Dynamic vs Static Typing Performance for Built-In Types in GDScript in the Godot Game Engine

DMITRY GORYACHKIN, University of Twente, The Netherlands

Modern “dynamically typed” languages such as JavaScript and Python are praised for their accessibility and ease of use by many developers. A key feature between all of them is type dynamicity, which allows the type of stored data to change at run-time, which can make writing some code easier. This, however, does not come without some significant overhead for the interpretation and casting involved when working with these dynamic types. The Godot game engine scripting language, GDScript, implements both dynamic and static typing systems, where all variables are dynamic and interpreted at runtime, unless an explicit type is given, in which case they become static to that one given type. Official documentation recommends static typing to eliminate overhead wherever possible. This research will first measure the performance of common variable use cases in a high-iteration set of loops, using dynamic and static typing for each case. This testing will include all types. Lastly, all results will be processed via averages and percentage-based analysis to find out what is the true impact on performance for each type’s use cases. The research is expected to add to the scientific body of knowledge of game development using GDScript, the impact of dynamic typing on GDScript performance and appropriate development practices when working in GDScript

Additional Key Words and Phrases: Godot, GDScript, gradually typed, dynamic typing, static typing, game development, performance, overhead

1 INTRODUCTION

In the modern landscape of programming, a trend of interpreted, “dynamically-typed" languages has emerged. Languages like Python and JavaScript are praised and beloved for their ease of use by many developers [1, 6], in large part due to the ability of these languages to dynamically choose data types on the fly and to change them whenever it may be convenient. This eliminates the need for explicit type declaration, and thus allowing for faster work and prototyping by allowing programmers to focus less on the specifics of syntax and data structures. Many other areas of software engineering followed this trend, with game development being no exception. The FOSS game engine “Godot" created in 2001 and made open source in 2014, implements the principle of “gradual typing" [4] into its native scripting language GDScript. The language has relatively syntax, and is tightly integrated with the engine, making it a popular choice with Godot developers. Its inclusion of gradual typing, allows for programmers to choose between dynamic and static typing by either leaving out type annotations or by adding them in [8].

This is because GDScript requires explicit variable declaration before use in its syntax. All variables in GDScript by default are of a special type called “Variant" when their type goes unspecified, which then gets interpreted at runtime into the assumed correct data type. That is, unless the variable is given an explicit type annotation at declaration, in which case the variable will be locked into that one type both on compile and run time.

However the dynamic typing supposedly has a large enough impact on performance due to the associated overhead, that the engine’s official documentation recommends static typing for performance improvements [5]. Except, how much of a difference this makes, and in which situations is the overhead the worst is never made clear in the official documentation for the project. Taking a look at their blog, official numbers exist for the VM instruction performance difference, claiming a variety of improvements from 5% all the way up to 150% [7]. However, as these figures are not from actual GDScript code, they are not practical for quantifying performance benefits from static typing.

Furthermore, there have been no studies of the impact dynamic typing has on the performance GDScript directly at the time of writing. The benchmarking that has been done by the Godot engine community at large, has been rather limited in scope, with existing community figures being limited to only a specific combination of type and use case .

As well, even related research to this field of typing performance and general overhead is relatively scarce, but a few relevant papers do exist. For example in their research into optimizing the engine of JavaScript V8, Dot et al. [2] found that a very significant part of the execution time of JavaScript using V8 was due to all the overhead associated with its dynamic typing system, and claimed that the optimization they found would work for both JavaScript and other dynamically typed languages as well.

Meanwhile Goch-Zech et al. [3] tackled the problem of optimizing Python by looking to reduce overhead caused by various instruments and performance monitors. In the process, they found that not only is the overhead caused by instrumentation significant, but that controlling it can result in up to 20% better performance in their use case, illustrating the fact that there may be cases when performance drops may occur for less apparent reasons.

Both papers together indicate that modern implementations of dynamic typing can not only be very costly, but also are yet to be fully optimized. Combined, the mentioned works show that dynamicity can and will impact performance, while at the same time indicate that performance of the languages must be evaluated carefully to account for other overhead not directly related to dynamicity.

The insights provided by the studies, combined with the previously mentioned lack of concrete figures, shows modern literature has yet to fully address the specific performance impact of using static typing in GDScript. This absence of widely accepted figures, limits the ability for developers using GDScript to assess and compare the performance implications of static versus dynamic typing, underscoring the need for a more rigorous and comprehensive approach.

