A Data-Driven System for Reactive, Multi-Language Build Pipelines

1. Introduction and Motivation

In modern software development, managing and building complex, multi-language projects often requires navigating a minefield of configuration, dependency management, and platform-specific quirks. Existing tools, while powerful in their niches, tend to suffer from a set of recurring limitations: they are either tied to a single language ecosystem, require verbose and error-prone configuration, or assume a rigid build model that does not generalize well to modular, dynamic applications.

This research introduces a new approach—a fully data-driven, reactive project management and build system—that leverages conventions and structured project layouts to infer and control project state without requiring explicit configuration files. Instead of shifting cognitive and maintenance overhead onto users and administrators, this system offloads complexity onto the tool itself, guided by a small number of clearly defined rules and extension points.

The system is designed with modularity by default, supporting both small tooling utilities and large-scale applications such as game engines. It allows projects to be detected, processed, extended, and built dynamically based on the contents of the file system, language-specific analysis, and modular plugins. It is implemented in C#, but not limited to .NET development—through plugins, it supports languages like C++ and Rust as first-class targets.

A unique strength of the system is its ability to "bootstrap" itself. Starting from a minimal setup, the tool can build and reconfigure itself through additional plugins, offering extensibility without requiring complex environment setups. As a result, developers can compose and evolve entire build pipelines, package repositories, and tooling ecosystems purely through data and convention

2. Core Architecture Overview

At its core, the system is structured as a stream-oriented, data-driven engine. The following architectural principles define the behavior:

- Convention Over Configuration: Project structure and file types are used to infer intent
- Entity-Component-System (ECS): All files, folders, projects modules, build artifacts and any resulting output are represented as Entities with modular Components
- Reactive Streams: Components are published to type-specific data streams. Observers ("Systems") attach to these and transform, modify or enrich the data
- Flush-Oriented Control Loop: A central controller orchestrates the pipeline by flushing all registered streams. As long as any stream produces new output, another pass is triggered

2.1 Entity-Component Model

What is ECS?
The Entity Component System (ECS) is a data-oriented architectural pattern that replaces traditional inheritance trees with a flat structure composed of:

- Entities: Simple identifiers
- Components: Typed data blocks
- Systems: Logic processors that operate on specific component combinations

ECS excels at performance-critical applications by reducing indirection, improving cache locality, and separating data from logic

2.2 Reactive Streams

What is Reactive Programming?
Reactive programming is a paradigm centered around data streams and change propagation. In this model, components react to incoming events, state changes, or time-based triggers by emitting updates through observable channels

2.3 Data Model

Projects are automatically discovered by scanning upwards for a Config/ directory or downward for qualifying source files. Each discovered project becomes an Entity, enriched with information about its structure, language, and role via Component-based inference. These entities propagate through various processing streams: detection, classification, linting, dependency resolution, transformation, and finally build or export.

The data model is fully extensible. Plugins can register new component types, observers, or pipelines. Entities can carry metadata, build instructions, output specifications, or language-specific traits. The system supports cross-project dependencies and builds the overall project graph dynamically as information becomes available.

The use of flush-based reactivity rather than asynchronous event propagation makes the pipeline predictable and deterministic, while remaining highly extensible. This design allows developers to inject tooling at any stage of processing without altering global flow logic

3. Package System and Dependency Management

The system employs an integrated package and dependency model inspired by the flexibility of established ecosystems like NPM or rust cargo. Every project module can be represented as a package, described either by a conventional package.json/cargo.toml file or by the upcoming Alchemy file format. These packages can be stored locally, downloaded from remote registries, or hosted in self-managed package servers

3.1 Package Resolution

Each package defines its metadata, entry points, exposed files, and declared dependencies. Peer dependencies are used extensively to express plugin relationships—e.g., a plugin package that enhances a language parser may declare a peer dependency on the language handler itself.

The dependency resolution mechanism is automatic and recursive: when a project is selected for processing, the system locates its package definition, resolves its dependencies, and registers any linked or peer-dependent modules as additional build targets. If packages provide streams, components, or systems, they are loaded and activated dynamically during the flush cycle

3.2 Naming Convention

Namespaces and naming conventions (e.g., host.language.module) help to organize packages and decouple internal APIs from public-facing modules. Traditional package management systems such as NPM or Cargo often operate within language-specific silos, assuming a 1:1 relationship between registry and ecosystem. This constraint limits reuse across language boundaries and complicates the integration of heterogeneous toolchains. To address this, the proposed system adopts a modular naming convention that explicitly encodes both origin and domain context into each package identifier.

This convention dissolves the artificial boundary between languages by allowing a unified registry to host C#, C++, Rust, or even shell-based modules side by side, while still preserving semantic clarity and separation of concerns. A single build system can then resolve dependencies across ecosystems without ambiguity or tooling fragmentation, enabling seamless orchestration of mixed-language projects within a shared pipeline.

