

atabase)

' to track flow

```python
used_transfer_manager_name: Optional[str] = None

for name, transfer_manager in transfer_managers.items():
    if transfer_manager.valid:
        try:
            transfer_manager.transfer(
                local_path=local_path, remote_path=remote_path
            )

            # We used this.
            used_transfer_manager_name = name
            break
        except PermissionError:
            raise LibrarianError(f"Could not set permissions on {remote_path}")
    else:
        print(f"Warning: transfer manager {name} is not valid.")

if used_transfer_manager_name is None:
    raise LibrarianError("No valid transfer managers found.")

return used_transfer_manager_name
```

Anatomy of an upload

```
if (
    info.size != transfer.transfer_size
    or info.md5 != transfer.transfer_checksum
):
    # We have a problem! The file is not what we expected. Delete it quickly!
    self.store_manager.unstage(staging_directory)

    transfer.status = TransferStatus.FAILED
    session.commit()

    raise ValueError(
        f"File {staged_path} does not match expected size/checksum; "
        f"expected {transfer.transfer_size}/{transfer.transfer_checksum}, "
        f"got {info.size}/{info.md5}."
    )
```

Find lo

Compute

Select available T

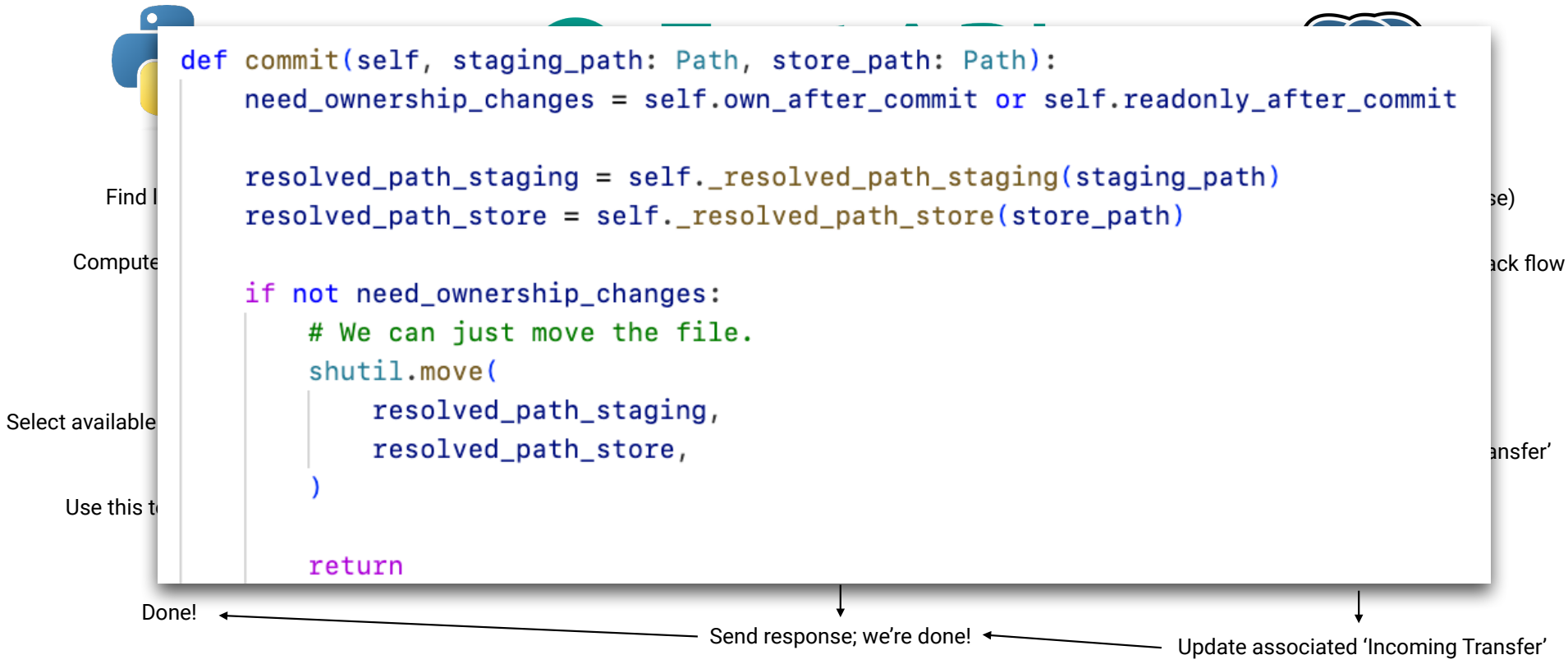
Use this to

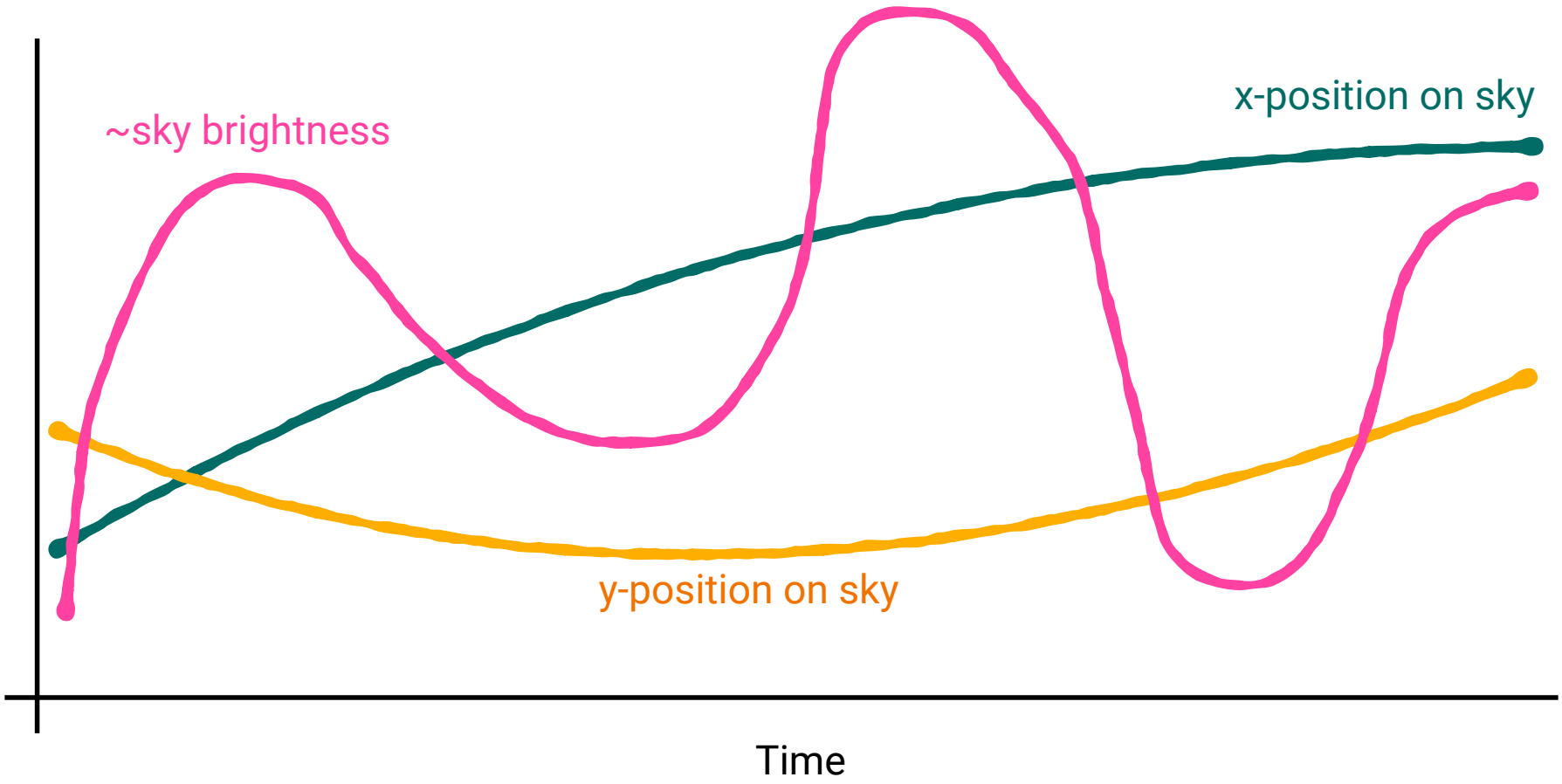
ase)

