Under the Blueprints: Parsing Unreal Engine's Visual Scripting at Scale

Kalvin Eng
Department of Computing Science
University of Alberta
Edmonton, Canada
kalvin.eng@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
abram.hindle@ualberta.ca

***Abstract*—In Unreal Engine, a popular game engine for AAA (high budget, high profile) title video games, Blueprint Visual Scripting is a widely used tool for developing gameplay elements using visual node and edge-based source code. Despite its widespread adoption, there is limited research on the intersection of software engineering and Blueprint-based visual programming. This dataset aims to address this gap by providing parsed Blueprint graphs extracted from Unreal Engine's binary UAsset files. We developed extractors and a custom parser to mine Blueprint graphs from 335,753 Blueprint UAsset files across 24,009 GitHub projects. By providing this dataset, we hope to encourage future research on the structure and usage of Unreal Engine Blueprints, and promote the development of tools—such as code smell detectors and language models for code completion—that can optimize visual programming practices within Unreal Engine.**

***Index Terms*—Unreal Engine, Blueprints, Visual Code**

I. Introduction

Visual programming has become a transformative tool in video game development. Unlike traditional text-based programming, visual programming enables users to create programs using graphical elements and connections. This approach provides a more intuitive way of coding, making it more accessible to individuals with design backgrounds [1, 2]. One commonly used visual programming system in video games is the Unreal Engine Blueprint Visual Scripting system [3], which facilitates the creation of gameplay elements through node and edge-based source code. An example of a Blueprint can be seen in Figure 1.

Visual programming is a widely used approach in video game development and is notably used in AAA (high budget, high profile) title video games [1, 2, 4]. Studying visual programming is economically valuable as many game studios rely on visual code for developing their AAA titles. Understanding visual programming patterns and structures is crucial for developers and educators, as it reveals their effectiveness in game development and highlights areas for improvement.

This paper presents a dataset of Unreal Engine Blueprint graphs parsed from 335,753 asset files using a custom developed parser to help researchers studying visual programming.

II. Unreal Engine and Blueprints

Unreal Engine 4 (UE4) is a game engine for developing video games, publicly released in March 2014 [5]. Among its features is Blueprints, a robust and accessible visual scripting system enabling game designers and artists to create gameplay logic/functionality without writing code line-by-line.

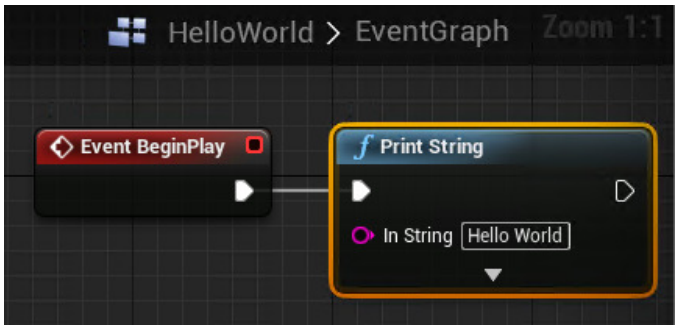

Fig. 1: An example of a Blueprint event graph where "Hello World" is printed to the screen when the game begins playing.

Unreal Engine allows for the creation of *game worlds* that contain *game objects* and the objects can be built with *components*. The game world and game objects can be controlled via a GUI, and components can be defined via the Blueprint visual scripting system. The game world, game objects, and components can also be defined via C++ code.

While the Blueprints visual scripting system is mainly used for defining gameplay elements, it should not be confused with the graph editor for Blueprints, which is also used elsewhere in the Unreal Engine Editor. Game developers and designers are able to use nodes and edges: to define how objects should be drawn on the screen in the *Material Editor* [6]; to control character animations with the *Animation Graph* [7]; to develop AI logic with *Behavior Trees* [8]; to create special effects with the Niagara FX system [9]; to manage the logic of gameplay widgets with *Widget Blueprints* [10]; to generate content for worlds with the procedural content generation framework [11]; and to generate sounds with the MetaSounds system [12].

