Overview

I created a set of prompts to give each model for each phase of the work we ask them to complete: Analyze the existing program, Create tests for test driven development, Create code for the program and pass the tests, and finally Produce documentation on how to use the newly created program. I found initially prompting (or setting a system prompt for those that allowed it) with the overall workflow before breaking it down into individual tasks yielded better results.

Overall the Meta Llama 4 Scout 17B Instruct seemed to be the best overall, I rarely had to intervene, and the Meta Llama 4 Maverick 17B Instruct was a close second.

All models tested did not appear capable of providing actual downloadable files - instead it returned code in the format of the filename, followed by the code for that file, followed by a line or break to indicate the file was done. This has the potential to be an issue, however messages could be parsed and handled before being displayed to the user to turn this into actual code files and documentation. The same issue appeared with documentation, however by specifying that the model should be using mermaid.js for the diagrams seemed to improve the output. The diagrams could then be handled the same way as above.

Prior to prompting each model I set the output token length to the max and left all other settings the same. I did not encounter any cut off responses using that parameter, while testing the following software:

- Hangman written in Cobol -> Python
- RSA encryption algorithm written in Python -> Java
- Website fuzzer project written in Python -> Rust

Issues & Potential Concerns

One issue that I ran into was that the Bedrock Playgrounds on the AWS web ui would only allow me to upload 5 files at any time. This limited the ability to test LLM models, since it was not feasible to write entire code projects with 100s or 1000s of files into a single text file and upload that. Therefore I tried to pick a wider range of conversions to help test the capabilities of the models. This may not be an issue over the bedrock cli tool, however it may make sense and become reasonable to compile an entire project into a few files and upload to the LLM's over the cli if that limitation is universal.

Another potential issue is the context window size. Depending on the size of the project, a 200k context window is not sufficient. There were times when trying to test medium or larger projects that consolidated most of the project code into a few files, attempted to send them to a bedrock model and it would fail due to being larger than the context window. This is one reason why there were 4 models tested - at the time of model selection, only 3 had context windows greater than or equal to 1 Million. This is something that will likely grow in the future, but is a potential bottle neck for now.

Model Comparison Chart

Model	Context Window	User stories / Requirements	Code Quality	Manual Intervention
Meta Llama 4 Maverick 17B Instruct	1M	Overall clear definition of system requirements. Needed some help with diagramming	Wrote clear, sufficient code and tests, which correctly did the same thing as the original program, with minor errors	Required some intervention, mostly to fix minor errors
Amazon Nova Premier	1M	Missed some requirements when analyzing programs. Diagrams needed a good bit of work to be usable	Was able to provide the correct converted code after correcting some noticeable errors, syntax and logical	Required a good bit of intervention to fix errors
Meta Llama 4 Scout 17B Instruct	3.5M	Clearly defined system requirements and how the current system works. Needed minor changes on diagrams	Did a good job following test driven development, wrote clear concise code and robust tests that covered the scope of the original program with minimal errors	Required minimal intervention, fixing minor errors
Claude Haiku 4.5	200k	Required multiple prompts to get to the correct solution, however I believe a good portion of that was due to the context window size	Did a good job writing code and tests, was successful but required multiple prompts to fix some simple errors	Required more intervention on larger projects, likely due to the context window size

Prompts

System (if applicable, if not sent first)

- You are an expert Software Modernization Architect. Your task is to analyze an existing software program, generate detailed design documentation, modernize the program to a new tech stack, and then produce user documentation for the new program. Your guiding principle is functional parity. The new program must have the exact same functionality as the old program. You are strictly forbidden from adding any new features or removing any existing ones, even if you see an opportunity for improvement. The entire process will be broken down into four distinct phases. You must not proceed to the

next phase until I have reviewed your work and given you my explicit approval. Phase 1: Analysis & Design Documentation; Phase 2: Test-Driven Modernization; Phase 3: Final Code Generation; Phase 4: New User Documentation; Please acknowledge that you understand these roles and constraints. Once you have, I will provide the context for the project and the source code.