As such, this research aims to systematically evaluate the execution time differences between static and dynamic typing in GDScript. It benchmarks nearly all built-in GDScript data types across a defined

TScIT 43, July 4, 2025, Enschede, The Netherlands

set of common use cases and variable access patterns, in order to establish baseline performance metrics for dynamic typing and compare them against their statically typed counterparts. The objective is to produce concrete, data-driven figures that quantify the impact of static typing on execution time relative to dynamic typing.
To structure this investigation and maintain a focused analytical scope, this paper poses the following research questions. These questions are designed to explore the fundamental performance implications of static versus dynamic typing within practical GDScript contexts:

Research Question 1 (RQ1) What is the difference in performance for built-in types?
Research Question 2 (RQ2) What are the differences in performance impact between access patterns?
Research Question 3 (RQ3) Which common use cases have the most significant performance impacts?

To answer these questions, this paper conducts an empirical investigation based on systematic performance measurements within the Godot Engine. By analyzing execution times across a range of data types and use case, it aims to provide clear, quantifiable insights into the impact of static versus dynamic typing in GDScript. The remainder of the paper details the measurement methods, presents and interprets the results, and discusses their implications for performance-conscious development in Godot.

2 MEASUREMENT METHODS

Godot is a game engine made open source in 2014. It features its own scripting language called GDScript, which supports optional type hints for variable declarations, allowing the interpreter to enforce type rules when specified. This flexibility gives developers the choice between static and dynamic typing, depending on their needs. This section outlines the approach used to measure execution time differences between these two typing modes. The goal is to create controlled and repeatable benchmarks that reflect real-world usage patterns while isolating the effects of typing. To achieve this, the tests are designed to focus solely on performance factors directly attributable to the type system, excluding unrelated sources of variation.
GDScript features a rich and extensive type system, comprising 38 built-in types alongside hundreds of user-defined or engine-provided types derived from the base“Object" type, each with their own implementations and nuances. This research limits its scope to the built-in types, as these are implemented at the engine level and exhibit consistent behavior across projects. In contrast, Object-derived types often introduce performance variability due to custom logic and implementation-specific overhead, which falls outside the scope of this typing-focused performance analysis.
A variable defined with a specific type can behave differently depending on how it is used. To accurately measure the performance impact of dynamic typing, this research first establishes three key concepts that form the foundation of the analysis:

- The distinct common use cases to be tested.
- The data types that will be subject to each test case.
- The way both combined will be measured and compared.

These concepts are defined prior to implementation to ensure consistency across all stages of the research. Most importantly, they help keep the focus on measuring performance differences in common, real-world use cases.

2.1 Use Cases

Before selecting which built-in types to include in testing, it is first necessary to define the test cases themselves. Since the objective is to measure the relative performance impact of static typing, only the overhead directly associated with the variable should be considered. This research assumes that the behavior of a variable can be meaningfully represented through the following defined test cases:

Test Case	Example Syntax
Declaration	`var variable: type = value`
Setting	`variable = value`
Getting	`variable`
Operation	`variable + value`
Comparison	`variable == value`
Method Call	`variable.method()`
Type Coercion	`variable as other_type`

Table 1. Test cases and their representative syntax

This classification is based on distinct syntax associated with each test, as well as assumptions about how interpreted languages typically operate. For example, operation and comparison are treated as separate tests despite their similar syntax, because it is assumed that operations generally incur less overhead than comparisons. Additionally, type checking is considered as a separate test; however, GDScript disallows type checking on statically typed expressions by throwing a compile-time error, making this test applicable only to dynamic typing.
Additionally, GDScript provides two generic collection types:“Array" and “Dictionary". Both can store values of any data type, with Dictionary also supporting any data type as a key. As a result, variables can be accessed either directly or through indexing into these collections. Furthermore, both Array and Dictionary support static typing of their value types, and Dictionary also allows for static typing of its key type. Taken together, this results in five distinct access patterns: direct variable access, Array access, inner-typed Array access, Dictionary access, and inner-typed Dictionary access. Each test is therefore repeated across all five access patterns to address **RQ2**.

2.2 Data Types

GDScript provides 38 built-in types. Of these, this research includes 36, excluding “RID" and “Signal". The RID type represents a numerical resource ID used internally by the engine to manage objects in memory, and it cannot be instantiated directly without active objects. Similarly, Signal requires explicit declaration syntax similar to that of functions and lacks a literal form. Due to these limitations, neither type can be reliably tested in isolation and is therefore excluded from this research.

