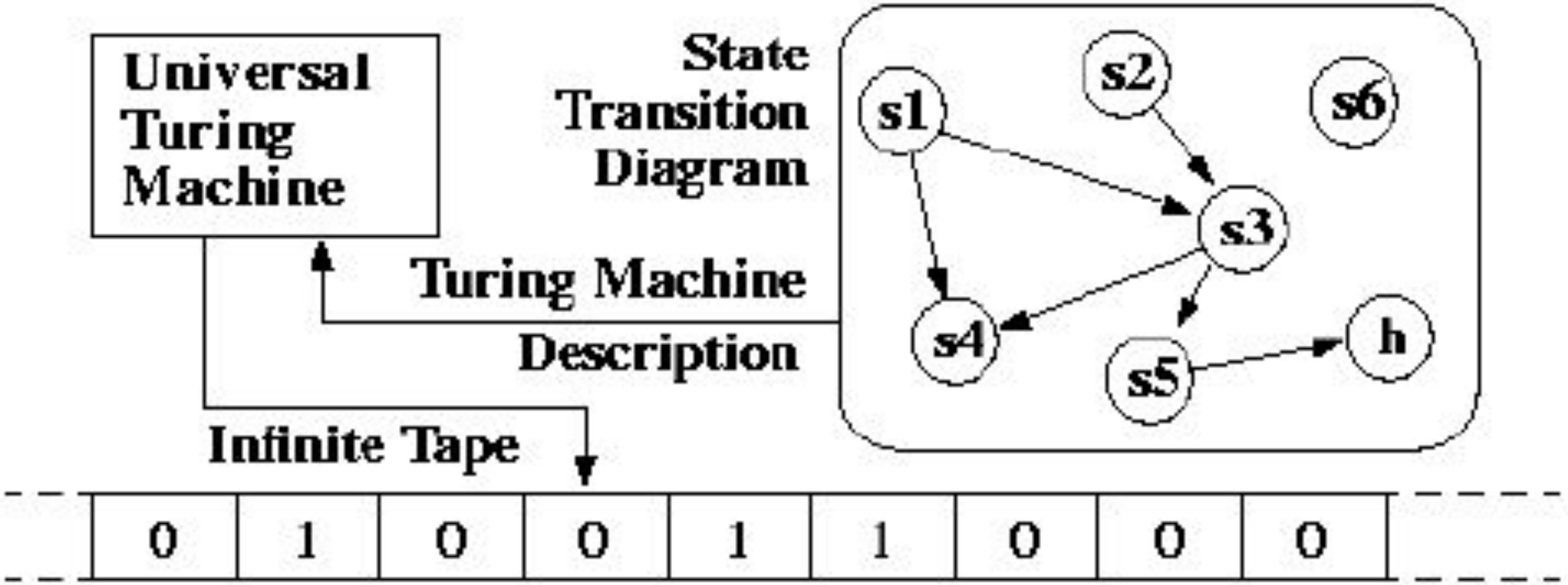
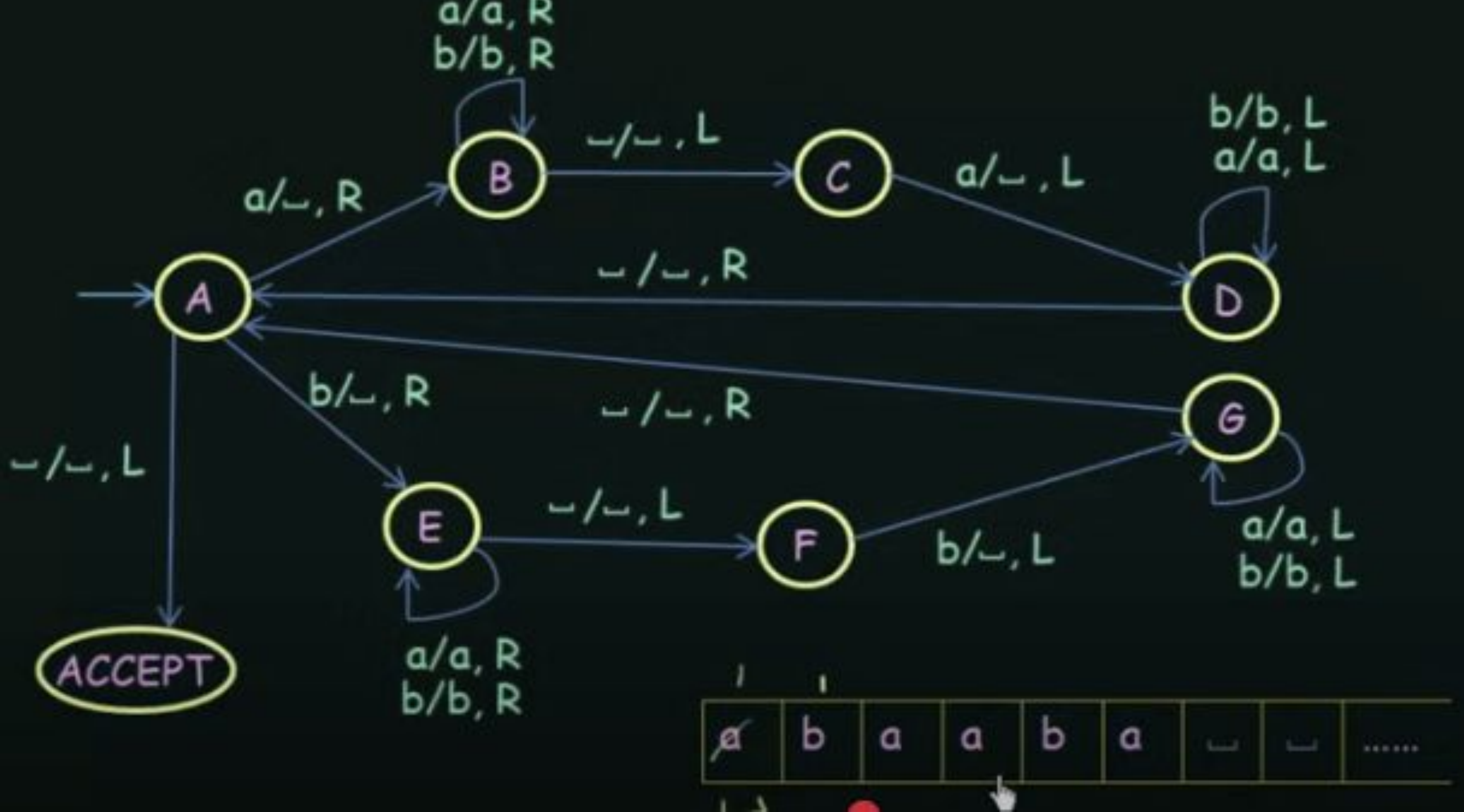
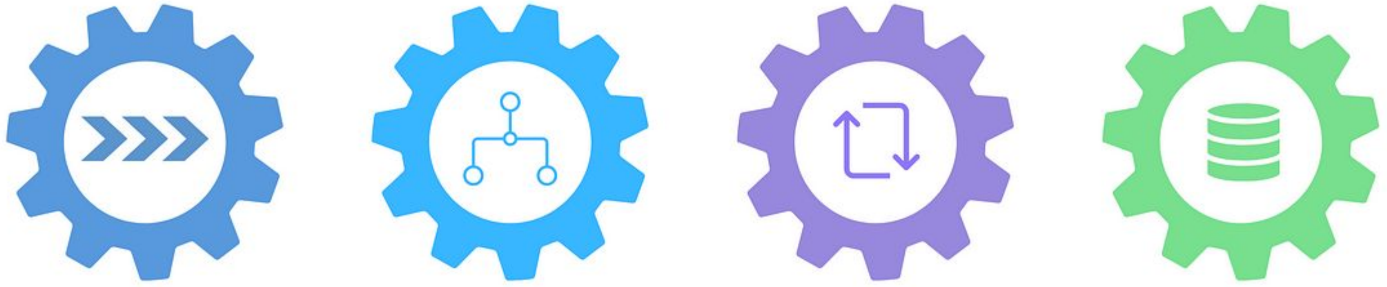

