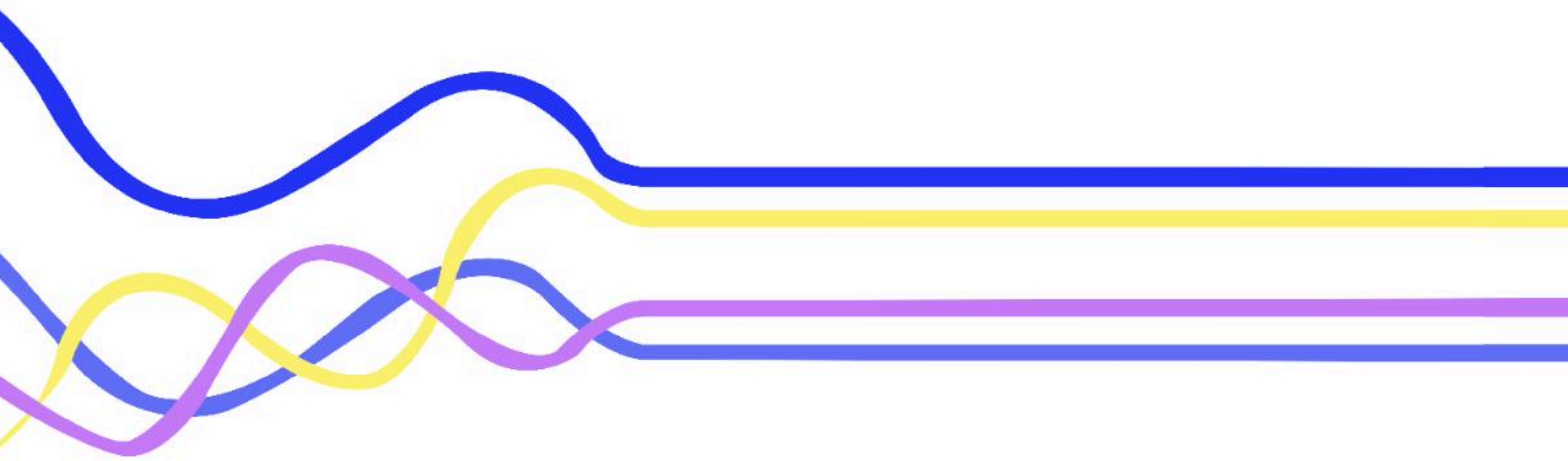
Whitepaper

Accelerating COBOL modernization - closing the knowledge gap



Introduction: The COBOL knowledge gap in modernization

Since its initial development in the 1950s, COBOL has expanded to hundreds of versions and over 200 billion lines of code powering critical systems in banking, insurance, government, and beyond.

Many enterprises face pressure to modernize part or all of their COBOL applications. Yet, despite significant investment, mainframe modernization projects are notoriously difficult. One industry survey revealed 79% of modernization initiatives failed. A second Gartner study found that 82% of initiatives take longer than expected.

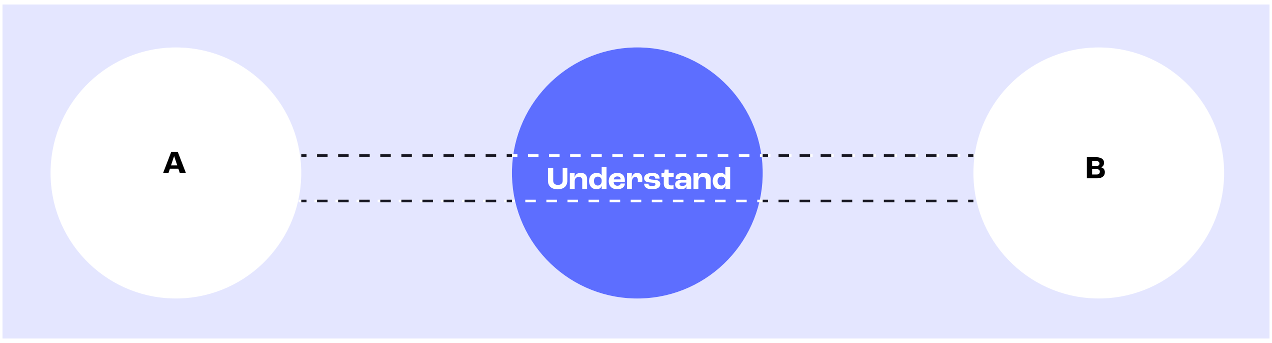
The major contributing reason is the “knowledge gap” that exists in COBOL which causes unexpected delays and risks due to missing information.
The main components of the gap are twofold:

- **Application understanding gap**: Knowing all the components, connections, where and what are the dependencies, and how it works from business and technical perspectives
- **Skills gaps**: Having enough subject matter experts to analyze the code and institutionalizing expert’s knowledge

These gaps make it notoriously difficult to successfully modernize or even maintain applications. Swimm uses new techniques intertwining static analysis with generative AI to eliminate the understanding gap and reduce critical dependencies on skilled labor.

Application understanding

Put simply - going from A to B requires understanding your application.



At a high level, a modernization process involves an overall map of the application, understanding functionality, building a spec, developing the new code, and validating the code before and during transitions.

At each of these stages application understanding is a core outcome for a successful migration.

Map	Understand/Plan	Develop	Validate
What do we have? What part should we modernize? What is the right modernization path? Where are the bottlenecks to improving performance? Where do we want to change or add functionality?	What are the business rules in the application? What dependencies exist for any given part of the application and what do they do? How do users experience the application and what are the related flows in the code? What are the product requirements for the modernized application?	How can we generate code with AI with all the relevant context?	What tests do we need to run to cover all potential cases and ensure a successful transition? What features do we need to ensure are implemented?

As anyone that has gone through modernization knows - and those about to will know - building an understanding for an entire application isn't easy. Key challenges include:

- **Scattered business logic and rules**: Business rules— if/then/else logic, constraints, and actions governing business processes—are spread across millions of lines of code, intertwined with technical logic (e.g., database access via CICS or IMS). For example, a rule like "High-value transfers require verification" may span multiple COBOL programs, making it hard to trace.

- **Complex control flow**: COBOL's use of PERFORM statements, GOTOs, and dynamic calls (e.g., EXEC CICS XCTL PROGRAM(CDEMO-TO-PROGRAM)) creates intricate execution paths that are difficult to follow.

- **Technical environment**: A significant portion of COBOL code handles infrastructure (e.g., DB2, VSAM datasets), obscuring business logic.

- **Cryptic naming**: Variable names like WS-V-02 or ACTIDIN often lack descriptive meaning, complicating analysis.

- **Obsolete logic**: Legacy systems may include outdated rules no longer aligned with current policies, requiring careful validation. Some of the existing code is actually "dead code", as it is no longer executed, but this is challenging to understand from the code itself.

- **Knowledge loss**: Original developers and business analysts have often left, leaving no institutional memory - i.e. documentation - to explain the code.

An overview of existing attempts to close the gap

There are 3 main approaches that we've observed to improve understanding of complex applications:

- Manual engineering effort
- Static analysis tools
- Generative AI

The first labor intensive approach has a poor track record of failures along with cost overruns and timeline extensions. It usually involves many people who first have to go through the existing system and understand how it operates. Some organizations rely on their own engineers, who are then unable to use their precious time on developing new features.

Other organizations outsource this task, which often leads to challenges as the outsource task force does not fully comprehend the specific business logic and needs of the codebase.

Existing static analysis approaches and new attempts to use generative AI to automatically bridge the gap but fall significantly short of an effective solution. In the table below, we've broken down the strengths and gaps of using these approaches during the critical step of extracting business rules.

