

# CLOUD ENGINEERING

---

## Programming Best Practices

Ashish Pujari

# Lecture Outline

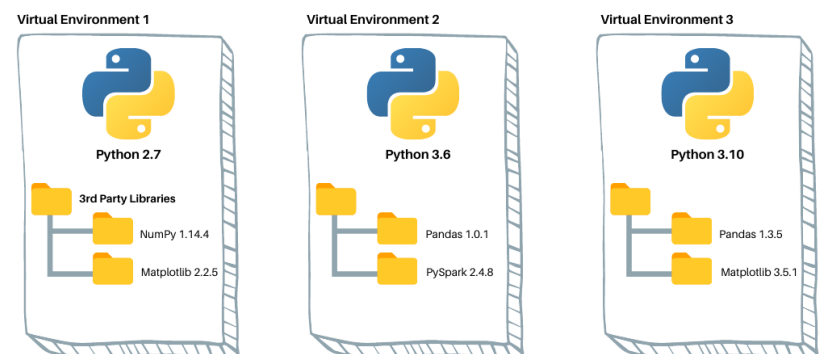
- Environment Management
- Code Management
- Logging and Exception Handling

# ENVIRONMENT MANAGEMENT

---

# Virtual Environments

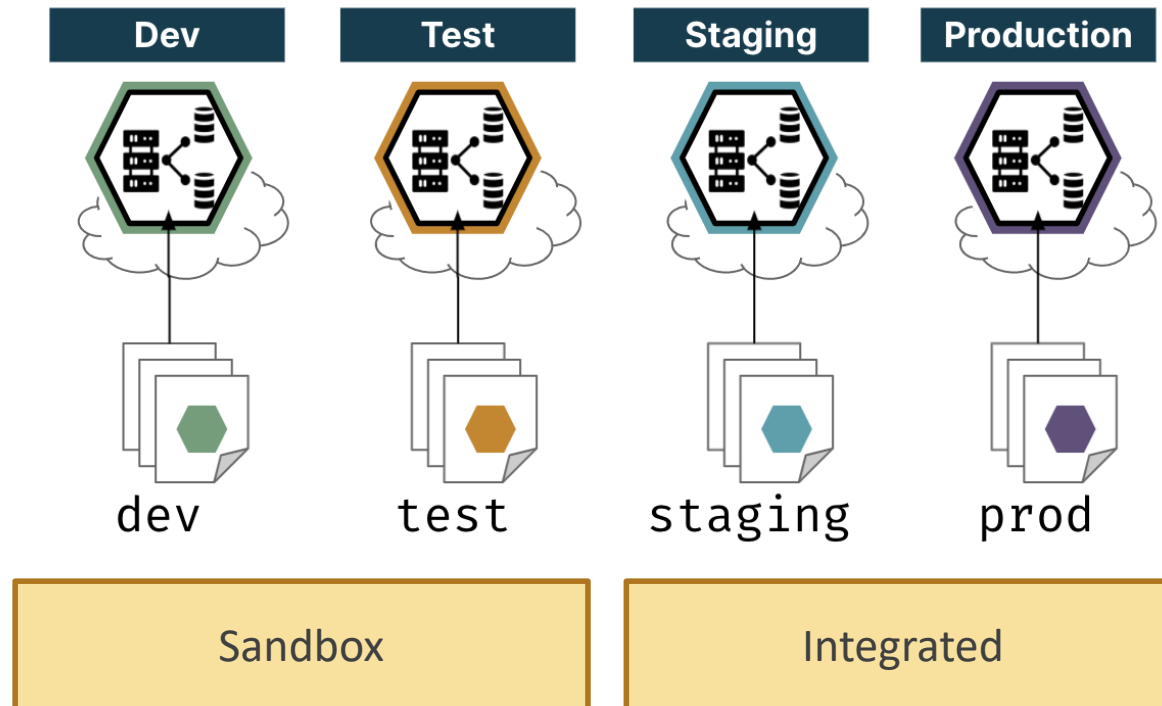
- A Python virtual environment consists of
  - the Python interpreter that the virtual environment runs on
  - a folder containing third-party libraries installed in the virtual environment.



