# Component Modules and Decentralized Method Execution - A Secure Architecture for Interoperable Toolchains

## 1. Introduction

Modern software environments often require extensibility, interoperability, and toolchain integration. However, most current approaches rely on loosely coupled CLI tools or runtime-specific plugin architectures. These patterns pose challenges in terms of robustness, isolation, security, and reusability.

This research introduces a cross-platform, modular execution architecture that enables tools to be securely embedded, independently executed, or distributed across cooperating systems. It introduces two complementary systems:

- **Component Modules** (CM/XCM): Loadable modules akin to Linux kernel modules, featuring exportable APIs, reactive data stream interaction, and standalone execution capabilities.

- **Decentralized Method Execution** (DME): A cross-language, memory-mapped file-based execution layer for interpreted instruction sets, enabling structured and reversible data exchange between isolated tools.

Both systems may additionally operate under a sandboxed environment, providing strict control over file system access, network usage, and intermodule interaction. This architecture targets developer tools, game engines, simulation environments, and CI/CD pipelines requiring fine-grained extensibility and control

## 2. Component Module System

Component Modules (files with .cm extension) and Executable Component Modules (.xcm) represent binary artifacts loaded by a host application or launched as standalone entities via a dedicated execution shell. Inspired by Linux kernel modules, these components offer structured extension points and system-wide function discovery

### 2.1 Features

Modules are capable of exporting their own functionality to other parts of the system. These exports are made discoverable through a centralized registry, enabling composition across modules and allowing them to build upon one another. For example, a build module may export transformation functions which a packaging module can invoke directly.

Additionally, modules can observe predefined data streams that exist within the host application. These streams may include engine loops, asset compilation events, or user input events. A reactive, publish-subscribe mechanism enables modules to respond to or filter data in real time.

Modules may also create their own data streams and publish data into the system.

These streams are announced through a global stream registry, which allows other modules to discover, connect, and process data as needed. This decentralized approach empowers modules to interact without tight coupling

## 2.2 Standalone Execution Layer

Modules with the .xcm extension are executable independently of a host application. A lightweight execution shell, similar in concept to dotnet run, is responsible for parsing CLI arguments, instantiating input/output streams, and initializing the module. In standalone mode—the module receives its data and event triggers through this embedded shell—allowing it to behave identically as if integrated into a host environment.

This execution layer makes it possible to reuse modules across vastly different environments—such as running a build tool in a CI pipeline without relying on an IDE or engine host

## 2.3 Host APIs and Dependency Modeling

To support structured extension of a host application, modules may be built against a dedicated Host API library. This is typically provided as a dynamic library (.dll on Windows, .so on Linux) that includes placeholder functions or interfaces representing the host's exposed capabilities. These dependencies are explicitly flagged during the post-compilation process, ensuring they are not subjected to the same security constraints as third-party libraries.

The system distinguishes between dependencies meant for sandbox operation and those tied to a specific host. Modules can express conditional logic depending on their execution context. For instance, file I/O might be routed through a restricted sandbox API in standalone mode, while a richer file system interface may be available in a trusted host.

The Host API may also be provided directly by the sandbox environment. In this case, foundational operations such as internet access, file creation, or querying user-specific information are made available through controlled, permission-gated calls. These APIs define the minimal and auditable interface between sandboxed modules and the external system

# 3. PostCompiler and Module Loader

The PostCompiler is responsible for transforming compiled binary artifacts (such as .dll, .so, or .exe files) into sandbox-aware, validated module containers with the .cm or .xcm extension. These files preserve all code and metadata necessary for secure execution, while adding integrity and compatibility guarantees

## 3.1 PostCompilation Process

1. **Binary Parsing**
   The PostCompiler reads input binaries in platform-native formats such as PE/COFF (Windows) or ELF (Linux). It parses the section headers, extracts function definitions, entry points, and metadata tables.

2. **Symbol Analysis & Method Extraction**
   Exported methods are enumerated

and registered. Functions that conform to specific module interfaces—such as module_load, module_execute, or module_unload—are marked as lifecycle hooks. Additional metadata (e.g., parameter signatures, stream bindings) can be embedded via attributes or custom sections.

3. **Dependency Inspection**
   The binary's import table is scanned for external dependencies. Libraries declared as Host APIs are whitelisted based on developer intent and excluded from sandbox enforcement. All other dependencies are flagged and must be resolvable within the sandbox or explicitly allowed.