Universal Turing Machine
State Transition Diagram
s1
s2
s3
s4
s5
s6
h
Turing Machine Description
Infinite Tape
0
1
0
0
1
1
0
0
0


a/a, R
b/b, R
⎵/⎵ , L
b/b, L
a/a, L
a/⎵, R
a/⎵ , L
⎵ /⎵ , R
A
B
C
D
b/⎵, R
⎵ /⎵ , R
G
⎵/⎵, L
E
⎵/⎵, L
F
b/⎵, L
a/a, L
b/b, L
a/a, R
b/b, R
ACCEPT
a
b
a
a
b
a
⎵
⎵
......

What Does Turing Complete Mean?

Key characteristics that define a Turing complete system



Sequence

Execute a series of computational steps in the order they are provided.

Conditionals

Execute different computational paths conditionally based on certain criteria.

Iteration

Repeat computational sub-processes over and over.

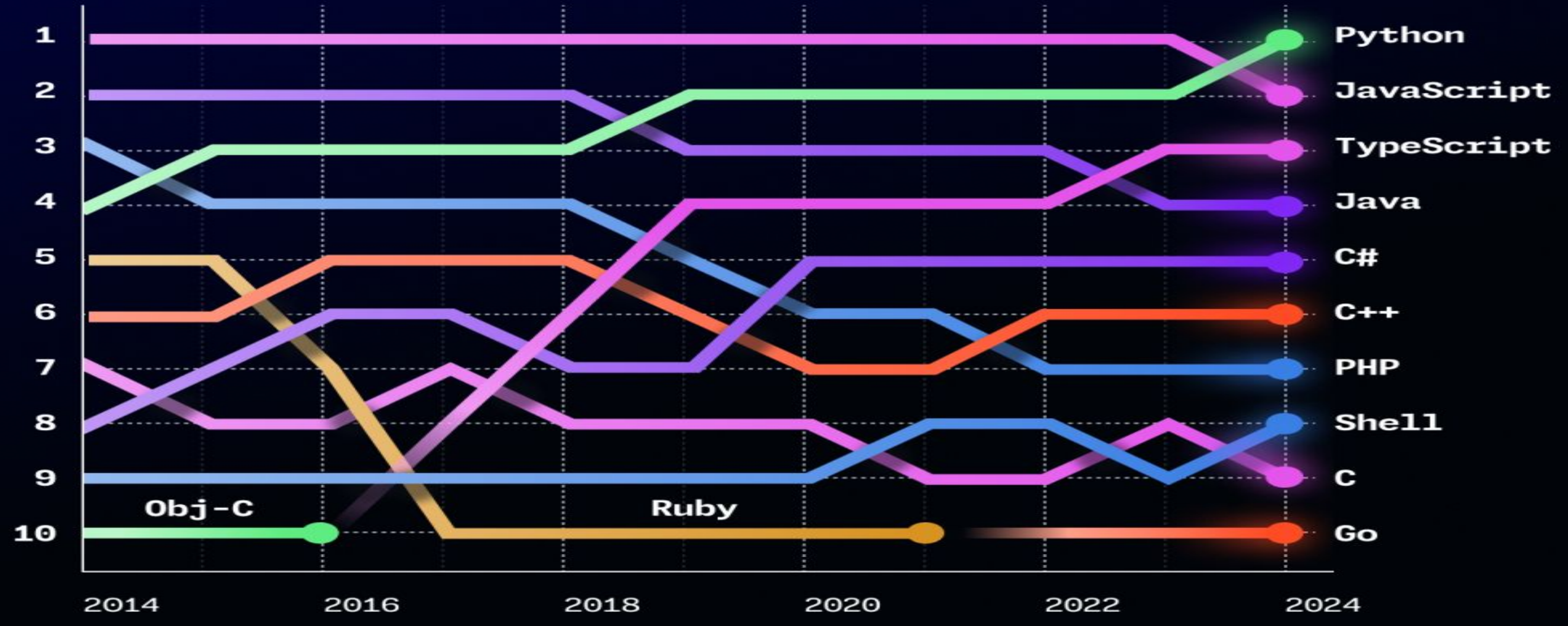
Store Data

Store intermediate results for later use in memory.

@miyokoshimura

Top programming languages on GitHub

RANKED BY COUNT OF DISTINCT USERS CONTRIBUTING TO PROJECTS OF EACH LANGUAGE.



What do all these languages have in common?

Non-turing complete examples?

What is the difference between a language, framework and module?

Compiled vs Interpreted

Compiled

-Code is translated from Source Code to Machine Code.

-Compiler .

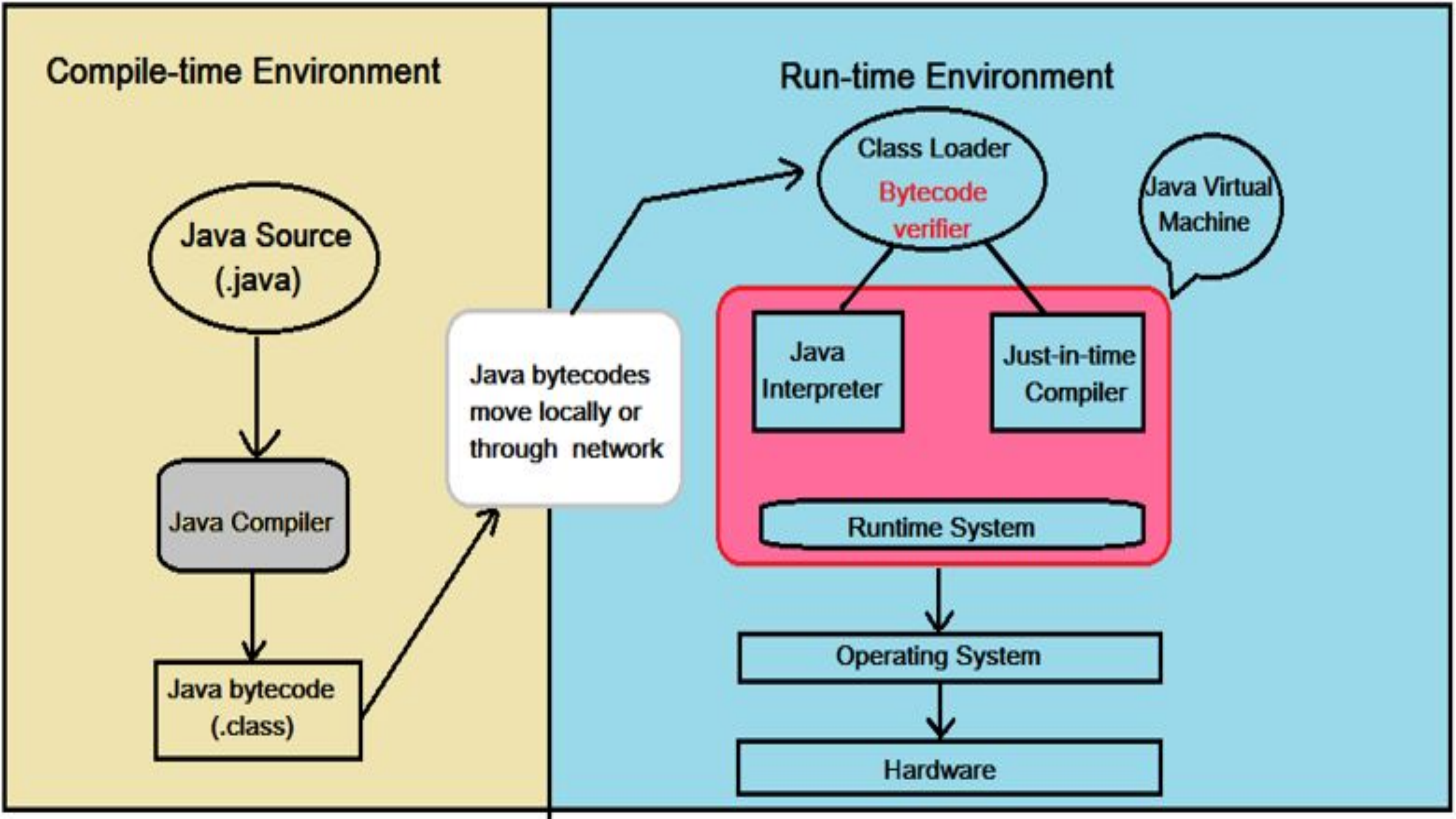
https://en.wikipedia.org/wiki/Compiled_language

