Development of a Domain Specific IDE and JavaScript Comparison Engine for Tomorrow

Gregory Mitten
University of Sussex

Tomorrow is a Copenhagen based start-up attempting to reduce the effects of climate change with their new mobile application, a prerequisite for the application to see widespread use is contributions from members of the open-source community. This project attempts to aid their efforts by easing development for volunteer engineers, in the hope of increasing the number of "integrations" developed. The project provides a domain-specific IDE eliminating configuration and providing a wealth of information to these volunteer engineers. Additionally, as all integrations share a common task, a JavaScript comparison engine is provided to identify similarities in the integration code. It is hoped that this can both inform volunteer engineers about how to overcome certain technical problems as well as allowing Tomorrow engineers to extract often repeated code, into easy to use templates where applicable.

Introduction

Problem

Context. Tomorrow's new mobile application also named tomorrow, is soon to enter its beta. The application aims to quantify the carbon impact of everyday actions, allowing users to comprehensively track their impact on the environment. How can tomorrow hope to handle all these vastly heterogeneous interaction modalities the user has transitively with the environment? Tomorrow is relying on open source community to build what are known as "integrations".

An integration is a suite of functions that connect to and retrieve data from a service, then parse it to match a schema. This integration can then be used by tomorrow to display the impact of said action(s) both graphically and textually. It should be evident that the development of integrations are a necessary requisite step for the application to see wide use. Hence it is pivotal an integration should be convenient and trivial to develop, as not to deter any potential volunteer engineers.

Definition. Currently, in order for volunteer engineers to test their integrations, they must run a server locally which provides limited feedback and capabilities. This project's aim is to reduce the initial discomfort, confusion and workload often present when beginning development on a system. The desired effect of this is to increase both the number of volunteer engineers and the volunteer engineer retention rate.

Proposed Solution

This project focuses on a technical route of achieving this outcome, attempting to automate the "toil" (Beyer et al, 2016) of integration development. Transpiling, hot loading, executing, rendering results, handling environment variables et cetera. The idea is to eliminate the rigmarole often involved in project configuration, providing a pre-configured IDE with live servers ready to execute volunteer engineer code.

Additionally, there are a finite number of discrete ways to develop the interface functions (connect, collect and disconnect). Meaning there will likely be a great deal of similarity between integrations. Necessarily the greater the integration count the greater the chance two of which are similar, and with Tomorrow anticipating over 100 integrations in an optimistic scenario we would expect a number of them to be very similar in places. Because of this, a code similarity engine is planned to allow Tomorrow and volunteer engineers to identify and learn from similar integrations. The volunteer engineers could use this to learn and fix their integrations, whereas the Tomorrow engineers may be able to, over time, refine their interface functions or add templates to further trivialise integration development.

Relevance

As much of modern software has moved from the desktop to the web, some are left questioning why the desktop IDE is still so widely used by developers, a group who by all means should be ahead of the curve. There are, of course, concerns that web applications cannot hope to provide the computational power necessary but attaching a cloud server could overcome this problem. It hoped throughout the development of this project provides a greater understanding of the limitations and boons of web-based IDE systems.

Finally, although there is a much literature on code plagiarism detection and the modern IDE often provides information about when the code is duplicated. There seems to be a lack of focus on identifying similar but not equivalent

code. It is often joked about that the contemporary software developer should concern themselves more with the literacy of sites like StackOverflow than they should with code. The reason this statement holds so much truth is software developers infrequently solve a unique problem. Although specific domain details make one problem superficially different, these are often easily identified by developers. It is hoped that this similarity identifying technology touches on an area largely unexploited, identifying similar code. In the future, this could be the basis of a JavaScript template engine facilitating the expedition of feature development. Many large technology companies use such technology internally but as of yet, there is surprisingly not widespread use of these techniques in other spheres.

In this scenario instead of thousands of developers coding solutions to many different problems, we instead see a small number of volunteer engineers all solving essentially the same problems with different solutions. Due to the hyperspecificity provided by the tomorrow integration development microcosm, there is a significant problem relaxation, it is hoped enough so that a similarity identification tool will prove useful.

Objectives

The project focuses on two discrete areas, of which is developing an IDE Web Application that eliminates all toil for volunteer engineers and can be used throughout tomorrow's lifecycle, the Tomorrow IDE (TIDE).

The other domain is developing a performant code similarity tool. The JavaScript Comparison Engine (JeSSE), JavaScript is the language all integrations must be developed in.

Background

Code Similarity

There is a wealth of literature in code similarity with much of it focussing on detecting plagiarism, although the intent behind detecting code similarities may be vastly different from TIDE pragmatically there is little difference in technique. Often plagiarism detection is concerned with being fooled or deceived, JeSSE does not share this concern as integrations are not assessed, hence worries of deliberate obfuscation are misplaced. Furthermore, the comparison scope is large for many of these tools often spanning university-wide and/or including previous years, whereas JeSSE will be used in a much more confined space. Regardless code plagiarism literature is regarded as extremely relevant.