A. Blueprints Visual Scripting System

The Blueprints visual scripting system consists of a graph containing nodes with pins. Each node represents an action or logic, such as a variable, function, or flow control operation. The connections made between nodes with pins represent the flow of logic. Figure 1 illustrates a basic Blueprint in which

the "Event BeginPlay" node activates the "Print String" node through connections established by "exec" pins, resulting in the display of "Hello World" at the start of the game.

Blueprints are similar to Abstract Syntax Trees (ASTs) because they illustrate the structure of a program without including detailed lower-level information. Blueprints are visual representations that focus on logical flow and function like an AST by outlining key operations. During the runtime compilation process, the Blueprint graph is transformed into bytecode for execution by a virtual machine. Blueprints made with Unreal Engine 4 can be converted into C++ code using Blueprint Nativization [4], which reduces runtime overhead. Blueprints are stored in their uncompiled form within `.uasset` files, allowing the node and edge-based source code to be parsed independently, without requiring a full set of project files.

There are five types of Blueprints [13]: (1) *Level Blueprint* to define gameplay logic at a level scope; (2) *Blueprint Class* to define gameplay functionality as a class; (3) *Data-Only Blueprint* to tweak properties of a parent Blueprint Class; (4) *Blueprint Interface* to allow Blueprint Classes and Level Blueprints to share and send data with each other; and (5) *Blueprint Macro Library* stores self-contained graphs that can be added to other blueprints.

A Blueprint is similar to a C++ class and takes in a parent class to inherit. When creating a Blueprint or Blueprint Macro Library, there is a GUI prompt that allows you to select the parent class of your Blueprint. The most common parent classes presented are: Actor, Pawn, Character, Player Controller, Game Mode, Actor Component, Scene Component. A parent class is the base class of your Blueprint providing predefined functions and variables for an object. Data-Only Blueprints use a Blueprint as its parent class. While the Blueprint Interface class uses the object interface class as its parent class. A Blueprint augments an inherited parent class and produces a "Blueprint Generated Class" for runtime use.

With the exception of the *Level Blueprint*, the other four types of Blueprints are stored as an `.uasset` Unreal Engine asset binary file, thus we extract the four types of Blueprints in our dataset. Within the binary file contains information about the asset filename, parent classes, nodes, and pins for the Unreal Engine editor. Because the asset filename is embedded within the binary file, the asset cannot be renamed outside the Unreal Engine editor as the binary file needs to be updated with the new filename. The recommended naming convention for Blueprints is to append an asset name with "BP_" [14].

Blueprints mainly consist of *events* and *functions*, each with a graph editor for manipulating nodes. A function graph creates local variables and typically returns a value. Every object Blueprint contains a *Construction Script* function graph, which is executed when the object is instantiated in Unreal Engine. In contrast, an event graph allows functions or actions to be triggered based on occurrences in the game, such as player inputs, collisions, or changes in game state, enabling timed execution. The graph editor also includes *event dispatchers* for one-to-many communication between Blueprints and *macros*, which condenses node groups into a single node.

There are four primary graph types: Event Graph (UberGraph), Function Graph, Macro Graph, and Delegate Signature Graph, with an additional Implemented Interfaces Graph for Blueprints that implement interfaces. The Event Graph is where gameplay logic is created and stored as an UberGraph Page, which is then compiled into a single UberGraph to manage all event-based logic. Function and Macro Graphs define reusable logic, while Delegate Signature Graphs represent event dispatchers, enabling communication between objects.

III. Related Work

To the best of our knowledge, there has been no work in analyzing Unreal Engine Blueprints at scale. This makes our dataset a unique contribution to the domain of visual programming research. Visual programming involves the creation of visual code, which is also known as block-based code, low-code, no-code, or visual scripts [1, 2].

Two published datasets related to our research on visual programming exist. One dataset [15] focuses on Scratch, a block-based visual programming language used for coding education. The other Pure Data dataset [16] is more closely related this work as Pure Data is a visual programming language that employs nodes and edges akin to Unreal Engine Blueprints. Pure Data is simpler and primarily utilized in audio engineering, while Unreal Engine employs Blueprints for video game development.

IV. Methodology

We used World of Code [17] to find projects and clone the projects' git repositories for the binary Blueprint `.uasset` files, then we parsed the files using our parser tool and stored the Blueprint graphs, nodes, and pins into a relational database.