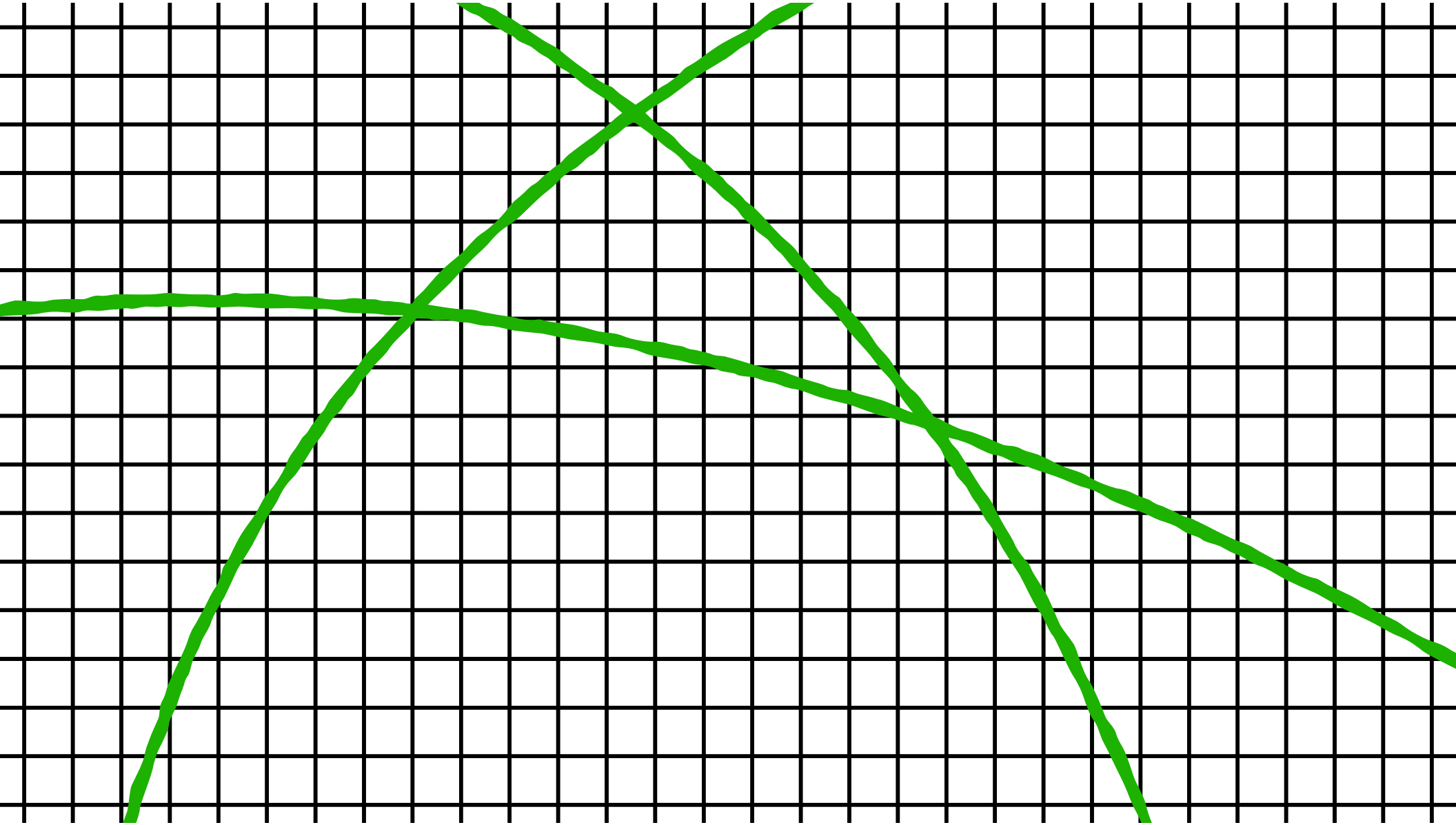
rack flow

ransfer'

Anatomy of an upload

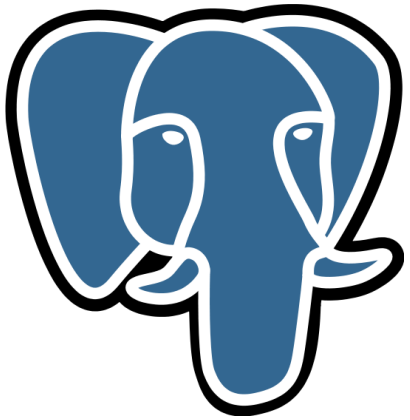



~sky brightness
x-position on sky
y-position on sky
Time



Data tracking

- Two universal rules for software development:
  - Never write your own database
  - Postgres is always enough
- We use SQLAlchemy to transform our abstractions to python objects that can be stored in a postgres database.
- SQLAlchemy allows us the flexibility to use SQLite for testing.



SQLAlchemy

System interaction

- We use a HTTP server as the primary touchpoint for clients.
- FastAPI is designed around building APIs easily (who'd have guessed..?) rather than web pages.
- In the abstract: allows you to consume python objects over the web, run arbitrary python code, and return a python object.