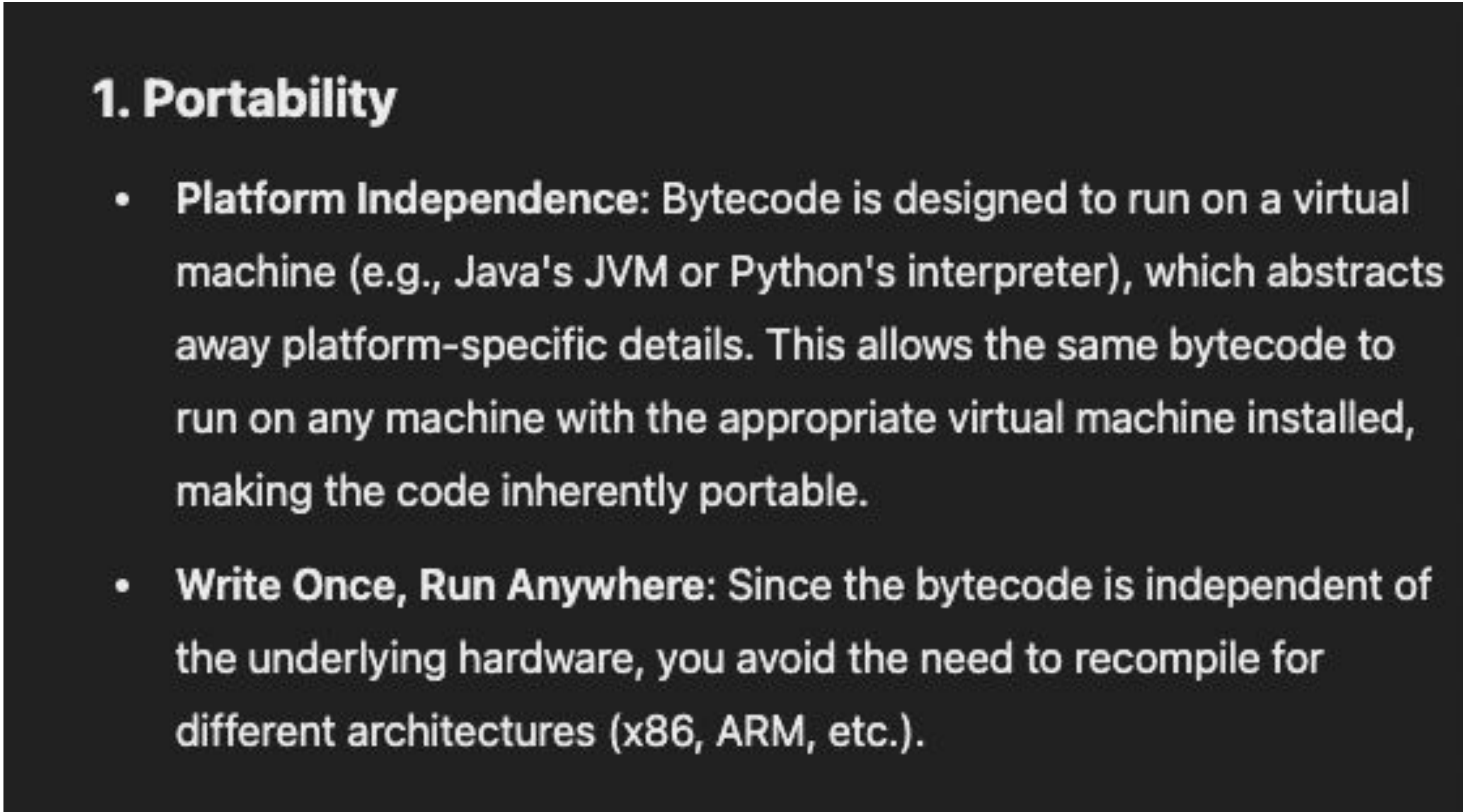
Interpreted

-Source code executed directly without prior compilation.

-Interpreter.

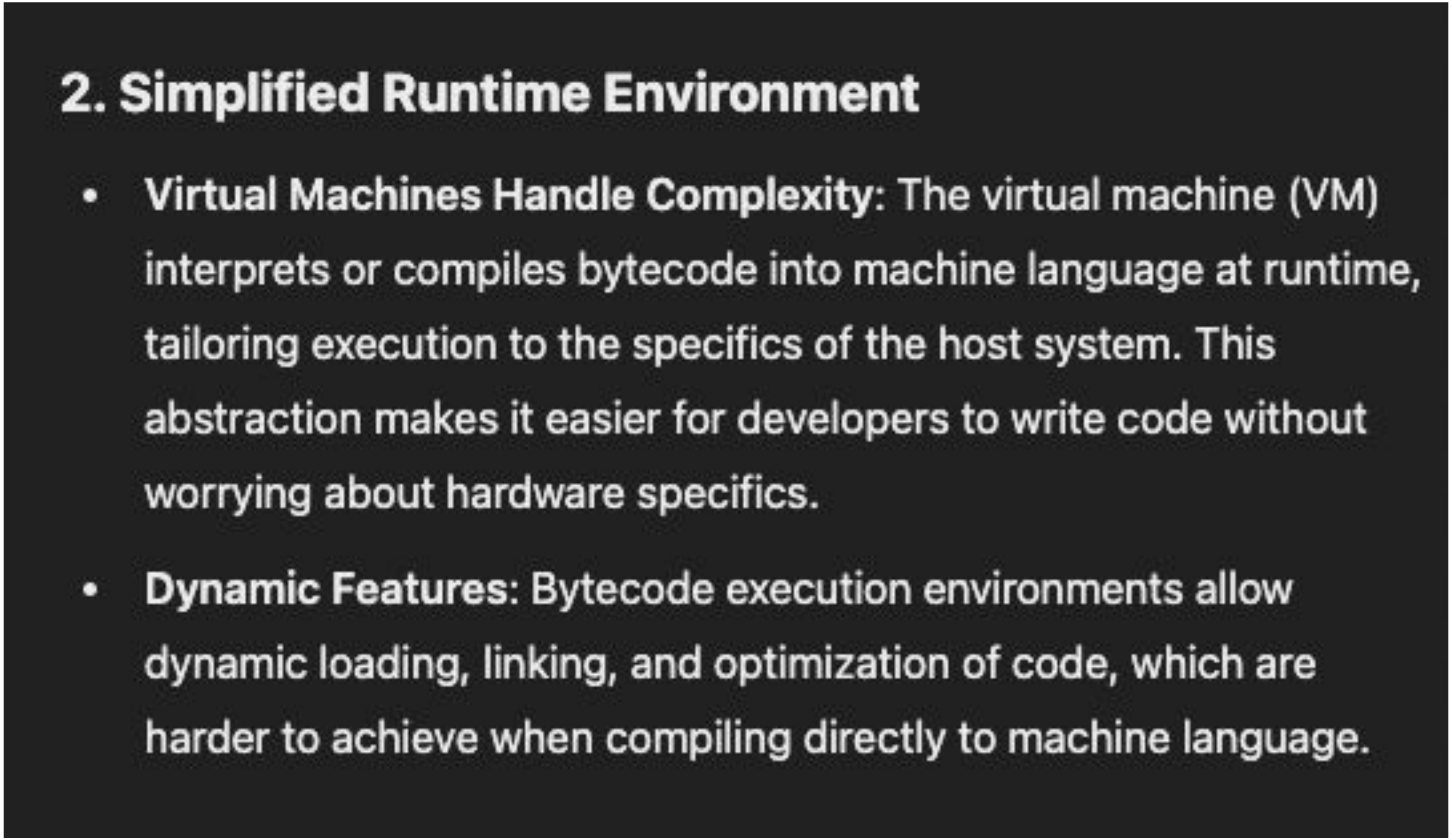
https://en.wikipedia.org/wiki/Interpreted_language