Additionally, while “null" is officially documented as a type, it simply represents the absence of a value or type. Since null has no behavior and cannot participate in any test case, it is also excluded from testing, despite technically being listed as a type.
Some data types are not applicable to certain test cases. In such cases, the incompatible test is omitted for that type.
Finally, although GDScript supports static typing of Dictionary keys, fully testing every data type as a key for every combination of inner-typed dictionaries would produce an unmanageable volume of data, exceeding the scope of this research. To maintain focus and feasibility, all inner-typed dictionary keys are set to the “int‘" type in this study.

2.3 Measurement & Comparison

Each test consists of one of the seven defined actions, repeated at least one million times to amplify any overhead introduced by the type system. To reduce the impact of noise in the results, a minimum of 10 samples is collected per test, and their average is used for analysis. Every test is executed as a pair: once using dynamic typing syntax and once using static typing syntax. Each pair is then run across all five access patterns described earlier. The total number of test cases per data type varies depending on the specific behaviors supported by that type.
For every test run, several pieces of metadata are recorded alongside execution times: the data type, the test action, the access method, the collection type, and whether the test uses static or dynamic typing. This metadata allows the analysis to precisely isolate and compare specific types of test runs as needed. Additionally, an extra reference test, referred to as “baseline", is always executed for all types. This baseline run contains an identical loop structure but omits the actions under test, allowing measurement of the loop’s execution time alone for comparison purposes.
Once all data is collected, the samples for each test are averaged. The baseline value for each data type is then subtracted from its corresponding test averages to isolate the raw overhead introduced by the code under test. Each dynamic run is paired with its static counterpart, and the percentage difference between them is calculated. The results are then grouped by data type and test action, and further filtered by access pattern, producing five distinct datasets, one for each access method. Each dataset contains the percentage difference that static typing introduces for every test case across all supported types for the corresponding access pattern. These five datasets are then visualized as heatmaps, with data type on one axis and test action on the other, to illustrate how static typing affects performance in each scenario.

3 MEASUREMENT IMPLEMENTATION

Implementation[1] is carried out using the Godot Engine version 4.4.1 editor. The project is set up as a new, empty Godot project with default settings. It contains a single saved scene with one root node of type “Node", to which the testing script is attached. Three folders are created for organization: “Scripts", “Scenes", and “Results". The attached script extends the “Node" class and executes all relevant code within the “_ready()" function, ensuring that the benchmarking begins automatically when the project is run.
The tests are implemented using code generation, which allows the script to dynamically include or exclude type annotations as each test is run. This is achieved through the “GDScript" class, which enables the creation of new scripts at runtime by assigning a string containing the source code. Each test is constructed by formatting a code template using string substitution, where placeholders (denoted by curly braces) are replaced with the appropriate code segments. Each test is implemented as a newly generated function, and all functions are executed in the order they are generated. The template used for generating test functions is as follows:

```
static func {func_name}(repetitions: PackedInt32Array) ->
      Dictionary[StringName, Variant]:
  var results: PackedFloat64Array = []
  for loop_count:int in repetitions:
    var iter_arr: Array = range(loop_count)

    {outer}

    var start_time: float = Time.get_ticks_usec()
    for i: int in iter_arr:

      {inner}

    results.append((Time.get_ticks_usec() - start_time))
  return {
    &\"name\":\"{func_name}\",
    &\"type\": {type},
    &\"test_type\": {test_type},
    &\"in_collection\": {in_collection},
    &\"collection_type\": {collection_type},
    &\"dynamic\": {dynamic},
    &\"results\": results,
  }
```

As shown in the template, each test receives an array of integers and iterates over it once in an outer loop. For each element, it performs an inner loop a number of times equal to the integer value from the outer array. The execution time of this inner loop is then measured and recorded. To avoid including unrelated overhead in the results, the generation of the iterator array using “range()" is placed outside the timed section of the loop. The most important template placeholders are `{outer}` and `{inner}`, which are generated dynamically for each test, all other parts of the test template are filled in automatically based on the test configuration. These sections are constructed based on the data type, the specific test action, the access pattern, and the collection type. A dedicated function assembles each test by combining predefined code snippets and metadata specific to each type. These snippets and data include:

(1) **A default value** - a string snippet containing a value of the current type.
(2) **A loop-based variable declaration** - a string snippet of a value of the current type derived from the loop index, included in all tests to ensure consistency across types.
(3) **An operation** - a string snippet containing a valid operation for the type.
(4) **A comparison** - a string snippet containing a valid comparison for the type.

