dagster

Dagster Deep Dive

Cooking with Gas: Building a Data Platform at US Foods

April, 1st 2025

Table of contents

dagster

Speakers

Lee Littlejohn

Lead Machine Learning Engineer

Alex Noonan

Developer Advocate

dagster

Disorganization is the Enemy

Platform Principles

Platform Principles

Disorganization is the Enemy

dagster

Platform Principles

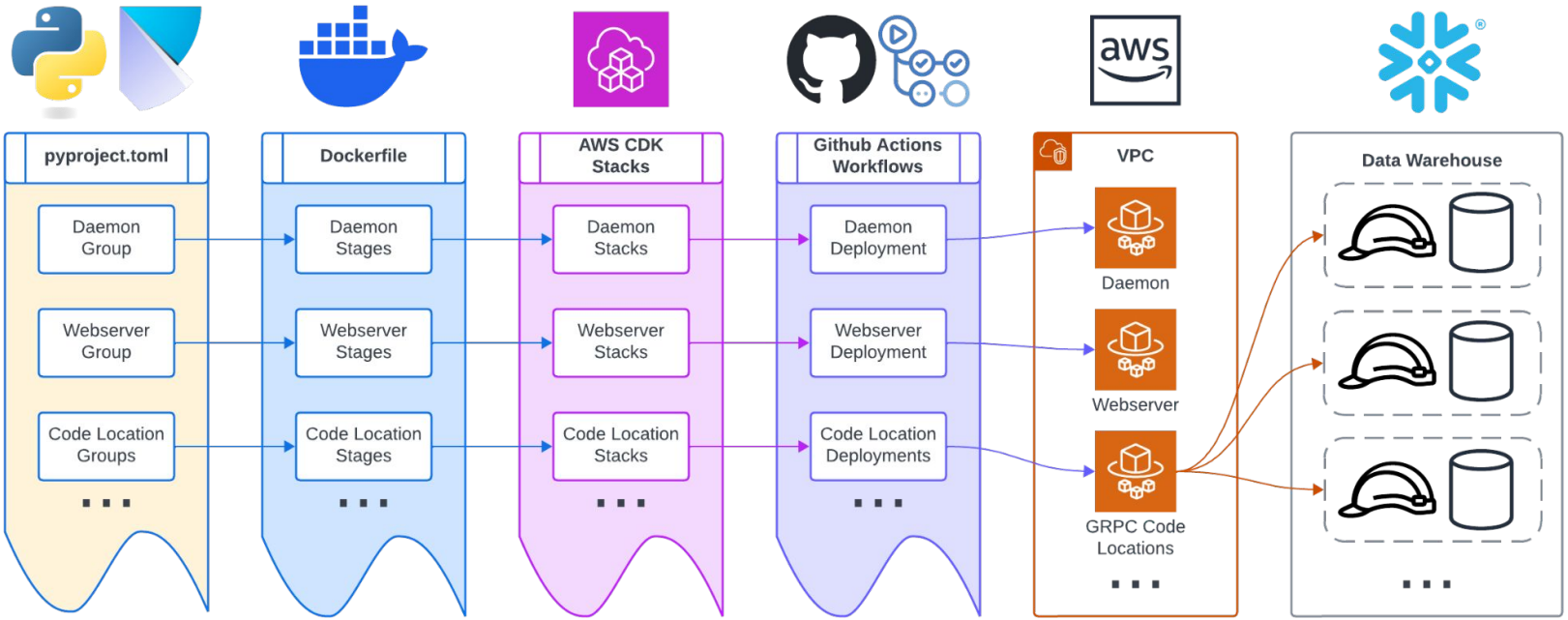
Efficiency at Every Level

- Reduce cognitive load
  - Focus on competency area
- Keep things maintainable
- Build in efficiency with modular design
- Establish single source of truth
- Share parallel patterns
- Minimize action, maximize output
- Transparency

"Always be knolling."