Compile-time Environment
Java Source (.java)
Java Compiler
Java bytecode (.class)
Java bytecodes move locally or through network
Run-time Environment
Class Loader
Bytecode verifier
Java Virtual Machine
Java Interpreter
Just-in-time Compiler
Runtime System
Operating System
Hardware



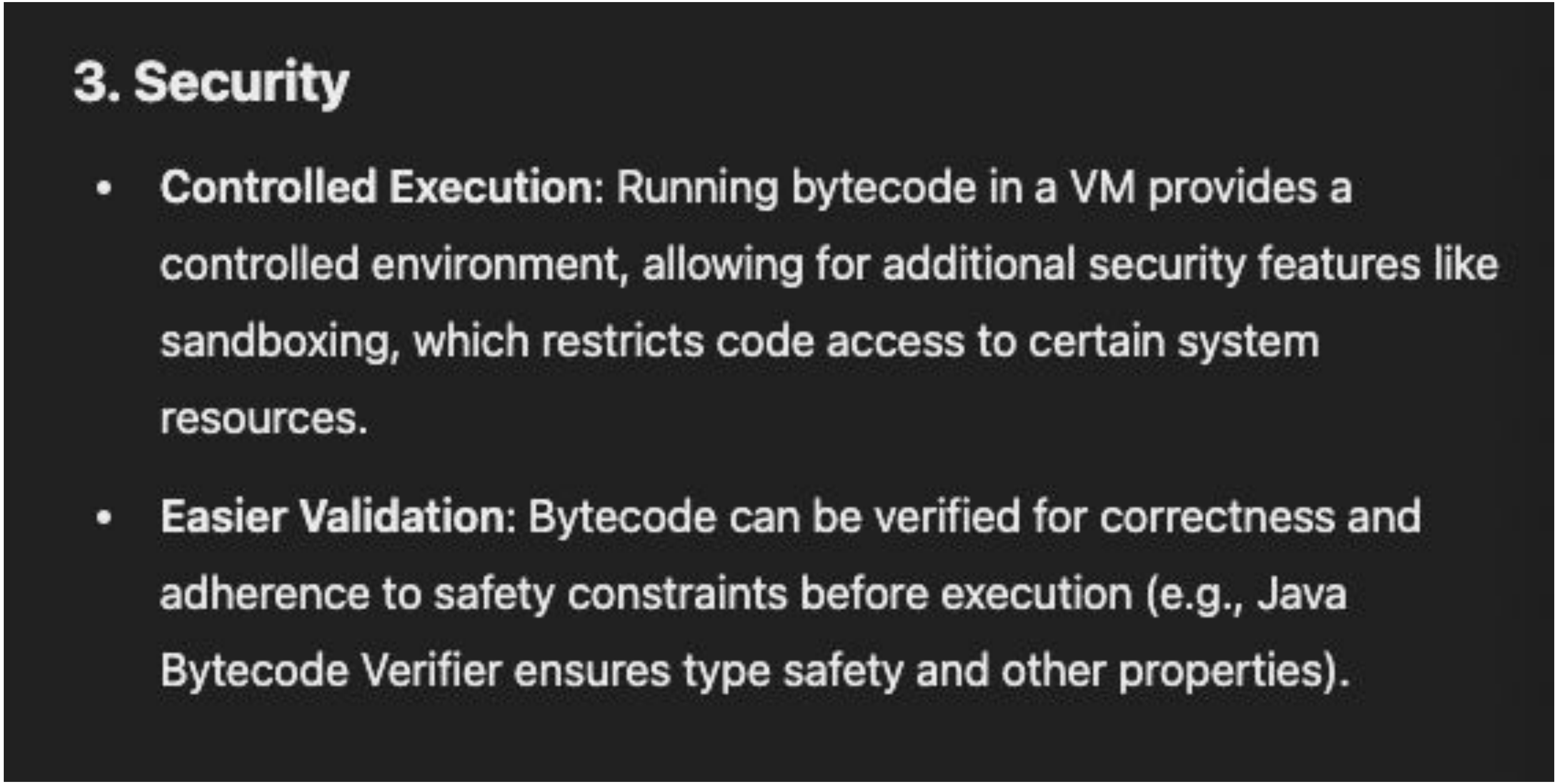
1. Portability

- **Platform Independence**: Bytecode is designed to run on a virtual machine (e.g., Java's JVM or Python's interpreter), which abstracts away platform-specific details. This allows the same bytecode to run on any machine with the appropriate virtual machine installed, making the code inherently portable.
- **Write Once, Run Anywhere**: Since the bytecode is independent of the underlying hardware, you avoid the need to recompile for different architectures (x86, ARM, etc.).