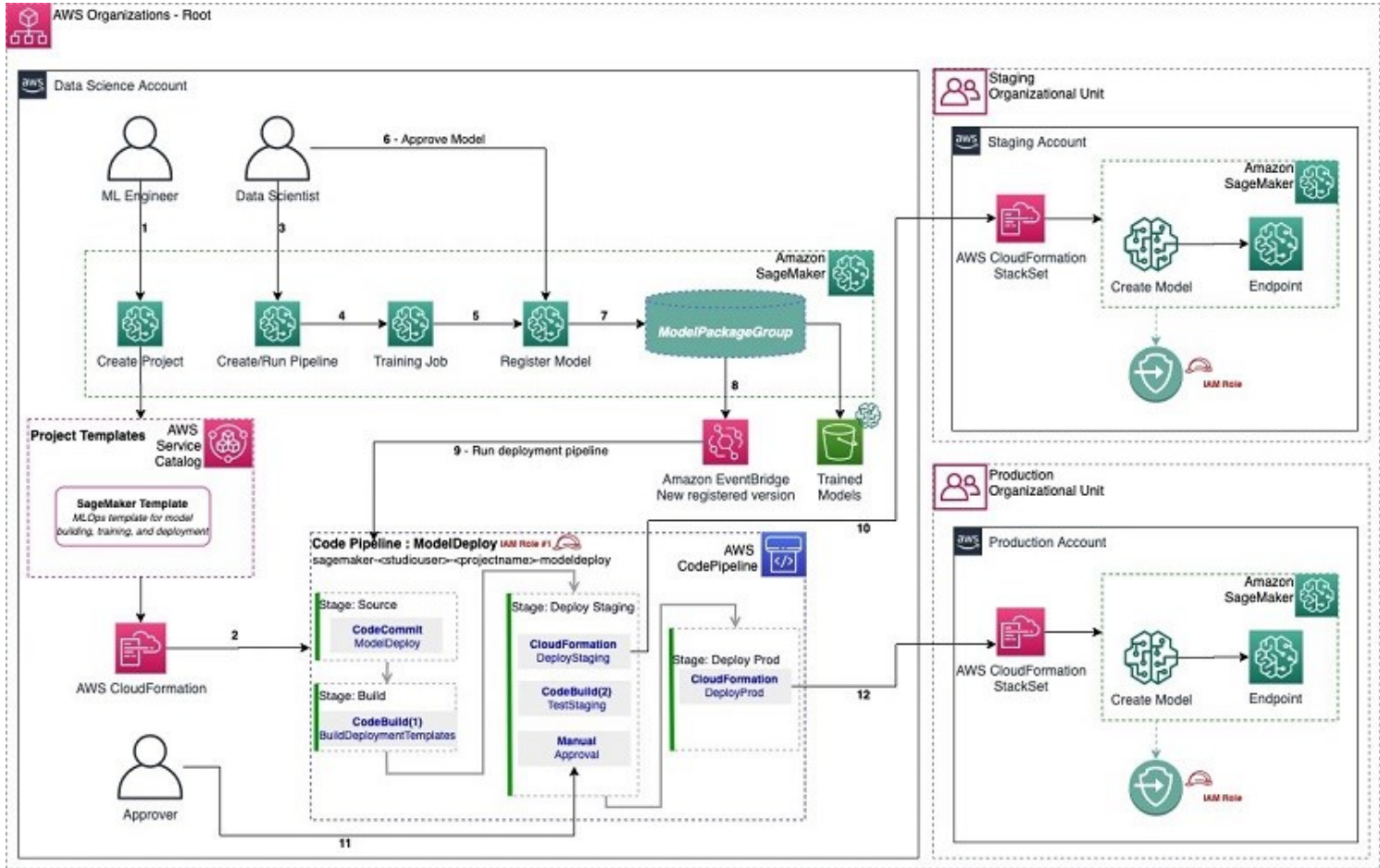
dataquest.io

- Best Practices
  - Use a Separate Virtual Environment for Each Project
  - Don't Forget to Activate Your Python Virtual Environment
  - Don't Use `>=` for Package Versioning in a Python Virtual Environment

# ML/AI Environments



# ML/AI Environments: AWS



# Configuration Variables

- Set up different configuration options for different programming environments
  - E.g., API keys, connection strings, logging configuration, or debug options, or other configuration settings
- Methods to store configuration:
  - Built-in data structures
  - Using dynamic loading
  - Environment variables - global variables; should be minimal or avoided
  - External Configuration Files -.ini, .yaml, .json, .xml, etc.