dagster

The Big Picture



dagster

Pain Points

Pain Points

Common Issues

Isolated Data Silos

Difficulty aligning cross-functional teams

Resources and Access Friction

Inconsistent access roles and unstandardized methods

Poor Overall Visibility

Redundant work, duplicate code and data

Suboptimal Governance

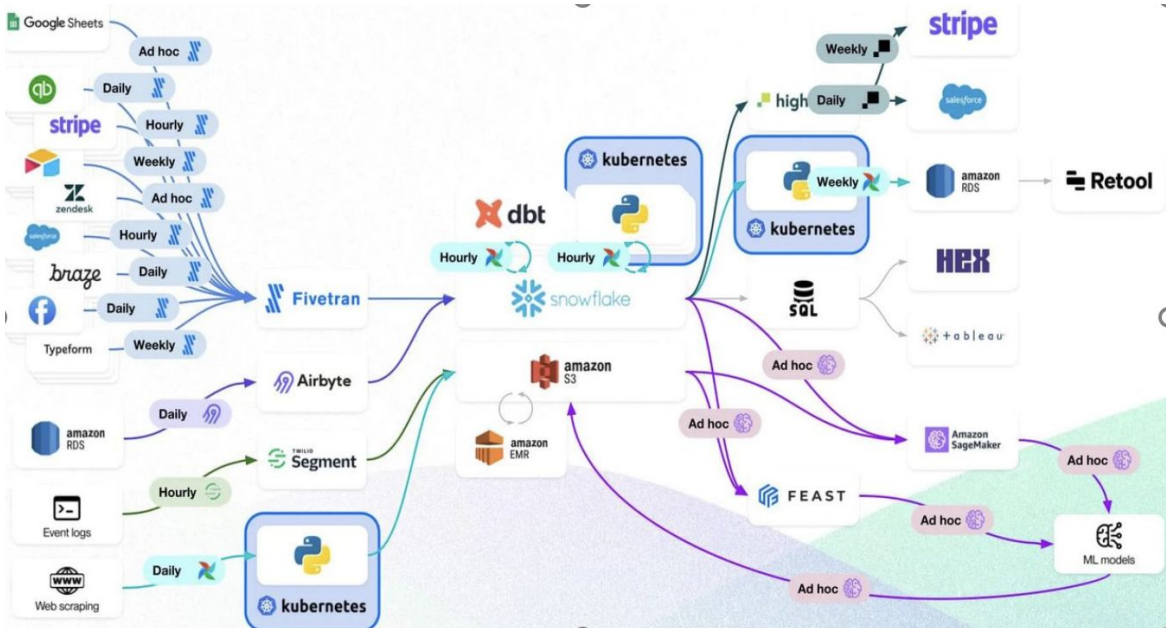
Loose environmental and database boundaries

dagster

Pain Points

Where We Were

- Fragmented “DAGs” in Alation
- Ad hoc scripting
- Infrastructure soup
- “cust_data_dev_smith”