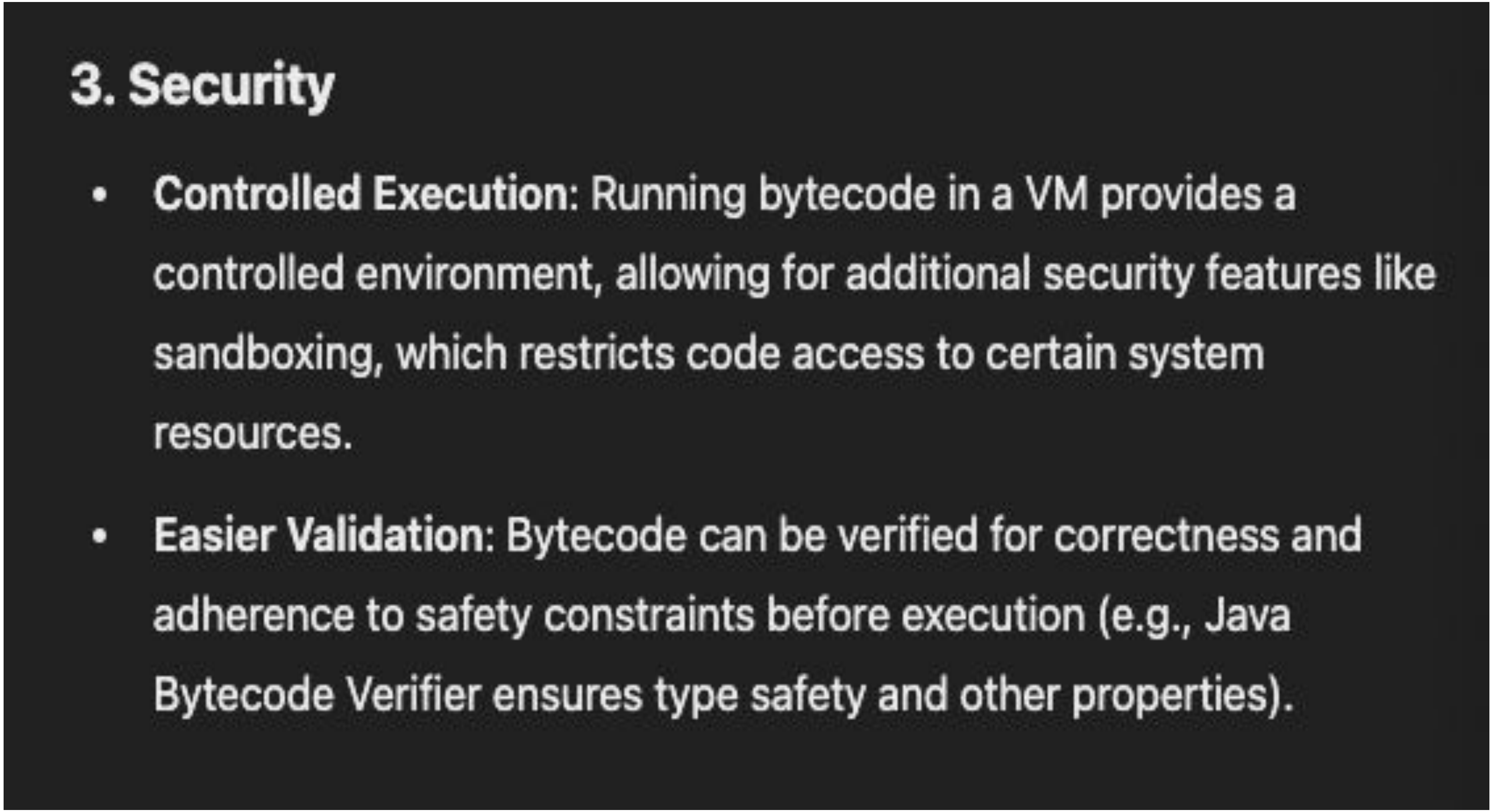
2. Simplified Runtime Environment

- **Virtual Machines Handle Complexity:** The virtual machine (VM) interprets or compiles bytecode into machine language at runtime, tailoring execution to the specifics of the host system. This abstraction makes it easier for developers to write code without worrying about hardware specifics.
- **Dynamic Features:** Bytecode execution environments allow dynamic loading, linking, and optimization of code, which are harder to achieve when compiling directly to machine language.



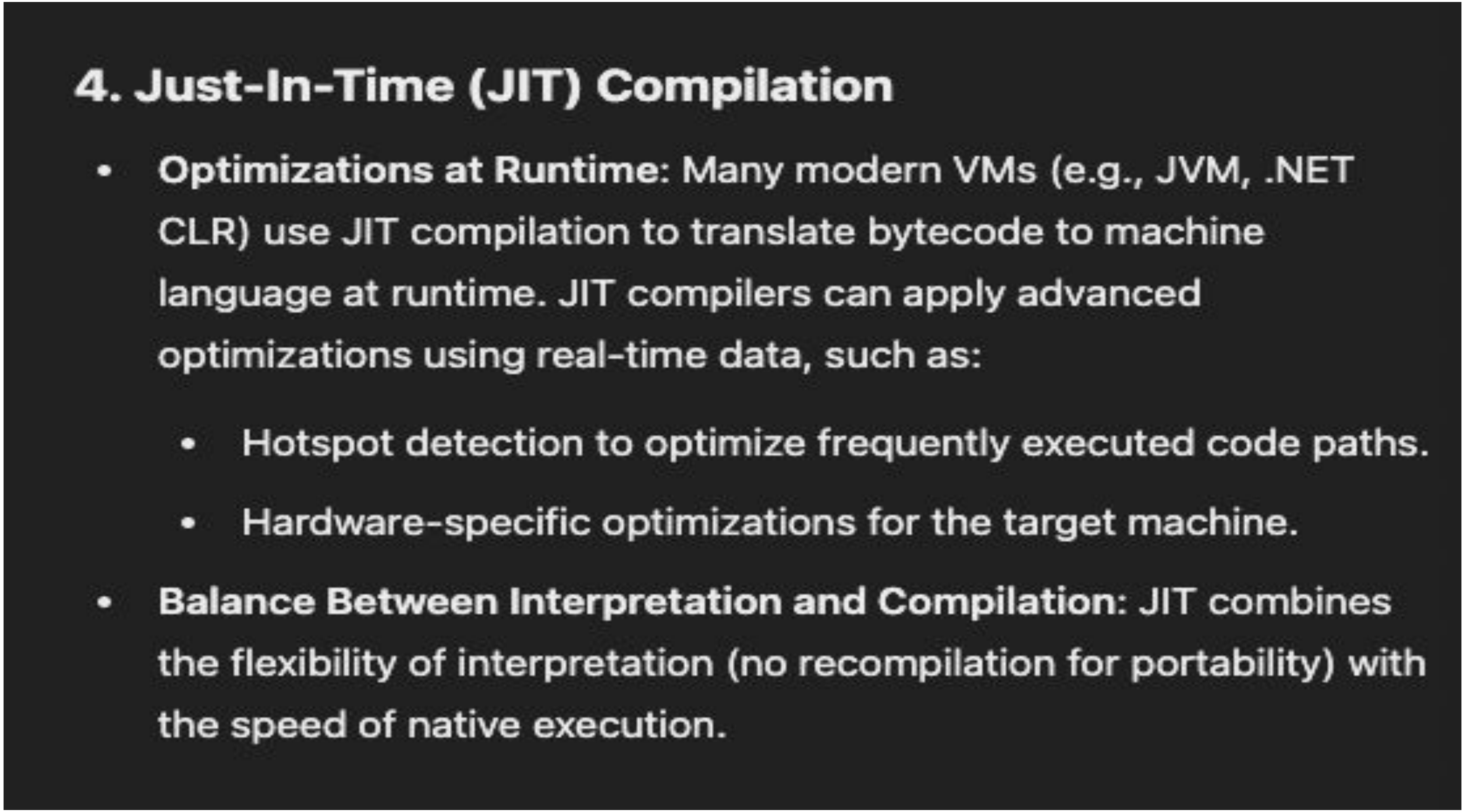
3. Security

- **Controlled Execution**: Running bytecode in a VM provides a controlled environment, allowing for additional security features like sandboxing, which restricts code access to certain system resources.
- **Easier Validation**: Bytecode can be verified for correctness and adherence to safety constraints before execution (e.g., Java Bytecode Verifier ensures type safety and other properties).