Duričc & Gašević (2013) enumerate several of the potential false flags involving lexical and structural code similarity, the relevant items for each are enumerated below.

Lexical

- Modification of source code formatting
- Renaming of identifiers
- Addition, modification or deletion of modifiers
- Modification of constant values

Structural

- Changing the order of statements within code blocks
- Reordering of code blocks
- Modification of control structures
- Method inlining and method refactoring

Duričc & Gašević follow a token-based analysis and use RKR-GST - Efficient randomized pattern-matching algorithms (Karp & Rabin, 1987). This searches for longest contiguous match greater than a minimum value and then marks it as to avoid duplication. Additionally, they make use of The Winnowing algorithm. This algorithm strips white spaces and forms n-grams from their hashes, these are then checked for a certain amount of overlap, for each value the hash with the lowest overlap is selected as representative. This reduces the number of hash values facilitating for greater efficiency in substring detection. They combine the results of these algorithms to retrieve a final similarity value.

Lancaster & Culwin (2004) complete a review of many source code plagiarism tools, they delineate plagiarism tools into two categories. Structure metric systems where every submission is extracted down to a vector of representative numbers and attribute counting systems, where certain attributes (such as identical values) are counted. Verco & Wise (1996) found attribute counting systems to have greater accuracy when the programs were very similar however they were unable to detect partial plagiarism and overall the performance metrics for structural metric systems were much greater.

Although it appears that many plagiarism identifiers are based on Token comparison there is an alternate approach in newer literature which involves a strategy of AST comparison. Feng et al (2013) suggest that token-based comparison is unable to effectively detect if the order of tokens has been rearranged, this problem does not occur in AST based comparison. Feng et al use an algorithm called AST-CC to hash nodes and compare the hash of each node with each other node with the same number of child nodes.Zhao et al 2015 also opt to use an AST-based approach, in order to compare these nodes, they first transform the AST into a group into a linear list and then into a group of subtrees based on the number of child nodes. Once this is complete they also use the AST-CC algorithm over it for comparison. Their AST-CC hash function includes only the type and children of each node omitting other comparison considerations but certainly a more efficient solution for scale.

It has been found so far that comparison techniques unanimously use hashing for the comparison of nodes, however, it is believed, that this hashing although providing greater for comparison efficiency, ultimately obfuscates the weighting

of "difference types". Should the same type of element edited greatly provide a greater similarity than if it was removed entirely? This consideration and many others can be accounted for by using seperate penalisation values for each difference type, rather than tweaking a hashing algorithm. Tuning the algorithms weights to provide more accurate similarity appears to be a powerful tool that needs to be considered. It should also be noted all of these previous tools were developed as plagiarism tools, as mentioned encountering slightly different problems.

Similarity Evaluation

In order to compare the effectiveness of plagiarism detection systems many researchers employ other well-established systems, Duri\‘ć \& Gašević use Measure Of Software Similarity (MOSS) (Aiken, 2018). Research from Engels et al (2007) found MOSS to be the most effective plagiarism system when utilised neural networks to assess the performance of 12 different engines, MOSS also invented the aforementioned Winnowing algorithm (Aiken et al, 2003) which is employed in MOSS. However, in order for MOSS's values to have any meaning, the project must first be analysed to determine if it's inner workings apply to this scenario.

MOSS is white space insensitive, suppresses noise by ignoring trivially short k-grams and position-independent. MOSS does not opt to remove identifiers but instead replaces all identifiers with the character “V”, but MOSS is sensitive to comments. Additionally MOSS uses asymmetric comparison, i.e. compare(a,b) does not necessarily have to yield the same result as compare(b,a).

The Issue of Similarity. Code similarity is not objective, although it may be asserted that two pieces of identical code are very similar, how many changes can one make before this similarity is lost? Plagiarism scarcely addresses this issue and instead defers to alternate plagiarism systems to provide answers, however, if other plagiarism systems are somewhat unsuccessful using as a baseline is counterproductive. Krinke (2002) discusses a solution to this issue, in order to tune his code similarity engine he used a list of code excerpts with code duplication and then manually checked whether they were duplicates. Although this is somewhat hazier in the example of similarity manual checking must be used.

Web IDEs

The idea of an online IDE is far from a new one, many companies already offer this service: codesandbox.io, codepen and JSFiddle to name a few. However, typically these environments are for prototyping and experimenting rather than developing complete projects. Much of the literature involving remote code execution has a heavy focus on how secure systems are, this is not so much a concern with TIDE as there no sensitive data accessible or even associated with the system. However, previous solutions will be analysed to derive a set of best practices and technology considerations.

Timo et al (2011) preach the benefits of an online IDE, namely the developer need not worry about configuring or updating the environment. They continue to suggest the use of a development server in which the code can be stored and ran on. In this case, they opt to use a Java Servlet to fulfil this role.

A solution that aims to provide a comprehensive IDE experience in the browser is codeanywhere (2019), the system works by providing each user with their own containerised virtual machine with dedicated memory and storage. This VM can be fully customised and set up with many development stacks and offering over 120 languages. As impressive as this solution may be spinning up a virtual machine for each volunteer developer this appears to be excessive for TIDE's scenario.

