

Morgan Stanley



Building and leveraging Python on the mainframe for application developers

Calder Sacks, Executive Director

August 2026

Bio Page

CALDER SACKS

Executive Director

Enterprise Z

Phone: +1 212 276-1952

Calder.Sacks@morganstanley.com

Years of Experience: 34 Years

Years at Morgan Stanley: 27 Years

Biography

Worked in many roles within the mainframe technology landscape. Started as a Natural mainframe programmer and moved into the Adabas DBA role. Lead the database engineering team at Morgan Stanley for 10+ years and now lead the team working on Mainframe integration strategy and technologies for our Cobol/DB2 mainframe environment.

Been working on technologies between mainframe and distributed since 2003.

- One of the first companies to call Java RPC services from our mainframe Batch and CICS processes back in 2005
- Setup Sybase access from the mainframe.
- Implemented SQL & ACI access to mainframe Adabas and CICS
- Running Kafka Connectors running on the mainframe to distributed Kafka

Our goal is integrating mainframe to be just another platform and to leverage what has already been built where possible

Table of Contents

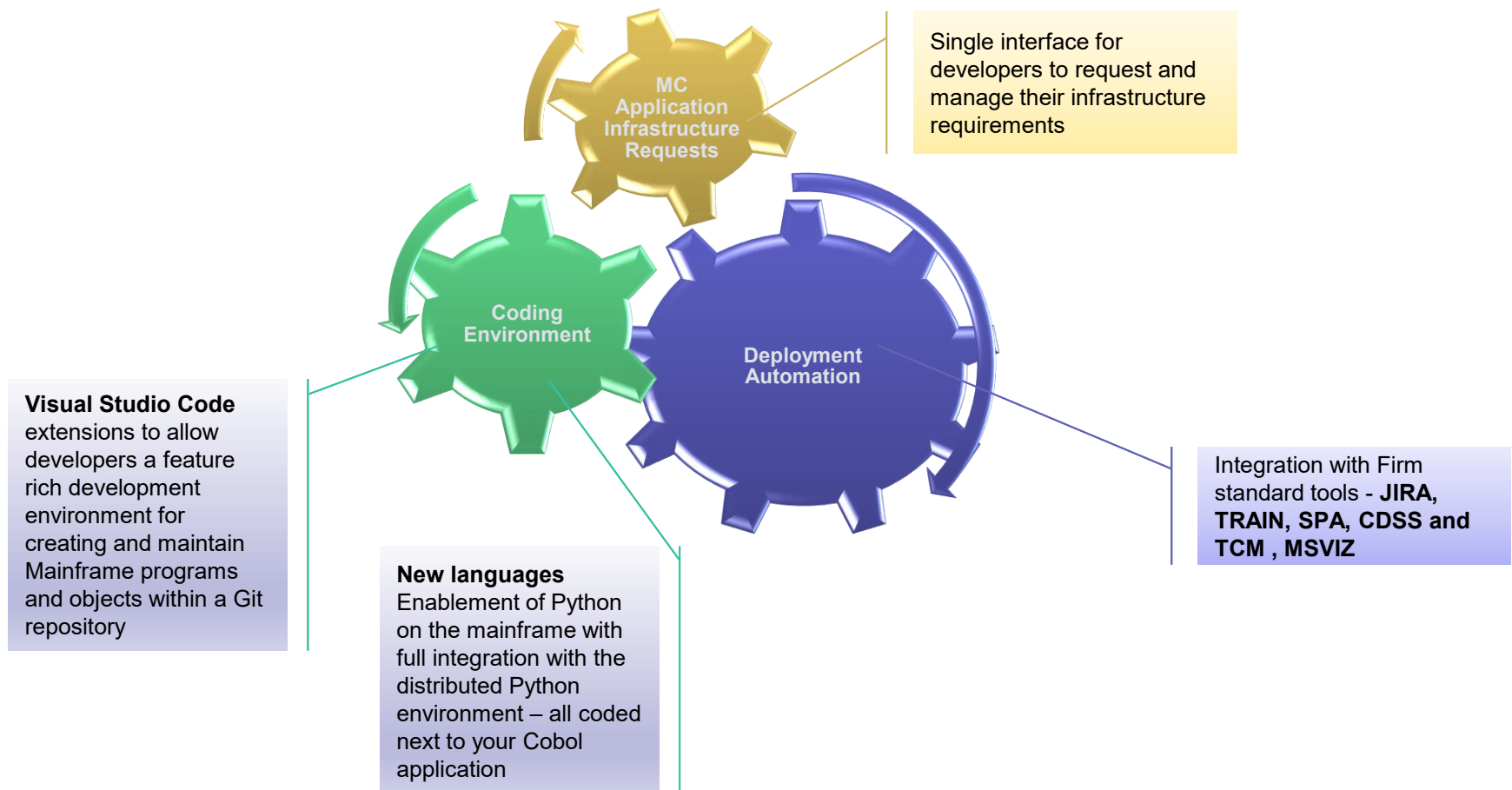
Python on the mainframe - Why do it ?