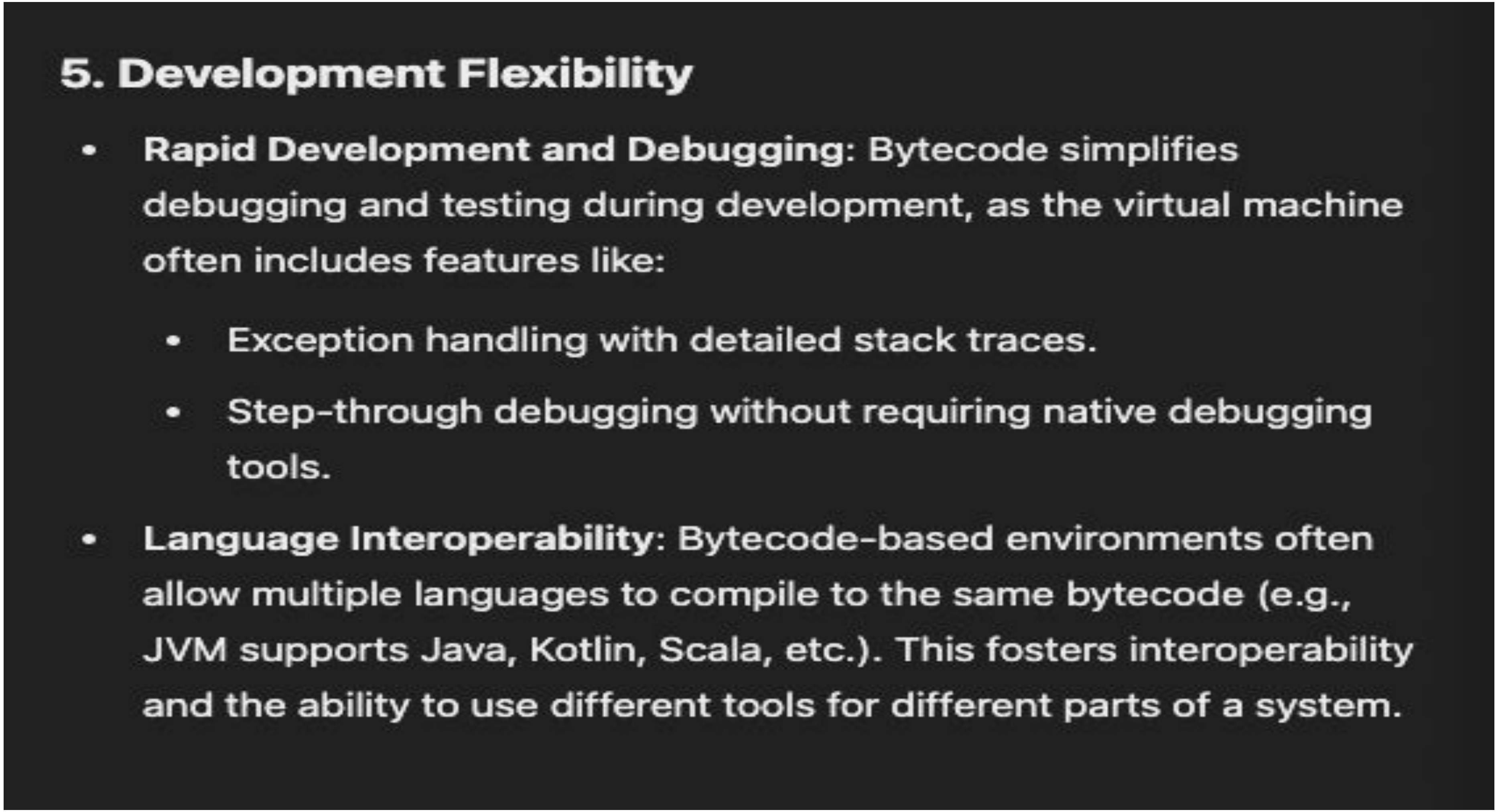
3. Security

- **Controlled Execution:** Running bytecode in a VM provides a controlled environment, allowing for additional security features like sandboxing, which restricts code access to certain system resources.
- **Easier Validation:** Bytecode can be verified for correctness and adherence to safety constraints before execution (e.g., Java Bytecode Verifier ensures type safety and other properties).



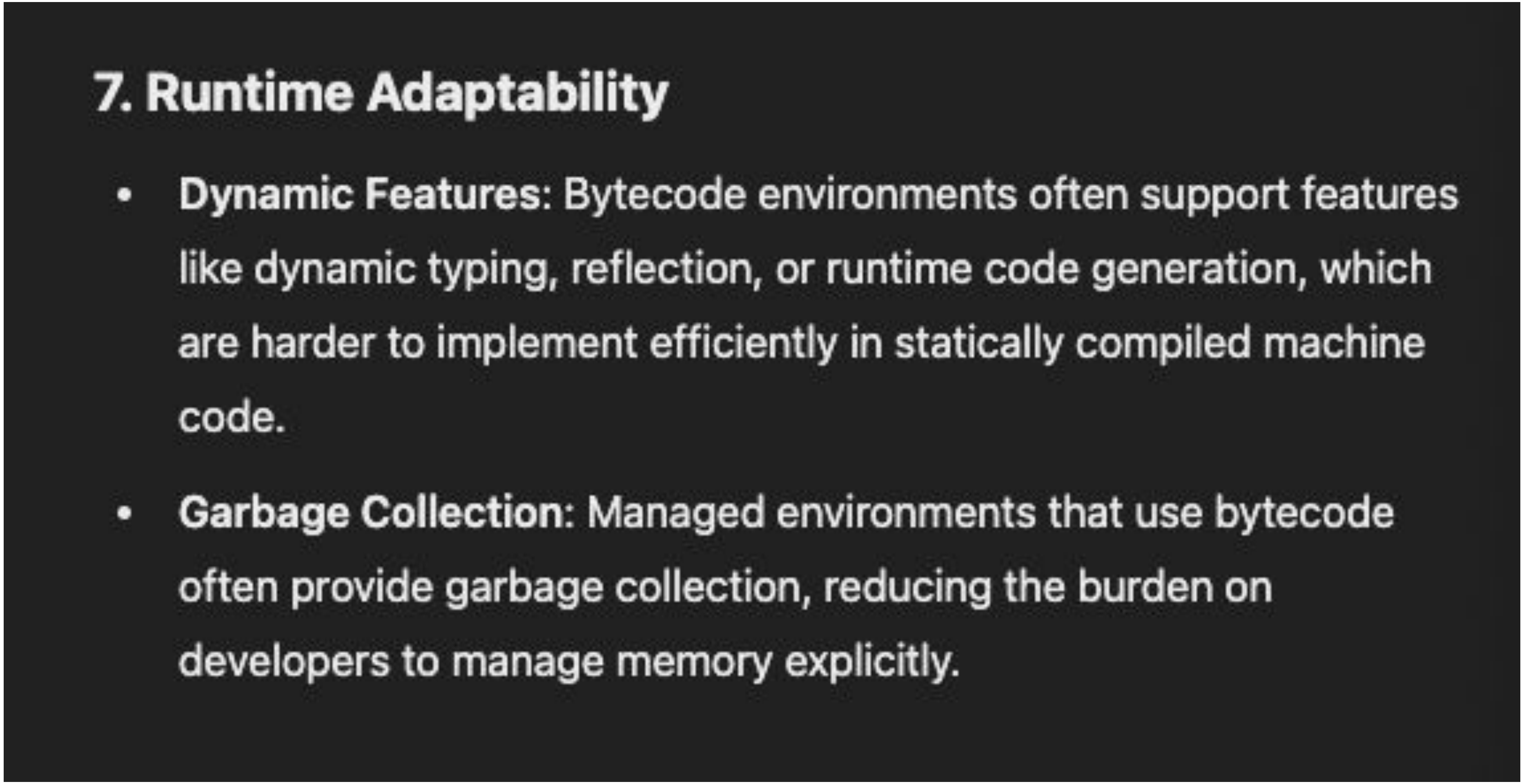
4. Just-In-Time (JIT) Compilation

- **Optimizations at Runtime:** Many modern VMs (e.g., JVM, .NET CLR) use JIT compilation to translate bytecode to machine language at runtime. JIT compilers can apply advanced optimizations using real-time data, such as:
    - Hotspot detection to optimize frequently executed code paths.
    - Hardware-specific optimizations for the target machine.
- **Balance Between Interpretation and Compilation:** JIT combines the flexibility of interpretation (no recompilation for portability) with the speed of native execution.



