OPEN
MAINFRAME
PROJECT

Modernization Working Group

Mainframe programming languages, Should I Stay or Should I Go

v2 Dec 2024

Executive summary

COBOL, PLI, and other classic mainframe programming languages have been around for decades. Despite numerous predictions of their demise, these languages continue to power critical business applications. However, the ever-increasing challenge with maintaining and improving legacy applications is a growing concern for CIOs and technology leaders.

In this whitepaper, we look into a pressing question: Are these legacy programming languages past their prime? To answer this, we propose a framework for assessing the suitability of continued use of a few mainframe programming languages. Our analysis reveals that while COBOL benefits from a dedicated community and targeted innovation, languages like PLI and REXX may require more immediate modernization efforts.

While we focus on a specific set of programming languages, comparing and contrasting with a few more modern ones, the proposed framework can be applied to any programming language ecosystem, providing a methodical approach for organizations to prioritize and inform their application modernization initiatives.

Navigating the Legacy Landscape: A Complex Decision

The decision to modernize legacy applications written in COBOL, PLI, or other historical mainframe languages is far from straightforward as it involves a complex interplay of factors, including:

Fig1. Mainframe Languages involve a complex interplay of above features.

Application modernization techniques and tools to facilitate the conversion of applications from older programming languages to modern ones have emerged over the last years, more recently powered by AI Large Language Models (LLMs). While AI-powered tools offer potential solutions for automating the conversion process, they have limitations. Scaling these tools to large, complex applications can be challenging, and their ability to capture business logic and handle underlying datasets may be limited.

These tools will keep evolving and eventually will get to a maturity point where they will enable a high degree of automated modernization through AI LLMs. Right now they are still at the initial stage of the hype cycle and a lot of architecting and manual development effort around outputs is still required.

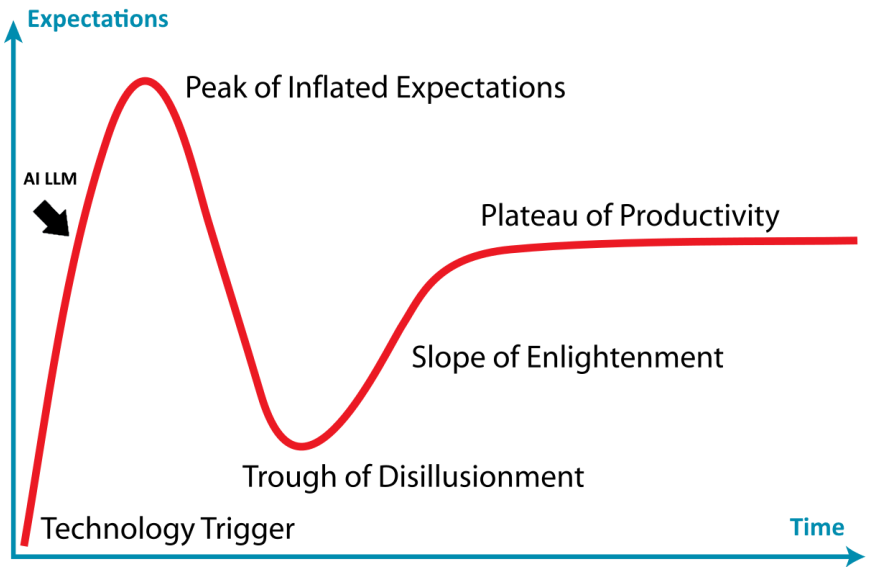


Fig2. Gartner hype cycle applied to AI LLMs automating mainframe applications language conversion

Another important consideration is that moving away from legacy programming languages (and runtimes) requires careful consideration of non-functional aspects. Mainframe runtimes historically required developers to focus primarily on business logic and business level error handling. When modernizing applications to a different platform and runtime, developers will likely need to reevaluate application architecture, non-functionals and technical error handling.

Therefore, to make informed modernization decisions, organizations should evaluate factors like the programming language ecosystem's health, supportability, and the availability of developer tools. Understanding the "shelf-life" of programming languages enables better prioritization of modernization efforts.