dagster

Chefs Love the Kit

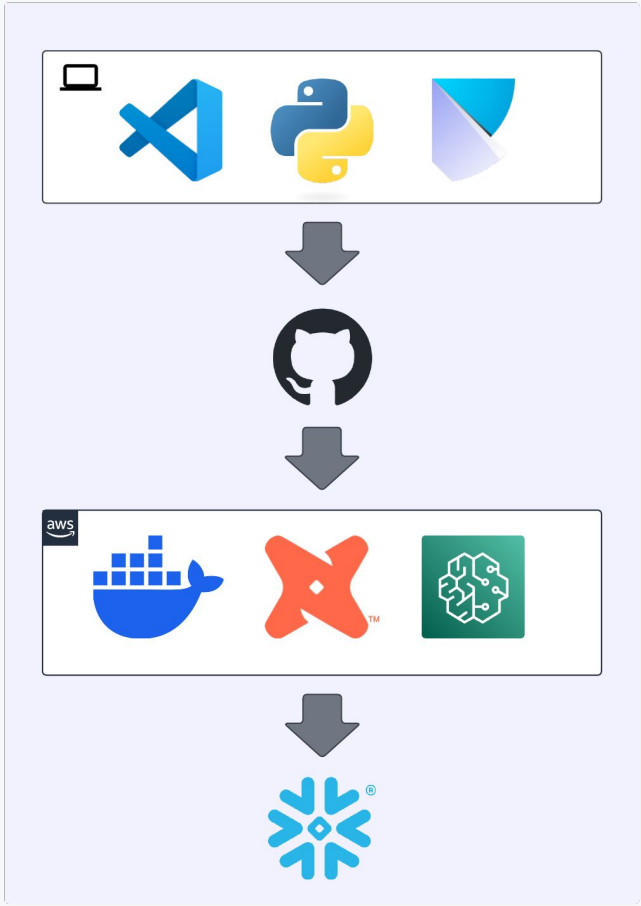
Tool Selection

Tool Selection

Why Dagster?

- Asset-based orchestration
- Rich UI and monitoring
- Extensive integration capabilities
- Strong typing
- Familiar interface and dependency injection
- Anything Python can be done in Dagster
- VISIBILITY
- Local/cloud development parity

Supporting Tools



dagster

We Built This City

Platform Foundation

Poetry & pyproject.toml

The First Source of Truth

- Everything else is downstream of this file
- Dependency groups for modular deployment
- Consistency across environments - local, development, staging, production
- No multiple requirements.txt file sprawl
- Streamlined task execution with Poe

```
[tool.poetry.dependencies]
python = ">=3.11, <3.13"
boto3 = "^1.28.20"
snowflake-connector-python = {extras = ["pandas", "secure-local-
storage"], version = "^3.12.3"}
...

[tool.poetry.group.daemon.dependencies]
dagster-postgres = "^0.25.4"
...

[tool.poetry.group.saute.dependencies]
statsforecast = "^1.7.4"
...

[tool.poetry.group.infra.dependencies]
aws-cdk-lib = "2.162.0"
...

[tool.pytest.ini_options]
...

[tool.ruff]
...

[tool.poe.tasks.dev]
help = "Starts a local dagster instance with the dev deployment
configuration. Requires at least one '-m' flag to specify a module to
load. ex: `poetry dev -m saute`"
cmd = "dagster dev -d ./src"
env = { DAGSTER_HOME = "${PWD}/dagster_home", DEPLOYMENT = "local" }
deps = ["_pwd", "_install_deps"]
uses = { PWD = "_pwd" }
```

dagster

Containerization Strategy

TODO catchy something

- Multi-stage Dockerfile
- Poetry dependency groups one-to-one mapping to container stages
- Build optimization techniques
  - Don’t invalidate the cache!
  - Use layer caching on ECR with Buildx

```
FROM python:3.11-slim AS dagster-base

# poetry setup
RUN curl -sSL https://install.python-poetry.org | python3 -
...

# dagster setup
ENV DAGSTER_HOME=$PYSETUP_PATH
# build context = project root, so the paths here are relative to that
COPY ./infra/workspace.yaml $DAGSTER_HOME/workspace.yaml
COPY ./infra/dagster.yaml $DAGSTER_HOME/dagster.yaml
WORKDIR $DAGSTER_HOME
...

# final daemon and webserver images
FROM dagster-base AS daemon
RUN poetry install --only daemon --no-root --no-cache

FROM dagster-base AS webserver
RUN poetry install --only daemon,webserver --no-root --no-cache
...

# final saute stages
FROM dagster-base AS saute-builder
RUN poetry install --only main,daemon,saute --no-root --no-cache

FROM dagster-base AS saute
COPY --from=saute-builder $VENV_PATH $VENV_PATH
COPY ./src/saute /src/saute/
...
```

dagster

Infrastructure as Code

AWS CDK

- Single upstream infrastructure stack to define shared components
  - infra buckets, load balancers, ECR repos, etc
- Team stacks define ECS Fargate components and other resources unique to each team