For a deeper dive into these shortcomings, read our primer: Shortcomings of traditional COBOL business rule extraction methods

Comparison of common automated approaches for cobol business rule extraction

Core functionality needed	Static analysis	LLMs / AI assistants
Parse COBOL syntax & structure	Good	Partial
Analyze complex control flow	Partial	Partial
Trace data flows across programs	Partial	Poor
Interpret external dependencies (JCL etc.)	Partial	Poor
Identify semantic meaning / business intent	No	Partial
Handle ambiguity & legacy code quirks	Partial	Partial
Provide holistic system view	Poor	Poor
Validate extracted rules accuracy	No	No

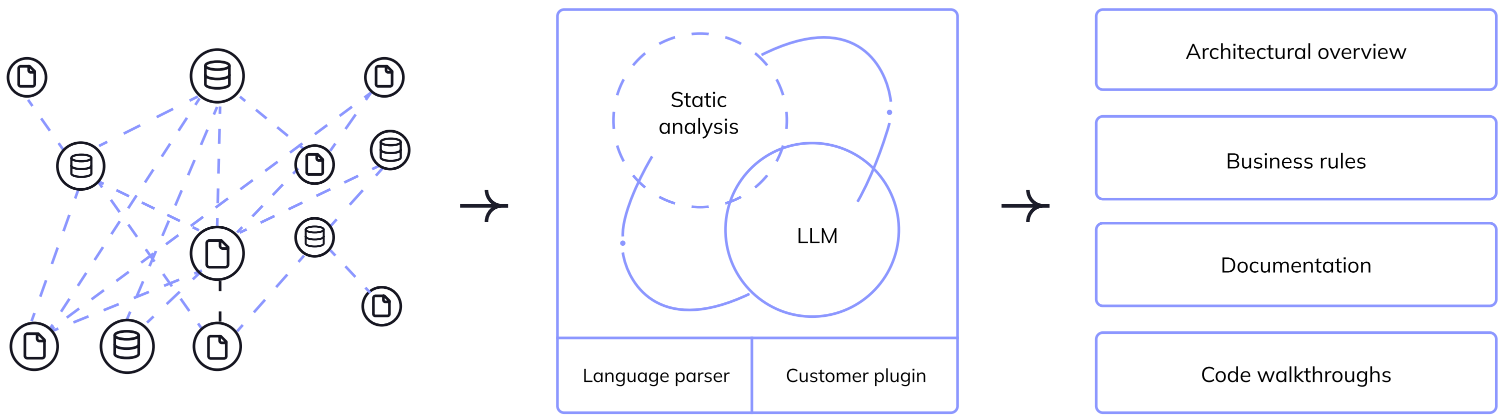
Swimm's approach to application understanding

Exceptional results intertwining static analysis and LLMs

Swimm uses proprietary static analysis with LLMs first to break down a codebase into knowledge artifacts. A knowledge artifact can be as small as a variable or a paragraph, or bigger such as a full control flow, process, job, or other construct that can reside in a single program or expressed across the entire codebase.

These artifacts represent the application functionality. Brought together these artifacts make up a local knowledge base, that is Swimm's internal representation of the codebase.Swimm then turns these artifacts into a comprehensive understanding of the application for architects, engineers, and business analysts.

Reliably turn code into insights



research-based approach - evaluate new ideas and apply quality assurance since we know the expected output at each stage. LLMs alone do not inherently allow this. By relying on static analysis, we can ensure that the output is always correct.

- **Optimized for LLMs**: LLMs aren't designed for analysis. They are probabilistic at their core. Over providing them with too much context has negative returns for quality and reliability. Our approach enables us to provide the exact context needed to each step in the process.

- **Comprehensive and scalable**: We can generate information for massive codebases in hours without losing details since we shared the codebase and individually build only the necessary information.

Swimm also provides additional benefits for a holistic solution to solving the knowledge gap:

- **Ensures up to date knowledge**: Code continues to evolve as regulatory or product requirements continue to change even during a modernization project. Swimm connects code to each artifact in the knowledge base, tracks changes to the code, and continuously updates the knowledge base as needed. Significant changes to the codebase can be escalated to engineers for adding additional insights that may not be apparent in the code. By continuously updating the knowledge base, rather than regenerating it, Swimm can retain the unique knowledge added by engineers or business analysts even as the code evolves.