This whitepaper aims to provide a structured approach for evaluating the ecosystem around COBOL, PLI, and REXX, comparing them to more recent languages like Java, Python, and Go. Our goal is to equip technology leaders with insights and where to prioritize their application modernization initiatives for mainframe-focused programming languages

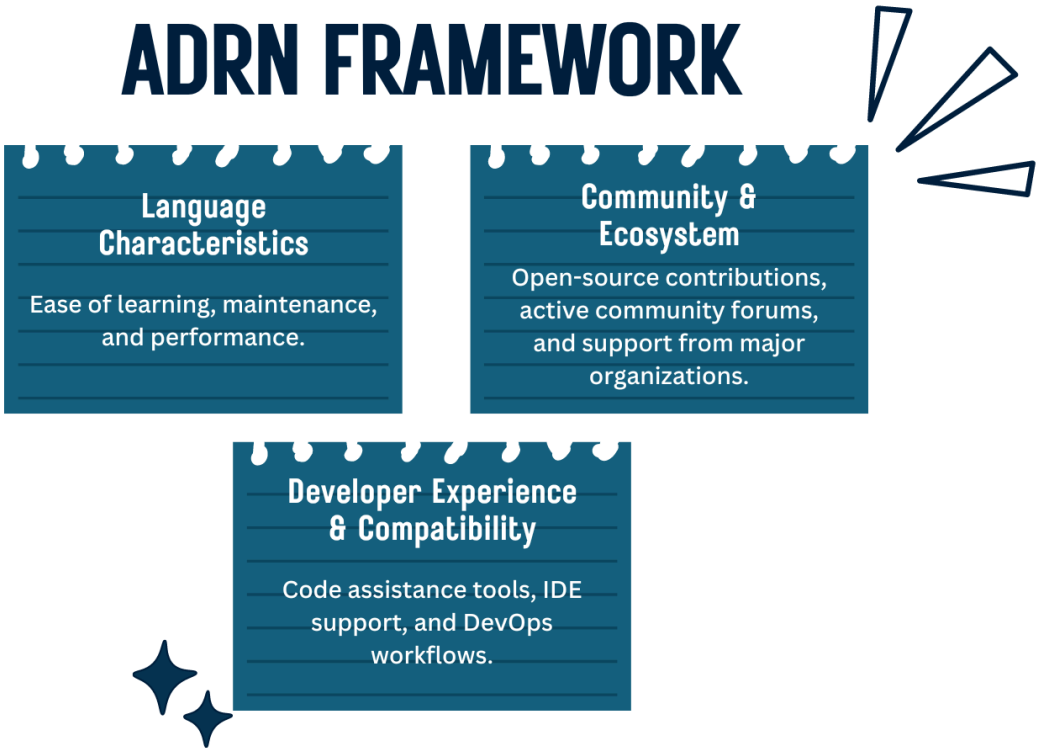
A decision framework

We started the journey to evaluate the supportability and viability of legacy programming languages by exploring existing assessment frameworks and guidelines. Unfortunately, all frameworks we came across didn't provide a straightforward and practical approach .

Frameworks like the Technology Acceptance Model (Davis 1989), Programming Language Evaluation Criteria (Sebesta 2012, #), Balanced Scorecard ("The Balanced Scorecard—Measures that Drive Performance" 1992), FURPS model, and DVF Development Methodology ("Prioritizr - Confluence", n.d.) offer valuable perspectives, but they fall short in assessing the liveness in the ecosystem and supportability of a programming language. For instance, TAM primarily focuses on user acceptance, BSC lacks technical depth, and PLEC often overlooks strategic and community-related aspects.

Recognizing these limitations, we developed a framework that addresses both practical, technical and strategic factors. Our framework (ADRN) covers the following key areas:

- **Language Characteristics:** Ease of learning, maintenance, and performance.
- **Community & Ecosystem:** Open-source contributions, active community forums, and support from major organizations.
- **Developer Experience & Compatibility:** Code assistance tools, IDE support, and DevOps workflows.



To validate the proposed framework, we applied it on a subset of popular mainframe and non-mainframe languages, covering legacy languages like COBOL and modern languages like Go. This comparison demonstrated the framework's versatility and applicability to any programming language, providing a holistic view for informed decision-making regarding a programming language's relevance and risk of continued use.

Are mainframe programming languages past due?