A. Project Discovery

To discover projects that use Unreal Engine on GitHub, we used the World of Code (WoC) version V dataset [17]. The WoC version V dataset contains 16,252,394,678 blobs and 1,31,171,380 distinct repositories retrieved by Mid May, 2023 [18]. We searched for filenames with the `.uasset` file extension using the "b2f" basemap. The name "b2f" stands for "blob to filename"; a blob is a SHA-1 of a file's content that is used for lookup, while a filename refers to the name of a file in a git repository. A blob can have multiple filenames. 38,172,994 "b2f" entries were found to contain the `.uasset` file extension. A major challenge to using the WoC dataset is that the binary contents of blobs are not stored — only text blobs as defined by git are stored. Therefore, we manually mirrored git projects containing the blobs.

Using the 38,172,994 entries, we retrieved possible Unreal Engine git projects with the "b2P" lookup script. The name "b2P" stands for "blob to a distinct repository". A "distinct repository" is identified as the most central repository within a cluster of repositories, as determined by the Louvain community detection algorithm [19]. Using distinct repositories, one can avoid many duplicated projects (forks) during analysis. 33,899 distinct GitHub repositories were discovered.

We were able to successfully clone 27,108 git repositories as of October 20, 2023. Cloning was done using the "mirror" option ensuring a complete copy of the source git repository including branch names. We use the latest updates of the git repositories as of July 3, 2024 for our analysis. In total there is 21 TB of repository data cloned.

We traversed the git trees of cloned projects on the main branch to find binary files ending with `.uasset`. We hashed the binary files with MD5 to use as an identifier for our database schema described in Section IV-C. The main branch is considered to be the current branch of a repository when running the `git branch` command. We parsed the binary `.uasset` files, ignoring null files and filenames that mismatch the embedded filename. Some files were unparseable due to being Git LFS placeholders, while others were partially parsed due to unresolvable graphs and nodes. Of the 27,108 cloned git repositories, we find that 24,060 repositories contain at least 1 partially parseable Blueprint file. There are 341,519 unique Blueprint files in total with: 2,462 serialized by an unknown custom Unreal Engine Version; 336,479 serialized by Unreal Engine 4; and 2,578 serialized by Unreal Engine 5. We fully parse graphs, nodes, and pins from 335,753 Unreal Engine 4 Blueprint files and store them in an SQLite database.

B. Blueprint Parser Tool

Our Blueprint parser tool is developed to extract Blueprint graphs, nodes, and pins from Unreal Engine 4 (UE4.0 - UE4.27) `.uasset` files by de-serializing its binary contents. It was created by analyzing the Unreal Engine source code. The Blueprint UAsset binaries contain serialized objects (instances of classes) and their properties. Unreal Engine's serialization process has developed over various versions of Unreal Engine, increasing its complexity. Our work on parser development reveals several noteworthy serialization changes:

- **UE4.0 - 4.12:** Blueprint graphs, nodes, and pins are serialized as objects.
- **UE4.13:** Major change where pins are no longer objects, but stored as properties of a node object.
- **UE4.14:** Changes to pin serialization with a new custom version scheme to serialize data for determining pin types.
- **UE4.17:** Pin serialization was changed to using container types to handle data serialization.
- **UE4.19:** Pin names are now serialized as a new type `FName` instead of `FString`.

We have released our parser with the dataset package [20] to support further development of the parser for Unreal Engine 5 and research into the Unreal Engine ecosystem.

C. Data Storage

The extracted Blueprint data about graphs, nodes, and pins are stored in a SQLite database file and a simplified schema can be seen in Figure 2. The schema consists of several interrelated tables, each storing different properties and relationships. The main tables include:

- **git_files**: Stores metadata for UAsset files in a Git repository, including file paths, commit details, and hash values.

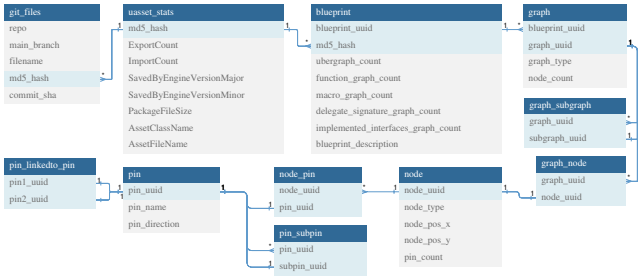

Fig. 2: Simplified database schema of extracted Blueprints.