# Continuous Integration/Deployment (CI/CD)

- Continuous integration, continuous delivery, and continuous deployment
- Incremental code changes are made frequently and reliably
- Benefits
  - Accelerated time-to-value
  - Increases efficiency
  - Streamlining workflows through built-in automation, testing
  - Higher team collaboration



# CI/CD: Features

- A single source repository
- Frequent check-ins to main branch
- Automated builds
- Self-testing builds
- Frequent iterations
- Stable testing environments
- Maximum visibility
- Predictable deployments anytime

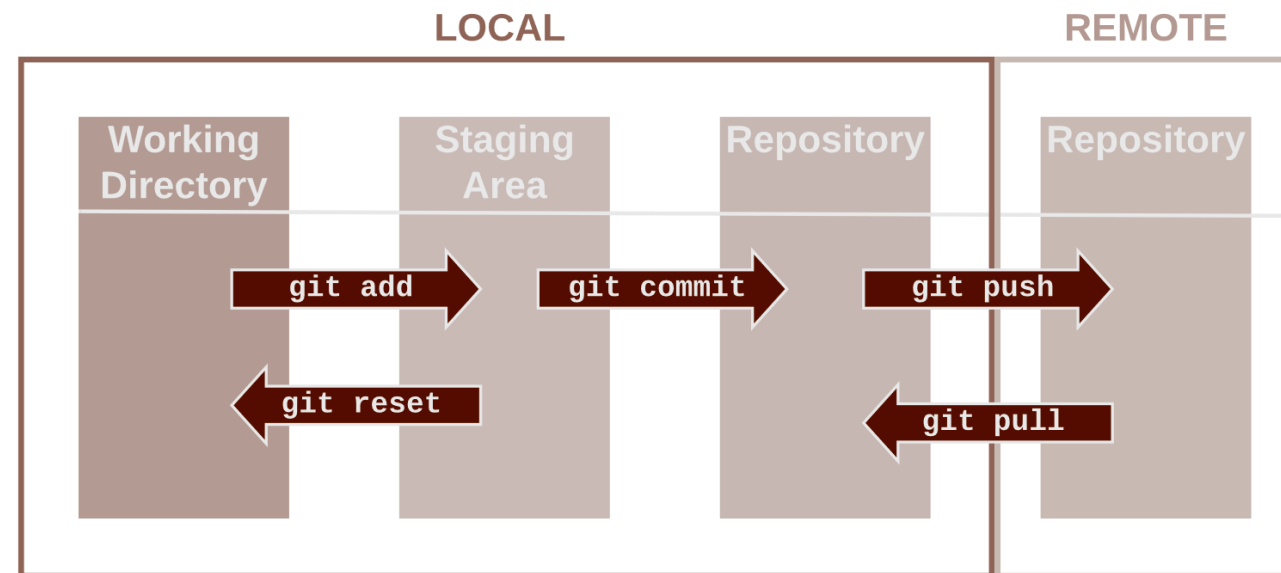
# CODE MANAGEMENT

---

Versioning, Git

# Code Versioning

- Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later
- Git is a free and open-source distributed version control system



# Git Commands

## Create and Config

```
$ git init
```

Turn an existing directory into a git repository

```
$ git clone [url]
```

Clone (download) a repository that already exists on GitHub, including all of the files, branches, and commits

```
$ git config --global user.name "[name]"
```

Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```

Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```

Enables helpful colorization of command line output

## Make Changes

```
$ git log
```

Lists version history for the current branch

```
$ git log --follow [file]
```

Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```

Shows content differences between two branches

```
$ git show [commit]
```

Outputs metadata and content changes of the specified commit

```
$ git add [file]
```

Snapshots the file in preparation for versioning

```
$ git commit -m "[descriptive message]"
```

Records file snapshots permanently in version history

# Git Commands

## Synchronize Changes

```
$ git fetch
```

Downloads all history from the remote tracking branches

```
$ git merge
```

Combines remote tracking branch into current local branch

```
$ git push
```

Uploads all local branch commits to GitHub

```
$ git pull
```

Updates your current local working branch with all new commits from the corresponding remote branch on GitHub.

`git pull` is a combination of `git fetch` and `git merge`

## Branches

```
$ git branch [branch-name]
```

Creates a new branch

```
$ git checkout [branch-name]
```

Switches to the specified branch and updates the working directory

```
$ git merge [branch]
```