- **Verifiable**: Engineers can easily click into the code itself from Swimm documents enabling them to verify key details quickly and efficiently.

- **Allows additional insights**: Swimm documents are easily edited in the IDE allowing experts to further enrich documents as necessary. Since code is directly linked to relevant documentation it is trivial to know exactly where to add more insight. When a human adds additional insights, Swimm will not overwrite them but will alert you if the underlying code they refer to changes.

Insights that matter

Swimm surfaces important insights based on the core aspects of understanding your application while enabling deep diving into any part of the code as necessary.

Returning to our core modernization process:

Map	Understand/Plan	Develop	Validate
What do we have? What part should we modernize? What is the right modernization path? Where are the bottlenecks to improving performance? Where do we want to change or add functionality?	What are the business rules in the application? What dependencies exist for any given part of the application and what do they do? How do users experience the application and what are the related flows in the code? What are the product requirements for the modernized application?	How can we generate code with AI with all the relevant context?	What tests do we need to run to cover all potential cases and ensure a successful transition? What features do we need to ensure are implemented?

Map

Visibility into the entire application in an easy to comprehend format is essential to understand what exists and how it is connected.

Swimm surfaces all the key elements of the application. Many applications were written for technical, industry, or business reasons with cryptic naming conventions that make applications particularly difficult to visualize at both high levels and when diving into the code itself.

Swimm organizes and displays programs, jobs, screens, and flows with natural language names and overview descriptions. Connections between the parts of the application are shown along with clear dependency diagrams and the ability to further deep dive into any part of the applications to peel away its layers.

Understand

Understanding the entirety of the application requires the easy ability to understand core entities and the business rules that run behind them.

Business rules

Extracting all of the business rules from an application is the single most important action once the decision has been made to modernize the application into a new application. Before we discuss business rules at length, let's define the meaning of business rules since the term can be used in different contexts.

There have been many attempts to define "business rules" formally (for example, by the Business Rules Group), which stated that a business rule is "a statement that defines or constrains some aspect of the business".

Some business rules embedded in the application protect organizations from legal or regulatory consequences. For example, the collection of a tax like a VAT or customer identification in Know Your Customer laws.

There are lots of business rules are embedded in code "Information systems normally implement a large number of business rules, for example, the 627 business rules applied in a 12,000-line COBOL application and the 809 in a 30,000-line COBOL application"

In essence, business rules are the specific constraints, conditions, and actions embedded within a software system that reflect the policies and procedures of an organization. Legacy systems contain strategic business rules that govern the business processes, whether these rules are explicit or implicit.

Joubert, Pieter, Jan Hendrik Kroeze and Carina de Villiers. "A grammar of business rules in Information Systems." The Journal for Transdisciplinary Research in Southern Africa 9 (2013): 36.

The concept of business rule extraction has been heavily covered in academic research since the 1990s. Many companies in the 2000s from IBM to smaller vendors have recognized the importance of business rules to modernization projects and attempted to bring products to market to automatically extract them from code. A review of the shortcomings of popular approaches was discussed previously in this whitepaper.

Automatically extracting business rules is difficult. Only through adding the power of LLMs into the process with static analysis has it been possible to overcome the technical challenges and approach the non-technical.

Technical challenges

- **Scattered and Intertwined Logic**: In COBOL, tracking the flow of logic is challenging, both within a single file and across multiple files. Furthermore, this logic is frequently intertwined with presentation logic, technical code, and auxiliary code. COBOL programs can have complex control flow structures, including deep calling hierarchies (PERFORMs) and sometimes GOTOs, making it hard to trace the sequence of operations that constitute a business rule.