Malan (2013) documents his approach to developing a web IDE for students attending programming modules at Harvard University, known as CS50. CS50 is a much less distant scenario than a fully-fledged IDE replacement and has many features akin to the requirements of TIDE. However, it does aim to provide many languages rather than just JavaScript. CS50 is built with NodeJS as its event-driven architecture is well-suited to handle asynchronous demands. Many developers may be concerned that a web IDE would incur non-trivial latency issues, enough so to make it worth using an offline alternative. Malan addressed this concern with the use of WebSockets for continuous interaction between server and client.

Goldman et al (2011) developed a collaborative web IDE, they opted to import their text-based portion of the IDE in its entirety rather than engineering their own, this is very much applicable to TIDE. Building a web input for code complete with autocomplete and syntax highlighting would take up a great deal of development time and hence one of the alternatives they offer may be used.

System Decisions

Frontend Technology

Choosing the frontend technology was fairly unimportant, the primary consideration was maintainability for Tomorrow engineers, its capacity to support WebSockets and a library that functions as a code editor. JavaScript is the most widely used frontend language but a number of JavaScript frameworks met these criteria. As Tomorrow develop their application in ReactNative it logically followed to use ReactJS for maintainability purposes.

Editor. Considering the number of online code playgrounds it is somewhat surprising how few are available to be integrated into another project, many of these systems do not provide any autonomy into another environment. However,

Microsoft has open-sourced their editing engine for VSCode, Monaco Editor, this has been imported and integrated into the frontend.

Backend Technology

One of the most important considerations for the backend was how it handles the execution of JavaScript code, all languages except JavaScript itself must execute JavaScript with an external tool such as Google's V8 interpreter (used in their browser Google Chrome) alternatively code can be executed using a node in the command line. It was deemed sensible to skip the middle man especially as node's non-blocking architecture combines with WebSocket best practices fluidly. The WebSocket framework chosen was socket.io as this is used in tomorrow, for ease of maintainability.

Comparison Specification

Taking into consideration the literature review the final specification of the code comparison is as follows. The comparison engine ignores the following aspects of code.

- Location Data
  - It is regarded as unlikely that exactly the same operations take place in exactly the same place, this does not mean that these items are code should not be considered similar. Not every operation is hyper dependent on its placement.
  - Location independence in this circumstance includes lines positions relative to each other and row and column numbers.
- Identifier names
  - This is common in plagiarism detection as to deceitful and artificially differentiate the copied code, however this is still very much relevant to volunteer engineers.
  - Although there may be some similarities in the naming of functions many identical variables and functions may be given entirely different names and should not be penalised in similarity score due to this.
- Comments
  - Comments have no effect on execution
  - Comments included in the template code to explain each method are likely to lead to false positives.
- Literal values
  - Fetching data from different APIS will always involve different access strings, penalising similarity values here in nonsensical
  - In other circumstances volunteer engineers may receive their data in a different measurement and need to transform it will involve different numbers but are essentially the same and still useful to learn from
  - Despite the fact that JavaScript is a dynamically typed language literals preserve their type information, this is the only information that should be stored about a value
- Logging
  - Logging should have no effect on execution

These items are to be ignored indiscriminate of circumstance as to never impact similarity calculation.

JeSSE will not attempt to test if two excerpts are strictly semantically equivalent. In this context, strict semantic equivalence means that code excerpt a and b perform the same operations in the same order. Instead, semantic similarity is used, JeSSE aims to determine if two code excerpts are similar in their contents rather than entirely identical, as this will very seldom be the case. One of the primary reasons for using this loose definition is interface functions necessarily interact with an external, potentially proprietary system to retrieve their data. It is naive to assume that because two API endpoints appear to be similar they perform similar operations. All instructions that can potentially be called inside a method must be included, this is to say the nodes are collected regardless of conditional control structures. The engine must additionally support function inlining, elaborated upon later.

Approach

AST-based comparison was judged to be more suitable for this criteria as Feng et al mentioned location independence is ineffective in token-based systems. A structural based approach is also employed, the literature suggests this is more effective than its attribute-counting based counterpart. As this metric is not used internally to generate a binary condition like many of these tools the metric generated should be understandable by a developer with no background in code similarity.

JeSSE Implementation

Preprocessing

The code is first parsed into an AST, many JavaScript parsers are available however as babel is a de facto standard for transpiling ES2015+ code into the "ubiquitously" executable deprecated version (ES2015). If the project aims to support all JavaScript the babel parser should be used, as babel is actively updated for new JavaScript features, somewhat future-proofing the system. Once parsed all root-level

functions are extracted from the AST, in a well-formed integration this will include the interface functions and any member functions they call. All function nodes are preprocessed individually.

Tree Flattening. Each function is then recursively "flattened" from a tree into a list of its nodes, this is for ease of with the transformation of each tree into a structure in which they can easily be compared. This list includes each node and all their children but once the children are extracted from a node and added to the list they are removed from the node itself. Tree flattening takes into account all control structures, inner function to any degree of nesting.

