European Mainframe Academy I EMA

Wolfram‘s zWelt

Version 9



Wolfram Greis, European Mainframe Academy

wolfram.greis@mainframe-academy.de

Agenda

1. Go on the Mainframe (z/OS)
2. Git and DBB
3. How AI helps me as a Mainframe Educator
4. IBM Dual Architecture (optional)

© European Mainframe Academy

01 - Go on the Mainframe (z/OS)

We already have Assembler, COBOL and PL/I

The proven languages of z/OS – and what they are great at

Assembler (HLASM) Since System/360 (1964)
Maximum control and speed: system code, exits, performance-critical routines

COBOL Since 1959
Business logic, exact decimal arithmetic, batch and CICS/IMS transactions

PL/I Since the mid-1960s
Business and scientific computing in one language; strong in banking and insurance



They run the core business – reliably, for decades. But: experts are retiring, new developers rarely learn them, and modern libraries (HTTP, JSON, cloud APIs) are scarce.

Do we really need Go?

Short answer: not as a replacement – as a complement

Keep in COBOL / PL/I / HLASM

- Core batch and transaction logic that works
- Decimal-heavy business rules
- Performance-critical system code
- Code with no business reason to change

Write new in Go

- REST / JSON APIs and microservices
- Tools, CLIs and automation
- Integration with cloud, Kubernetes, messaging
- Projects for new developers



Go adds modern developers and modern libraries to z/OS – while the data and the proven programs stay where they are: on the mainframe.

Go has a huge talent pool

Developers you can actually hire – and train quickly

2.2 M

professional developers use Go as their primary language – twice as many as 5 years ago

5 M+

professionals use Go when secondary-language users are included

11 %

of all developers plan to adopt Go in the next 12 months

25

keywords only – a small language that is learned in weeks, not years



For z/OS teams: hire from a big market – and let experienced mainframers mentor them on the platform.

Sources: JetBrains, The Go Ecosystem in 2025 / State of Developer Ecosystem 2025; go.dev/ref/spec

A huge, open community

Open source since 2009 – developed in public, used worldwide



- **Open source** since 2009, BSD-style licence – free to use, also commercially
- **Developed in public** on GitHub: proposals, issues and code reviews are open to all
- **Predictable releases:** a new Go version every six months
- **Meet-ups, GopherCon conferences,** Slack, forums, many free tutorials
- **On z/OS:** IBM "Go on z/OS" community and open-source z/OS modules (github.com/ibmruntimes)

Why it matters: problems are usually solved already – someone has asked the question, written the library or fixed the bug. Your team is never alone.

A lot of existing packages

"Batteries included" standard library – plus a huge open-source ecosystem

Standard library – already on board

net/http web servers & clients	**encoding/json** JSON in / out
crypto/tls encryption, certificates	**database/sql** SQL databases
log/slog structured logging	**testing** unit tests, benchmarks



Open-source modules – find them on pkg.go.dev

- **Web:** Gin, chi, Echo **CLI:** cobra (used by kubectl, helm)
- **Logging/testing:** zap, testify **Cloud:** AWS, Azure, Google SDKs
- **z/OS:** go-recordio (VSAM) and other IBM z/OS modules

Package manager? Go handles this automatically

Go modules are built into the go command – no extra tool to install

1 go mod init	2 import "…"	3 go mod tidy	4 go build
creates go.mod (name of your module)	write the import in your code	downloads modules, records versions	verifies go.sum, builds the binary

```
// go.mod – written by the go command
module example.com/custapi

go 1.27

require github.com/go-chi/chi/v5 v5.2.1
```

- **One tool:** the go command fetches, versions and verifies dependencies
- **Reproducible:** exact versions in go.mod, checksums in go.sum
- **Air-gapped z/OS?** use a company proxy (GOPROXY) or go mod vendor

Compare: Python needs pip/venv, Java needs Maven/Gradle – Go has dependency management built in.

Import modules directly from GitHub

The import path is the download address

main.go

```
import (
    "fmt"                                  // standard library
    "github.com/ibmruntimes/go-recordio/v2"  // from GitHub
)
```

Terminal

```
$ go get github.com/ibmruntimes/go-recordio/v2
```

Your code – import path

↓

Module proxy – proxy.golang.org or company proxy

↓

Checksum DB – sum.golang.org verifies

↓

Module cache – local copy, used offline

Enterprise control: GOPROXY = approved company mirror, GOPRIVATE = internal Git servers, go mod vendor = all sources inside the project for audits and offline builds.