- **Technical Environment Complexity**: A significant portion of the COBOL code (as much as 70-80% according to some sources) is dedicated to the technical environment and infrastructure (e.g., IMS, CICS, DB2) rather than the core business logic and needs to be filtered out. Specifically, distinguishing between data variables that are relevant to the business and those that are part of the technical framework is crucial but difficult.

- **Naming Conventions**: It is common for COBOL code to include cryptic variable or paragraph names that do not provide any insight into the purpose of the code.

Non-technical challenges

- **Obsolete Business Logic**: Legacy systems may contain obsolete business logic that was never removed from the codebase. Identifying and distinguishing this from current, active rules adds to the difficulty of extraction.

- **Business Knowledge Loss**: Over time, the original developers and business analysts who understood the rationale behind certain rules may have left the organization making it challenging to validate extracted rules.

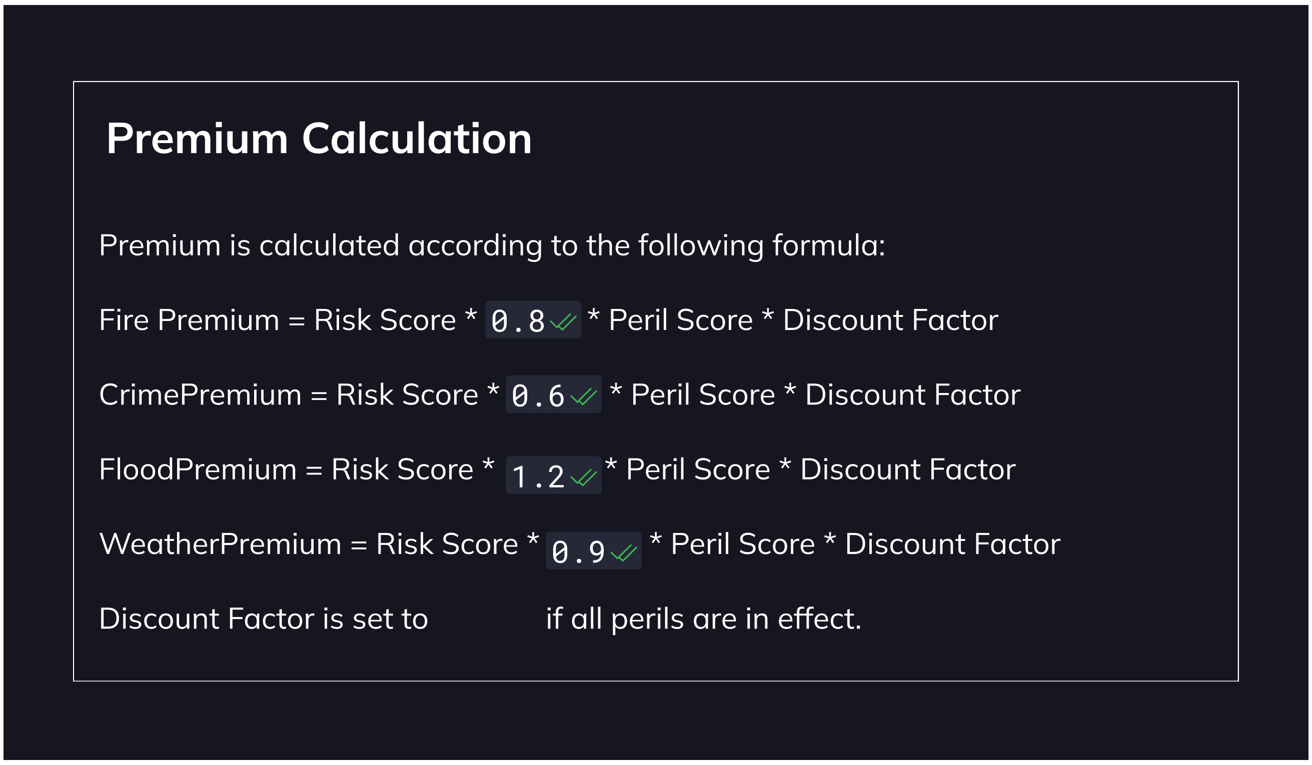
Business Rule Extraction with Swimm

Swimm overcomes the inherent challenges in business rule extraction with a symbiotic approach that fuses the deterministic precision of static analysis with the semantic understanding of Large Language Models (LLMs). This hybrid model ensures accuracy, completeness, and traceability of business rules.