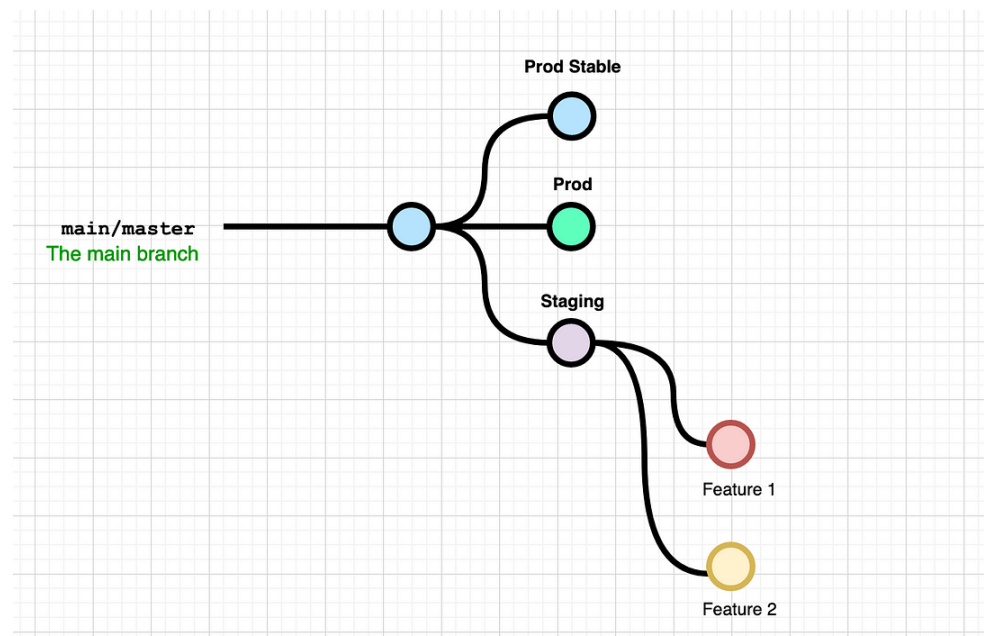
Combines the specified branch's history into the current branch. This is usually done in pull requests, but is an important Git operation.

```
$ git branch -d [branch-name]
```

Deletes the specified branch

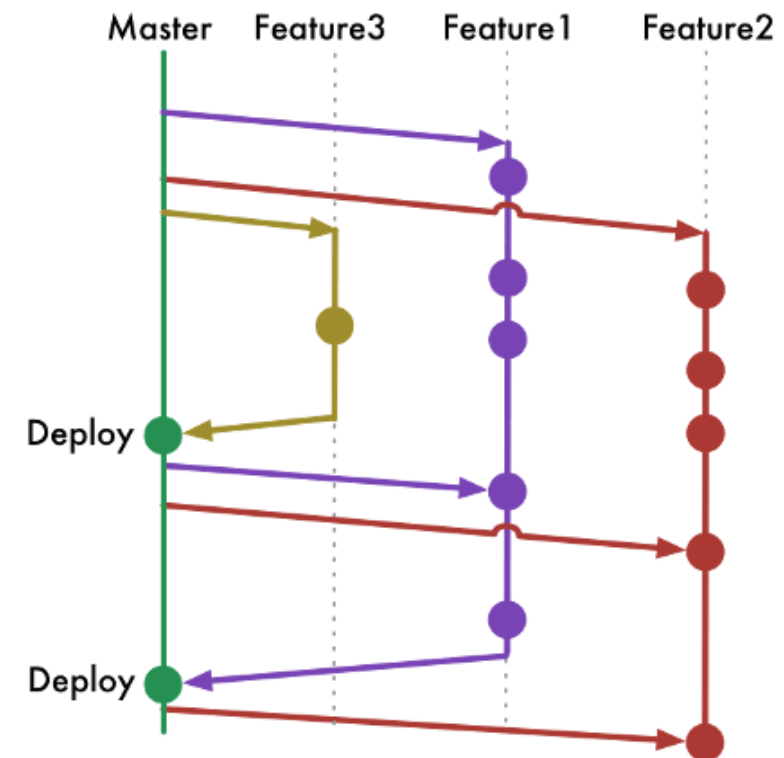
# Git Branching

- Branches are used as a means for teams to develop features giving them a separate workspace for their code.
- These branches are usually merged back to a main branch upon completion of work.