Our approach for leveraging Python

Managing Python in the mainframe world

Project Goal

Simplify and unify the development environment, by providing a modern feature and function rich set of tools in an easy-to-use environment, to improve the mainframe developer experience when making requests for mainframe resources and deploying changes to the mainframe environment with full integration into the Firm standard CI/CD tool set



Where are we using Python on the Mainframe?

- We are working to make the mainframe just another platform for people to run their applications on
 - Rewritten our mainframe cost allocation system into z/Python with DB2 database
 - Rewritten many of our Git Deployment scripts into z/Python
 - Multiple application teams have moved some of their business process steps to us z/Python
 - The Mainframe Storage management team have adopted z/Python for their disk management scripts and deploy these with our integrated MFGit solution

Why use Python on the Mainframe?

- Python is the most popular programming language in use today, by a wide margin
 - Access to Open Source Software is a big win – leveraging what the community has built already
 - A multi paradigm language - you choose the correct paradigm to solve the problem
 - Object orientated
 - Procedural
 - Functional programming
 - It is well suited to scripting
- Potential for new developers to work with and on mainframe systems, extending application expertise
- Many of our development teams are hybrid teams and they want to cross train
- Ability to break apart mainframe applications in components and rewrite over time
- Data Gravity concerns for large scale applications
- Many new tools and architectures leverage USS and Mainframe Python for automation and deployments

Python for the mainframe

Allows developers to

- Rewrite 1 step in a job instead of the whole job
- Rewrite single programs at a time if needed
- Reduces implementation risk and conversion risks
- Allows for non-mainframe developers to work with mainframe code



Some examples of use cases we are working on today

- Cobol to Python for business applications
- Rewrite REXX to Python
- Change JCL to Python scripts
- Adding Flask REST API's to run python services on the mainframe right next to the source of data

Currently in use in our production environment

- Installed a number of IBM packages
 - Utility packages to allow your Python code to work with mainframe components
 - Scientific and Calculation intensive package support

A use case example of the Python code

python/batches/dl_feed_processor.py

```
34
35 """
36
37 import os
38 import sys
39 import logging
40 from platform import system
41 from pathlib import Path
42 from typing import List, Any, Dict
43 from datetime import datetime
44
45 from utils.config_loader import load_config
46 from utils.logger_factory import setup_logger
47 from utils.db_util import import_ibm_db, connect, close_connect, call_stored_procedure
48
49 ▼ def process_feed(feed_name: str, environment: str, log_level: str = None) -> None:
50     """
51     Process a single datalake feed for the specified environment.
52
53     Arguments:
54     - environment: The environment to process the feed for (e.g., 'dev', 'qa', 'prod').
55     - feed_name: The name of the feed to process.
56     - log_level: The logging level to use (optional).
57     """
58     # Initialize logger (temporary until logger_factory is implemented)
59     logger = logging.getLogger(__name__)
60     ▼ try:
61         logger.info(f"Starting process_feed with feed_name: {feed_name}, environment: {environment}, log_level: {log_level}")
62
63         # Load configurations
64         sys_config: Dict[str, Dict[str, Any]] = load_config('config/sys_config.properties')
65         datalake_config: Dict[str, Dict[str, Any]] = load_config('config/datalake_config.properties')
66         logger.debug(f"Loaded configurations for {environment} environment and feed {feed_name}")
67         logger.debug(f"sys_config: {sys_config}")
68         logger.debug(f"datalake_config: {datalake_config}")
69
70         # Validate and retrieve environment and feed configurations
71         env_config = sys_config.get(environment)
```

26 Cobol programs replaced with 1 Python program

- Took the opportunity to redesign small parts of the application
- Cobol lines of code (Averaged 600 lines per module) = 15,600
- Python Lines of code = 330

Changing our JCL scripts to Python for Git processing

Existing JCL
loaded with ZOWE

z/Python scripting

Getting Co-pilot help

The image displays a VS Code editor interface with three main panes:

- Left Pane (JCL Script):** Shows the original JCL script `P814FH51.jcl`. It includes comments like `//*JOB=ARM S=VSYS` and JCL statements such as `ALLOCATE TEMP DATASET`, `EXEC PGM=IEFBRI14`, and `DD DSN=DSN, DISP=(NEW,PASS)`.
- Right Pane (Python Script):** Shows the new Python script `cli.py` using the Click library. It defines a command `update_dataset_with_dd` with options for `--data-definition` and `--help`. The script includes a `main` function and a `cli` group to register commands.
- Chat Pane (Copilot Help):** Shows a chat session where the user asks for help adding a step to create/allocate a temporary dataset. The assistant provides the necessary JCL code:


```

      *****
      /* ALLOCATE TEMP DATASET
      *****
      //ALLOC EXEC PGM=IEFBRI14
      //TEMP DD DSN=&&TEMP,
      //      DISP=(NEW,PASS),
      //      SPACE=(TRK,(1,1)),
      //      UNIT=SYSDA,
      //      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
      
```

The bottom status bar shows the terminal output of a git commit and push command, indicating successful execution.

Live coding against z/Python with VS Code

Coding with VS Code plugins to get auto complete and other help to increase developer productivity

Automatic movement of code to z/Python environment

Real time unit test execution using Pytest packages on z/Python

```
src > msmfpt > io.py
You, 2 weeks ago | 1 author (You)
1 import time
2
3 from click import echo
4
5 # change MIN to 100,000
6 MIN_DATASET_SIZE_FOR_PROGRESS = 100_000
7
8
9 def create_large_dataset(dataset_name: str) -> bool:
10     from zoutil_py import datasets # noqa: PLC0415
11
12     try:
13         datasets.create(
14             dataset_name,
15             dataset_type="SEQ",
16             record_format="FB",
17             record_length=120,
18             block_size=1_200,
19             primary_space="30M",
20             secondary_space="15M",
21         )
22     except Exception:
23         return False
24     return True
```

Terminal output:

```
(.venv) PS C:\MSDE\spranger\git\zpy
creating local archive ... done.
upload 'src/msmfpt/io.py'
sync: src/msmfpt/io.py
creating local archive ... done.
upload 'src/msmfpt/io.py'
sync: src/msmfpt/io.py
creating local archive ... done.
uploading archive ... done.
decompressing on z/OS ... done.
unpax on z/OS ... done.

/bin/sh ./livecoding.sh develop
sync'ed
exit code: 0

(sync entry 'src' watching)
upload 'src/msmfpt/io.py'
sync: src/msmfpt/io.py
creating local archive ... done.
uploading archive ... done.
decompressing on z/OS ... done.
unpax on z/OS ... done.

/bin/sh ./livecoding.sh develop
sync'ed
exit code: 0

(sync entry 'src' watching)
[]
```

SSH Session Output:

File	Size	Lines	Lines	Lines	Lines	Lines
src/msmfpt/api.py	103	7	10	0	94%	105-111
src/msmfpt/cli.py	27	4	2	1	83%	10, 61, 109, 122
src/msmfpt/core.py	6	0	0	0	100%	
src/msmfpt/io.py	66	57	8	0	12%	10-24, 28-43, 47-54, 58-64, 68-70, 74-128
TOTAL	205	69	22	2	65%	

Coverage XML written to file out/coverage-results/UNIT/coverage.xml
38 passed, 1 warning in 0.51s

Managing Python on the mainframe

Python Packages

Some Python packages will depend on other packages – creating a tiered package problem

- The top level package may work on OS390 but one of the underlying dependency packages may not
- There are of 729k (up from 648k 6 months ago) Python packages at PyPi.org.
- We have 4,500 that are allowed in our organization
 - z/Python compatible = 3,100 packages
- Setup a MF PyPi repository within our organization to load MF vetted packages from the approved internal list

Package in question: `sktime`

- very popular machine learning timeseries package
 - ◊ <https://www.sktime.net/en/stable/>

Initial Evaluation

- Package name :: `sktime-0.35.0-py3-none-any.whl`
- Based on the package name, looks like a possible candidate for adoption
 - ◊ Won't know until we download/evaluate all of its direct and indirect dependencies

```
$ pip download sktime
Looking in indexes: https://spranger:****@msdo-docker-prod.ms.com:4443/artifactory/api/pypi/msppyl/simple, https://spr
Collecting sktime
  Using cached https://msdo-docker-prod.ms.com:4443/artifactory/api/pypi/msppyl/ca/ad/6f673b34be739b70d5b8a8435e77c358b
Collecting joblib<1.5, >=1.2.8 (from sktime)
  Using cached https://msdo-docker-prod.ms.com:4443/artifactory/api/pypi/msppyl/91/29/d4b4b042f2be8b623cbd5e2148caFca2
...
Successfully downloaded sktime joblib numpy pandas scikit-base scikit-learn scipy packaging python-dateutil pytz thread
```