[1]The full Godot project implementation used in this research is archived on GitHub and can be accessed at: https://github.com/Dmitry-Gor/dynamic-vs-static-test-impl

(5) **A coercible type** - a string snippet containing a type into which the variable can be explicitly cast.
(6) **A method call** - a string snippet that calls a method applicable to the variable's type.
(7) **Applicable tests** - an array of integers indicating which test types are valid for this data type; only tests whose indices appear in the array are executed.

Each test constructs its `{outer}` and `{inner}` sections with test-specific logic. In most cases, the outer segment is responsible for declaring the variable under test, and it is formatted to optionally include a type annotation when the test is using static typing. One part of `{inner}` remains consistent across all tests: it always includes a variable derived from the current loop index, known as `i_based`, which is inserted as follows:

```
# ... {outer} above here

var start_time: float = Time.get_ticks_usec()
for i: int in iter_arr:
    var i_based: float = i + 0.1 # example for float

    # ... rest of {inner} below
```

This `i_based` is included in every loop because some tests rely on it to behave consistently and avoid being treated as constant expressions. Godot evaluates any expression that lacks a variable as a constant at compile time, as part of an optimization, an engine behavior discovered during implementation.
Beyond the shared elements, each test is constructed using a unique combination of the code snippets and metadata described above. The logic for generating the inner and outer sections varies depending on the test, with outer defaulting to a simple variable declaration unless otherwise noted. The approach for each test case is as follows:

Baseline Both `{inner}` and `{outer}` are left empty, containing only the aforementioned `i_based` variable declaration (2) to establish a timing baseline.
Declare The outer variable declaration is moved into `{inner}`, while `{outer}` remains empty. The variable is assigned its default value (1).
Set The `{inner}` contains an assignment where the variable is set to `i_based`.
Get The `{inner}` contains only the variable access.
Operation The `{inner}` contains an expression that performs an operation (3) involving the variable and `i_based`.
Compare The `{inner}` contains a comparison (4) between the variable and `i_based`.
Method The `{inner}` includes a method call (6) on the variable.
Type Coerce The `{inner}` contains an explicit type cast of the variable to another type (5) using `as`.

Once the `{inner}` and `{outer}` segments are constructed for a given test case, two versions of the test are generated: one using dynamic typing and one using static typing. The only difference between the two lies in the declaration of the subject variable. A typical pair of tests differ as follows:

```
## DYNAMIC typing version:
    # ... rest of the function above

    var thing = 0.05 # no type hint for dynamic test

    var start_time: float = Time.get_ticks_usec()
    for i: int in iter_arr:
        # ... {inner} here and below that uses the "thing
    " variable
```

```
## STATIC typing version:
    # ... rest of the function above

    var thing: float = 0.05 # type hint present, float
     for this example

    var start_time: float = Time.get_ticks_usec()
    for i: int in iter_arr:
        # ... {inner} here and below that uses the "thing
    " variable
```

Each new test pair is added to the complete source code being generated. The same test configuration is then repeated for each of the remaining access patterns. In these cases, the key difference is that, for container types, the value is accessed via indexing into the collection variable, rather than using the variable directly as the value itself. Continuing from the previous example, the difference between access patterns appears as follows:

```
## STATIC typing version with an inner-typed Dictionary:
    # ... rest of the function above

    var thing : Dictionary[int,float] = {} # static or
     dynamic affects if the hint appears here
    for i: int in iter_arr:
        thing[i] = 0.05 # filling the collection variable

    var start_time: float = Time.get_ticks_usec()
    for i: int in iter_arr:
        # ... {inner} here and below that uses "thing[i]"
      access in stead of "thing" directly
```

Once the full source code is assembled with all test cases in place, the tests are executed using the specified repetitions parameter, and the results are recorded. After execution completes, the resulting data file is copied and transferred to RStudio for processing.
In RStudio, the processing script begins by averaging the sample values for each test case. It then subtracts the baseline value for each data type, taken from the corresponding baseline test, from its related test averages. This step eliminates the influence of the `i_based` variable and default loop overhead specific to each type.
The adjusted averages are then split into two sets: one for dynamic tests and one for static tests, with baseline entries excluded. These two sets are joined by matching rows based on shared type, test case, and access pattern. A percentage difference in execution time is computed between each dynamic-static pair and added as a new column to the metadata.
Finally, the results are grouped by data type and test case, producing value matrices suitable for heatmap visualization. One heatmap is generated for each of the five variable access patterns, illustrating how static typing affects performance across different operations and types.