- **uasset_stats**: Contains information about all UAssets extracted from Git repositories, including compatibility with different engine versions, asset types, file sizes, and export/import counts. The "md5_hash" column is used to link a uasset with a blueprint.
- **blueprint**: Represents Blueprints, storing graph counts, and blueprint properties like description. Blueprints are associated with one-or-many graphs via the "blueprint_uuid" column.
- **graph**: Represents graphs, storing node counts, and graph properties like type. Graphs are associated with one-or-many nodes with the "graph_uuid" column.
- **node**: Represents nodes, storing pin counts, and node properties such as x and y position on the graph editor. Nodes are associated with one-or-many pins via the "node_uuid" column.
- **pin**: Represents pins, storing pin properties such as name and direction. Pin-to-pin relationships are stored in the **pin_linkedto_pin** table linking pins with the "pin_uuid" column and can be used to reconstruct a full graph.

Each of the major entities (blueprints, graphs, nodes) also has a corresponding table for extra properties, allowing the schema to store additional properties extracted from the UAsset Blueprint that do not commonly appear in other Blueprints. There are also tables to model subgraphs (**graph_subgraph**), subpins (**pin_subpin**), and linked pins (**pin_linkedto_pin**).

D. Data Validation

We developed our parser by testing Blueprint files across UE4 versions (UE4.0–UE4.27), ensuring error-free parsing and refining it for compatibility with files we made and the dataset files. From the parsed dataset, we randomly sampled 19 UAsset files without replacement. The sample size was determined using Cochran's formula (z-score of 1.31, 90% confidence level, 0.5 variability, and 15% precision). The small sample size was due to the labour-intensive process of manually comparing graphs in the Unreal Engine Editor to our parsed data, which involves downloading each project containing the file, opening it in the appropriate Unreal Engine version, and matching nodes and connections one-by-one.

V. Dataset Analysis

In this section, we provide a summary of our dataset queried with SQL. The extent of our dataset can be seen in Table I.

	Count	Mean	SD	Min	25%	50%	75%	Max
Event Graph (UberGraph Page) Per Blueprint	324,767	1.01	0.16	1	1	1	1	18
Function Graph Per Blueprint	300,320	1.68	2.97	1	1	1	1	236
Macro Graph Per Blueprint	7,903	2.38	3.69	1	1	1	2	72
Delegate Graph Per Blueprint	6,749	1.74	1.95	1	1	1	2	42
Implemented Interfaces Graph Per Blueprint	16,040	0.93	2.61	0	0	0	1	73
Total Graphs Per Blueprint	335,757	2.62	3.32	0	2	2	2	255
Nodes Per Graph	857,207	13.25	35.51	1	1	4	12	3,435
Pin Connections Per Graph	475,394	23.36	52.45	1	4	10	24	4,432
Pin Connections Per Node	10,661,298	2.31	1.55	1	1	2	3	359
Unique Blueprints Per Project	24,009	22.50	33.11	1	9	13	24	902
Extracted Blueprint File Size (Bytes)	335,753	158,348.31	318,540.25	2,409	32,862	108,369	165,526	39,789,332
Blueprint Clones Across Projects	16,585	13.32	265.70	2	2	2	3	12,523

TABLE I: Summary Statistics for Blueprints

Node Type	Count	%
CallFunction	3,683,795	32.44
VariableGet	2,310,101	20.34
VariableSet	845,151	7.44
Event	654,211	5.76
IfThenElse	541,540	4.77
FunctionEntry	532,236	4.69
Knot	528,746	4.66
CommutativeAssociativeBinaryOperator	328,224	2.89
MacroInstance	245,828	2.16
DynamicCast	203,711	1.79

TABLE II: Top 10 Node Types

Baseline Comparison: We compare our dataset to the Pure Data dataset [16] as a rough baseline using the summary statistics in Table I. While Islam *et al.* [16] assumes one graph per file, our dataset has multiple graphs per Blueprint UAsset file, making our comparison approximate. Islam *et al.* [16] report a mean of 94 nodes and 85 connections per graph, with a median of 25 nodes and 16 connections. Our dataset has a mean of 13 nodes and 23 connections, with a median of 4 nodes and 10 connections per graph, indicating simpler graphs.