Go understands Unicode

UTF-8 everywhere – and conversion from / to EBCDIC on z/OS

- **Source code and strings are UTF-8:** ä, ß, €, 世界, emoji – all fine
- **rune** = one Unicode character; **for range** walks characters, not bytes
- **z/OS data is often EBCDIC** (IBM-1047, IBM-037, IBM-1140): convert with golang.org/x/text

```
s := "Grüße, 世界"
fmt.Println(len(s))                  // 15 bytes
fmt.Println(utf8.RuneCountInString(s))  // 9 characters

dec := charmap.CodePage1047.NewDecoder()
text, err := dec.Bytes(record)         // EBCDIC -> UTF-8
```



Benefit: international names, addresses and web data are handled correctly – no code page surprises.

Go is the lingua franca of cloud computing

If you talk to the cloud team, you talk Go



> 75 % of the projects of the Cloud Native Computing Foundation (CNCF) are written in Go

Built with Go:

Kubernetes	Docker	Terraform	Prometheus
etcd	Helm	Istio	containerd

Official Go SDKs: AWS, Microsoft Azure, Google Cloud

Mainframe in the hybrid cloud: the same language for z/OS services, Kubernetes operators, CLIs and API gateways – one skill set across platforms.

Source: go.dev/solutions/cloud

Part 2

Go on z/OS in practice

Getting started, VSAM access, REST APIs and Go vs Python

Getting started: IBM Open Enterprise SDK for Go

IBM-supported Go compiler and runtime for z/OS

- **Current release 1.27** (October 2026) – in step with community Go
- **Native 64-bit z/OS programs,** run in z/OS UNIX or as batch jobs (JCL)
- **z/OS extras:** VSAM access, zEDC compression, call C and COBOL
- **Same toolchain** as on Linux, Windows, macOS: build, test, modules
- **Wharf** helps to port existing Go packages to z/OS

hello.go

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, z/OS!")
}
```

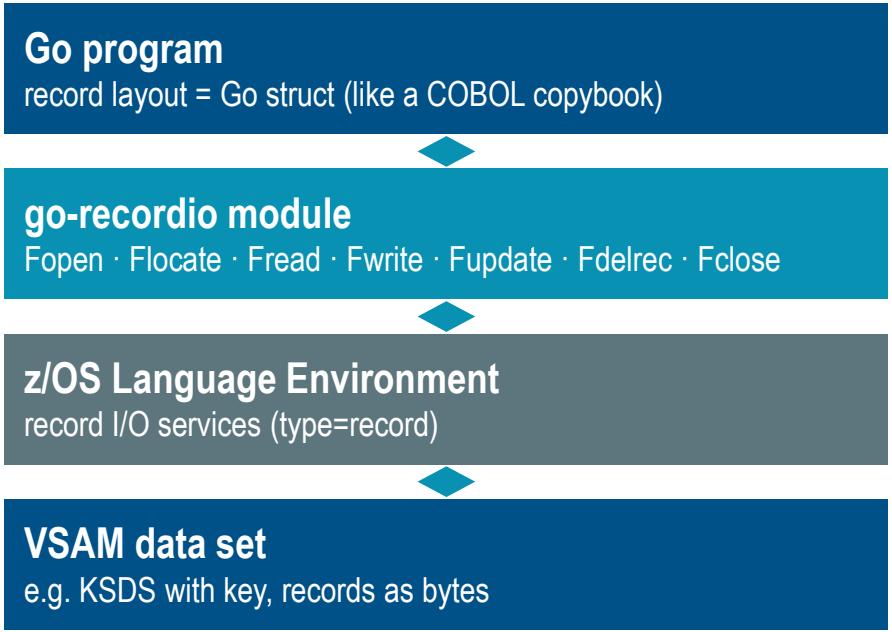
z/OS UNIX shell

```
$ go mod init example.com/hello
$ go build
$ ./hello
Hello, z/OS!
```

Result: one self-contained executable – no runtime or interpreter to install on the target system.

Example: Go accessing VSAM data on z/OS

IBM's open-source module go-recordio – direct record I/O, no C code needed



COBOL analogy

COBOL	Go (go-recordio)
OPEN	**Fopen**
START / READ KEY	**Flocate + Fread**
READ NEXT	**Fread**
WRITE	**Fwrite**
REWRITE	**Fupdate**
DELETE	**Fdelrec**
CLOSE	**Fclose**

Records are plain bytes: map them to Go structs; convert EBCDIC text fields where needed.

VSAM example: read one customer by key

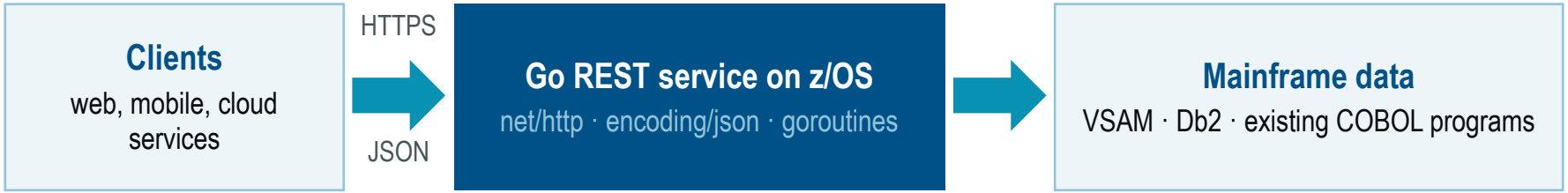
Simplified from IBM's go-recordio v2 examples