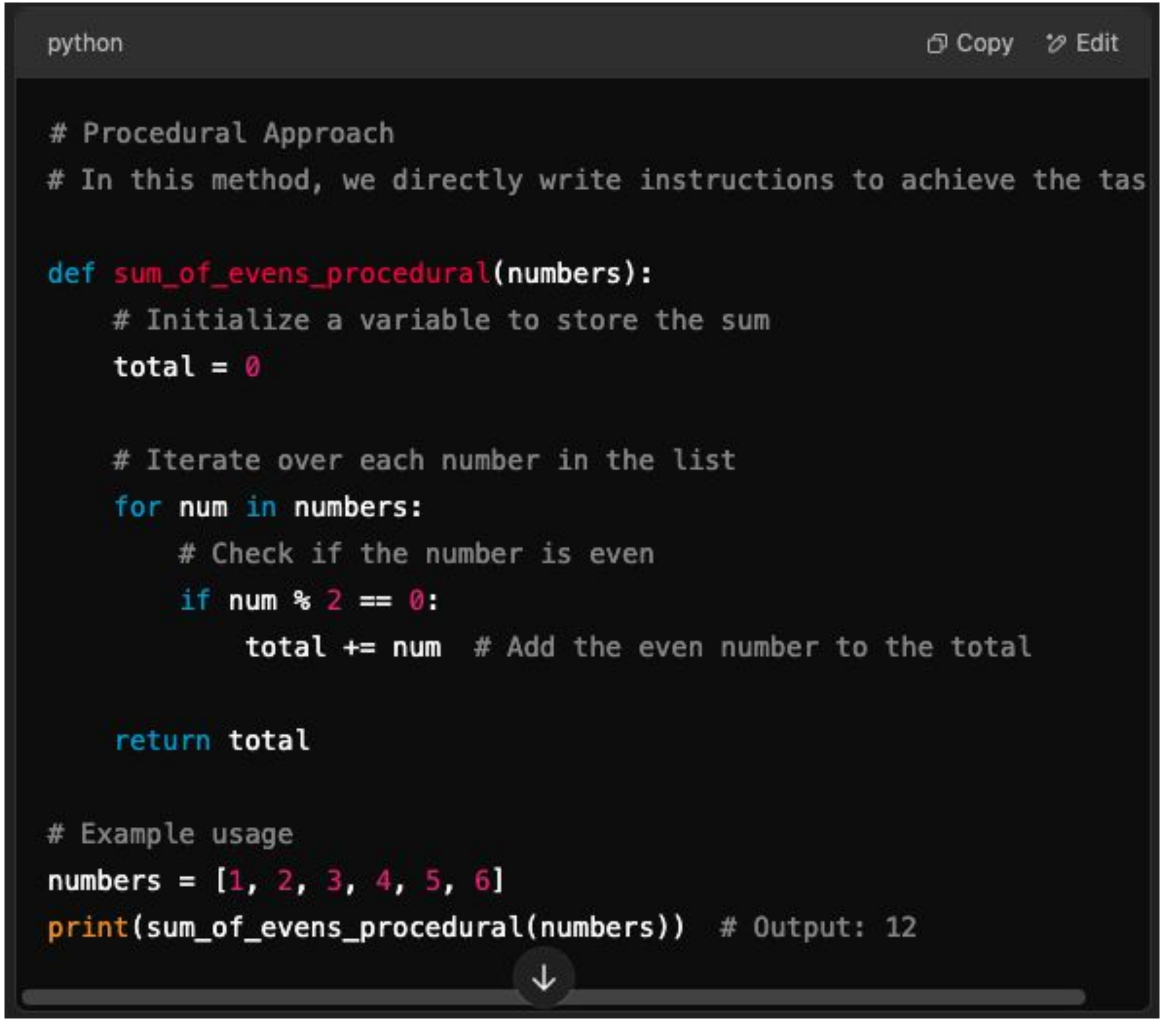
5. Development Flexibility

- **Rapid Development and Debugging**: Bytecode simplifies debugging and testing during development, as the virtual machine often includes features like:
  - Exception handling with detailed stack traces.
  - Step-through debugging without requiring native debugging tools.
- **Language Interoperability**: Bytecode-based environments often allow multiple languages to compile to the same bytecode (e.g., JVM supports Java, Kotlin, Scala, etc.). This fosters interoperability and the ability to use different tools for different parts of a system.