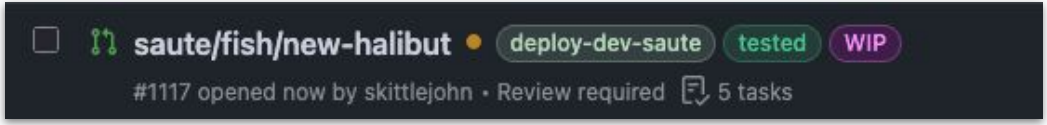
```python
IMAGE_TAG = os.getenv("IMAGE_TAG", "latest")  # branch name

# GRPC server image parameters
GRPC_ECR_REPO = "ia-dagster/saute"
GRPC_DIGEST = os.getenv("GRPC_DIGEST")
COMPLETE_IMAGE_TAG = ...

# get env/secrets keys from template.env to set them as CDK parameters
secrets_from_dotenv = dotenv_values("../template.env")
local_only_keys = ...
secrets_keys = [key for key in secrets_from_dotenv if key not in local_only_keys]

class SauteStack(Stack):
    """Stack for the Saute team's Dagster ECS service."""

    def __init__(self, ...):
        secrets_dict = {key: ecs.Secret.from_secrets_manager(secret, key) ...}
        env_dict = {...}
        self.log_group = logs.LogGroup.from_log_group_name(...)
        self.task_role = iam.Role(...)
        self.saute = ecs.FargateTaskDefinition(...)
        self.saute.add_container(
            container_name="saute",
            image=ecs.ContainerImage.from_ecr_repository(
                repository=ecr.Repository.from_repository_name(
                    repository_name=GRPC_ECR_REPO
                ),
                tag=COMPLETE_IMAGE_TAG,
            ),
            command=[
                "dagster",
                "code-server",
                "start",
                "-m",
                "saute",
            ],
            ...
        )
        self.saute_service = ecs.FargateService(...)
        self.saute_bucket = s3.Bucket(...)
        self.saute_reload_lambda_trigger = triggers.TriggerFunction(...)
```

dagster

CI/CD

Github Actions

- Team-agnostic label-based workflows
- Use team names to target correct upstream constructs
- Deployment pipeline
  - Builds container for specific team
  - Environment-specific configurations (dev/staging/prod)
  - Deploys corresponding CDK stack




```yaml
- name: Build and push to ECR
  id: build-push
  uses: docker/build-push-action@v5
  with:
    context: .
    file: ./infra/Dockerfile
    target: ${{ inputs.team_name }}
    tags: ${{ steps.ecr-login.outputs.registry }}/ia-dagster/${{ inputs.team_name }}:
${{ inputs.image_tag }}
    cache-from: type=registry,ref=${{ steps.ecr-login.outputs.registry }}/ia-dagster/
${{ inputs.team_name }}:buildcache
    cache-to: type=registry,mode=max,image-manifest=true,oci-mediatypes=true,ref=${{
steps.ecr-login.outputs.registry }}/ia-dagster/${{ inputs.team_name }}:buildcache
    push: true
...

- name: Install infra dependencies
  run: poetry install --only infra --no-root
...

- name: Deploy stack
  id: deploy
  working-directory: ./infra
  run: poetry run cdk deploy ${{ inputs.stack_name }} --require-approval never -c
env=$DEPLOYMENT
```

dagster

How we do it

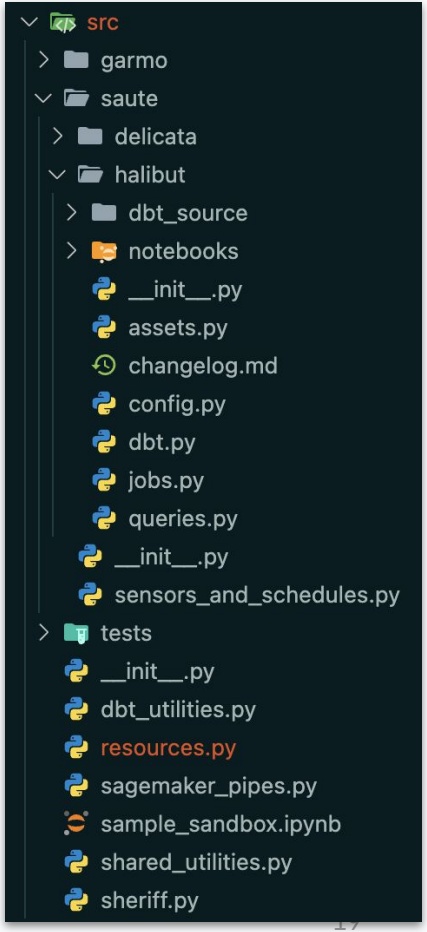
Dagster Implementation

Code Location Directory Strategy

Avoiding the “import circus”