```
import (
    "fmt"
    "log"
    rio "github.com/ibmruntimes/go-recordio/v2"
    "github.com/ibmruntimes/go-recordio/v2/utils"
)
type Customer struct {        // record layout (like a copybook)
    ID   [8]byte              // VSAM key
    Name [32]byte
    City [20]byte
}
func main() {
    rs := rio.Fopen("//'HLQ.CUSTOMER.KSDS'", "rb+,type=record")
    if rs.Nil() { log.Fatal("cannot open VSAM data set") }
    defer rs.Fclose()
    var c Customer
    buf := utils.ConvertTypeToSlice(&c)  // struct <-> record bytes
    if rs.Flocate([]byte("00004711"), rio.Loc_KEY_EQ) == 0 {
        rs.Fread(buf)
        fmt.Println(string(c.Name[:]), string(c.City[:]))
    }
}
```

1. **Struct = record layout**
   fields with fixed byte lengths, key first
2. **Fopen**
   open the cluster in record mode
3. **defer Fclose**
   closes automatically at the end
4. **Flocate + Fread**
   position on key, read the record
5. **Use the fields**
   convert EBCDIC text if needed

Building RESTful apps with Go

Expose mainframe data as JSON APIs – with the standard library only




```go
func main() {
    http.HandleFunc("GET /customers/{id}", getCustomer)
    log.Fatal(http.ListenAndServe(":8080", nil))
}