Flattening method takes two parameters, the top level nodes in each function body, and a list of all other member functions this is discussed in function inlining. The nodes types:

`IfStatement`, `WhileStatement`, `ForStatement`, `DoWhileStatement`, `FunctionDeclaration`

are referred to as "deep nodes" as they typically have children.

Deep nodes almost unanimously require leaf node extraction. The algorithm traverses through each node in the tree if children are present the parent node is added to the list and then its children are processed. As JavaScript uses pass by reference to accomplish this the node's children are deep cloned, once this cloning has taken place the child nodes are then dereferenced in the parent to be cleared by JavaScript's automatic memory management. The parent node is then added to the list of flat nodes, the child nodes are recursively passed into the flatten algorithm and the result is concatenated with the list.

Other statements, such as lambda expressions, may also be deep nodes but they have a different structure. These nodes must pass a gauntlet of cascading conditions as there is so much diversity in the node structure there is no one simple condition to determine whether children are present. For example, if a node has a consequent, there is no certainty that consequent has a body and even if a consequent's body is present it may be empty. When a node succeeds in passing these conditions their children are extracted in the same fashion as deep nodes, but simply referring to a different route to the body.

In Listing 1 & 2 exactly the same instructions are called in a semantically equivalent manner but they would yield vastly different similarity results if we were to ignore function inlining.

```
1 function connect() {
2   return getConnection()
3 }
4
5 function getConnection() {
6   return http.get("https://api.com/get-
        session/username").execute();
7 }
```

Listing 1: Function returning call expression

```
1 function connect() {
2   return http.get("https://api.com/get-
        session/username").execute();
3 }
```

Listing 2: Function returning contents of previous call expression

Once function inlining is performed the resulting code bears a much greater resemblance, seen in Listing 3. JeSSE does not execute the code and so even though line 3 is unreachable it is still processed.

```
1 function connect() {
2   return getConnection()
3   return http.get("https://api.com/get-
        session/username").execute();
4 }
```

Listing 3: Function returning contents of previous call expression

Function inlining is limited to member functions as to not exponentially expand the number of nodes being evaluated. In the other case where every method is processed, the nature of programming abstraction this would almost certainly lead to an enormous number of nodes. This is not considered representative of the integration code, but rather every single operation that takes place within it.

Each expression is analysed and if it is both (1) a call expression (2) and its method signature is present in the list of functions passed into the flattening method. The method name is then as a key to access the map of all member functions, it will then extract and flatten recursively the body of the called function. The result is the concatenation of the current list of nodes with the expression followed by the contents of the member function called. A function's body can only be added to the list once per interface function, in the same manner, that the contents of a for loop are once rather than the number of times the loop iterates.

As the list is not a list of every node called, rather a unique list of each node that can be called, a function's body can only be added to the list once per interface function. Consistent with the manner that the contents of a for loop is not added the number of times the loop iterates (if deterministic).

Eliminate Irrelevant Nodes. The next phase of preprocessing is to eliminate the nodes that have been determined as irrelevant for comparison, which here only includes logging nodes. JeSSE has several specific logging method signatures hardcoded, the flattened node list is scanned and for call expressions matching these signatures dereferencing any matches.

Eliminate Irrelevant Node Details. Finally although some similarity tools replace information they do not want accounted for with a consistent token and others opt to simply ignore these components in their comparisons. JeSSE instead eliminates this information from the nodes entirely by dereferencing it, meaning all data in the remaining structure is to be considered. This must occur last as previous phases rely on this information.

This completes pre-processing, Figure 1 & 2 are examples of pre-preprocessing and post-preprocessing respectively.

Figure 1. Parsed AST

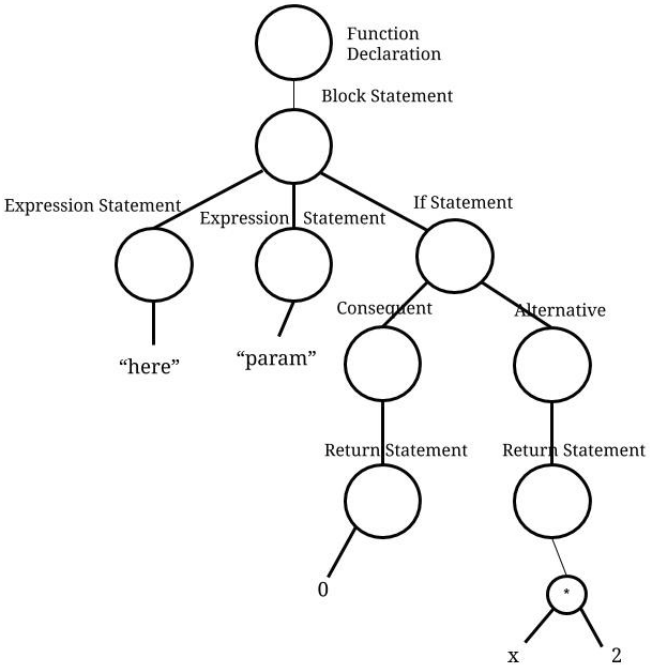
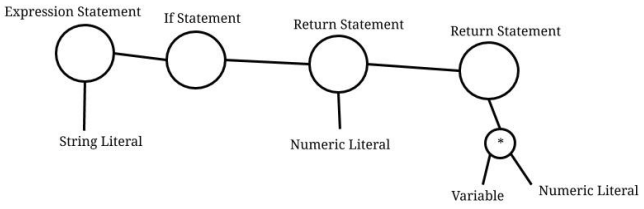