- One-to-one modular mapping
- Team separation by directory
- Team directories copied into team images
- Shared utilities for resources
- Absolute imports only - works regardless of where the code runs - notebook, dev, prod, etc

```
from saute.fish.config import (
    FishConfig,
    metadata,
    project,
    tags,
    team,
    ...
)
from saute.fish.feature_engineering import (
    engineer_features,
    impute_bycatch,
    ...
)
from saute.fish.utils import (
    butter_baste,
    debone,
    ...
)
```



dagster

Dependency Injection

Environment Configuration

dagster

Environment Configuration

Dependency Injection

- Triplicate deployments for each code location - development, staging, and production environments
- Very flexible - some teams need different resources - Kafka, MSteams, etc.
- Standardized configuration patterns set by environmental variables
- Environment-specific resources and configurations per team

```python
if os.getenv("DEPLOYMENT") in ["prod", "staging", "dev"]:
    io_manager = S3PickleIOManager(
        s3_resource=BaseS3Resource(),
        s3_bucket=EnvVar("BACKEND_BUCKET"),
        s3_prefix="io_manager",
    )
    pipes_bucket = f"{os.getenv('DEPLOYMENT')}-dagster-pipes"
    s3_resource = S3Resource()
    location_resource = LocationResource()
else:
    io_manager = FilesystemIOManager()
    pipes_bucket = "dev-dagster-pipes"
    s3_resource = S3Resource(profile_name=EnvVar("AWS_SSO_PROFILE"))
    location_resource = LocationResource(profile_name=EnvVar("AWS_SSO_PROFILE"))

kafka = KafkaResource(
    bootstrap_servers=EnvVar("KAFKA_BOOTSTRAP_SERVER"),
    key=EnvVar("KAFKA_KEY"),
    secret=EnvVar("KAFKA_SECRET"),
)

snowflake_resource = SnowflakeResource(
    account=EnvVar("SNOWFLAKE_ACCOUNT"),
    user=EnvVar("SAUTE_SNOWFLAKE_USER"),
    password=EnvVar("SAUTE_SNOWFLAKE_PASSWORD"),
    warehouse=EnvVar("SAUTE_SNOWFLAKE_WAREHOUSE"),
    database=EnvVar("SAUTE_SNOWFLAKE_DATABASE"),
    role=EnvVar("SAUTE_SNOWFLAKE_ROLE"),
    echo=EnvVar("DEBUG"),
)
```

dagster

Environment Configuration

Reinforce the Paradigm

- Config.py - the power of tags and definition metadata
- Specify default compute, RAM
- Change at-will in launchpad
- Bespoke Dagster resource instantiation

```python
# config.py
team = "saute"
project = "halibut"
tags = {"team": team, "project": project}
metadata = {
  "Project": project,
  "Documentation URL": "fish.halibut.tasty.com",
  "Owners": ["my_email@host.com", ...]
}
```