4 RESULTS

Running the tests for ten samples of 100k iterations each, then plotting the heatmaps, we get 5 figures that each correlate to an

access pattern: Fig.1 for direct access, Fig.2 and Fig.4 for Array access, and Fig.3 and Fig.5 for Dictionary access.

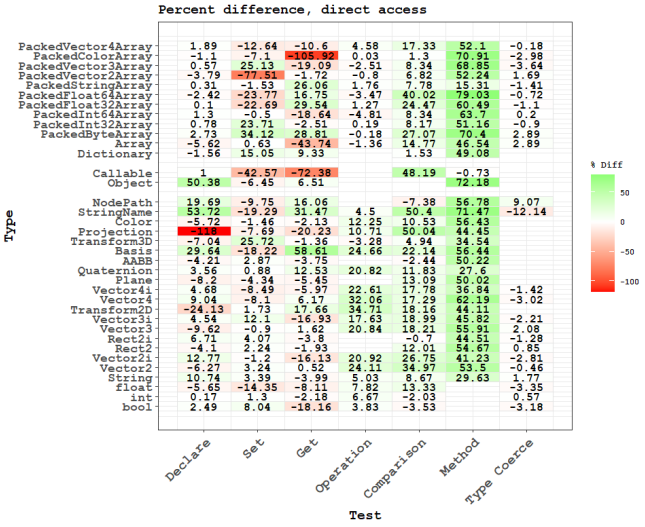

Fig. 1. Test Type vs. Type percent execution time difference (direct access)

For direct access (Fig.1), declaration, setting, and getting all show mixed results. Most types exhibit only minuscule performance differences that could be attributed to noise, but a few notable exceptions are that `Callable`, `Projection`, `PackedVector2Array`, and `PackedColorArray` all experience execution time increases above 70% from baseline in at least one of these three cases. By contrast, `Object`, `StringName`, and `Basis` each see execution time decreases exceeding 50% in one of the aforementioned cases. Type coercion, on the other hand, shows no anomalous cases outside what is plausibly noise, except for perhaps `StringName` and `NodePath`, which show both slight increases and decreases.

The remaining three test cases, operation, comparison, and method calls, are at worst unaffected by static typing, with the majority of cases benefiting substantially, ranging from approximately 10% up to 80%. Notably, operation changes the least of the three, with 11 types showing benefits, while method calls benefit the most, with a minimum 15% decrease in execution time across all but one type.

This pattern of mixed results continues for `Array` and `Dictionary` without inner typing (Fig. 2 and Fig. 3). Both access patterns exhibit a trend of inconsistent outcomes across several test categories, with certain types standing out in performance, and other categories showing more consistent benefits from static typing.

For `Array`, the most variable categories are `declare` and `set`. Noteworthy performance loss appears in the types of `Transform3D`, `Basis`, `bool`, `PackedInt32Array`, and `PackedColorArray`, all showing over a 10% increase in execution time during setting. `Callable` and `PackedFloat32Array` meanwhile see performance gains above 10% in the same test. In the `declare` test, `PackedInt32Array` and `Transform2D` both show more than 10% improvements. The remaining test cases, by contrast, tend to benefit consistently from static typing, with performance improvements reaching up to 45% in some cases, and most gains falling in the 10-30% range.

Similarly, `Dictionary` access shows significant variance in the `declare` and `set` tests. For example, `Basis` sees a 15% increase in execution time, while `PackedFloat64Array` and `Quaternion` each show reductions exceeding 20%. Unlike direct or array access, however, the purely beneficial categories are less distinct. The most consistent gains appear in the `method` test, where all but one type benefit. The other test cases, `set`, `get`, `operation`, `comparison`, and `coercion`, also mostly show improvements, though these are generally more modest, typically peaking around 16%.