Figure 2. The AST after preprocessing



Comparing Function Bodies

In order to compare function bodies, each node must find the node most similar to itself in the alternate list, node matches are exclusive, meaning each node can match with a maximum of one other node; this is true bilaterally. Once a list of these matches has been compiled the similarity of each can be calculated. The similarity value between any two nodes is calculated using a map of chosen values for each difference type, found in table 1. The calibration of these values will be discussed later.

Table 1

Addition	A value is present in the compare node not found in the original node
Deletion	A value is not present in the compare node that is present in the original node
Edit	The field is present in both but different values are found
Array	An array is present in both places but a different number of values are found

Having control of how the system treats each individual difference in terms of the capacity to alter the punishment value is extremely valuable, although node hashing offers greater efficiency it is found lacking in this domain (without the creation of a specific hashing algorithm). On comparison of each node, fields are compared and the corresponding punishment values are applied, the punishment values of each node are summed. Note that although the comparison is similar to each base node can match with a maximum of 1 compare node, but if there are fewer nodes on one side than the other unilateral similarity calculator may say the functions are very similar despite the fact one is much smaller than the other.

Finding each node's best match is far less trivial than it seems. If this is done naively the match of one node will prevent all future nodes from matching with that node regardless of whether their similarity is greater than the original match, this only occurs due to the exclusivity of the problem. One solution to this would be to compare these values and if the new node shows greater similarity then return and repeat the process with the initial node. This results in unnecessary processing. The preferred solution that has been chosen is an algorithm developed named "exclusive match".

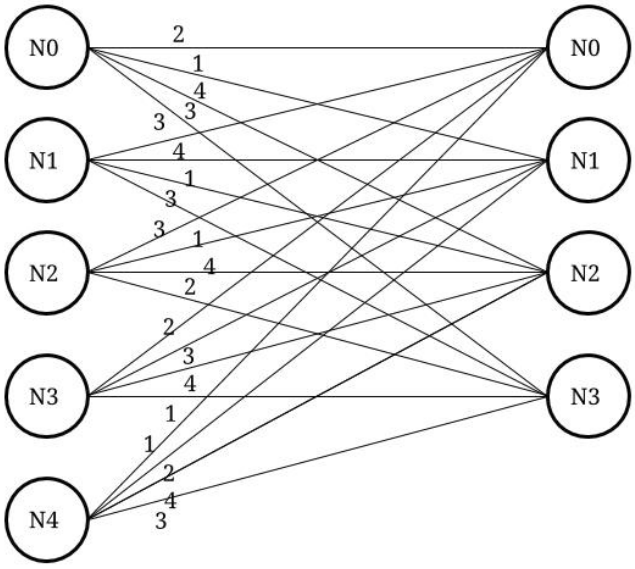
Exclusive match builds a hierarchy of matches ordered by similarity as the nodes are compared. Each difference is assigned its corresponding value and these are summed to form the similarity value, due to the nature of the parser two ASTs can contain vastly different keys despite not being extremely different in nature on occasion, so an upper boundary has been set as a maximum difference punishment value. Once all hierarchies have been built, the uniqueness of the first item in each hierarchy is then tested for duplicates. If a duplicate is identified a loop is entered, the loop will continually pop the index from the hierarchy with the lower similarity value. This continues until the first index in the current node's hierarchy in the hierarchy it is either: Unique or has the highest similarity value of all it's duplicates. Figure 3 represents the first phase where all hierarchies are built, Figure 4 represents

Upon completion, populated hierarchies are summed and stored including their similarity values. Empty hierarchies are not simply ignored, these items are identified and their count is used as a modifier to multiple S (The similarity value) by (where a lower S represents greater similarity).