1. **Foundation of Static Analysis**: The process begins with Swimm's proprietary static analysis engine mapping the entire application. It meticulously traces data flows across programs, deciphers complex control flows (including GOTOs and PERFORMs), and filters out technical "noise" from the underlying business logic. This creates a complete, accurate, and structured knowledge graph of the application—a reliable foundation that is impossible for an LLM to build on its own.

2. **Context-Rich AI Enrichment**: With this structured blueprint in place, Swimm then leverages an LLM. Instead of pointing the AI at millions of lines of raw, ambiguous code, Swimm provides it with precise, context-rich snippets from the knowledge graph. This targeted approach prevents AI "hallucinations" and allows the LLM to do what it does best: provide natural language explanations from clear instructions.

The result is a comprehensive and verifiable catalog of business rules. Each rule is presented in plain English, linked directly back to the specific lines of code it came from, and accompanied by diagrams illustrating its logic flow. This allows analysts to not only understand a rule like “A premium customer's credit limit is increased by 15%” but to instantly verify its implementation in the source code, ensuring accuracy and building trust in the modernization process.



Premium Calculation

Premium is calculated according to the following formula:

Fire Premium = Risk Score * `0.8` ✓ * Peril Score * Discount Factor

CrimePremium = Risk Score * `0.6` ✓ * Peril Score * Discount Factor

FloodPremium = Risk Score * `1.2` ✓ * Peril Score * Discount Factor

WeatherPremium = Risk Score * `0.9` ✓ * Peril Score * Discount Factor

Discount Factor is set to   if all perils are in effect.

Core application functionality

While extracting business rules, modernization teams also need to ensure a deep knowledge of the workings of the application to write the specifications of the modern application.

Mainframe applications can be broken down into three core entities which are used to understand the application as a whole:

- **Online operations**: Screens and the workflows they interact with
- **Batch operations**: Jobs, often written in JCL, that handle high-volume, repetitive tasks
- **Utilities**: Copybooks and complex shared logic

To understand an application, each of these can be further broken down into the most useful questions.

- **User screens**: What screens are available to the user? What business rules apply to each screen?
- **Screen interactions**: What actions can be performed on each screen? What are the limitations and validations for each screen? What happens when a user interacts with the screen? How is the underlying logic implemented?
- **Batch operations**: What batch jobs exist, and when do they run? What files are used as input? How are records parsed and processed? What fields impact downstream calculations?
- **Batch job logic**: What does each batch job do? What business rules are involved in a batch job?
- **Shared logic**: Are they complex routines used across multiple operations? If so, what do they handle?