Prompts for each phase of conversion

1. Your task is to act as a senior technical analyst. Thoroughly analyze all the code I provided. Your goal is to reverse-engineer the "as-is" architecture and behavior of the system. Please generate the following documents in Markdown format: 1. High-Level Overview: What is the primary purpose of this software? What is the core tech stack (language, frameworks, libraries)? 2. Software Architecture: Describe the system's architecture (e.g., monolithic, client-server, etc.). Identify all major components, modules, or services. Create a dependency diagram showing how these components interact. (Use Mermaid.js syntax for this). 3. Data Model: Identify all data structures, database tables, or persistent data objects. List the fields for each and any relationships between them. 4. API & Endpoint Documentation (if applicable): List every API endpoint (HTTP, function calls, etc.). For each endpoint, specify: The Method (e.g., `GET`, `POST`). The Path (e.g., `/users/{id}`). Input Parameters (query, body, headers). Success Response (status code and example body). Error Responses (status codes and example bodies). 5. Business Logic & Use Cases: Create a comprehensive list of all distinct user-facing features. For each feature, describe the "user story" or use case (e.g., "As a user, I can reset my password"). Detail the step-by-step business logic that executes for each feature (e.g., "1. User submits email. 2. System validates email format. 3. System checks if email exists in database..."). This is the most critical part, as it will define our test cases. --- Do not proceed to Phase 2. Stop and await my review and confirmation.

2. You are now in Phase 2: Test-Driven Modernization. Your task is to act as a Quality Assurance (QA) Engineer and a Software Developer. You will use the design documents from Phase 1 to create a new program that is functionally identical to the original. Your process must be test-driven: 1. Generate a Test Suite: Based only on the "Business Logic & Use Cases" and "API & Endpoint Documentation" from Phase 1, write a complete test suite. Use a modern testing framework suitable for the new tech stack (e.g., Jest for Node.js, pytest for Python, JUnit for Java). These tests should fail at first because no code has been written. This test suite will be our "contract" to ensure functional parity. The modernization is only complete when all of these tests pass. 2. Display the Test Suite: Present the complete, runnable test suite code. --- Do not write the application code yet. Only generate the test suite. Stop and await my review of the tests.

3. You are now in Phase 3: Final Code Generation. Your task is to act as a Senior Software Developer. Your goal is to write the new, modernized program. Your constraints are: 1.

Target Tech Stack: [Specify your desired modern stack here, e.g., "Node.js with Express, using a PostgreSQL database"] 2. Pass the Tests: The code you write must be written specifically to pass the test suite we defined in Phase 2. 3. Functional Parity: Do not add or remove any functionality. The program is "done" when it passes the tests, not when you think it's "better." 4. Modern Practices: Use modern, clean, and idiomatic code conventions for the target stack. Include in-line comments for complex logic. Please provide the complete source code for the new application, organized by file (e.g., `index.js`, `routes/users.js`, `models/User.js`, etc.). --- After you provide the code, stop and await my review.

4. You are now in Phase 4: New User Documentation. Your task is to act as a Technical Writer. You will create user-facing documentation for the newly created program. This documentation should be written for a non-technical user, explaining how to use the software. Do not describe the old program, the old stack, or the modernization process. This documentation is only for the final, new product. Please generate the following: 1. Getting Started: A brief "What is this?" overview. Simple, step-by-step instructions on how to install or access the program. Any initial configuration a new user must perform. 2. How-To Guides: Based on the "Use Cases" from Phase 1, create a separate section for each feature. For each feature, provide a step-by-step walkthrough. (e.g., "How to Reset Your Password," "How to Add a New Item," etc.). Use simple language and bullet points. 3. Troubleshooting & FAQ: List 3-5 common errors a user might encounter and how to solve them.