Through our research, we compiled information across a set of key characteristics that should influence an organization's decision to keep using or replacing a programming language:

1. Easiness of access to source code
2. Syntax easiness of understanding
3. Relative performance for use cases
4. Relative number of open source repositories and forums
5. Key open communities
6. Key enterprise backers
7. Availability of code assistants
8. Availability of IDEs
9. Easiness of integrating with modern app dev patterns

We then analyzed popular mainframe languages (COBOL, PLI, REXX) and non-mainframe languages (Java, Python, and Go), leading to the findings summarized below. It is important to note that while Java, Python, and Go are supported on IBM mainframes, we have not looked into implications of modernizing onto these languages on mainframe as such capability is not widely available for other legacy mainframe systems.

Language Characteristics

- **Legacy challenges:** Compiled legacy languages, especially those where source code was lost, can be difficult to modernize as reverse engineering is hard to execute against compiled code. Documenting end-to-end processes is required to successfully reimplement functionality.
- **Verbosity:** Legacy languages tend to be verbose which is associated with complexity. However, verbosity can enhance readability at a micro level while it can hinder understanding at a macro level. Thus, a targeted and progressive modernization is the best option.

Community and Ecosystem

- **Open source impact:** Open-source communities, forums, and enterprise backing have significantly influenced the longevity and supportability of programming languages.
- **COBOL Revival:** COBOL has experienced a resurgence with the emergence of new communities, tools, and modernization initiatives. Its continued use in industries like finance and government has driven investment and innovation, extending its longevity.
- **PL/I and REXX Challenges:** PL/I and REXX have not been as fortunate in terms of community support and innovation as COBOL.

Developer Experience & Compatibility:

- **Developer tools:** COBOL saw gaps being filled with new DevOps tools being released and code assistants powered by AI which lowers developers ramp-up. However the adoption of new IDEs and DevOps tooling for mainframe hasn't been consistently prioritized.
- **Modern architecture patterns:** There are quite a few software industry heavyweights that made easier integrating legacy applications with modern applications that make use of microservices, serverless and event-driven architectures. Such solutions become a stop-gap for the short and medium term.

Overall COBOL is currently in a stronger position regarding supportability than other mainframe languages, partly due to its widespread use across various industries as depicted in Fig.3. However, it's important to note that COBOL was primarily designed for large-scale data processing and may not be the ideal choice for new general-purpose programming projects. Despite this, existing COBOL applications continue to benefit from a relatively healthy ecosystem compared to other mainframe languages.

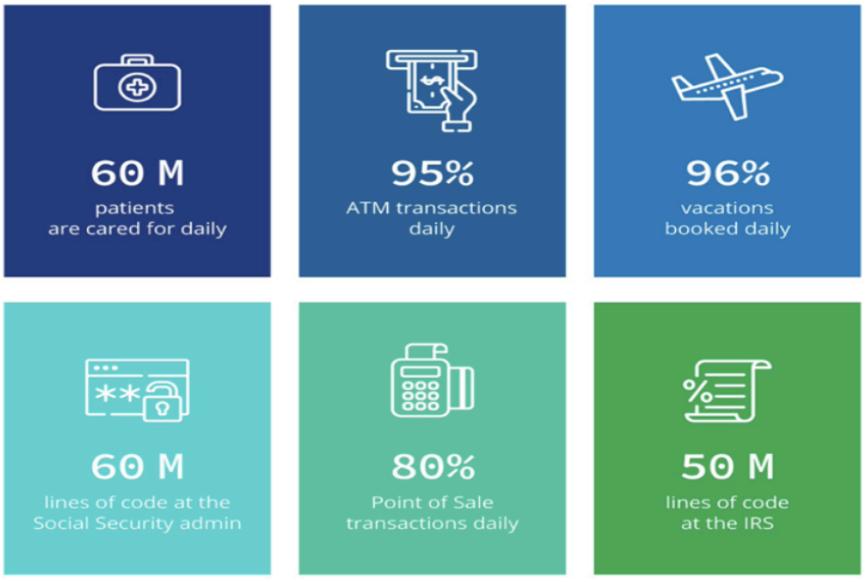


Fig3. COBOL current use across different North America services, Source: Rocket software / MicroFocus, 2023

Conclusion

In conclusion, the proposed framework provides a useful approach for assessing the viability of programming languages and their ecosystems. When applied to mainframe languages, we found that COBOL possesses a reasonable ecosystem and support, making it suitable for continued use where requirements have not changed dramatically, and where reliability and stability in large-scale, data-intensive applications is required.