Algorithm 1 Exclusive Match

```
hierarchies ← []
for each node n in nodes do
    hierarchies.add(sort(match(node, compareNodes))
for each hierarchy h in hierarchies do
    while compareResult ≠ 0 do
        if compareResult = 1 then
            hierarchies[h].pop()
        else if compareResult =-1 then
            hierarchies[compareResult.index].pop()
        compareResult ← compare(h, hierarchies)
procedure MATCH(node, compareNodes)
    hierarchy ← []
    for each node cn in compareNodes do
        hierarchy.add(similarity(node, cn.AST), cn.idx)
```

Figure 3. The hierarchies built for each node

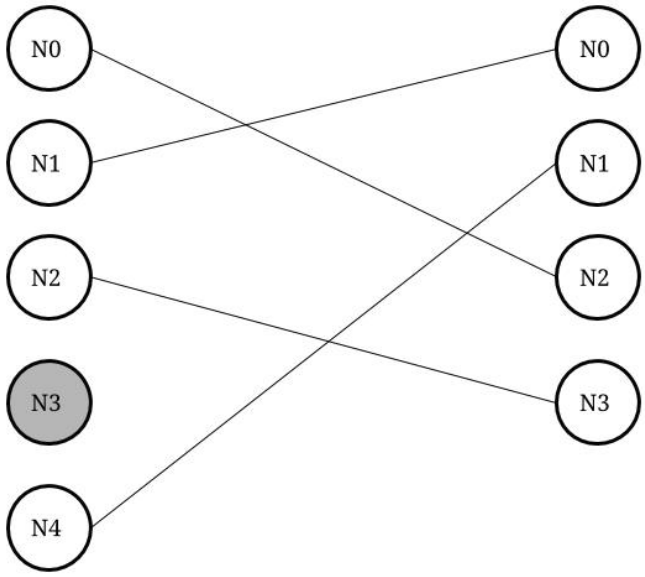


To make S relative to its size of the nodes that generated it the total S of each function is divided by the number of keys present in all of its nodes. Key-count is always directly proportionate to the comparison as deep-diff works on the basis of comparing each individual key in the node. What this does mean however is that more complicated nodes are weighted heavier in the generation of S but this was seen as appropriate.

Once a score is generated for a function in the form `compare(a, b)` only one side of the comparison has taken place and the system is unilateral. This result is weighted as 50% and the other half is derived from an inverse comparison for `compare(b, a)`.

The purpose of this transformation is so the engineers perceive the value as a percentage, this should intuitive to the engineers without much thought. Although it should be noted that a TS of 75 is not 75% similar, it was judged this is representative enough that engineers will get the general idea of the system. JeSSE has been tweaked so an S greater than 1 is seen to be very dissimilar and will be displayed to volunteer and tomorrow engineers alike as 0. If these values are displayed in such a way why even give multiple pieces of code the possibility of having a difference value less than 0? This is because as the most similar integration must still be selected i.e. a dissimilar function of similar length is still considered more similar than the contrary. Providing a TS to the engineer of -56 was judged to be confusing and break the percentage-based illusion, after all how could the code be -56% similar. TS for each method is displayed alongside a visual indicator of similarity client-side and a copy of the most similar code to allow the developer to inspect it to look for places they could complete their task differently.

Figure 4. The hierarchies with pruned duplicates



Each method of each integration undergoes the comparison process with the volunteer engineer's code, matching each function to the corresponding interface function. After every function has been assessed the code is imported from the most similar integration for each interface function, to prevent memory leaks these modules must be deleted. First emitted to the client-side to reduce further waiting time and then the modules are removed from the cache.

TIDE Implementation

Code Execution

There are many ways to dynamically execute code in JavaScript, one of which is the notorious `eval()`, many of the reasons eval is widely disliked is due to its security im-

plications, not particularly concerning to TIDE. However, it also runs into many efficiency concerns. Meawad et al (2012) posited that `eval()` is, in almost every scenario, replaceable by a superior alternative, even developing a tool that automatically replaced 97% of eval usage. There are many alternatives to `eval`, some are assessed below.

- JavaScript provides a text parameter for its Function class allowing the passing of source code directly into the object, i.e. `new Function("console.log('test')").apply()` prints `test` in the console. This appears to be a thinly veiled eval it is not quite, the code input here is parsed into a JavaScript Function whereas eval runs the input expression with whatever scope it is called in. The issue with passing code in only functions can be passed in whereas the volunteer developers code entire modules with constants and imports which will not have the correct scope inside a wrapper function.
- Another strategy is by Document Object Model modification, this would involve appending the script to the HTML of the current page their viewing and then running the methods from the JavaScript initially provided, besides the fact this solution is inelegant and would likely be very error-prone there are additional reasons why this should not be used. The first of which is it would require their browser to support the JavaScript they are writing, although this may seem like a reasonable request it is not unlikely some experimental feature is used for some element of their integration that is not supported by many old versions of browsers. Finally, there are numerous modules which may be imported from the Tomorrow project which would be programmatically awkward to selectively provide based on the DOM.
- Although node.js is a JavaScript framework perhaps it is better practice to write the code to a file stored on the backend and then initiate JavaScript execution via the command line using a tool such as Google's V8 JavaScript interpreter. However, retrieving the console output is regarded to be inconsistent and cannot possibly be the best solution.
- The final and perhaps only viable idea is to provide the code to the backend, write it to a file, treat the file as a normal module and import it. This must be synchronous of course due to the race condition of the file being populated. But once this is complete the code will be indistinguishable from a module imported on boot, allowing for the full capabilities of a module. Meaning the results do not need to be parsed or converted, they will natively retrieve JSON objects that can be directly sent to the client.