func getCustomer(w http.ResponseWriter, r *http.Request) {
    c, err := lookupCustomer(r.PathValue("id")) // e.g. VSAM
    if err != nil {
        http.Error(w, "not found", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(c)
}
```

- **No framework needed:** routing, JSON and TLS are in the standard library
- **Scales:** each request runs in its own lightweight goroutine
- **Simple to deploy:** one binary, starts in milliseconds
- **Optional:** Gin or chi for bigger APIs

Go vs Python: different use cases

Both are available on z/OS – choose the right tool for the job

	Go	Python
Type system	static – errors found at compile time	dynamic – errors found at run time
Execution	compiled to native machine code – fast	interpreted – slower, C libraries for speed
Concurrency	goroutines and channels built in	limited by the GIL for CPU-bound threads
Deployment	one self-contained binary	interpreter + packages (pip, venv)
Best for	APIs, microservices, CLIs, system tools, cloud	data analysis, AI/ML, scripting, automation
On z/OS	IBM Open Enterprise SDK for Go	IBM Open Enterprise SDK for Python, ZOAU, Ansible

Not either-or: use Python to explore, analyse and automate – use Go to build and run services.

Key takeaways

What to remember from this module

1 — Complement, not replace
COBOL, PL/I and Assembler stay; Go adds new capabilities

2 — People
huge talent pool and an open, active community

3 — Packages
rich standard library; modules managed by the go command

4 — GitHub imports
pinned versions, checksums, company proxy for control

5 — Unicode + EBCDIC
UTF-8 native, code page conversion available

6 — Cloud and z/OS
language of the cloud; supported on z/OS with VSAM and REST

Next step: try the Go and VSAM challenge on IBM Z Xplore, or install the SDK on a test LPAR.

02 – Git and DBB

Agenda and learning objectives

What you will be able to explain after this module

1. **What is Git?**
   Distributed version control and the Git servers GitHub, GitLab, Bitbucket

2. **What is Dependency Based Build?**
   Groovy for IBM Z, scanner, metadata database, build reports

3. **What is Jenkins?**
   Automation server and its agent role on z/OS

4. **Putting it together**
   End-to-end pipeline, team benefits, key takeaways



Modern tools, shared code, one mainframe platform

Why change how we build mainframe software?

From manual hand-offs to an automated, shared workflow

	Traditional way	With Git + DBB + Jenkins
Source code	Members in PDS libraries, locked by a host-only SCM	✓ One Git repository, shared with every team and tool
Parallel work	Check-out locks: one developer per member	✓ Branches: many developers in parallel, merged safely
Builds	JCL submitted by hand, often full rebuilds	✓ DBB rebuilds only what changed and what depends on it
Feedback	Errors found late, in test or production	✓ Every push is built and checked within minutes
Traceability	Who changed what and why? Hard to answer	✓ Commit history, reviews and build reports

What is Git?

The source code manager behind most open source projects

What is Git?

A distributed version control system (VCS)

- **Git is the source code manager** for the majority of open source projects
- Created in 2005 by Linus Torvalds for the Linux kernel
- Distributed: every developer has a full copy of the history
- Fast branching and merging enable parallel work
- Every change is a commit: who, what, when and why
- Free, open source and supported by almost every IDE and tool



Repository: the project and its history

Commit: a saved snapshot

Branch: an independent line of work

Merge: combine branches

Git servers: where teams share repositories

Platforms that contain Git and build upon it

There are many Git servers (repositories) that contain Git and build upon it. They add collaboration, security and automation around the same Git core.

GitHub

Largest public host of open source; pull requests, Actions CI, Enterprise Server for on-premises

GitLab

All-in-one DevOps platform; merge requests, built-in CI/CD, often self-hosted in companies

Bitbucket

Atlassian product; pull requests, tight Jira integration, Data Center for on-premises

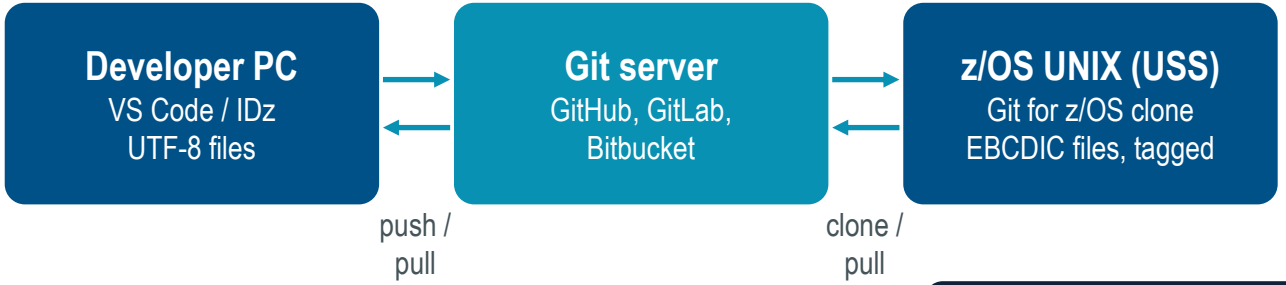


What the servers add on top of Git

Central remote repository • Pull / merge requests and code review • Access control and audit trail • Issue tracking • Webhooks that trigger CI servers such as Jenkins

Git on z/OS: same Git, EBCDIC-aware

Code is stored in UTF-8 in Git and converted on z/OS UNIX



Same commands everywhere:
git clone, add, commit, push, pull

Key points

- Git for z/OS runs in z/OS UNIX System Services (e.g. IBM Open Enterprise Foundation for z/OS)
- Git stores text as UTF-8; mainframe tools need EBCDIC (e.g. IBM-1047)
- .gitattributes tells Git which code page to use per file type
- Check special / non-printable characters before migrating from PDS

.gitattributes (example)

```
# line endings
*       text=auto eol=lf
# z/OS working-tree code pages
*.cbl  zos-working-tree-encoding=ibm-1047
*.cpy  zos-working-tree-encoding=ibm-1047
*.jcl  zos-working-tree-encoding=ibm-1047
*.json zos-working-tree-encoding=utf-8
```

A daily Git workflow for the team

Feature branch workflow: small changes, reviewed, merged often



Good habits

- Pull before you start, push at least daily
- One branch per task or ticket
- Small commits with clear messages
- Never commit build outputs or passwords

Why it improves efficiency

- No member locks: parallel development
- Reviews catch defects before the build
- Every push can trigger an automatic build
- Easy rollback: every state is recoverable

What is Dependency Based Build?

IBM Dependency Based Build (DBB): intelligent builds for z/OS

Dependency Based Build (DBB) at a glance

IBM's build toolkit for z/OS applications stored in Git

1 Groovy for IBM Z

Build scripts in Apache Groovy with Java APIs for z/OS (MVSExec, JCLExec, data sets)

2 Generated automation

Reusable frameworks generate the build: zAppBuild (Groovy) or zBuilder (YAML, DBB 3.x)

3 Dependency scanner

Reads COBOL, PL/I, ASM … and finds copybooks, includes and called programs

4 Dependencies in a database

Scan results are stored in the metadata store (Db2 for z/OS or Db2 LUW)

5 Access to stored information

DBB 1.x: application server with REST API; DBB 2.x+: metadata APIs and CLI

6 Build reports

JSON and HTML report of every build: what was built, return codes, outputs

Groovy for IBM Z and generated automation

From hand-written JCL to scripted and configured builds

Groovy with DBB APIs: compile one COBOL program