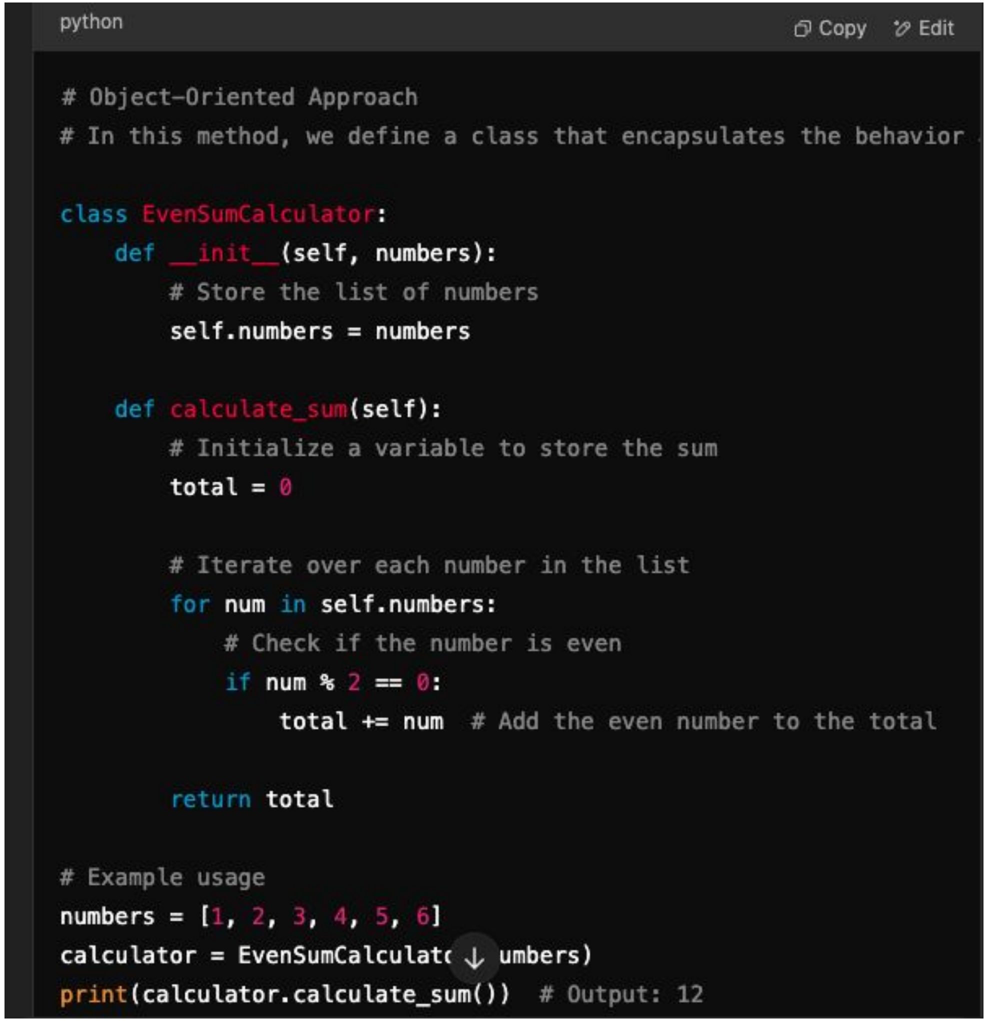


7. Runtime Adaptability

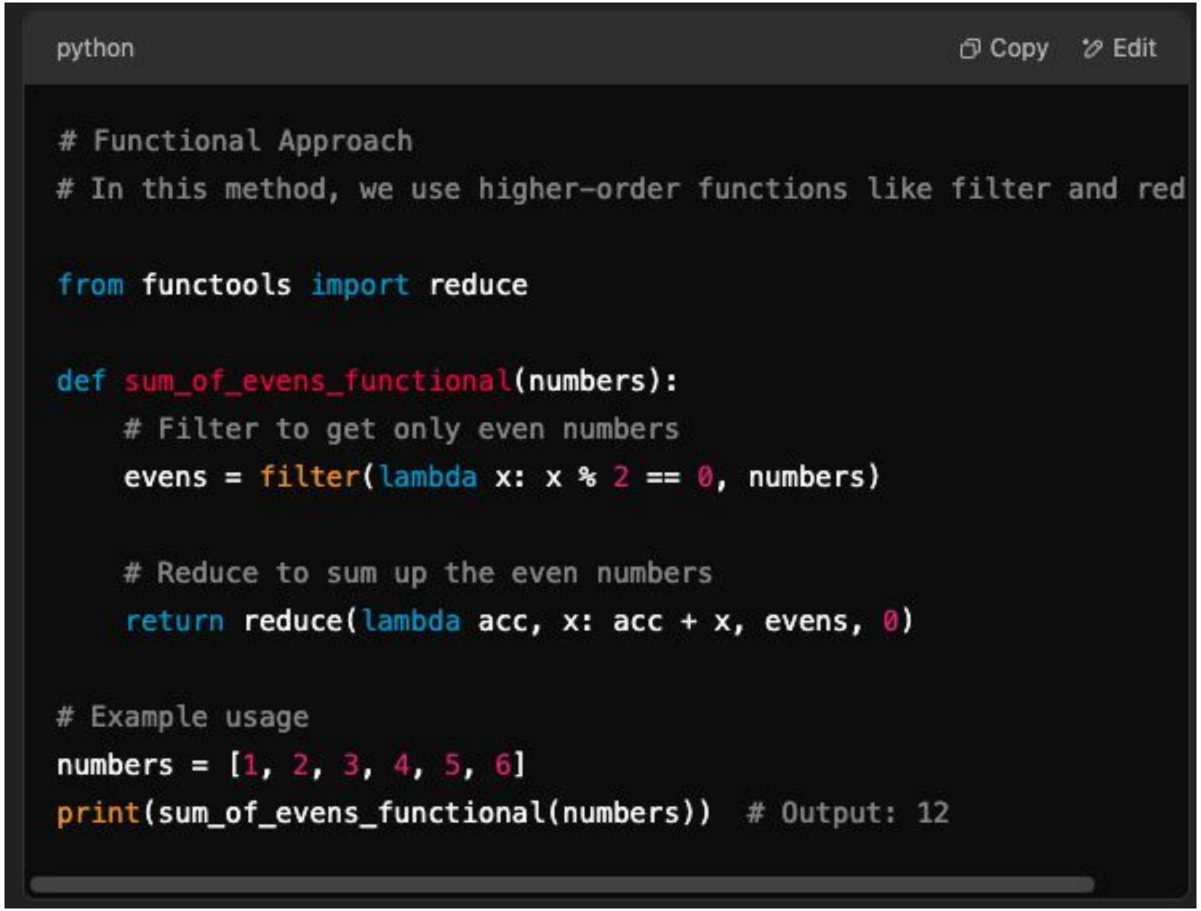
- **Dynamic Features:** Bytecode environments often support features like dynamic typing, reflection, or runtime code generation, which are harder to implement efficiently in statically compiled machine code.
- **Garbage Collection:** Managed environments that use bytecode often provide garbage collection, reducing the burden on developers to manage memory explicitly.



```python
# Procedural Approach
# In this method, we directly write instructions to achieve the tas

def sum_of_evens_procedural(numbers):
    # Initialize a variable to store the sum
    total = 0

    # Iterate over each number in the list
    for num in numbers:
        # Check if the number is even
        if num % 2 == 0:
            total += num  # Add the even number to the total

    return total

# Example usage
numbers = [1, 2, 3, 4, 5, 6]
print(sum_of_evens_procedural(numbers))  # Output: 12
```



```python
# Object-Oriented Approach
# In this method, we define a class that encapsulates the behavior

class EvenSumCalculator:
    def __init__(self, numbers):
        # Store the list of numbers
        self.numbers = numbers

    def calculate_sum(self):
        # Initialize a variable to store the sum
        total = 0

        # Iterate over each number in the list
        for num in self.numbers:
            # Check if the number is even
            if num % 2 == 0:
                total += num  # Add the even number to the total

        return total

# Example usage
numbers = [1, 2, 3, 4, 5, 6]
calculator = EvenSumCalculatc umbers)
print(calculator.calculate_sum())  # Output: 12
```



```python
# Functional Approach
# In this method, we use higher-order functions like filter and red

from functools import reduce

def sum_of_evens_functional(numbers):
    # Filter to get only even numbers
    evens = filter(lambda x: x % 2 == 0, numbers)

    # Reduce to sum up the even numbers
    return reduce(lambda acc, x: acc + x, evens, 0)

# Example usage
numbers = [1, 2, 3, 4, 5, 6]
print(sum_of_evens_functional(numbers))  # Output: 12
```

Explanation of Differences

1. **Procedural Approach:**
   - Focuses on the direct flow of control.
   - The code is simple and procedural but does not emphasize modularity or reusability.
2. **Object-Oriented Approach:**
   - Encapsulates data and methods in a class.
   - Promotes reusability, readability, and maintenance, especially for more complex problems.
3. **Functional Approach:**
   - Emphasizes immutability and the use of higher-order functions.
   - Code is concise and declarative, focusing on *what* to do rather than *how*.

1. **Function**: A reusable block of code that performs a specific task and typically returns a value to the caller.
2. **Method**: A function that is associated with an object or class and operates on its data (e.g., in object-oriented programming).
3. **Procedure**: A block of code similar to a function but does not return a value, focusing on executing actions or processes.
4. **Macro**: A preprocessor directive or set of instructions that gets expanded at compile-time or runtime, often used to replace repetitive code with a template.

Project #1 Description
Dockerized Python Hello World