Graphs: Graphs, consisting of nodes and pins, are central to Blueprints with four primary types (Event, Function, Macro, Delegate) representing different gameplay logic. Table I shows that Event Graphs (UberGraph Pages) and Function Graphs are the most common, with significantly fewer Macro, Delegate, and Implemented Interfaces Graphs. Most Blueprints contain between 0-2 graphs. Interestingly, we can see that at least one of the Blueprints has 236 Function graphs which suggests high complexity for a Blueprint.

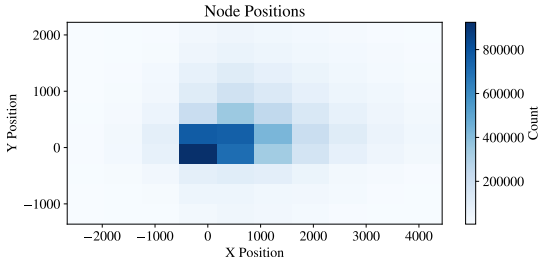

Fig. 3: Density heatmap showing the common positions of nodes in the Blueprint Editor. Outlier positions are omitted.

Nodes: Nodes in Blueprints represent actions or operations. From Table I, 25% of graphs contain only 1 node, 50% contain 1-4 nodes, and 75% have 1-12 nodes, showing most graphs are simple. However, one graph with 3,435 nodes indicates that some Blueprints are highly complex.

Additionally, Figure 3 presents a heatmap showing the distribution of node positions across all graphs. The heatmap reveals that most nodes are located within a range of -285 to 432 on the y-axis and -533 to 888 on the x-axis, highlighting that the majority of nodes across all graphs are concentrated within a relatively confined space in the coordinate plane.

Table II lists the top 10 most common Unreal Engine Blueprint node types and their percentage out of all node types. Nodes like **CallFunction** (32%), **VariableGet** (20%), and **VariableSet** (7%) are key for variable access and function calls, while **Event** (5.8%) and **IfThenElse** (4.8%) handle game logic. Other nodes, such as **FunctionEntry** (4.7%) and **Knot** (4.7%), serve specific formatting and entry purposes. These counts reflect the core functions of Blueprint visual scripting.

Pin Connections: Pin connections in Blueprints, similar to graph edges, link nodes to enable data flow. Table I shows the average number of pin connections per node is 2.31, with most nodes having between 1-3 connections, up to a maximum of 359 pin connections per node. This suggests that while most nodes have few connections, some are highly interconnected, representing high complexity. These highly connected nodes may indicate areas of high interdependency, where changes in one part of the Blueprint could impact several others.

Blueprint Clones: In Table I, it is shown that Blueprints with cloned binary file contents are cloned an average of 13 times. However, some Blueprints are reused significantly, with the highest clone count reaching 12,523. This underscores the importance of Blueprints in the development process, as many projects depend on the reuse of essential Blueprints. While cloning enhances efficiency, it also raises concerns regarding the proliferation of defects through code replication.

VI. Future Research

With our extensive Unreal Engine Blueprint graphs dataset, researchers can explore static analysis, quality assessment, and code completion. Potential research questions (RQs) include:

- How can static analysis techniques be adapted to assess the quality of Unreal Engine Blueprint graphs?
- What common defects and code smells exist in Unreal Engine Blueprints, and how do they impact performance?

Our dataset supports developing tools that leverage past examples like code completion, auto-layout, refactoring suggestions, and language models. By analyzing node details, pin types, connections, and graph structures, researchers can train models on the syntax and semantics of Unreal Engine Blueprints to develop these tools.

VII. Conclusion

We present a dataset of Blueprint graphs, nodes, and pins from 335,753 Unreal Engine asset files across 24,009 public Github projects, along with a parser tool for extracting Blueprint graphs, nodes, and pins from Unreal Engine 4 UAsset files. All code for extraction and dataset construction is available in our dataset package [20].