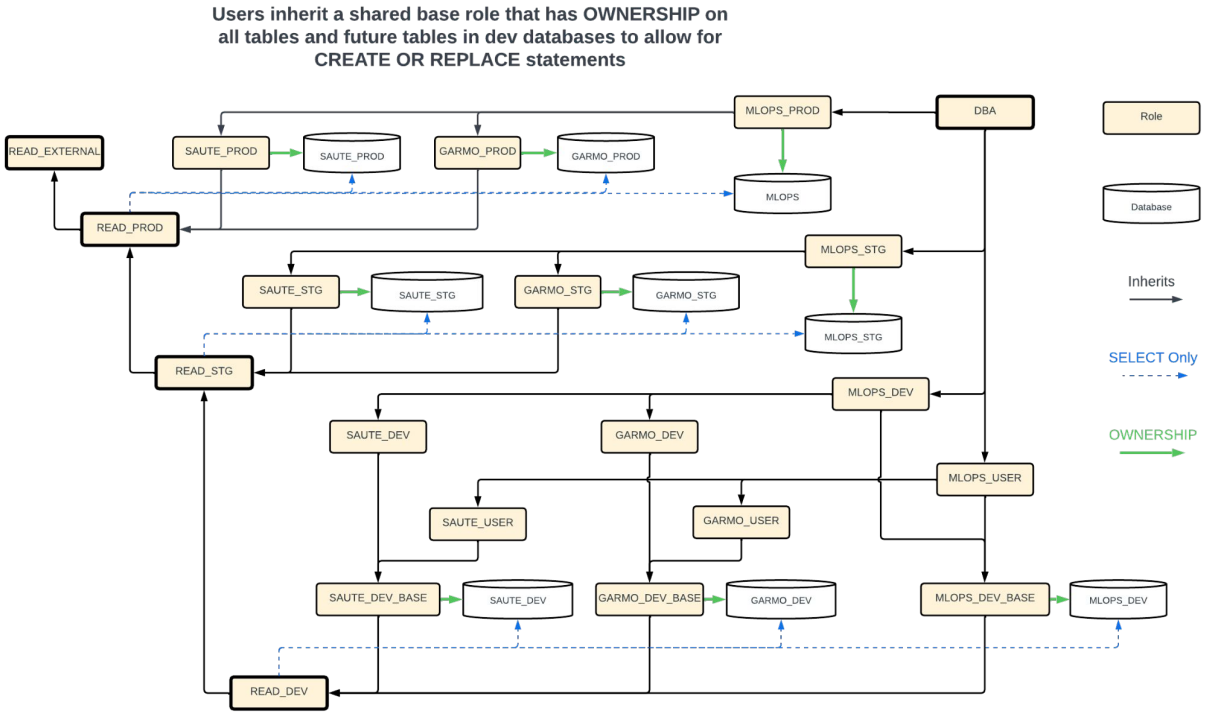
```python
# jobs.py
summer_halibut = define_asset_job(
    name="summer_halibut",
    tags={"ecs/cpu": "8192", "ecs/memory": "32768", **tags},
    selection=AssetSelection.asset(...),
    ...
)
```

```python
# resources.py
def setup_for_execution(self, context: InitResourceContext):
    """Dynamically sets default schema by group/project name."""
    self._run_id = context.run_id
    team = context.dagster_run.tags["team"].upper()
    project = context.dagster_run.tags["project"].upper()
    self._snowflake_schema = project
    self._query_tag_string = (
        f"team={team} project={project}"
        f"deployment={DEPLOYMENT.upper()} run_id={self._run_id}"
    )
```

dagster

Snowflake

- Database per team per environment - direct mapping from code locations to databases with consistent naming conventions
- Hierarchical RBAC based on team structure with environment-specific permissions
- Zero-Copy Cloning
    - Develop on real data