```
import com.ibm.dbb.build.*
def compile = new MVSExec().pgm("IGYCRCTL")
compile.dd(new DDStatement().name("SYSIN")
        .dsn("APP.COBOL(PAYCALC)").options("shr"))
compile.dd(new DDStatement().name("SYSLIN")
        .dsn("APP.OBJ(PAYCALC)").options("shr"))
int rc = compile.execute()
```

Groovy for IBM Z

- Run with groovyz on z/OS UNIX
- Same steps as JCL, but versioned in Git and reusable

zAppBuild (Groovy framework)

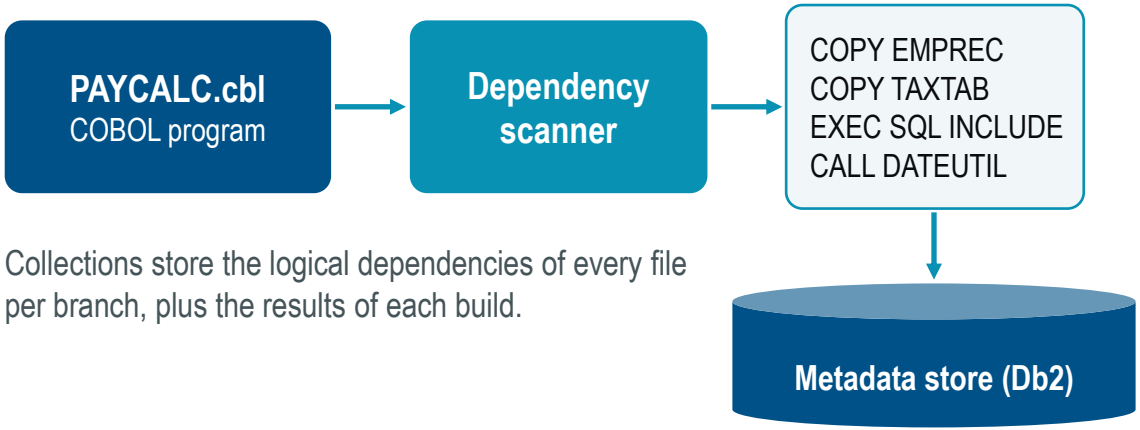
- Open source on GitHub (IBM/dbb-zappbuild)
- Language scripts for COBOL, PL/I, ASM, BMS …
- Full, impact and user builds out of the box

zBuilder (YAML, DBB 3.x)

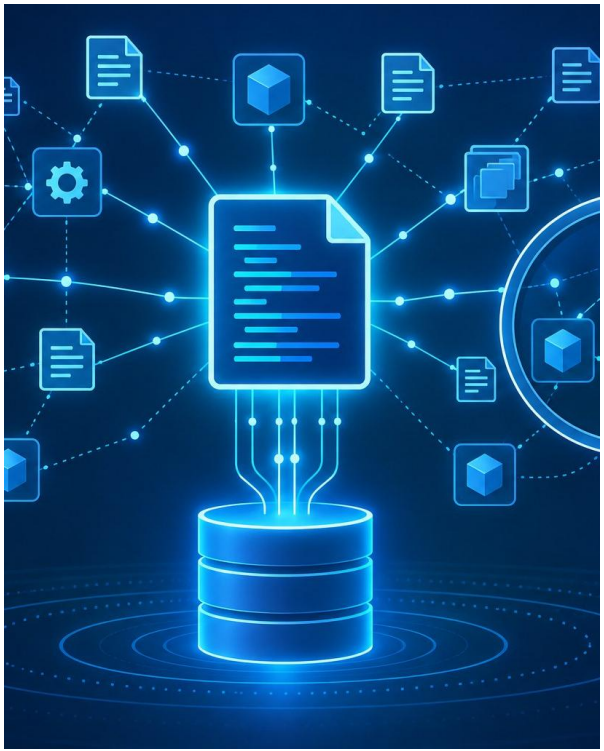
- Configuration instead of code: dbb-build.yaml
- Lifecycles: full, impact, user, merge, pipeline
- Started with: dbb build impact

Dependency scanner and dependency database

Know exactly what is affected by a change



Collections store the logical dependencies of every file per branch, plus the results of each build.



Impact analysis in action (example)

- **Change:** copybook EMPREC is modified in Git
- **DBB asks the database:** which programs use EMPREC?
- **Result:** only those programs are rebuilt, not the whole application

Accessing the stored build information

From a REST application server to direct database access

How the information in the dependency database is accessed