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2022-03464.

References

[1] K. Eng, A. Hindle, and A. Senchenko, "Identifying Defect-Inducing Changes in Visual Code," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 474–484.

[2] ——, "Predicting Defective Visual Code Changes in a Multi-Language AAA Video Game Project," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 485–494.

[3] Epic Games, "Blueprints Visual Scripting," 2017. [Online]. Available: https://web.archive.org/web/20170708113303/https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html

[4] ——, "Nativizing Blueprints," 2017. [Online]. Available: https://web.archive.org/web/20170708170345/https://docs.unrealengine.com/latest/INT/Engine/Blueprints/TechnicalGuide/NativizingBlueprints/index.html

[5] ——, "Epic Games Releases Unreal Engine 4 for All," 2014. [Online]. Available: https://web.archive.org/web/20140701175219/http://epicgames.com/news/epic-games-releases-unreal-engine-4-for-all/

[6] ——, "Essential Material Concepts," 2014. [Online]. Available: https://web.archive.org/web/20140907144008/https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/IntroductionToMaterials/index.html

[7] ——, "Animation Blueprints," 2014. [Online]. Available: https://web.archive.org/web/20140827015058/https://docs.unrealengine.com/latest/INT/Engine/Animation/AnimBlueprints/index.html

[8] ——, "Behavior Trees Nodes Reference," 2014. [Online]. Available: https://web.archive.org/web/20140906114310/https://docs.unrealengine.com/latest/INT/Gameplay/AI/BehaviorTrees/NodeReference/index.html

[9] ——, "Niagara Overview," 2023. [Online]. Available: https://web.archive.org/web/20231202152827/https://docs.unrealengine.com/5.0/en-US/overview-of-niagara-effects-for-unreal-engine/#modules

[10] ——, "Widget Blueprints," 2024. [Online]. Available: http://archive.today/2024.09.04-191950/https://dev.epicgames.com/documentation/en-us/unreal-engine/widget-blueprints-in-umg-for-unreal-engine

[11] ——, "Using PCG Generation Modes," 2024. [Online]. Available: http://archive.today/2024.09.04-150128/https://dev.epicgames.com/documentation/en-us/unreal-engine/using-pcg-generation-modes-in-unreal-engine

[12] ——, "Creating Procedural Music with MetaSounds," 2024. [Online]. Available: http://archive.today/2024.09.04-150746/https://dev.epicgames.com/documentation/en-us/unreal-engine/creating-procedural-music-with-metasounds

[13] ——, "Types of Blueprints," 2024. [Online]. Available: http://archive.today/2024.09.04-153544/https://dev.epicgames.com/documentation/en-us/unreal-engine/types-of-blueprints-in-unreal-engine

[14] ——, "Recommended Asset Naming Conventions," 2021. [Online]. Available: https://web.archive.org/web/20211128215252/https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/AssetNaming/

[15] E. Aivaloglou, F. Hermans, J. Moreno-León, and G. Robles, "A Dataset of Scratch Programs: Scraped, Shaped and Scored," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 511–514.

[16] A. Islam, K. Eng, and A. Hindle, "Opening the Valve on Pure-Data: Usage Patterns and Programming Practices of a Data-Flow Based Visual Programming Language," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 492–497.

[17] Y. Ma, T. Dey, C. Bogart, S. Amreen, M. Valiev, A. Tutko, D. Kennard, R. Zaretzki, and A. Mockus, "World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS Data," *Empirical Software Engineering*, vol. 26, pp. 1–42, 2021.

[18] A. Mockus, *Version V stats*, 2023, https://bitbucket.org/swsc/overview/commits/9163c503277fa4d4aeb5fbf336e41e5eb4c7d28b.

[19] A. Mockus, D. Spinellis, Z. Kotti, and G. J. Dusing, "A Complete Set of Related Git Repositories Identified via Community Detection Approaches Based on Shared Commits," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 513–517.

[20] K. Eng and A. Hindle, "Under the Blueprints: Parsing Unreal Engine's Visual Scripting at Scale (Dataset)," https://zenodo.org/records/14247703, 2025.