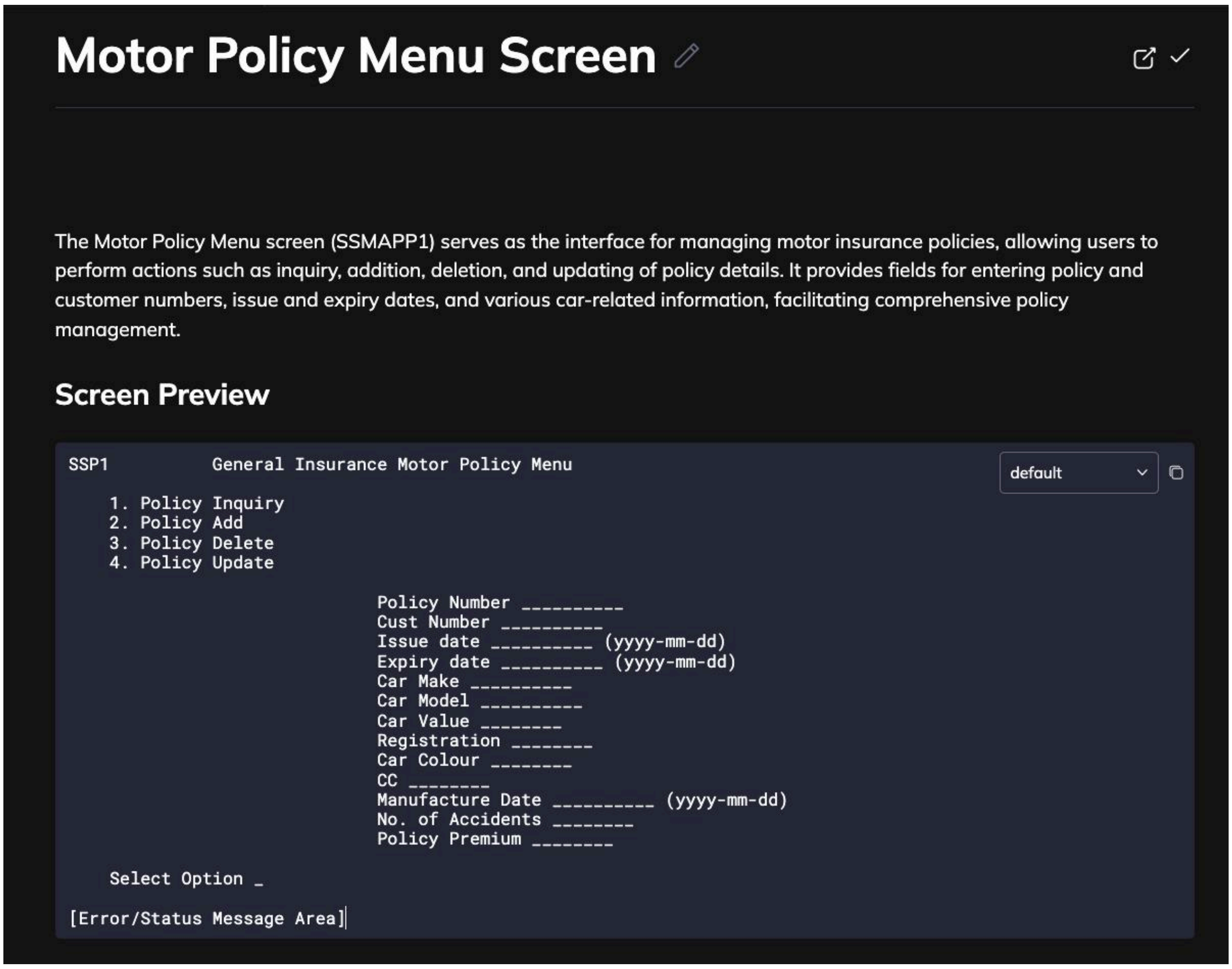
Swimm delivers structured insights to answer each of these questions while also being flexible to dive deeper into any element an analyst or engineer needs to further focus on.

Screens

Swimm reconstructs screen visuals from COBOL BMS and related code. These visual representations are accompanied by explanations of the purpose of the screen along with drill downs into every consideration involved in the screen.

1. **Understand the screen's purpose**
2. **Input limitations and validations**
3. **Understanding user interactions and constraints**
4. **Step-by-step code walkthroughs**
5. **Dependency mapping**

Just seeing a screen already provides significant insight. Swimm's insights go far further giving clarity to any line of questions relevant to modernizing or working with the screen.



Motor Policy Menu Screen

The Motor Policy Menu screen (SSMAPP1) serves as the interface for managing motor insurance policies, allowing users to perform actions such as inquiry, addition, deletion, and updating of policy details. It provides fields for entering policy and customer numbers, issue and expiry dates, and various car-related information, facilitating comprehensive policy management.