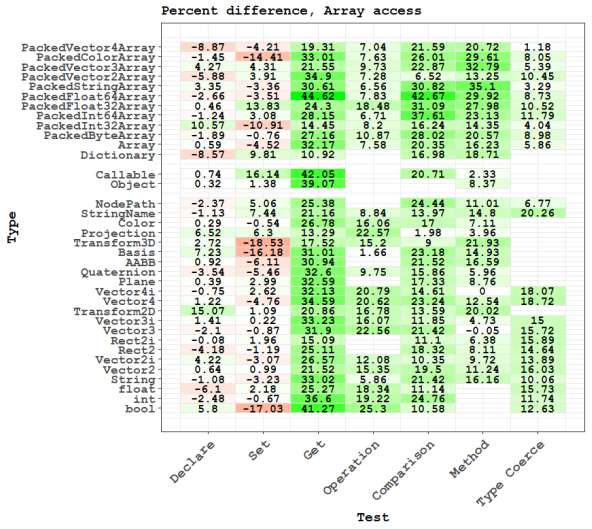

Fig. 2. Test Type vs. Type percent execution time difference (array access)

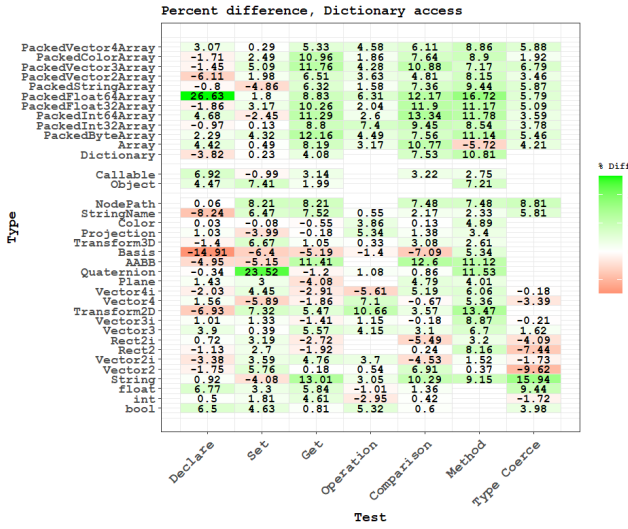

Fig. 3. Test Type vs. Type percent execution time difference (dictionary access)

For `Array` and `Dictionary` access with defined inner types (Fig. 4 and Fig. 5), the results are significantly more definitive. Both access

patterns exhibit substantial performance degradation during `declare`, with most cases requiring roughly twice as long to declare as their dynamic counterparts, and some `Dictionary` cases reaching up to three times the execution time. In the `set` test, results show a wide variance, with both gains and losses observed across types. However, all remaining test cases consistently benefit from static typing. Performance improvements for these cases typically range from 20-50% for `Array`, and 10-30% for `Dictionary`.

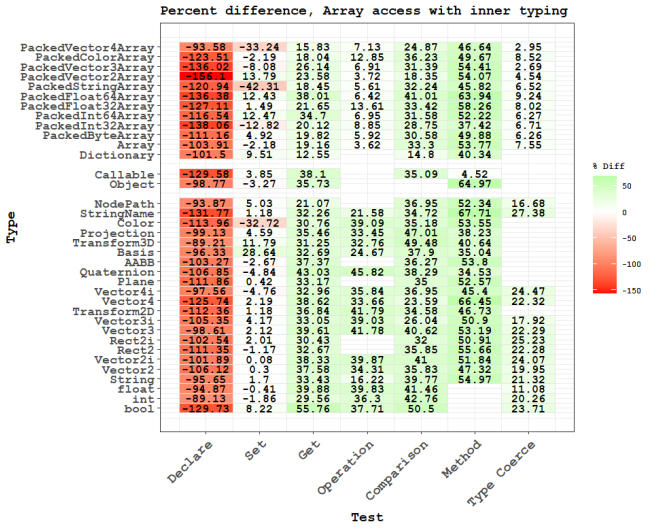

Fig. 4. Test Type vs. Type percent execution time difference (subtyped array access)

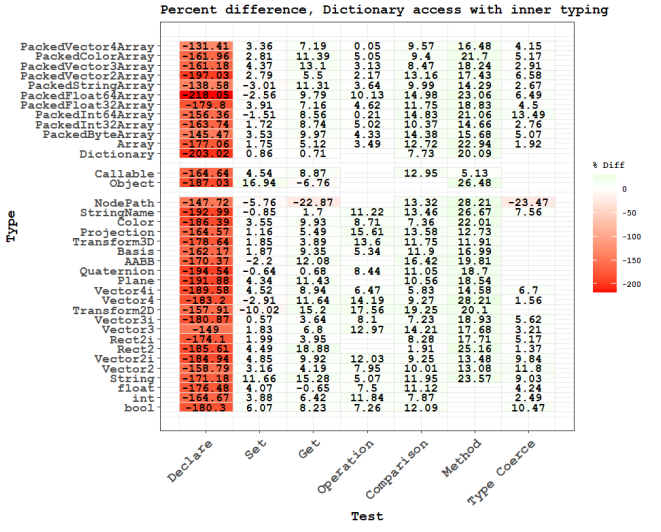

Fig. 5. Test Type vs. Type percent execution time difference (subtyped dictionary access)