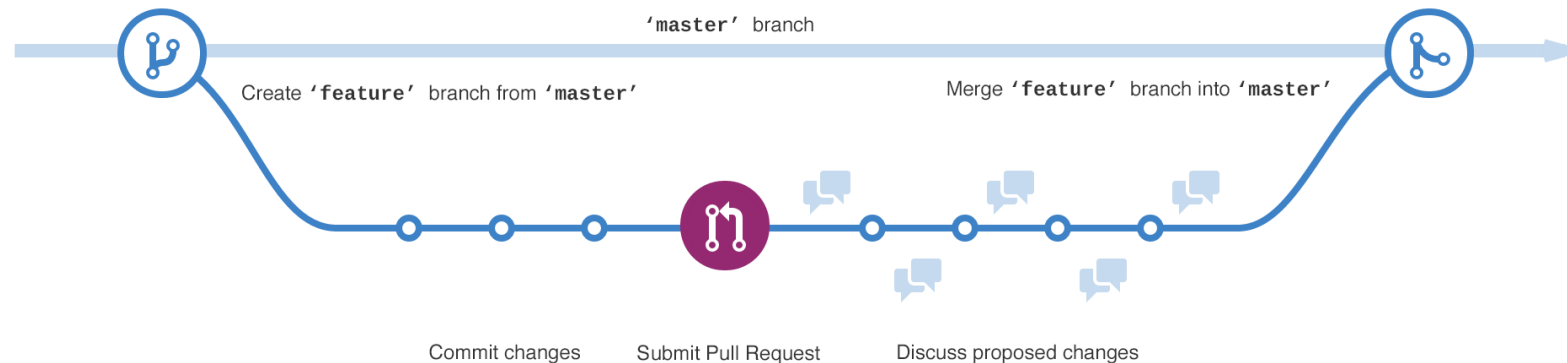
# Branching Strategies

- Designed to enable parallel development and structure releases by avoiding merge conflicts and allow for the easier integration of new features
- E.g., GitFlow, GitLab Flow, Trunk development



# Pull Request (PR)

- To start a pull request, you need to push your local branch to the remote Git repository.
- Write a description in your pull requests that explain:
  - What problem(s) are being solved
  - The approach and changes made to address it
  - Any special notes or highlights in the code review.
- Your peer will review the code and will merge it with main branch





# Common Types of Branches

Branch Type	Description
Trunk branch	Main, mainline, or the master branch; Implicit first branch when a repo is created
Development branch	Long-lived feature branch that holds changes made by developers before they're ready to go to production; Often parallels the main or trunk branch
Feature branch	Used for the lifetime of a new feature during its development Often used by a single developer, but possible to share it with others
Release branch	Reflects a set of changes that are intended to go through the production release process
Hotfix branch	changes related to emergency bug fixes; used in teams with explicitly versioned products, such as installed applications

# Git: Best Practices

- Each commit should be minimal but complete.
- Store large files outside your repo
- You should commit very often
- Each commit message consists of a header, a body and a footer
- Branches (other than master and development) must be short and as much as possible they should have a single purpose.
- The name of the branch should refer to its purpose.
- Before merging your feature branch to a public branch (e.g., main) always rebase your code to the latest version of the public branch and solve the conflicts locally.

# Coding Standards

- Rules, techniques, and guidelines to create cleaner, better readable, and more efficient code with minimal bugs and errors
- E.g., PEP-8 is a set of guidelines for writing Python code that is easy to read and maintain.
- Further Reading:
  - <https://effectivepython.com/>
  - <https://datasciencecampus.github.io/coding-standards/python.html#style-guide>

# PEP-8: Best Practices

Naming Conventions	<p>Use a consistent naming convention for all variables, functions, and classes.</p> <p>Variable names should be lowercase, with words separated by underscores (snake_case), while class names should be in CamelCase.</p> <p>Function names should also be lowercase, with words separated by underscores, and should be descriptive of their purpose.</p>
Indentation	<p>Use 4 spaces for indentation, instead of tabs. This helps ensure that code is readable across different platforms and text editors.</p>
Line Length	<p>Keep lines of code to a maximum of 79 characters in length, to make it easier to read and understand code. This can be extended to 120 characters if necessary.</p>
Whitespace	<p>Use whitespace judiciously to make code more readable.</p> <p>Separate functions and classes with two blank lines, and code blocks within functions and classes with a single blank line.</p> <p>Use a single space around operators and after commas in function calls.</p>