Screen Preview

```
SSP1          General Insurance Motor Policy Menu

    1. Policy Inquiry
    2. Policy Add
    3. Policy Delete
    4. Policy Update

                              Policy Number __________
                              Cust Number __________
                              Issue date __________ (yyyy-mm-dd)
                              Expiry date __________ (yyyy-mm-dd)
                              Car Make __________
                              Car Model __________
                              Car Value ________
                              Registration ________
                              Car Colour ________
                              CC ________
                              Manufacture Date __________ (yyyy-mm-dd)
                              No. of Accidents ________
                              Policy Premium ________

    Select Option _

[Error/Status Message Area]
```

Batch Operations

While individual screens provide critical insight into a mainframe application, much of the core business logic resides in batch operations. These jobs process large volumes of data, execute key financial calculations, and generate essential system updates.

Swimm enables teams to understand batch jobs from high level overviews to uncovering layered flows and business rules.

1. **Decision logic flowcharts**
2. **Critical path identification**
3. **Input file definitions**
4. **File-handling routine traces**
5. **Exception handling routines**

Customizable, secure, fast - enterprise ready

In order for an Application Understanding Platform to successfully close the knowledge gap, it needs to work on real COBOL applications in security conscious environments.

Language and customer frameworks support
No two COBOL applications are the same. Firstly, there are approximately 300 dialects of COBOL in use. In addition, when many applications were written, standardization of solutions to common problems like open source libraries and question answer services like StackOverflow didn't exist.

The developers of the codebase had to reinvent the wheel every time they needed to do something that is now a one-liner in a modern language. They had to write their own libraries, and their own APIs. As a result, each COBOL codebase is unique and complex.

R. Lammel and C. Verhoef, "Cracking the 500-language problem," in IEEE Software, vol. 18, no. 6, pp. 78-88, Nov.-Dec. 2001, doi: 10.1109/52.965809. keywords: {Computer languages;Lab-on-a-chip;Software tools;Impedance;Investments;Helium;Books;Costs;Assembly;Search engines}, Swimm's architecture enables customization to a specific customer's codebase and frameworks. Unlike training an LLM, this process does not require access to a company's IP. To accomplish this, all access to the company's codebase is done via a local machine on the company's network, and communication is done only with the company's LLM instance. No code or artifacts ever leave the company's network.

Security

Many COBOL applications underpin highly sensitive infrastructure including global finance and government. Swimm is built from a security 1st perspective.

1. **On premise solution** - run Swimm without leaving your network

2. **Use your own LLM** - Swimm can use a customer's internally approved LLM infrastructure.

3. **Certified security** - SOC 2 & ISO 27001 compliant

4. **No infrastructure** - Swimm runs on a developer's laptop requiring no custom hardware (*a docker container with credentials is required for customers using their own LLM)

Conclusion: Closing the knowledge gap

The future of mainframe modernization hinges on solving a fundamental problem: understanding. Without a clear, comprehensive grasp of how COBOL applications work—across screens, jobs, flows, and deeply embedded business logic—modernization efforts stall.

Swimm is closing the knowledge gap with a new approach: by fusing deterministic static analysis with generative AI for a reliable, cost effective path forward. Swimm empowers teams to map, understand, and validate their applications in hours—not months. Modernizing COBOL isn't just about rewriting code. It's about reclaiming the institutional knowledge locked inside it. Swimm is how you do that—accurately, securely, and at scale.

[1] See: Nikiema, S. L., Samhi, J., Kaboré, A. K., Klein, J., & Bissyandé, T. F. (2025). The Code Barrier: What LLMs Actually Understand?. arXiv preprint arXiv:2504.10557.

Haroon, S., Khan, A. F., Humayun, A., Gill, W., Amjad, A. H., Butt, A. R., ... & Gulzar, M. A. (2025). How Accurately Do Large Language Models Understand Code?. arXiv preprint arXiv:2504.04372.

swimm

Swimm is an Application Understanding Platform for legacy and mainframe applications

Close the knowledge gap today