Once the module importing option was decided upon a new problem arose in order to allow execution of all valid JavaScript, the system must support also support ES2015+ syntax. However, node.js does not yet have full native support for all these features (node.js, 2019). Silva et al (2017) discuss one solution to this is transpiling the code, from ES2015+ to ES2015 as ES2015+ does not add any feature to the language which cannot be replicated in more verbose ES2015, which is fully supported by node.js. However, transpiling comes at a cost, all code must be transformed before execution this is in most cases a trivial issue as all code is transpiled at the beginning. In this circumstance as ES6 features are supported in the IDE every single code execution must be transpiled to ES2015 to run, a non-trivial performance cost. Another and perhaps better solution to this is via the use of ESM, a module loader that transforms ECMAScript2015+ modules at run time rather than transpiling them (Benny, 2018). This can be added to the entire system via simply running the server's root JavaScript file through the ESM module loader rather than directly. The elegance of this solution comes from the issue that the server itself uses ES16+ syntax, the ESM module loader combats both this and the issue of ES16+ in integrations simultaneously.

Figure 5. The Code Execution flow

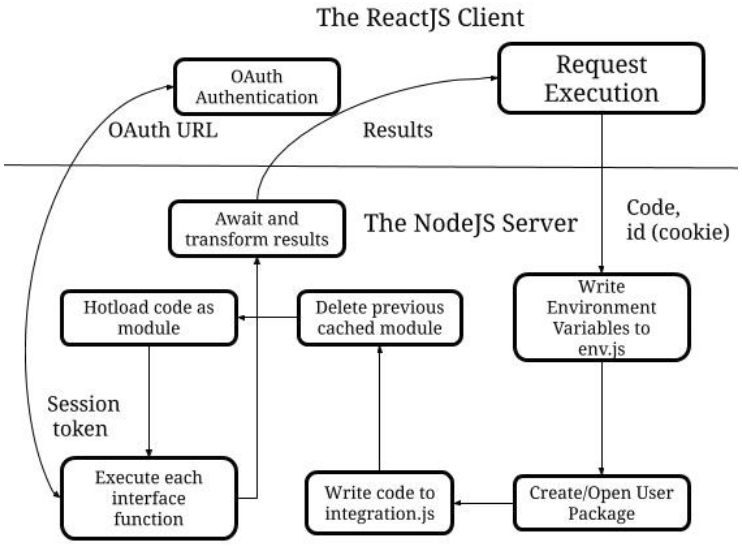


Figure 5 represents the phases each code execution passes through when an execution request is sent from the user. This was achieved using a promise chain, of course, some items on the list are conditional, i.e. a user package is only created if one currently does not exist. The environment is set up as an empty module exporting the fields specified by the user as a JSON Object. Additionally, in order for the stub to have its methods made accessible, code specifying all interface functions as exports is appended.

All integrations imported with ES2015 dependency management system, rather than with ES16. However, the

integrations themselves may import the modules using whichever syntax the volunteer engineer chooses, this is facilitated by the ESM module loader. ES15+ `import` statement specifically makes the dependency cache inaccessible during run-time. Even with ES2015's require, modules are cached and even if it is explicitly stated that the module should be reimported node will simply retrieve this cached version. In order to facilitate dynamic hot-loading multiple times in a session, the system must resolve the cached module and dereference it, it can then be re-imported.

Once imported the module's interface functions are executed, feeding them arguments identical to what they receive in tomorrow. The `connect` interface method receives parameters in the form of two functions, the first of which is the app fetches the username and password of the current user. In the IDE the volunteer engineer must instead fill out an authentication panel. The other method requests a web view to be resolved for OAuth, discussed later.

The outcome of the connect method is stored and then passed as the state parameter into collect, once this is complete the remainder of the operations benefit greatly from node.js's asynchronous nature. The calculation of code similarity, the transformation of the `collect` result in various representations and fetching the logs takes place simultaneously. These results are emitted to the client.

Miscellaneous Features

Concurrency. To allow many volunteer developers to access the system simultaneously, each user will be assigned a unique id as a cookie upon their first entry into the site, this id is used to generate their session package. A user package has all the necessary file components for one to develop: their simulated environment and the module, user packages are persistent as long as their corresponding cookie is, meaning that their code is stored on the server rather than locally on the browser. These ids must be generated in such a way that it is extremely unlikely one user should be able to guess the id of another user to interfere with their integration, because of this a Universally Unique Identifier (UUID) is used.

Logging and Debugging. For TIDE to see any long-form use, it must provide adequate information for debugging. This presents an issue and is clear why a virtual machine is used in the case of some web IDEs, the execution of the code is controlled by a server with a console that is not accessible. If volunteer developers are to have any idea why their code is not behaving as expected, they must have a means of receiving error messages or retrieving values in the program. Execution state, whether successful or otherwise is provided to the volunteer engineer using pop-ups, if no meaning error name can be found the message "an error occurred but not message could be retrieved appears" this was judged to be better than no message at all.