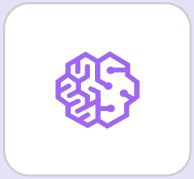


dagster

Integrations

aws

dbt

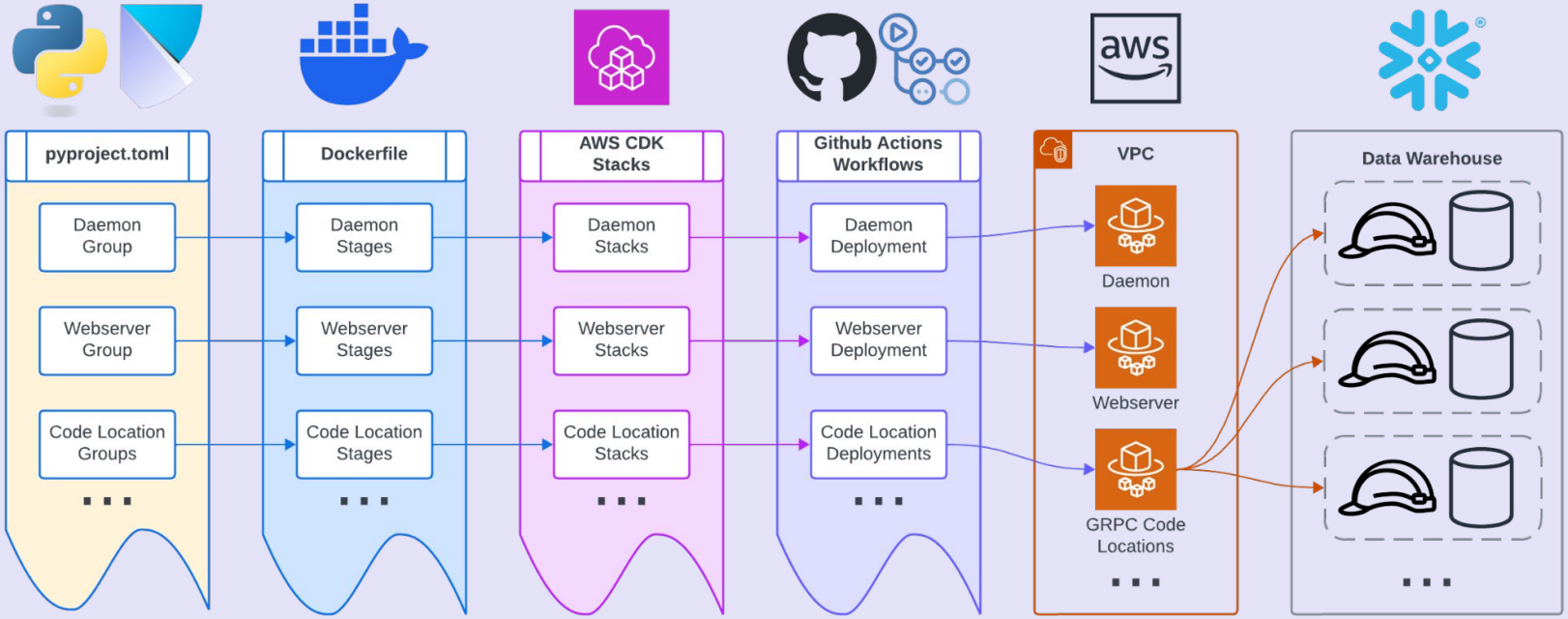


Sagemaker

dagster

```python
dbt_manifest_path, halibut_dbt_resource = \
    build_dbt_manifest_and_resource(__file__)

dbt_translator = build_project_dbt_translator(
    group_name=project,
    key_prefix=[team, project],
    metadata=metadata
)

@dbt_assets(
    manifest=dbt_manifest_path,
    partitions_def=weekly_partitions,
    backfill_policy=BackfillPolicy.single_run(),
    dagster_dbt_translator=dbt_translator,
)
def dbt_ingestion_assets(
    context,
    config: FishConfig,
    halibut_dbt_resource: DbtCliResource
):

    start, end = context.partition_time_window
    dbt_vars = json.dumps({
        "full_refresh_start_date": config.data_start_date,
        "partition_start": start.date().isoformat(),
        "partition_end": end.date().isoformat()
    })
    dbt_build_args = ["build", "--vars", dbt_vars]

    if config.full_refresh:
        dbt_build_args.append("--full-refresh")

    yield from halibut_dbt_resource.cli(
        dbt_build_args,
        context=context
    ).stream()
```

Putting it All Together



dagster

Putting it All Together

It Just Works

- Simplified troubleshooting and maintenance
- Improved developer velocity
- Standardized configurations

Faster Onboarding

- End-to-end workflow visualization
- Documentation as code in one repo
- Consistent organization across the platform

Empowered Stakeholders

- Self-service for teams.
- Clear ownership and responsibility
- Standardized yet flexible

dagster

Q&A

Join Our Beloved Slack Community

Try out Components

Register for the Components Webinar

dagster

Subtitle goes here

Thank you!