Organizations using COBOL still need to invest in modernization efforts to integrate their legacy applications with newer technologies and development patterns. This will ensure COBOL applications can keep up with today's rapidly evolving technological landscape. Where legacy applications require extensive code rewriting due to changing needs, a progressive and modular approach to modernization and replacement will be the best fit.

Furthermore, our analysis suggests that REXX and PLI may be more suitable candidates for expedite replacement than COBOL, especially for organizations considering application migration or modernization initiatives triggered by extensive business changes.

References

[1] IBM 2024 https://www.ibm.com/topics/mainframe-modernization

[2] “The Balanced Scorecard—Measures that Drive Performance.” 1992. Harvard Business Review. https://hbr.org/1992/01/the-balanced-scorecard-measures-that-drive-performance-2.

[3] Davis, Fred D. 1989. “Technology Acceptance Model (TAM).” Innovation Acceptance Lab. https://acceptancelab.com/technology-acceptance-model-tam.

[4] Kizior, Dr. Ronald J., Donald Carr, and Dr. Paul Halpern. 2000. “Does COBOL Have a Future?” 17 (126): 5. 035ab4d33540014200b329436671249ce251254b.

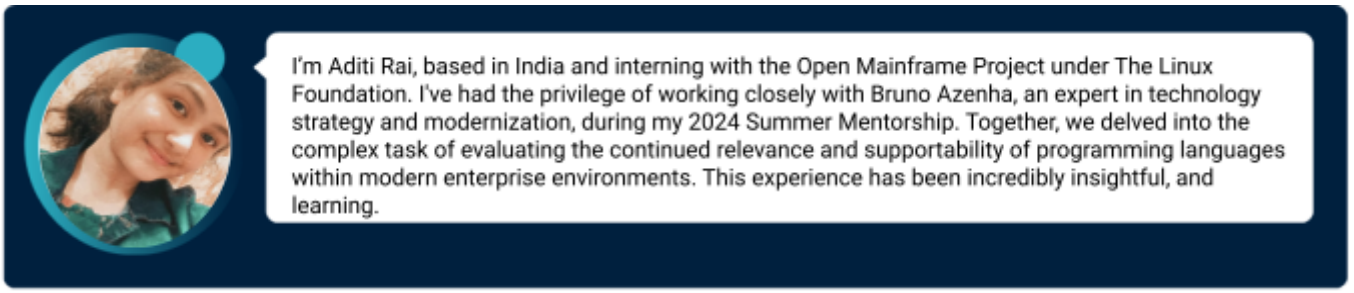
[5] “Prioritizr - Confluence.” n.d. Atlassian. https://tryble.atlassian.net/wiki/spaces/prioritizr/pages/254738466/Design+Thinking+Desirability+Viability+and+Feasibility+DVF.

[6] Sebesta, Robert W. 2012. *Concepts of Programming Languages*. N.p.: Pearson.

About this paper

Our authors

This document was developed under the Open Mainframe Project's Summer 2024 Mentorship program and sponsored by the Mainframe Modernization Working Group.



Special thanks

This work would not be possible without the useful suggestions and feedback of members of the Mainframe Modernization Working Group, part of the Open Mainframe Project. Special thanks to Vinu Viswasadhas, Swathi Rao, Misty Decker and Sonya Wilde for their support and feedback that greatly helped shape the content in this white paper.

About the Modernization Working Group

The Modernization Working Group, part of the Open Mainframe Project, intends to be a focal point for thought leadership around what it means to modernize mainframe applications.
We are an open group passionate about knowledge sharing, leading constructive discussions, challenging status quo, exploring creative solutions, and generating contents that will help the community in their journey.
Learn more about the working group at:
https://openmainframeproject.org/our-projects/working-groups/modernization-working-group/

Appendix I - Comparison of programming language

The definition for each key characteristic we have analyzed is as follows:

1. Easiness of access to source code
Interpreted languages provide source code access and portability.
Compiled applications require maintaining source code separately and have portability limitations.

2. Syntax easiness of understanding
An easy to follow language syntax is important to both beginners and experienced programmers.