To implement logging a very simple class was made available to the user, they simply import a logger class from the project files (a path is specified in the IDE documentation) and then build a stack of logs by calling functions within the logger. Typical logging levels are provided such as 'info', 'debug', etc. To access these logs and return them to the client-side, a logger is then added to the export function of the aforementioned module stub. However, due to the fact, the logger's presence is not guaranteed an additional line must be appended to the module exports, the contents of "logger" whether present or not and exports that instead.

Execution Results. Once the user has executed their integration they must be able to view the results, several display formats will be returned to the user. The first of which is a graph mapping the co2 usage over time allowing them to visualise their data as a time-series. Next cumulative co2 emissions will be mapped using an appropriate carbon model (provided by Tomorrow) to see if their results are reasonable. Finally, their data will be returned in raw JSON, the response the programmer might expect to see initially.

OAuth. Volunteer engineers are able to use OAuth in their integration, this is where the differences in the nature of server-based code execute become more evident. In order to initially resolve the web view to a value, i.e. a session token, the volunteer engineer must often enter values into this web view. This, of course, presents an issue as their code is tested on the node.js server making the volunteer engineer entirely unable to continue this process and leaving the system frozen.

The OAuth flow was implemented with the use of WebSockets, the request is initially sent via HTTP to the backend which will then begin the execution cycle as normal. Once the connect interface method is called a well-formed OAuth integration authorisation takes place via a web view, a promise is set up on the server that emits a signal to a receiving client-side WebSocket. On receiving this transmission the client creates a new window using the URL provided by the server, providing a means for the volunteer engineer to enter their authentication information. Once the user has logged in to the window (this can also take place automatically if details are stored) the server-side promise resolves as it receives a response from the client and can continue execution with valid authentication.

State Injection. Some volunteer developers may not want to run the `connect` component of their system on occasion, this may be to probe the system for a bug or perhaps because their API has a finite number of calls not well-suited to the development of the system. The tomorrow application itself will also rarely authenticate, instead opting to store this information where possible, hence engineers may be interested in testing the more likely circumstance of connection state already existing.

In a client-side modal, volunteer engineers can specify

well-formed JSON (enforced by component) that is passed as a parameter with the code into the server. If this state injection is present the server bypasses the execution of certain features and instead only executes the `collect` method, using the state provided. This may also be convenient for providing expired tokens for testing.

Testing

System testing is primarily focussed on the code similarity element of the project. This is a perfect domain for unit testing, the comparison engine produces an output of a float (S) indicating the two functions are considered identical. Many of the requirements of JeSSE are specified in such phrasing as "the inclusion of x should not affect the outcome of the comparison", because of these tests could be created in such a manner. Some such tests include:

- Same nodes in a different order
- Same nodes in a different order with some inside control structure
- If statement child nodes are flattened
- For loop child nodes are flattened
- ... for all deep nodes
- Lambda expression content flattened
- Logging nodes are ignored
- Comments are ignored
- Called functions are inlined

Hyperparameter Tuning

The final phase of optimising JeSSE involves tuning various numeric hyperparameters. (1) The Missing modifier increment, for each missing match increase a modifier by this value, this modifier is ultimately multiplied with the similarity score. This is used as it was judged to be logically more effective in identifying similarity, for every missing line the code becomes less and less likely to be largely similar. (2) Difference values, the similarity punishment value assigned to a specific difference type. (3) The maximum punishment value one node can receive, some nodes can be so expansive punishment values are disproportionate see Listing 4 & 5.

```
function sleepTest() {
    let start = new Date().getTime();
    sleep(1000 * 60 * 60 * 24);
    return new Date().getTime() - start;
}
```

Listing 4: Function returning contents of previous call expression

```
function sleepTest() {
    let start = new Date().getTime();
    sleep(86400000);
    return new Date().getTime() - start;
}
```

Listing 5: Function returning contents of previous call expression

If no maximum punishment value is set here, due to the manner that JeSSE recursively evaluates nodes even inside of call expressions this produces an TS of 30, which has been regarded as disproportionately low for such a change. A naive solution to this is to say why not treat all numeric operations and such as one numeric literal, the calculation in listing 4 is trivial and present for clarity sake but in real integrations, it is unlikely all calculations will be deterministic.

The initial tuning attempt involved the code comparisons used to assess the effectiveness of JeSSE vs MOSS (Listings x to y), however, once these were optimised (see table x,y,z) real comparison of integrations were judged to perform poorly. These tests were replaced with integration tests that were judged to perform poorly with the new tuning, however, MOSS performed very poorly here as the integrations are not in anyway plagiarised. So MOSS was dropped as a comparison, due to the problem of similarity a set of heuristics were formed. The first 4 tests must be greater than 1 but ideally as low as possible, the final 3 tests should be as close to 0 as possible.

Combinatorial Tuning. Although there is little doubt an approach of assessing combinatorial parameters thoroughly would lead to the system functioning more effectively, due to the oracle problem brought about as a result of the problem of similarity an enormous amount of manual work. This will be discussed further later on.