# PEP-8: Best Practices

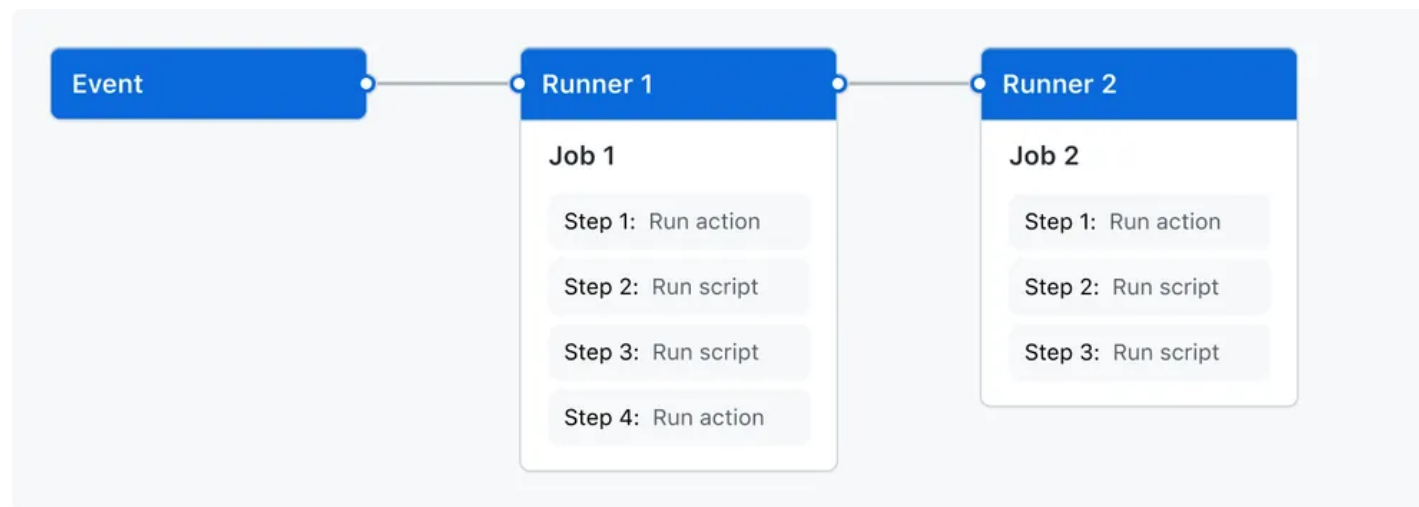
Comments	<p>Use comments sparingly, and only when they add value to the code.</p> <p>Use comments to explain why code is being done, not what it is doing, and to use complete sentences with proper grammar and spelling.</p>
Import Statements	<p>Place all import statements at the top of the file, and group them in the following order: standard (built-in) library imports, third-party library imports, local application imports.</p> <p>Use absolute imports, rather than relative imports (dot notation)</p>
Function and Method Arguments	<p>Use whitespace around the equals sign when defining function and method arguments with default values.</p> <p>Use a space before and after the equals sign</p>

# Code Analysis

- Static code analysis
  - Examines code to identify issues within the logic and techniques.
  - Linting is static analysis process used to flag patterns that might cause errors or other problems
  - Tools: *PyLint, pyflakes, Mypy, ast, etc.*
- Dynamic code analysis
  - Running code and examining the outcome
  - Performs testing possible execution paths of the code
  - Tools: *DynaPyt, pdb, etc.*

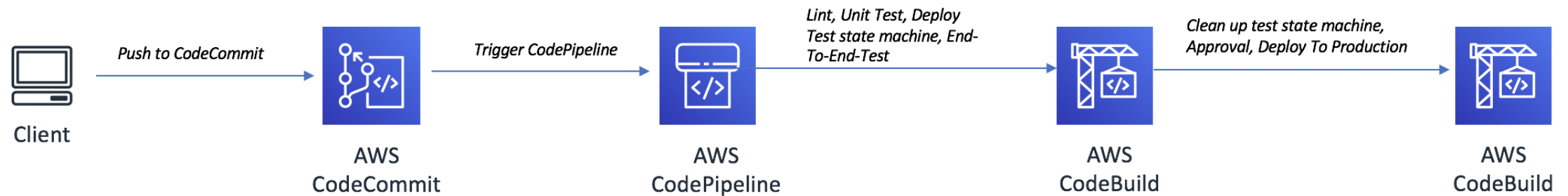
# Automation: Github

- GitHub Actions makes it easy to automate software workflows
- Use cases: CI/CD, Static/Dynamic analysis, Code building, deployment, etc.



- Further reading
  - <https://docs.github.com/en/actions>
  - <https://github.com/sdras/awesome-actions#machine-learning-ops>

# Automation: AWS Code Pipelines



[AWS CodePipeline](#)



# LOGGING AND EXCEPTION HANDLING

---

# Logging

- Logging is used to monitor system/model performance, troubleshooting errors, root cause analysis, cyber security incidents, etc.
- Typical log file contains:
  - Date and time when event occurred
  - Log level or severity level
  - An error code, if applicable
  - User id or process id that triggered the event
  - Message: description of the event
  - Function name
  - Name or IP address of the device where the event took place