3. Relative performance for use cases
How efficiently a programming language executes tasks compared to others and what are the primary use cases

4. Relative number of open source repositories and forums
A larger number of repositories often indicates a vibrant ecosystem and community support.

5. Key open communities
Communities like official foundations, non-profit organizations, or prominent online user groups that typically provide support and continuous innovation.

6. Key enterprise backers
Prominent software companies that offer software tools, compilers, runtimes and contribution to programming language standards evolution

7. Availability of code assistants
Tools and technologies that help developers write and debug code faster and more efficiently

8. Availability of multiple options for IDEs
A wide range of IDE options can provide developers with more flexibility and choices tailored to their workflow.

9. Easiness of integrating with modern App Dev patterns
How easily a programming language can be extended or integrated with cloud-native application architecture and principles (microservices, containers, serverless, event driven and others)

The next set of tables summarize all the findings for the languages in scope of this analysis.

Criteria	COBOL	PL/I	REXX	Java	Python	Go (Golang)
Language Characteristics						
Easiness of access to source code	Compiled - need to safeguard source code	Compiled - need to safeguard source code	Interpreted - source code always available	Compiled or Interpreted	Interpreted - source code always available	Compiled - need to safeguard source code
Syntax easiness of understanding	Descriptive English-like syntax but with high verbosity, leading to initially steep learning curve	Complex syntax with steep learning curve, combines features from multiple languages	Moderate learning curve, focused on scripting	Moderate learning curve, well-documented with extensive learning resources	Gentle learning curve, readable syntax, extensive tutorials	Moderate learning curve
Relative performance for use cases	High - Optimized for business data and records processing focused on non-technical developers	High - Optimized for system programming	Lower - Focused on sysadmins and scripting tasks around automation and text processing	Moderate - Good performance with JIT compilation and focused on general programming	Lower - slower than compiled languages with a strong focus on data analysis	High - Designed for performance and focused on general programming
Community & Ecosystem						
Relative number of open source repositories and forums	Just a few modern projects and communities focused on specialized use cases	A niche language with a limited number of repositories and forums	Very few but specialized forums	A robust ecosystem with a large number of open-source projects and active forums	An extensive community with numerous forums and open-source repositories	A growing number of projects and increasing community
Key open communities	OpenMainframeProject, SHARE, Free Software Foundation / GnuCOBOL	SHARE, IBM	Rexx Language Association, IBM Rexx	Java Community Process	Python Software Foundation, Google,	Google
Key enterprise backers	IBM, RocketSoftware, Broadcom, Fujitsu, Kyndril	IBM, Rocket Software / Microfocus	IBM , Rexx Systems	Oracle, IBM, Red Hat	Google, Microsoft, Facebook	Google, Red Hat
Developer Experience						
Availability of code assistants	Watson Code Assistant, cobol copilot	Watson Code Assistant,	IBM TSO/E	IntelliJ IDEA, Eclipse , Visual Studio Code	Github Copilot , Tabnine ,	VS Code, JetBrains GoLand, Kite

Criteria	COBOL	PL/I	REXX	Java	Python	Go (Golang)
Availability of multiple options for IDEs	Mix of legacy and modern IDEs like Visual Studio, Eclipse, Rocket Software Enterprise Developer, IBM Developer for z/OS, IBM WaziCode and Fujitsu NetCOBOL	Mostly legacy tooling with some recent IDEs provided by Rocket Software / Microfocus and IBM	Limited tooling, primarily within IBM environments. Examples: oorexx, IBM TSO/E	Extensive tooling and IDE support, including Eclipse, IntelliJ, VSCode and others	Extensive tooling and IDE support, including PyCharm, VS Code, and Jupyter Notebooks	Growing tooling and IDE support, including VS Code, JetBrains GoLand, LiteIDE and other modern tools
Compatibility with New Technologies						
Easiness of integrating with modern App Dev patterns	Moderate - Not designed for cloud native architectures but with an ecosystem of tools to facilitate integration. Examples: ZOWE, RocketSoftware, IBM and others)	Low - Not designed for cloud native architectures and not easy to integrate / extend	Low - Not designed for cloud native architectures	High - Supported by frameworks like Spring Boot, Eclipse Vert.x, Quarkus .	High - Excellent support for microservices, serverless functions	High - Designed with cloud native principles in mind