Application server (WebSphere Liberty) on Linux or z/OS

Liberty server removed; toolkit connects directly to Db2

Metadata store CLI with JSON output, build maps, zBuilder

Application server with REST access

- REST API to read and write collections and build results
- Web UI to browse dependencies and builds
- Concept still matters: tools query build data

Today: metadata APIs and CLI

- MetadataStore API in Groovy / Java scripts
- dbb CLI on z/OS UNIX, also via SSH from pipelines
- Security through RACF groups and Db2 JDBC

Build reports: transparency for every build

Generated automatically at the end of each DBB build

- Saved as JSON (for tools) and static HTML (for people)
- Lists every file built, the commands run and their return codes
- Shows outputs (load modules, listings) and deploy types
- Archived by Jenkins next to the build log

Who uses the report?

Developers: see why a build failed
Release managers: know what will be deployed
Auditors: proof of what was built, when and how

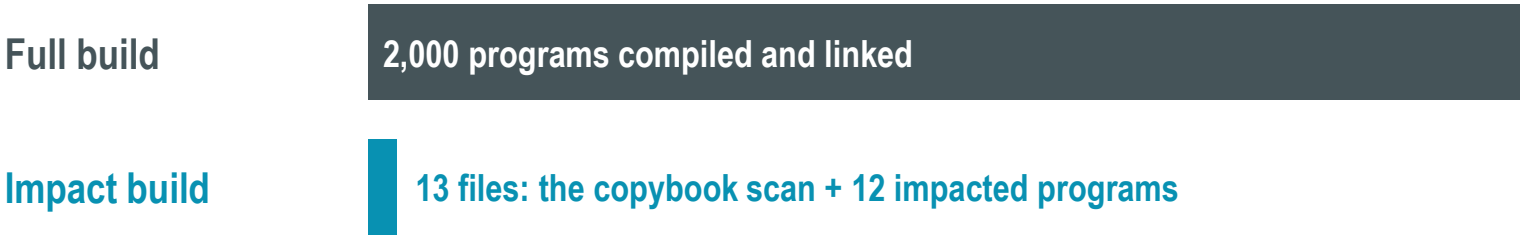


Tip: the deployment step reads the JSON report to know exactly which outputs to package.

Why dependency-based builds save time

Full build versus impact build (illustrative example)

Scenario: an application with 2,000 programs; one copybook used by 12 programs is changed



Faster feedback

Developers get build results in minutes, not hours