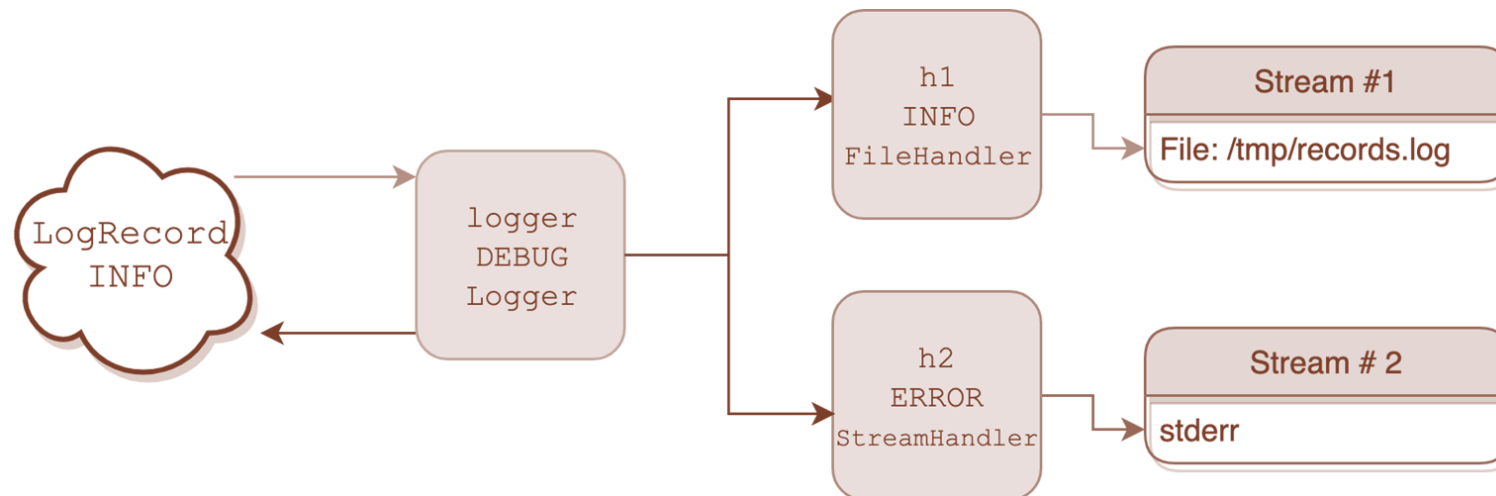
# Log Levels

- Describe the type and severity of a logged event based on the severity of the impact on users and the urgency of response required by the organization.

Log Level	Details	What to log	Environment
DEBUG	Detailed information, for diagnosis and troubleshooting.	Input and intermediate values, time taken for major/minor subroutines, etc.	Dev, Test
INFO	High level logs with small amount of information; Confirmation that things are working as expected.	Major branches in processing logic, time taken for major subroutines, size of input/output, record counts, etc.	Dev, Test, Prod
WARNING	Logs unexpected events and potential future problems.	Unused values/data, library or version deprecation, etc.	Dev, Test, Prod
ERROR	Serious problems; code has not been able to perform certain functions.	Unexpected input or parameter values, file not found, divide by zero, etc.	Dev, Test, Prod
CRITICAL	Serious errors; the program execution may have failed.	Resource Leaks, Data loss, Disk full, etc.	Dev, Test, Prod

# Log Handlers

- Object that handles how and where the logs must be directed
  - StreamHandler , FileHandler , RotatingFileHandler , etc.

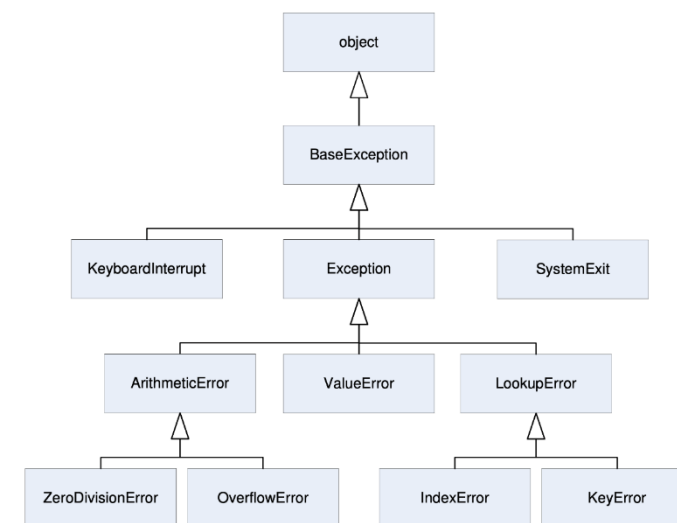


# Exceptions

- Errors detected during execution (run-time) are called exceptions
- Some exceptions are built in; sometimes useful to create user-defined exceptions
- It is possible to write programs that handle selected exceptions.