Custom registries like GitHub Packages or self-hosted servers can be configured to host private or segmented packages, enabling controlled internal development workflows alongside public ecosystems

3.3 Module Dependency

Dependency graphs are derived not from build scripts but from static analysis and structural rules—each source file's relationships are resolved through linting and language-specific inspection. This allows fully declarative package usage without requiring verbose configuration or pre-written build scripts.

The package system ensures reproducibility through lockfiles and content hashes and can generate templates for multiple IDEs or external build systems, bridging declarative structure with imperative tooling

4. Stream Model

The reactive stream architecture lies at the heart of the system. Each stream corresponds to a logical phase of the processing pipeline: discovery, parsing, linting, analysis, classification, build orchestration, and export. Entities are published into streams based on their type and state. Observers, known as systems, consume stream data and may mutate entities or emit new ones.

A central flush controller manages lifecycle and ordering. Every pass flushes all registered streams and collects the number of entities processed. As long as one or more streams yield new data, further passes are scheduled. This guarantees a fixpoint traversal without unnecessary recursion or asynchronous branching

4.1 Language specific Linting

Language-specific linting systems act as both analyzers and dependency resolvers. For C#, this includes analysis of using directives, namespace declarations, and reflection-based inspection of assemblies. For C++, the system processes #include directives, resolves translation units, and maps both static and dynamic linkage.

All file-based information is cached with hash-based deduplication. Conditional parsing paths are resolved via a preprocessing layer that supports token skipping based on #if/#ifdef constructs. The parser and linter both implement LR(1)-like logic, and can be extended via pattern-based rules that match against token sequences or AST fragments.

Because streams are observable and extensible, third-party systems can participate in the processing loop without requiring core modification. This enables language-specific extensions, security checks, or visualization tooling to operate transparently

5. Plugin Model and System Extensibility

Plugins are first-class citizens within the system. They are defined as packages that declare peer dependencies on core subsystems and expose streams, systems, or components. Plugins are auto-discovered during project analysis and activated if matching host versions are present.

Each plugin may extend the build graph, inject preprocessing logic, introduce new entity types, or provide additional stream observers. Plugins do not require manual registration—dependency analysis and naming conventions determine loading behavior.

The flush model ensures that plugin systems execute in deterministic order relative to other systems. By defining dependencies between streams, developers can build custom orchestration models without compromising the global execution contract.

Examples of plugins include:

- Language extensions (e.g., C++, Rust or Java)
- IDE generators (e.g., Visual Studio or Rider)
- Platform targets (e.g., linux-x64, web-wasm)
- DSL processors or schema validators

6. Alchemy File Format

While traditional package descriptors like package.json, Cargo.toml, or project.yaml are well-supported, the system introduces a novel textual format called Alchemy Files. These aim to be human-friendly and syntax-light, while remaining translatable to standard formats.

Alchemy Files:

- Use indentation and keywords instead of braces or delimiters
- Can embed templates, conditional logic, and parameter substitutions
- Act as declarative control surfaces for projects, tasks, and pipelines

Alchemy is not merely a configuration format—it is a lightweight, language-agnostic text processor designed to act as an intermediary between human-readable configuration and structured, machine-consumable formats like JSON, YAML, or TOML. Inspired by the behavior of C/C++ preprocessors, Alchemy supports conditional directives such as if, ifdef, define, and import, which are tokenized and compiled into an abstract stream of instructions.

Unlike static formats, Alchemy operates in active exchange with the target parser. While Alchemy processes and transforms its input, the downstream consumer (e.g., a JSON interpreter) can provide structural expectations or constraints back to the processor. This bidirectional communication allows Alchemy to resolve context-sensitive configurations, perform structural rewrites, or inject defaults dynamically—without requiring manual duplication or format-specific syntax.

The result is a system where domain-specific logic, platform rules, or plugin configuration can be centralized, abstracted, and modularized. Alchemy files may ultimately describe anything from build targets to plugin manifests to orchestration pipelines—but always through a token stream that bridges free-form syntax with structured expectations.

The actual surface syntax remains intentionally flexible and is subject to further development in a dedicated specification. For now, Alchemy serves as the connective tissue between expressive input and deterministic output, giving developers and tools alike a powerful shared representation

7. Conclusion

This research has introduced a highly modular, data-driven framework for managing and building projects across languages and domains. By combining reactivity, ECS-style structure, and flush-based synchronization, the system enables powerful, deterministic workflows with minimal user effort.

Its extensibility through packages and plugins allows for ongoing evolution, while the upcoming Alchemy format opens the door for more accessible configuration and integration. From static analysis to full build orchestration, the entire lifecycle is encapsulated in a set of predictable, observable transformations.

The approach outlined here addresses common pain points in modern software development—reducing manual configuration, eliminating fragile build scripts, and encouraging ecosystem-level modularity. It lays the foundation for a new kind of tooling system: declarative, composable, and introspective by design.

Future work will include formalizing the plugin API, refining language support layers, and completing the Alchemy specification. In doing so, we hope to provide a foundation for scalable, maintainable, and highly adaptable build tooling—one that grows with the software it helps to build