Lower cost

Less CPU consumption on the mainframe

Higher quality

No forgotten programs: the database knows every user of a copybook

More releases

Small, frequent, low-risk changes become possible

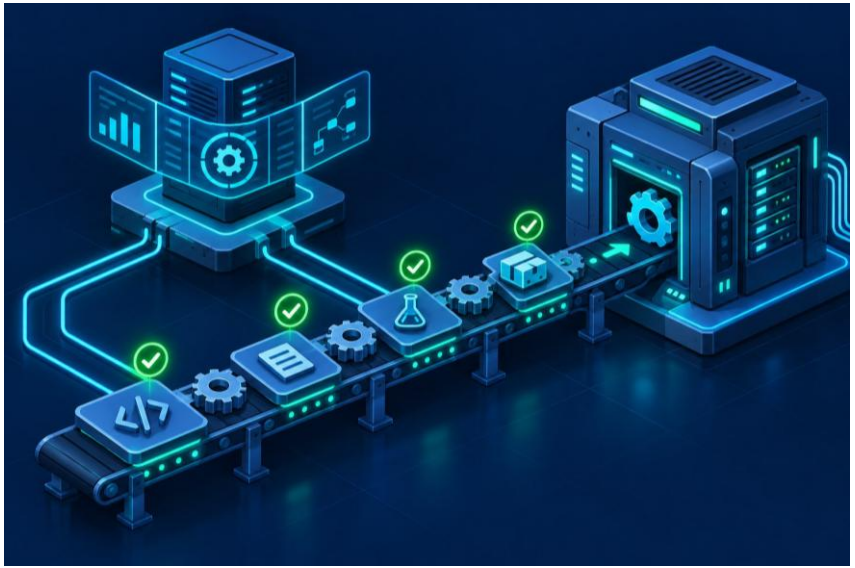
What is Jenkins?

The automation server and its role as an agent on z/OS

What is Jenkins?

Open-source automation server for continuous integration and delivery (CI/CD)

- Automates build, test and deployment on every change
- Pipelines as code: a Jenkinsfile stored in Git
- Over 1,800 plugins, e.g. Git servers, SSH, reporting
- Controller / agent architecture distributes the work



Controller

Schedules jobs, shows results, stores configuration

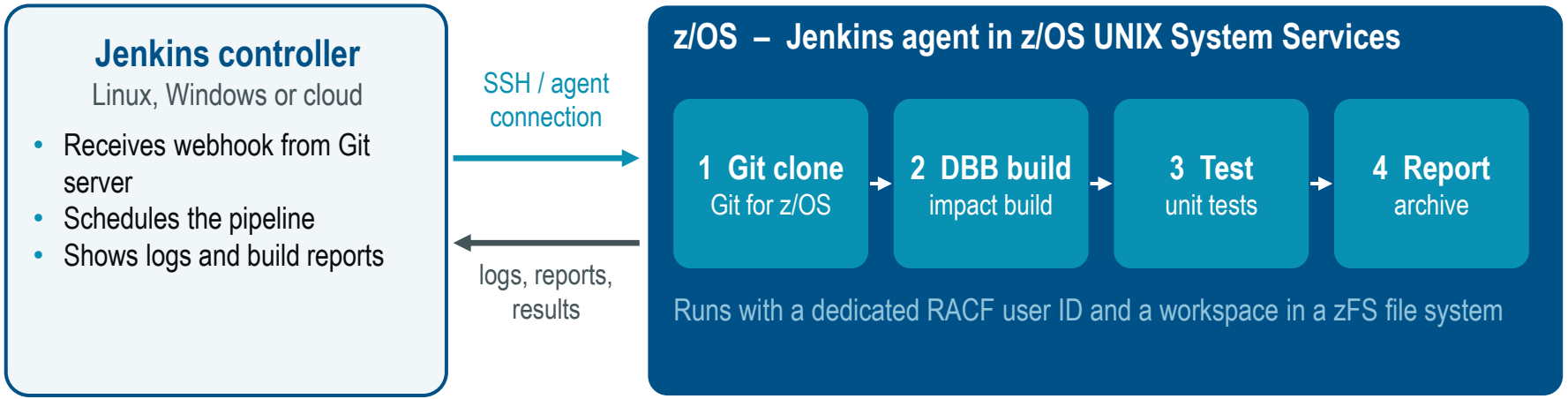
Agent

Executes the steps on its machine, e.g. on z/OS

Terminology: older documentation says master / slave; Jenkins now uses controller / agent.

Jenkins agent ("slave") on z/OS

The controller delegates the mainframe work to an agent on z/OS UNIX



Prerequisites on z/OS

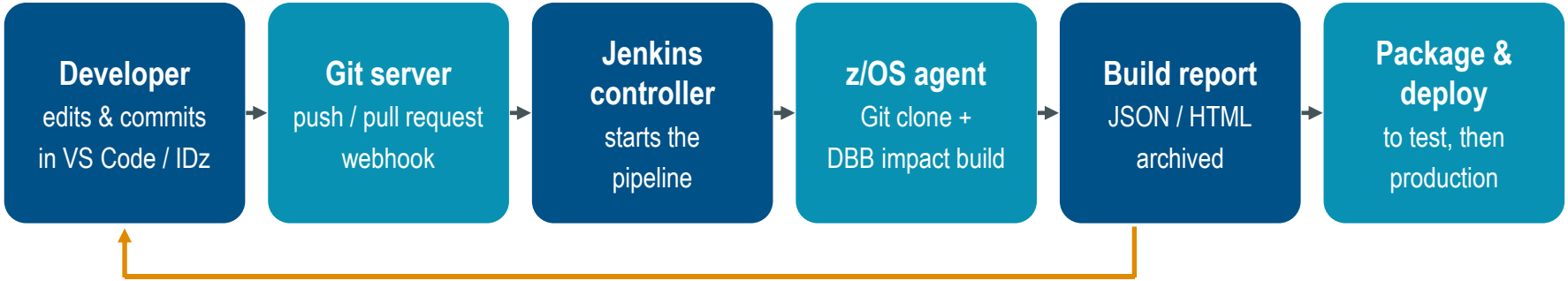
- Java (IBM Semeru) and Jenkins agent software
- Git for z/OS and IBM DBB installed in USS

Why an agent on z/OS?

- Compilers, data sets and Db2 are only on z/OS
- Same pipeline view as for Java or cloud apps

Putting it together: the end-to-end pipeline

From a code change to a tested, deployable build



Jenkinsfile (simplified)