4. **Format Encoding**
   The binary and all associated metadata are packaged into a .cm or .xcm file. This includes:
   - Executable sections
   - Metadata for exports and streams
   - Execution context (host-bound or standalone)
   - Optional cryptographic signature or manifest block

## 3.2 Module Loader Functionality

At runtime, a Module Loader is responsible for instantiating and executing .cm or .xcm modules within a controlled environment. Depending on the context (host application or standalone shell), the loader adapts its behavior accordingly.

1. **Memory Allocation & Loading**
   Executable memory is allocated using platform-specific calls:
   - On Windows: VirtualAlloc + VirtualProtect
   - On Linux: mmap with PROT_EXEC | PROT_READ | PROT_WRITE

2. The module's code is copied into the allocated space, and relocation is applied if necessary. Entry points are resolved, and the virtual address space is adjusted to allow controlled execution.

3. **Lifecycle Invocation**
   The loader calls the appropriate module lifecycle functions:
   - module_load() when the module is first loaded
   - module_execute() when triggered by host or shell
   - module_unload() during shutdown or hot-reload

4. These functions provide hooks for registering exports, binding to data streams, or initializing internal state.

5. **Contextual Execution**
   In host mode, the loader links the module against in-memory host APIs and registers it in the module registry. In standalone mode, the loader parses CLI arguments, binds them to synthetic streams, and simulates a host-like runtime.

## 3.3 Dependency and Method Mapping

After the module's memory has been initialized and lifecycle methods prepared, the loader proceeds to resolve its declared external dependencies. These may include:

- Other component modules (.cm) providing reusable functions or data streams
- Host-provided APIs for system-level access (I/O, networking, user interaction)
- Sandbox-exposed services, which may be conditionally available based on the runtime environment

Each dependency is resolved by querying the module registry and the symbol table, both of which are maintained by the host or the execution shell. When a module calls an external function or requests a stream, the loader performs the following steps:

1. **Symbol Linking**
   The module's internal references to external symbols (function names, stream identifiers) are matched against exported symbols from other modules. Function pointers are patched to point to the correct memory addresses or proxy stubs.

2. **Host API Binding**
   If the module depends on a host-defined API (as declared in the Host API placeholder DLL or stub), the loader maps the relevant functions to their actual implementations within the host or sandbox runtime. These functions are subject to permission checks and may be replaced with no-ops or restricted shims in sandboxed contexts.

3. **Permission Validation**
   Each linked dependency is validated against the module's declared permissions. If a module attempts to access unauthorized symbols or streams, the load process is aborted, or a degraded capability mode is enforced.

4. **Dynamic Stream Resolution**
   If a module registers as a stream observer or publisher, the loader facilitates the binding by inserting the module into the runtime stream graph. This enables modules to dynamically exchange data without hardcoded links.

This mechanism ensures that modules remain portable, secure, and extensible, while enabling fine-grained reuse of functions, services, and reactive interfaces across the runtime environment

# 4. Decentralized Method Execution

Decentralized Method Execution (DME) provides a cross-process communication protocol based on shared memory. Instead of relying on process-level messaging or sockets, DME uses memory-mapped files to enable real-time, bi-directional exchange of instructions and serialized data. This approach facilitates interoperability between tools written in different languages or hosted in isolated environments.

Each DME connection uses a shared memory region divided into two logical areas. The first is the instruction stack, a serialized queue of commands and execution metadata. The second is the data

segment, which holds shared objects, temporary buffers, and serialized structures to support instruction execution

## 4.1 Execution and Synchronization

At any point in time, only one process has write and execution privileges over the instruction stack. This ownership is encoded in metadata—such as process ID or access token—and governs write access to ensure consistency and avoid race conditions. Other processes may still read from memory, allowing non-blocking monitoring or validation.

Instructions are executed by an embedded interpreter running in the respective process. This interpreter is lightweight, stack-based, and structurally inspired by virtual machines like WebAssembly (WASM) or the .NET Intermediate Language (IL). It supports function calls, conditional branching, memory manipulation, and foreign function interface (FFI) calls.

A rollback mechanism allows previously valid instruction sets to be restored in case of execution failure. This ensures transactional execution semantics and prevents partial state corruption.

To notify peer processes of state changes or execution readiness, DME implementations rely on platform-specific signaling. On Windows, a dedicated WindowMessage may be registered to alert cooperating processes. On Unix-like systems, equivalents such as eventfd, UNIX domain sockets, or file locks serve the same role, ensuring compatibility and responsiveness

## 4.2 Native Method Integration

DME supports the invocation of both interpreted and natively registered methods. Interpreted methods reside in the shared memory region and are executed by the DME interpreter. Native methods are registered through a platform-specific method registry, exposing their function signatures and entrypoints in a secure, validated manner.