Indepth Evaluation

- After downloading all of `sktime` direct and indirect dependencies, red flags are thrown

```
$ ls
joblib-1.4.2-py3-none-any.whl
numpy-2.1.3-cp310-cp310-win_and64.whl # red flag !
packaging-24.2-py3-none-any.whl
pandas-2.2.3-cp310-cp310-win_and64.whl # red flag !
python_dateutil-2.9.0.post0-py2.py3-none-any.whl
pytz-2024.2-py2.py3-none-any.whl
scikit_base-0.12.0-py3-none-any.whl
scikit_learn-1.5.2-cp310-cp310-win_and64.whl # red flag !
scipy-1.15.1-cp310-cp310-win_and64.whl # red flag !
six-1.17.0-py2.py3-none-any.whl
sktime-0.35.0-py3-none-any.whl
threadpoolctl-3.5.0-py3-none-any.whl
tzdata-2024.2-py2.py3-none-any.whl
```

Package and artifact version control

Create and deploy an artifact that has custom code and all its dependencies baked in, for 2 reasons

- Avoiding dependency hell (this is a real term)
 - Managing packages is a complex matter
 - Dependencies that have dependencies that have other dependencies -- all with their own unique version specifications
 - Managing shared packages across teams and projects becomes unmanageable at scale
 - Having a project/application define and bundle their own dependencies is the only way to avoid this
- Reducing deploy time dependencies
 - What all needs to be working to install a package
 - it needs to know where PYPI is
 - it needs to have credentials to access said PYPI
 - it needs a working network connection
 - the package AND specific version defined in project needs to be available
 - repeat this for deps that have deps
 - there needs to be enough disk to download and install the package
 - Etc..

Package and artifact version control

By bundling the custom code and all deps into a single artifact, the deploy step becomes a much simpler operation and reduces the possibility for error

**** Think Containers in a distributed environment for a self contained execution contract ****

TRAINPACKAGE:[zpython](#)/[sysutil](#)/2025.09.17-30 

Overview

Summary Not Available

Project source of this release: **artifactory**

Namespace of this release: **trainpackage**

Release date: 2025-09-17 17:44:04 GMT

Train Metrics

Current Train compliance level:

L3 - Dependency registration

Dependencies

Train-Identified Direct Dependencies

VMS:msde/artifactory-eng-tools (1)	VMS:msde/traincli (1)	VMS:msde/msdlbuild (1)
--	---------------------------------------	--

Build Tool-Identified Dependencies

Direct Dependencies

PYPI:generate/train-metadata (1)	PYPI:3rd/msmfuapp (1)	VMS:python/core (1)
--	---------------------------------------	-------------------------------------

Indirect Dependencies

PYPI:3rd/packaging (1)	PYPI:3rd/pip-requirements-parser (1)	PYPI:3rd/pyparsing (1)	PYPI:3rd/pipdeptree (1)	PYPI:3rd/pip (1)
PYPI:3rd/setuptools (1)	PYPI:3rd/wheel-filename (1)	PYPI:3rd/click (1)	PYPI:3rd/terminaltables3 (1)	

Managing Python on the mainframe

Version upgrades

- Python developers use Python differently than what mainframe teams are used to
- Doing central version upgrades now have many dependencies that may or may not be ready
 - E.g. the IBM ZOAU toolkit package may not be ready for Python v3.13
 - This is a paradigm shift for the mainframe engineering teams that have always upgraded products in a centrally managed manner, e.g. Cobol version upgrades
- We have decided to let people code the version of Python in their code to ensure stability
 - We allow 3 versions of Python
 - This creates an upgrade problem though
 - We have built part of this tracking by leveraging our distributed systems that have been built to manage version dependencies and CVE reporting and tracking

Managing Python on the mainframe

Closing thoughts

- Python needs to be managed differently than what mainframe people are used to
- You will need people with good technical abilities to set it up initially
- Understand the differences between operating systems and technologies
- Make sure you look at how to manage versions and CVE's especially if you use Open Source packages
- We have integrated our Python deployments with our mainframe Git Solution
 - Deployments are done for our engineering teams to the controlling systems as well
 - Application teams can deploy as needed to the z/Python environment

Questions ?