5 DISCUSSION

A plausible explanation for the mixed results, particularly in the `declare` and `set` tests, is that the interpreter may lack sufficient low-level optimizations to fully capitalize on the presence of static type annotations. In many cases, especially for really specific operations, the cost of executing the action may not differ meaningfully between dynamic and static contexts, as the interpreter may still follow the same execution path regardless of type hints. Additionally, instances where performance slightly degrades, most notably in direct access or non-inner-typed `Array` and `Dictionary` use, could be due to implicit type coercion. Some built-in types may require conversion from input literals into internal representations before performing an action, and increased specificity in a statically typed context might trigger extra conversion steps, introducing marginal overhead. The most severe performance regressions, observed during declaration of inner-typed collections, may stem from how the GDScript engine handles type-specific memory allocation. When declaring a container such as `Array[int]`, the engine could be converting the literal (e.g., `[]`) into a form explicitly structured to store `int` values, instead of the default generic `Variant`, introducing a cost that dynamic declarations avoid.

These findings highlight an important consideration for developers working with GDScript: the performance implications of static typing are not uniformly positive, and in certain edge cases, particularly with inner-typed collections or coercion-prone types, static typing may introduce unexpected overhead. This underlines the need for caution and attention to anomalies when optimizing performance-critical code. However, when viewed in the broader context, the results strongly support the use of static typing. The consistent and often substantial performance gains observed in common operations such as method calls, comparisons, and arithmetic suggest that, in most practical scenarios, static typing offers clear advantages. As such, developers are encouraged to adopt static typing where possible, especially in parts of the codebase that are performance-sensitive or rely heavily on these frequently used operations.

It is also important to recognize that the usefulness of static typing in GDScript extends well beyond performance considerations. Type annotations help catch mismatches and misuse at the time of writing, significantly reducing the likelihood of runtime errors caused by unexpected values or incorrect assumptions about data types. This is particularly valuable in large or long-term projects, where code is maintained by multiple developers and system complexity increases over time. By enforcing clearer interfaces and enabling better tooling support, static typing improves maintainability and readability, key factors in reducing technical debt. These benefits, combined with the observed performance improvements in many typical use cases, make a strong case for the consistent use of static typing in GDScript development.

5.1 Threats to Validity

One significant limitation of this study lies in the computational constraints imposed by the scale of the test suite. Due to the high number of test combinations across types, access methods, and test actions, each individual test run is limited to 100,000 iterations per

sample, with only 10 samples per configuration. While this is sufficient to expose general trends, it introduces a considerable amount of statistical noise. Even after averaging across samples, the results can still reflect fluctuations that obscure the true impact of static typing, particularly for operations where the execution time differences are small.

Another threat to validity arises from the simplicity of the analysis approach. This research relies on percent increases and decreases relative to a baseline, followed by visual inspection through heatmaps. While effective at highlighting broad trends, this method lacks the statistical rigor needed to detect or account for outliers, variance, or potential correlations between type characteristics and performance outcomes. As a result, subtle interactions or anomalies may be either exaggerated or completely missed, which can compromise the strength of the conclusions drawn.

Finally, the absence of prior studies or published performance analyses specifically focused on static versus dynamic typing in GDScript presents a significant limitation. The broader topic of type system performance in dynamic languages is itself underexplored, and virtually no research exists in the context of the Godot engine. This lack of reference material makes it difficult to benchmark the methodology or validate the results against established findings, limiting the ability to situate this work within a broader academic or practical context.

Compounding this issue is the limited understanding of the internal behavior of Godot's interpreter on the part of the researcher. While the interpreter is open-source and theoretically accessible for inspection, gaining the depth of insight required to trace and explain the performance impact of type annotations would demand extensive familiarity with the engine's C++ implementation. Given the time constraints and scope of this project, such an investigation would be infeasible for a single researcher to undertake in parallel with the testing and analysis, leaving many of the observed performance behaviors largely up to speculation.