This hybrid execution model allows modules to dynamically invoke shared utility functions, call into runtime-provided services, or trigger host-specific behavior. Since native methods are exposed through a known interface and executed within sandbox constraints, the model maintains both flexibility and safety

## 4.3 Distributed and Networked Systems

While DME can be extended to support distributed execution across multiple physical or virtual machines within a network. This enables modular tools and components to collaborate securely and efficiently across system boundaries—such as between servers, development workstations, or client-server architectures.

In a distributed configuration, the shared memory model is emulated over the network using a thin synchronization layer, typically built on top of UDP or TCP. This transport layer mirrors the contents of the DME memory segment (instruction stack and data segment) between machines, enabling remote tools to observe, modify, and execute instructions as if they were local.

This approach allows DME to act as a lightweight, platform-independent remote

execution protocol, capable of supporting scenarios such as:

- Cross-machine tool orchestration in build farms or CI pipelines.
- Remote debugging and instrumentation in development environments.
- Client-server interaction for games or simulations where behavior trees, data transformations, or validation modules are executed in isolated backends.

To maintain transactional consistency and performance, the networked DME layer can implement write-lock arbitration, instruction framing, and differential memory updates. Compression and checksum validation can further optimize transmission and reliability over slower or unstable connections.

Security remains a core focus: all communication may be encrypted using TLS or symmetric ciphers (e.g., AES), and participants in a DME network must negotiate trust and permissions before memory synchronization is permitted. Additionally, sandboxing constraints continue to apply to all participating modules, regardless of network origin.

This extension of DME opens the door to building decentralized tool ecosystems within secure enterprise networks or across distributed infrastructure, while preserving the same unified programming and execution model as local use cases

# 5. Sandbox and Security Model

All modules—whether executed in host mode or standalone—run within a sandbox that enforces strict boundaries around system resources, APIs, and capabilities.

Access to the file system is restricted through mount namespaces and allowlisted paths. Modules can be granted read-only or read-write access to specific directories. This prevents arbitrary access to the user's file system and isolates build outputs, configuration, or cached data.

Network access is similarly gated. Permissions may be granted or denied based on domain names, IP ranges, or protocols. For example, a validation module may be permitted to retrieve remote schema definitions over HTTPS but denied general outbound access.

Inter-module communication is also tightly controlled. A permission model governs which modules may connect to which data streams or invoke which exported functions. This prevents unauthorized use of sensitive functionality or leakage of internal data between modules.

On Linux, this sandboxing is implemented using tools such as Bubblewrap (the core of Flatpak), Linux namespaces, seccomp (for syscall filtering), and AppArmor (for mandatory access control). Flatpak itself is a modern sandboxing system that allows desktop applications to run in isolated containers with well-defined permissions.

On Windows, isolation is achieved through AppContainer, a lightweight process sandbox introduced in Windows 8 and widely used for Store apps and system-isolated processes. AppContainer uses Security Descriptors and access control tokens to isolate processes from the file system, registry, and network unless explicitly permitted.

This dual-platform approach ensures consistent security semantics regardless of operating system.

## 6. Future Enhancements

Looking ahead, this architecture may incorporate advanced features for trust validation and protected execution.

Module authenticity and integrity can be ensured through cryptographic signatures. An ECDSA-based signature validation mechanism would allow hosts and shells to verify that a module has not been tampered with after publication. Signature metadata could be embedded in the .cm or .xcm file header and validated as part of the module loading process.

Additionally, modules may be optionally encrypted using symmetric algorithms such as AES to prevent reverse engineering or unauthorized access. This may be especially valuable in game engines or commercial toolchains where intellectual property must be protected.

Execution environments may restrict loading to only signed or trusted modules. In combination with sandboxing and stream registration, this establishes a robust trust boundary suitable for enterprise-grade deployment.

Further ideas include declarative sandbox profiles in JSON or YAML format, making permissions explicit and machine-readable. Module registries could evolve to include versioning, semantic metadata, and OCI-style distribution mechanisms. Future shells may support hot reloading of modules, dynamic stream routing, and introspection tools for debugging or inspection

## 7. Conclusion

This research outlines a secure, modern, and extensible approach to building, distributing, and executing tools and modules in a unified architecture. With support for reactivity, sandboxing, native exports, and reversible execution, this design serves as a foundation for robust development environments across platforms.

By combining the power of dynamic modules and cross-process method execution with secure runtime isolation, this system can significantly reduce integration complexity while increasing confidence in toolchain behavior and modular interoperability