**SundarBan Payment Gateway**

**Program Execution**

Executing a program involves several stages, from writing the code to running the final executable. The key steps in this process are:

1. **Writing Code**: A programmer writes source code in a high-level language like C, C++, or Python and saves it with an appropriate file extension (e.g., .c for C programs).
2. **Preprocessing**: The preprocessor processes the source code by handling macros, including necessary header files, and applying directives such as #include and #define.
3. **Compilation**: The compiler translates the preprocessed code into assembly language, providing a low-level representation of the program.
4. **Assembly**: The assembler converts the assembly code into object code, containing machine-readable binary instructions. Many modern compilers integrate this step with compilation.
5. **Linking**: The object code is linked with external libraries to generate an executable file, resolving dependencies and function references.
6. **Loading**: The operating system loads the executable file into memory, sets up the runtime environment, and initializes the program counter before execution begins.

An illustrative diagram (Figure 3.1) further clarifies this process.

**Executable Files**

An executable file contains compiled code that can be run directly by an operating system. When launched, the OS loads it into memory and executes it. Common formats include .exe for Windows, ELF (Executable and Linkable Format) for Linux, and Mach-O for macOS.

**Key Components of an Executable File**

1. **Header (Metadata)**: Stores essential information about the file, such as its format, entry point, memory layout, and dependencies, guiding the OS during execution.
2. **Program Counter (PC)**: A CPU register that tracks the address of the next instruction, ensuring proper execution flow.

**Load Testing**

Load testing evaluates a system's performance under varying levels of demand, ensuring stability, responsiveness, and efficiency. It helps prevent system failures due to high traffic and enhances user experience.

**Types of Load Testing**

1. **Smoke Testing**: A preliminary check to confirm basic system stability.
2. **Average-Load Testing**: Measures performance under normal operating conditions.
3. **Stress Testing**: Pushes the system beyond its limits to identify breaking points.
4. **Soak Testing**: Assesses long-term system stability under sustained load.
5. **Spike Testing**: Examines performance under sudden traffic surges.
6. **Breakout Point Testing**: Determines when performance starts degrading significantly.

**Load Testing Tools**

Various tools, such as LoadView, help conduct load tests by offering test recording, execution, distributed testing, and performance analysis.

**Load Testing vs. Stress Testing**

Load testing ensures stable performance under expected traffic, while stress testing evaluates system resilience under extreme conditions and measures recovery ability.

**Benefits of Load Testing**

* Prevents system crashes and unexpected failures.
* Enhances user experience by ensuring smooth performance.
* Identifies performance bottlenecks early, reducing future costs.
* Supports business scalability by ensuring applications handle growing user demands.

**Scaling in Software Architecture**

Scaling refers to a system’s ability to handle increased workloads while maintaining efficiency. It is crucial for ensuring reliability and optimal resource utilization.

**Types of Scaling**

1. **Vertical Scaling (Scaling Up)**: Enhances a single machine's performance by upgrading its CPU, RAM, or storage. Suitable for monolithic architectures and smaller applications.
2. **Horizontal Scaling (Scaling Out)**: Expands capacity by adding more machines or servers, distributing workloads effectively. Common in large-scale web applications and cloud-native architectures.
3. **Diagonal Scaling**: Combines vertical and horizontal scaling, dynamically adjusting resources based on demand, often seen in cloud environments.

**Function Calls**

Function calls dictate how programs handle tasks, affecting performance and concurrency.

**Synchronous Function Calls**

A synchronous function call requires completing one task before moving to the next. The calling process remains blocked until execution is finished, ensuring sequential task execution.

**Asynchronous Function Calls**

An asynchronous function call enables tasks to run concurrently, allowing the program to proceed without waiting. This is essential for handling time-consuming operations such as network requests or large-scale data processing.