Context where  
the exception  
occurred

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```



Exception  
details

# Handling Exceptions

A *try* may have more than one *except* clause, to specify handlers for different exceptions

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

Base class of all non-fatal exceptions

Cleanup  
action

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

# Handling Exceptions: Best Practices

- Use try-except-finally blocks to handle exceptions gracefully and ensure your code can handle errors effectively
- Explicit exception handling is better than implicit
- Use a finally block to perform cleanup operations such as releasing resources e.g., database connection, threads, etc.
- Keep code in your try block to a minimum and avoid handling too many exceptions
- Use custom exceptions only when absolutely necessary

# Monitoring



/aws/lambda/cwi/Demo

filter @type="REPORT"  
| fields @timestamp, @message

Run query

Actions

Sample queries

Have feedback? Email us.

Logs

Visualization

Distribution of log events over time

25 records matched | 128,780 records (56.3 MB) scanned in 2.9s @ 44,057 records/s (19.2 MB/s)

#	@timestamp	@message
1	2019-06-06T23:39:57.240-05:00	REPORT RequestId: 1dcac8bf-cdea-4102-a26a-5f966f5a0ee3 Duration: 156.83 ms Billed Duration: 200 ms Memory Size: 1...
2	2019-06-06T23:39:41.660-05:00	REPORT RequestId: cda0b260-e6d1-468e-b3f4-ad682105de37 Duration: 265.90 ms Billed Duration: 300 ms Memory Size: 1...
3	2019-06-06T23:39:40.899-05:00	REPORT RequestId: 7bef5c91-9dc9-45a8-a103-f6d5177851e5 Duration: 177.54 ms Billed Duration: 200 ms Memory Size: 1...
4	2019-06-06T23:39:40.040-05:00	REPORT RequestId: 5a6e120c-7507-4831-8ca2-b2b4591b8335 Duration: 164.99 ms Billed Duration: 200 ms Memory Size: 1...
5	2019-06-06T23:39:39.360-05:00	REPORT RequestId: b3f1ee2e-6c52-4a4b-be04-b890d3351113 Duration: 175.82 ms Billed Duration: 200 ms Memory Size: 1...
6	2019-06-06T23:39:38.699-05:00	REPORT RequestId: 06f83e08-154b-48a6-8d94-14d2e2b2c83c Duration: 160.86 ms Billed Duration: 200 ms Memory Size: 1...
7	2019-06-06T23:39:37.959-05:00	REPORT RequestId: 7a8baf4f-5f62-4629-9e3e-980fcf26efde Duration: 155.49 ms Billed Duration: 200 ms Memory Size: 1...
8	2019-06-06T23:39:37.240-05:00	REPORT RequestId: 7d212ccf-14ac-4920-bae4-c054e9882051 Duration: 197.19 ms Billed Duration: 200 ms Memory Size: 1...
9	2019-06-06T23:39:36.520-05:00	REPORT RequestId: 0b67a7c8-f102-4c63-9e67-e9e223c2a32c Duration: 179.91 ms Billed Duration: 200 ms Memory Size: 1...
10	2019-06-06T23:39:35.819-05:00	REPORT RequestId: 834cc50d-9d6a-42fb-8b95-9e4393b0edd8 Duration: 152.20 ms Billed Duration: 200 ms Memory Size: 1...
11	2019-06-06T23:39:34.999-05:00	REPORT RequestId: 100b21c7-4ee2-4443-8129-26e298cab311 Duration: 163.53 ms Billed Duration: 200 ms Memory Size: 1...
12	2019-06-06T23:39:34.219-05:00	REPORT RequestId: 2a9a7533-152b-4a1b-9063-84a4a01079a7 Duration: 221.69 ms Billed Duration: 300 ms Memory Size: 1...

Query help

Learn more

Commands

fields

filter

stats

sort

limit

parse

Discovered fields

Search for a field

@ingestionTime	100%
@logStream	100%
@message	100%
@requestId	100%
@timestamp	100%
accountId	99%
duration	99%
host	99%
httpVerb	99%
path	99%
requestId	99%
requestSize	99%
responseSize	99%
srcIp	99%
statusCode	99%
userAgent	99%
@type	<5%



# Programming Best Practices

- DRY Principle – Don't repeat yourself
- Keep code simple and readable
- Understand context
- Modular design
- Low coupling and high cohesion
- Avoid Deep Nesting
- Naming conventions, file and folder structure
- Follow coding standards
- Proper logging and exception handling
- Peer Review
- Unit Testing
- Code pipeline automation