```
pipeline {
  agent { label 'zos' }
  stages {
    stage('Build') { steps { sh 'dbb build impact' } }
  }
}
```

Roles of the three tools

Git stores and versions the code
DBB builds only what is impacted
Jenkins automates and orchestrates it all

Benefits for the team and the product

What changes when Git, DBB and Jenkins work together



Team efficiency

- Parallel work on branches instead of locks
- No manual JCL for builds
- Minutes instead of hours to get feedback
- Shared tools with non-mainframe teams
- Easier onboarding of new developers

Product outcomes

- Fewer defects reach production
- Smaller, more frequent releases
- Complete traceability for audits
- Repeatable, consistent builds
- Lower build cost through less CPU

Key takeaways and knowledge check

Five ideas to remember and four questions to test them

Key takeaways

- Git is the source code manager of most open source projects
- GitHub, GitLab and Bitbucket host Git and add collaboration
- DBB = Groovy for IBM Z + scanner + dependency database + build reports
- Impact builds rebuild only what a change affects
- Jenkins runs the pipeline; its agent on z/OS does the mainframe work

Knowledge check

Q1 Name two Git servers that build upon Git.

Q2 Where does DBB store the dependencies it scans?

Q3 A copybook changes. What does an impact build rebuild?

Q4 What is the role of the Jenkins agent on z/OS?

03 - How Artificial Intelligence Helps Me as a Mainframe Educator

Creating Training Material

Creating Training Materials

- Creating PowerPoint presentations ➔ superNinja & Gemini Notebook
- Audio files that go with the PowerPoint presentations.➔ Gemini Notebook
- Copy the audio files and **edit** them ➔ Audiate (Techsmith)
- Creating the videos using the slides and the finished audio files ➔ Camtasia (Techsmith)
- Translation and grammar➔ DeepL und Grammarly
- Ask questions about the materials ➔ Gemini Notebook & Gemini
- Creating exercises and hands-on activities ➔ Gemini Notebook
- Talk to AI tools ➔ wisprflow.ai

https://notebook.google.com/

© European Mainframe Academy

Beispiel „SuperNinja“

- Prompt an SuperNinja:

  „*Prepare training module slides on "Performance Analysis and Capacity Planning". The module should be interactive and easy to understand, designed to improve team efficiency and product outcomes. Produce .pptx powerpoint presentation with customizable slides and generate 1 or more images for each of the slides but only if these are useful and relevant. Include teacher notes. Review, refine and fix everything before finishing.*
  Content:“

- Aufgabe an SuperNinja um 9:05 Uhr gestartet
- Um 9:40 Uhr kommt die eMail:
  Your task result on Performance Analysis Training Module is ready

© European Mainframe Academy

Get Answers to Open Questions

© European Mainframe Academy

Question:

What are „National Characters“ and why are they called that?

© European Mainframe Academy

The Answer from Gemini (1/3)

fit for mainframe

In enterprise computing, **national characters** are three specific non-alphanumeric symbols that IBM operating systems (such as z/OS, MVS, and OS/390) treat syntactically as letters.

In the standard US EBCDIC character set (Code Page 037/1047), they are:

$ (Dollar sign – **0x5B**)

@ (At sign – **0x7C**)

(Number / Hash sign – **0x7B**)

In contexts like JCL (Job Control Language), assembler labels, RACF profiles, and data set naming conventions, names typically must begin with an alphabetic character (A–Z) or a national character.

© European Mainframe Academy

The Answer from Gemini (2/3)

fit for mainframe

Why Are They Called "National Characters"?

The name originates from early internationalization and code page variances in EBCDIC and 7-bit ASCII/ISO standards (specifically **ISO 646**):

Country-Specific Substitution: Early encoding systems did not have enough code space for accented letters, local currency symbols, or regional glyphs. Standards bodies and IBM reserved a handful of code points (`0x5B`, `0x7B`, and `0x7C` in EBCDIC) to be customized by individual **nations**.

Visual Variation Across Borders: While a US terminal displayed `$`, `@`, and `#`, a terminal configured for another country's code page displayed its own local currency symbol or linguistic character at those exact same binary code points.

© European Mainframe Academy

The Answer from Gemini (3/3)

In the **UK** (e.g., Code Page 285), the $ slot (0x5B) often displayed as the British Pound sign (£).

In **Germany** (Code Page 273), these positions were reassigned to German umlauts (ä, ö, ü, or §).

In **Scandinavia**, they mapped to characters like Æ, Ø, and Å.

Syntactic Equivalence: Because system parsers (like compilers, linkers, and the JCL interpreter) needed consistent rules regardless of the country where the system was running, the architecture defined these three code points as **national characters** and treated them as valid alphabetic equivalents everywhere.

This allowed programmers in different countries to use their local symbols or letters in member names, labels, and identifiers without breaking syntax validation rules.

© European Mainframe Academy

Resources and Links

© European Mainframe Academy

- IBM Developer for z/OS
  IBM Developer for z/OS
- Open Mainframe Project
  https://openmainframeproject.org/
- Open Mainframe Blogs
  https://openmainframeproject.org/blog/
- Skool
  https://www.skool.com/

© European Mainframe Academy

(C) European Mainframe Academy

fit for mainframe

Questions? Remarks?

wolfram.greis@mainframe-academy.de

© European Mainframe Academy