5.2 Further Work

There are several promising directions for future research that could expand on the findings and address the limitations of this study. Notably, while this work includes `PackedInt32Array`, `PackedFloat64Array`, and other packed-array types in the overall benchmarks, their unique memory handling and internal representation merit a dedicated performance investigation. Packed arrays are designed specifically for efficiency and low-level data access in Godot, yet the impact of static typing on their behavior remains largely unexplored beyond surface-level trends observed here. Future studies could isolate these types and test them across a broader set of operations to determine whether their performance profile is distinct from other built-in types.

Additionally, the current study simplifies the testing of typed dictionaries by using `int` as the static key type across all configurations. A more exhaustive investigation into the performance impact of varying key types, especially those more complex or expensive to compare, like `StringName`, `Object`, or `Vector2`, would provide deeper insights into how static typing interacts with dictionary internals. Similarly, the analysis treats each access pattern independently, but future research could explore the relationships between patterns, identifying whether static typing benefits or penalizes certain sequences of access (e.g., retrieving a value immediately after setting it) differently depending on context.

Finally, replicating this study under more favorable conditions could yield more precise and generalizable results. The use of a more powerful machine, larger iteration counts, and more sophisticated statistical methods, such as hypothesis testing, variance analysis, or confidence interval modeling, would reduce noise and increase the reliability of findings. In parallel, a deeper dive into Godot's interpreter and its optimization strategies would allow future researchers to better explain anomalous or counterintuitive performance results. With Godot's interpreter being open-source, such an investigation is feasible, though time-intensive. Understanding its memory allocation patterns, type resolution mechanisms, and JIT behaviors (if any) would be essential to uncovering the root causes of performance variability, ultimately turning current speculation into well-supported conclusions.

6 CONCLUSIONS

This study set out to explore the execution time differences between static and dynamic typing in GDScript by benchmarking nearly all built-in data types across common operations and access patterns. Through automated testing and analysis, the research revealed several key insights into the performance implications of using static typing within the GDScript.

In response to **RQ1**, the performance impact of static typing varies by type and operation; while the results are mixed in some areas, most common use cases, particularly method calls, comparisons, and arithmetic operations, consistently benefit from static typing. For **RQ2**, differences between access patterns are notable: non-direct patterns such as array and dictionary access generally show greater performance gains with static typing, though inner-typed declarations in these patterns suffer from significant slowdowns. As for **RQ3**, the tests indicate that method calls, comparisons, and operations benefit the most from static typing across all patterns, while actions like variable declaration and assignment remain inconsistent, especially when used with inner-typed containers.

Ultimately, despite some anomalies and edge cases, the findings support the practical use of static typing in GDScript, not only for its runtime benefits in most scenarios but also for the additional advantages in code clarity and error prevention.

REFERENCES

[1] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. 303–312. https://doi.org/10.1109/COMPSAC.2013.55

[2] Gem Dot, Alejandro Martínez, and Antonio González. 2015. Analysis and Optimization of Engines for Dynamically Typed Languages. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 41–48. https://doi.org/10.1109/SBAC-PAD.2015.20

[3] Andreas Gocht-Zech, Alexander Grund, and Robert Schöne. 2021. Controlling the Runtime Overhead of Python Monitoring with Selective Instrumentation. In *2021 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. 17–25. https://doi.org/10.1109/ProTools54808.2021.00008

[4] Ariel Manzur Juan Linietsky and Godot contributors. 2025. GDScript reference. https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_

basics.html. Documentation for the engine itself, version at time of writing is 4.4.1, future versions may be subject to change..

[5] Ariel Manzur Juan Linietsky and Godot contributors. 2025. Static typing in GDScript. https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/static_typing.html. Documentation for the engine itself, version at time of writing is 4.4.1, future versions may be subject to change..

[6] Dongdong Lu, Jie Wu, Yongxiang Sheng, Peng Liu, and Mengmeng Yang. 2020. Analysis of the popularity of programming languages in open source software communities. In *2020 International Conference on Big Data and Social Sciences (ICBDSS)*. 111–114. https://doi.org/10.1109/ICBDSS51270.2020.00033

[7] George Marques. 2020. GDScript progress report: Typed instructions. https://godotengine.org/article/gdscript-progress-report-typed-instructions/. Official article from the Godot engine development team, but quite old since this pre-dates 4.0 by almost 2.5 years..

[8] Jeremy Siek. 2014. What is Gradual Typing. https://jsiek.github.io/home/WhatIsGradualTyping.html. Personal github page, link may change, date of access is 30-Apr-2025..